A Sufficient Introduction to R

Derek L. Sonderegger

2016-09-01

Contents

Chapter 1

Introduction

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

R is a script based language, so there is no point and click interface. (Actually there are packages that attempt to provide a point and click interface, but they are still somewhat primitive.) While the initial learning curve will be steeper, understanding how to write scripts will be valuable because it leaves a clear description of what steps you performed in your data analysis. Typically you will want to write a script in a separate file and then run individual lines. The saves you from having to retype a bunch of commands and speeds up the debugging process.

This document is a very brief introduction to using R in my course. I highly recommend downloading and reading/skimming the manual "An Introduction to R" which is located at cran.r-project.org/doc/manuals/R-intro.pdf.

Finding help about a certain function is very easy. At the prompt, just type `help(function.name)` or `?function.name`. If you don't know the name of the function, your best bet is to go the the web page www.rseek.org which will search various R resources for your keyword(s). Another great resource is the coding question and answer site stackoverflow.

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.

When you first open up R (or RStudio) the console window gives you some information about the version of R you are running and then it gives the prompt `>`. This prompt is waiting for you to input a command. The prompt + tells you that the current command is spanning multiple lines. In a script file you might have typed something like this:

```
for( i in 1:5 ){
    print(i)
}
```

But when you copy and paste it into the console in R you'll see something like this:

```
> for (i in 1:5){
+    print(i)
+ }
```

If you type your commands into a file, you won't type the `>` or `+` prompts. For the rest of the tutorial, I will show the code as you would type it into a script and I will show the output being shown with two hashtags (`##`) before it to designate that it is output.

1.1 R as a simple calculator

Assuming that you have started R on whatever platform you like, you can use R as a simple calculator. At the prompt, type 2+3 and hit enter. What you should see is the following

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

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

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

```
6*8
```

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

```
4^3
```

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

```
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 hereSee next section.. Don't do that as the `name=value` notation is making an association mapping and not a permanent assignment.

1.2 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[1]. *If you don't assign the result to a variable, you have no way of accessing the result.* [2]

1.3 Scripts and RMarkdown

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 script (or RMarkdown) 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 scripts (and RMarkdown) from the beginning. Furthermore, having a script file fully documents how you did your analysis, which can help when writing the methods section of a paper. Finally, having a script makes it easy to re-run an analysis after a change in the data (additional data values, transformed data, or removal of outliers).

It often makes your script more readable if you break a single command up into multiple lines. R 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.

[1] To paraphrase Beyonce, "Cause if you liked it, then you should have put a name on it."

[2] This isn't strictly true, the variable `.Last.value` always has the result of the last expression evaluated, but you can't go any farther back.

1.3.1 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.

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

```
# 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.

1.3.2 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[3] 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.

[3] Clark's third law states "Any sufficiently advanced technology is indistinguishable from magic."

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 document using RMarkdown.

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...`

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.

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

To get information about a package, you can use either of the following afer the library has been loaded:

```
library(help='faraway')
```

1.5 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

Vectors

R operates on vectors where we think of a vector as a collection of objects, usually numbers. The first thing we need to be able to do is define an arbitrary collection using the `c()` function[1].

```
# Define the vector of numbers 1, ..., 4
c(1,2,3,4)
```

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

There are many other ways to define vectors. The function `rep(x, times)` just repeats `x` a the number times specified by `times`.

```
rep(2, 5)              # repeat 2 five times... 2 2 2 2 2
```

```
## [1] 2 2 2 2 2
```

```
rep( c('A','B'), 3 )   # repeat A B three times  A B A B A B
```

```
## [1] "A" "B" "A" "B" "A" "B"
```

Finally, we can also define a sequence of numbers using the `seq(from, to, by, length.out)` function which expects the user to supply 3 out of 4 possible arguments. The possible arguments are `from`, `to`, `by`, and `length.out`. From is the starting point of the sequence, to is the ending point, by is the difference between any two successive elements, and `length.out` is the total number of elements in the vector.

```
seq(from=1, to=4, by=1)
```

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

```
seq(1,4)        # 'by' has a default of 1
```

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

```
1:4             # a shortcut for seq(1,4)
```

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

```
seq(1,5, by=.5)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```
seq(1,5, length.out=11)
```

```
##  [1] 1.0 1.4 1.8 2.2 2.6 3.0 3.4 3.8 4.2 4.6 5.0
```

If we have two vectors and we wish to combine them, we can again use the `c()` function.

[1] The "c" stands for collection.

```
vec1 <- c(1,2,3)
vec2 <- c(4,5,6)
vec3 <- c(vec1, vec2)
vec3
```

```
## [1] 1 2 3 4 5 6
```

2.1 Accessing Vector Elements

Suppose I have defined a vector

```
foo <- c('A', 'B', 'C', 'D', 'F')
```

and I am interested in accessing whatever is in the first spot of the vector. Or perhaps the 3rd or 5th element. To do that we use the `[]` notation, where the square bracket represents a subscript.

```
foo[1]  # First element in vector foo
```

```
## [1] "A"
```

```
foo[4]  # Fourth element in vector foo
```

```
## [1] "D"
```

This subscripting notation can get more complicated. For example I might want the 2nd and 3rd element or the 3rd through 5th elements.

```
foo[c(2,3)]  # elements 2 and 3
```

```
## [1] "B" "C"
```

```
foo[ 3:5 ]   # elements 3 to 5
```

```
## [1] "C" "D" "F"
```

Finally, I might be interested in getting the entire vector except for a certain element. To do this, R allows us to use the square bracket notation with a negative index number.

```
foo[-1]          # everything but the first element
```

```
## [1] "B" "C" "D" "F"
```

```
foo[ -1*c(1,2) ] # everything but the first two elements
```

```
## [1] "C" "D" "F"
```

Now is a good time to address what is the `[1]` doing in our output? Because vectors are often very long and might span multiple lines, R is trying to help us by telling us the index number of the left most value. If we have a very long vector, the second line of values will start with the index of the first value on the second line.

```
# The letters vector is a vector of all 26 lower-case letters
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

Here the `[1]` is telling me that `a` is the first element of the vector and the `[18]` is telling me that `r` is the 18th element of the vector.

2.2 Scalar Functions Applied to Vectors

It is very common to want to perform some operation on all the elements of a vector simultaneously. For example, I might want take the absolute value of every element. Functions that are inherently defined on single values will almost always apply the function to each element of the vector if given a vector.

```
x <- -5:5
x
```

```
##  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

```
abs(x)
```

```
##  [1] 5 4 3 2 1 0 1 2 3 4 5
```

```
exp(x)
```

```
##  [1] 6.737947e-03 1.831564e-02 4.978707e-02 1.353353e-01 3.678794e-01
##  [6] 1.000000e+00 2.718282e+00 7.389056e+00 2.008554e+01 5.459815e+01
## [11] 1.484132e+02
```

2.3 Vector Algebra

All algebra done with vectors will be done element-wise by default.For matrix and vector multiplication as usually defined by mathematicians, use `%*%` instead of `*`. So two vectors added together result in their individual elements being summed.

```
x <- 1:4
y <- 5:8
x + y
```

```
## [1]  6  8 10 12
```

```
x * y
```

```
## [1]  5 12 21 32
```

R does another trick when doing vector algebra. If the lengths of the two vectors don't match, R will recycle the elements of the shorter vector to come up with vector the same length as the longer. This is potentially confusing, but is most often used when adding a long vector to a vector of length 1.

```
x <- 1:4
x + 1
```

```
## [1] 2 3 4 5
```

2.4 Commonly Used Vector Functions

Function	Result
`min(x)`	Minimum value in vector x
`max(x)`	Maximum value in vector x
`length(x)`	Number of elements in vector x
`sum(x)`	Sum of all the elements in vector x
`mean(x)`	Mean of the elements in vector x
`median(x)`	Median of the elements in vector x
`var(x)`	Variance of the elements in vector x

Function	Result
`sd(x)`	Standard deviation of the elements in x

Putting this all together, we can easily perform tedious calculations with ease. To demonstrate how scalars, vectors, and functions of them work together, we will calculate the variance of 5 numbers. Recall that variance is defined as

$$Var\left(x\right) = \frac{\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2}{n-1}$$

```
x <- c(2,4,6,8,10)
xbar <- mean(x)          # calculate the mean
xbar
```

```
## [1] 6
```

```
x - xbar                 # calculate the errors
```

```
## [1] -4 -2  0  2  4
```

```
(x-xbar)^2
```

```
## [1] 16  4  0  4 16
```

```
sum( (x-xbar)^2 )
```

```
## [1] 40
```

```
n <- length(x)           # how many data points do we have
n
```

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

```
sum((x-xbar)^2)/(n-1)    # calculating the variance by hand
```

```
## [1] 10
```

```
var(x)                   # Same thing using the built-in variance function
```

```
## [1] 10
```

2.5 Exercises

1. Create a vector of three elements (2,4,6) and name that vector `vec_a`. Create a second vector, `vec_b`, that contains (8,10,12). Add these two vectors together and name the result `vec_c`.
2. Create a vector, named `vec_d`, that contains only two elements (14,20). Add this vector to `vec_a`. What is the result and what do you think R did (look up the recycling rule using Google)? What is the warning message that R gives you?
3. Next add 5 to the vector vec_a. What is the result and what did R do? Why doesn't in give you a warning message similar to what you saw in the previous problem?
4. Generate the vector of integers $\{1, 2, \ldots 5\}$ in two different ways.
   a) First using the `seq()` function
   b) Using the `a:b` shortcut.
5. Generate the vector of even numbers $\{2, 4, 6, \ldots, 20\}$
   a) Using the seq() function and

b) Using the a:b shortcut and some subsequent algebra. *Hint: Generate the vector 1-10 and then multiple it by 2.*

6. Generate a vector of 1001 elements that are evenly placed between 0 and 1 using the `seq()` command and name this vector `x`.
7. Generate the vector $\{2, 4, 8, 2, 4, 8, 2, 4, 8\}$ using the `rep()` command to replicate the vector c(2,4,8).
8. Generate the vector $\{2, 2, 2, 2, 4, 4, 4, 4, 8, 8, 8, 8\}$ using the `rep()` command. You might need to check the help file for rep() to see all of the options that rep() will accept. In particular, look at the optional argument `each=`.
9. The vector `letters` is a built-in vector to R and contains the lower case English alphabet.
   a) Extract the 9th element of the letters vector.
   b) Extract the sub-vector that contains the 9th, 11th, and 19th elements.
   c) Extract the sub-vector that contains everything except the last two elements.

Chapter 3

Statistical Tables

Statistics makes use of a wide variety of distributions and before the days of personal computers, every statistician had books with hundreds and hundreds of pages of tables allowing them to look up particular values. Fortunately in the modern age, we don't need those books and tables, but we do still need to access those values. To make life easier and consistent for R users, every distribution is accessed in the same manner.

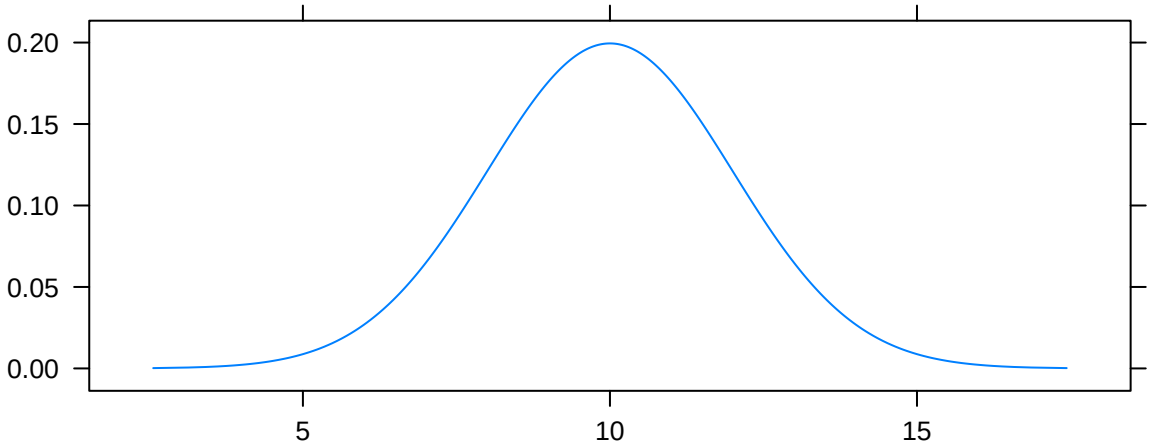
3.1 `mosaic::plotDist()` function

The `mosaic` package provides a very useful routine for understanding a distribution. The `plotDist()` function takes the R name of the distribution along with whatever parameters are necessary for that function and show the distribution. For reference below is a list of common distributions and their R name and a list of necessary parameters.

Distribution	Stem	Parameters	Parameter Interpretation
Binomial	`binom`	`size prob`	Number of Trials Probability of Success (per Trial)
Exponential	`exp`	`lambda`	Mean of the distribution
Normal	`norm`	`mean=0 sd=1`	Center of the distribution Standard deviation
Uniform	`unif`	`min=0 max=1`	Minimum of the distribution Maximum of the distribution

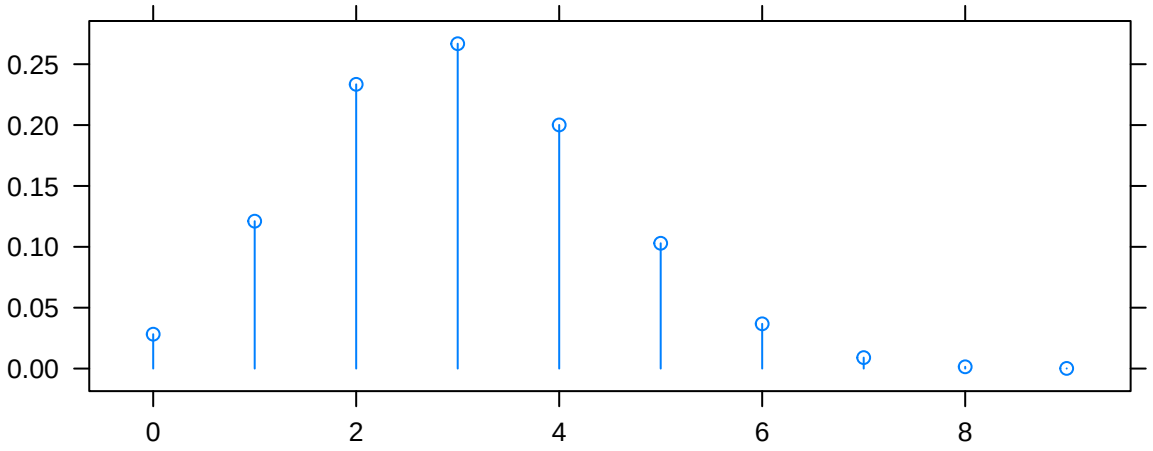
For example, to see the normal distribution with mean $mu = 10$ and standard deviation $\sigma = 2$, we use

```
library(mosaic)
plotDist('norm', mean=10, sd=2)
```



This function works for discrete distributions as well.

```
plotDist('binom', size=10, prob=.3)
```



3.2 Base R functions

All the probability distributions available in R are accessed in exactly the same way, using a `d`-function, `p`-function, `q`-function, and `r`-function. For the rest of this section suppose that X is a random variable from the distribution of interest and x is some possible value that X could take on. Notice that the `p`-function is the inverse of the `q`-function.

Function	Result
`d`-function(x)	The height of the probability distribution/density at x
`p`-function(x)	$P(X \leq x)$
`q`-function(q)	x such that $P(X \leq x) = q$
`r`-function(n)	n random observations from the distribution

For each distribution in R, there will be be this set of functions but we replace the "-function" with the distribution name or a shortened version. `norm`, `exp`, `binom`, `t`, `f` are the names for the normal, exponential, binomial, T and F distributions. Furthermore, most distributions have additional parameters that define the distribution and will also be passed as arguments to these functions, although, if a reasonable default value for the parameter exists, there will be a default.

3.2.1 d-function

The purpose of the d-function is to calculate the height of a probability mass function or a density function (The "d" actually stands for density). Notice that for discrete distributions, this is the probability of observing that particular value, while for continuous distributions, the height doesn't have a nice physical interpretation.

We start with an example of the Binomial distribution. For $X \sim Binomial\,(n = 10, \pi = .2)$ suppose we wanted to know $P(X = 0)$? We know the probability mass function is

$$P\,(X = x) = \binom{n}{x} \pi^x\,(1-\pi)^{n-x}$$

thus

$$P\,(X = 0) = \binom{10}{0} 0.2^0\,(0.8)^{10} = 1 \cdot 1 \cdot 0.8^{10} \approx 0.107$$

but that calculation is fairly tedious. To get R to do the same calculation, we just need the height of the probability mass function at 0. To do this calculation, we need to know the x value we are interested in along with the distribution parameters n and π.

The first thing we should do is check the help file for the binomial distribution functions to see what parameters are needed and what they are named.

```
?dbinom
```

The help file shows us the parameters n and π are called size and prob respectively. So to calculate the probability that $X = 0$ we would use the following command:

```
dbinom(0, size=10, prob=.2)
```

```
## [1] 0.1073742
```

3.2.2 p-function

Often we are interested in the probability of observing some value or anything less (In probability theory, we call this the cumulative density function or CDF). P-values will be calculated this way, so we want a nice easy way to do this.

To start our example with the binomial distribution, again let $X \sim Binomial\,(n = 10, \pi = 0.2)$. Suppose I want to know what the probability of observing a 0, 1, or 2? That is, what is $P\,(X \leq 2)$? I could just find the probability of each and add them up.

```
dbinom(0, size=10, prob=.2)   +       #   P(X==0) +
  dbinom(1, size=10, prob=.2) +       #     P(X==1) +
  dbinom(2, size=10, prob=.2)         #     P(X==2)
```

```
## [1] 0.6777995
```

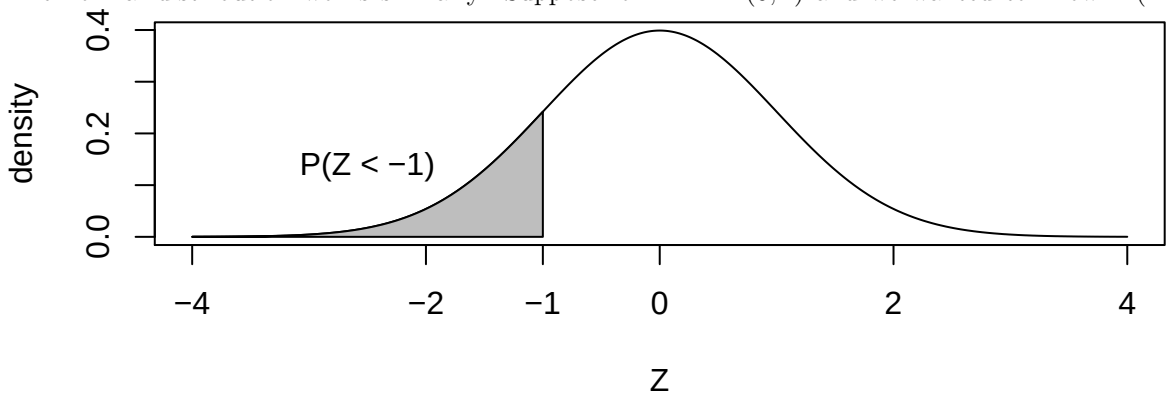
but this would get tedious for binomial distributions with a large number of trials. The shortcut is to use the `pbinom()` function.

```
pbinom(2, size=10, prob=.2)
```

```
## [1] 0.6777995
```

For discrete distributions, you must be careful because R will give you the probability of less than or equal to 2. If you wanted less than two, you should use `dbinom(1,10,.2)`.

The normal distribution works similarly. Suppose for $Z \sim N(0,1)$ and we wanted to know $P(Z \leq -1)$?



The answer is easily found via `pnorm()`.

```
pnorm(-1)
```

```
## [1] 0.1586553
```

Notice for continuous random variables, the probability $P(Z = -1) = 0$ so we can ignore the issue of "less than" vs "less than or equal to".

Often times we will want to know the probability of greater than some value. That is, we might want to find $P(Z \geq 1)$. For the normal distribution, there are a number of tricks we could use. Notably

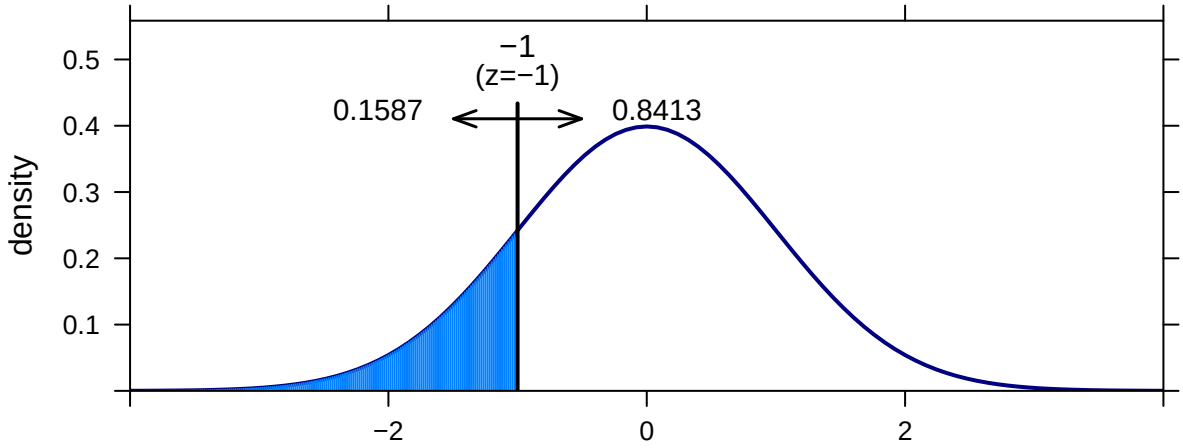
$$P(Z \geq -1) = P(Z \leq 1) = 1 - P(Z < -1)$$

but sometimes I'm lazy and would like to tell R to give me the area to the right instead of area to the left (which is the default). This can be done by setting the argument $lower.tail = FALSE$.

The `mosaic` package includes an augmented version of the `pnorm()` function called `xpnorm()` that calculates the same number but includes some extra information and produces a pretty graph to help us understand what we just calculated and do the tedious "1 minus" calculation to find the upper area. Fortunately this x-variant exists for the Normal, Chi-squared, F, Gamma continuous distributions and the discrete Poisson, Geometric, and Binomial distributions.

```
library(mosaic)
xpnorm(-1)
```

```
##
## If X ~ N(0,1), then
##
##  P(X <= -1) = P(Z <= -1) = 0.1587
##  P(X >  -1) = P(Z >  -1) = 0.8413
```




```
## [1] 0.1586553
```

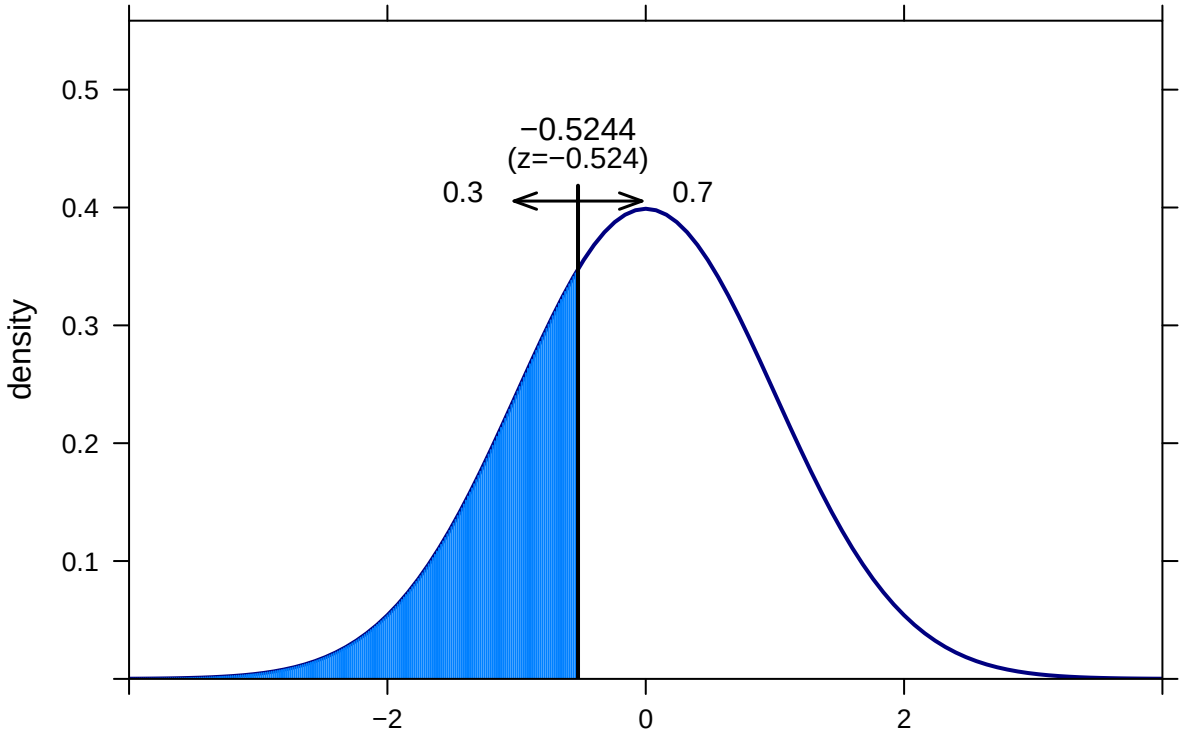
3.2.3 q-function

In class, we will also find ourselves asking for the quantiles of a distribution. Percentiles are by definition 1/100, 2/100, etc but if I am interested in something that isn't and even division of 100, we get fancy can call them quantiles. This is a small semantic quibble, but we ought to be precise. That being said, I won't correct somebody if they call these percentiles. For example, I might want to find the 0.30 quantile, which is the value such that 30% of the distribution is less than it, and 70% is greater. Mathematically, I wish to find the value z such that $P(Z < z) = 0.30$.

To find this value in the tables in a book, we use the table in reverse. R gives us a handy way to do this with the `qnorm()` function and the mosaic package provides a nice visuallization using the augmented `xqnorm()`. Below, I specify that I'm using a function in the `mosaic` package by specifying it via `PackageName::FunctionName()` but that isn't strictly necessary but can improve readability of your code.

```
mosaic::xqnorm(0.30)   # Give me the value along with a pretty picture
```

```
##  P(X <= -0.524400512708041) = 0.3
##  P(X >  -0.524400512708041) = 0.7
```




```
## [1] -0.5244005
```

```
qnorm(.30)              # No pretty picture, just the value
```

```
## [1] -0.5244005
```

3.2.4 r-function

Finally, I often want to be able to generate random data from a particular distribution. R does this with the r-function. The first argument to this function the number of random variables to draw and any remaining arguments are the parameters of the distribution.

```
rnorm(5, mean=20, sd=2)
```

```
## [1] 22.05720 23.49668 16.48997 19.03776 17.67445
```

```
rbinom(4, size=10, prob=.8)
```

```
## [1] 9 7 9 9
```

3.3 Exercises

1. We will examine how to use the probability mass functions (a.k.a. d-functions) and cumulative probability function (a.k.a. p-function) for the Poisson distribution.

   a) Create a graph of the distribution of a Poisson random variable with rate parameter $\lambda = 2$ using the mosaic function `plotDist()`.

   b) Calculate the probability that a Poisson random variable (with rate parameter $\lambda = 2$) is exactly equal to 3 using the `dpois()` function. Be sure that this value matches the graphed distribution in part (a).

c) For a Poisson random variable with rate parameter $\lambda = 2$, calculate the probability it is less than or equal to 3, by summing the four values returned by the Poisson `d`-function.

d) Perform the same calculation as the previous question but using the cumulative probability function `ppois()` or the mosaic function `xppois()`.

2. We will examine how to use the cumulative probability functions (a.k.a. p-functions) for the normal distributions.

a) Use the mosaic function `plotDist()` to produce a graph of the standard normal distribution (that is a normal distribution with mean $mu = 0$ and standard deviation $\sigma = 1$.

b) For a standard normal, use the `pnorm()` function or its `mosaic` augmented version `xpnorm()` to calculate

i. $P(Z < -1)$
ii. $P(Z \geq 1.5)$

c) Use the mosaic function `plotDist()` to produce a graph of an exponential distribution with rate parameter 2.

d) Suppose that $Y \sim Exp(2)$, as above, use the pexp() function to calculate $P(Y \leq 1)$. (Unfortunately there isn't a mosaic augmented `xpexp()` function.

3. We next examine how to use the quantile functions for the normal and exponential distributions using R's q-functions.

a) Find the value of a standard normal distribution ($\mu = 0$, $\sigma = 1$) such that 5% of the distribution is to the left of the value using the `qnorm()` function or the mosaic augmented version `xqnorm()`.

b) Find the value of an exponential distribution with rate 2 such that 60% of the distribution is less than it using the `qexp()` function.

4. Finally we will look at generating random deviates from a distribution.

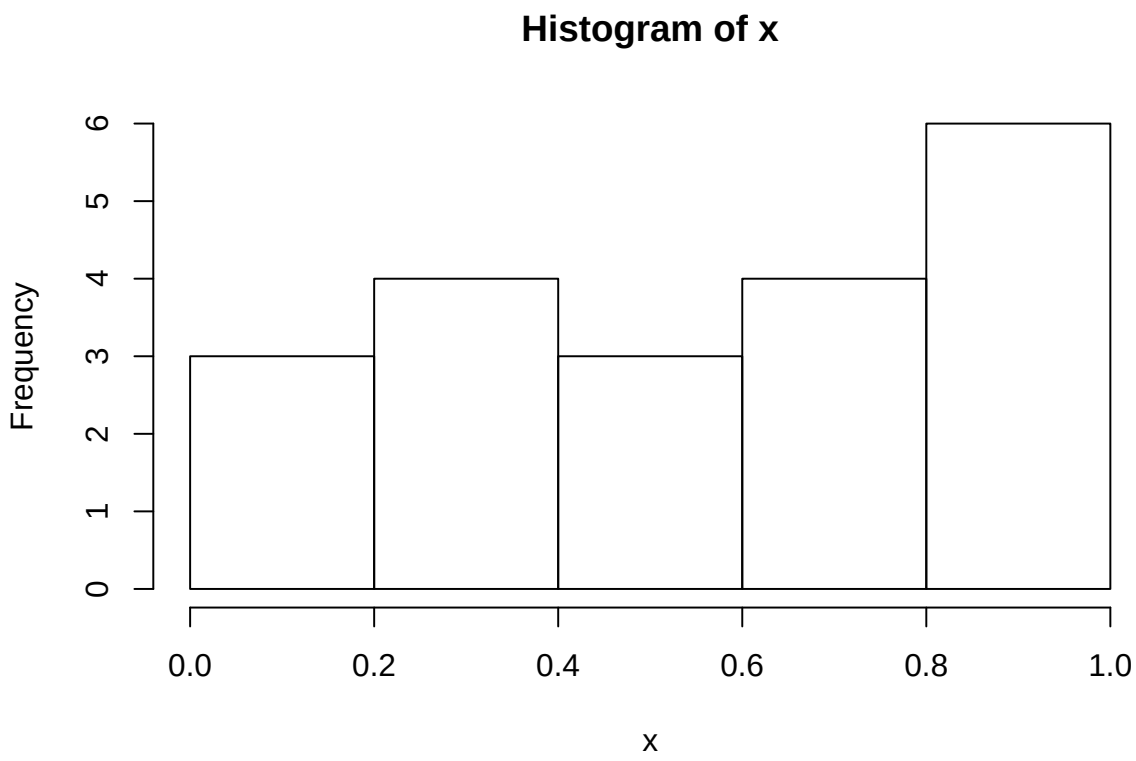
a) Generate a value of 1 from a uniform distribution with minimum 0, and maximum 1 using the `runif()` function. Repeat this step several times and confirm you are getting different values each time.

b) Generate a sample of size 20 from the same uniform distribution and save it as the vector `x` using the following:

```
x <- runif(20, min=0, max=1)
```

Then produce a histogram of the sample using the function `hist()`

```
hist(x)
```



c) Generate a sample of 2000 from a normal distribution with `mean=10` and standard deviation `sd=2` using the `rnorm()` function. Create a histogram the the resulting sample.

Chapter 4

Data Types

There are some basic data types that are commonly used.

1. Integers - These are the integer numbers $(\ldots, -2, -1, 0, 1, 2, \ldots)$. To convert a numeric value to an integer you may use the function `as.integer()`.
2. Numeric - These could be any number (whole number or decimal). To convert another type to numeric you may use the function as.numeric().
3. Strings - These are a collection of characters (example: Storing a student's last name). To convert another type to a string, use as.character().
4. Factors - These are strings that can only values from a finite set. For example we might wish to store a variable that records home department of a student. Since the department can only come from a finite set of possibilities, I would use a factor. Factors are categorical variables, but R calls them factors instead of categorical variable. A vector of values of another type can always be converted to a factor using the as.factor() command.
5. Logicals - This is a special case of a factor that can only take on the values `TRUE` and `FALSE`. (Be careful to always capitalize `TRUE` and `FALSE`. Because R is case-sensitive, TRUE is not the same as true. Using the function `as.logical()` you can convert numeric values to `TRUE` and `FALSE` where `0` is `FALSE` and anything else is `TRUE`.

Depending on the command, R will coerce your data if necessary, but it is a good habit to do the coercion yourself. If a variable is a number, R will automatically assume that it is continuous numerical variable. If it is a character string, then R will assume it is a factor when doing any statistical analysis.

To find the type of an object, the `str()` command gives the type, and if the type is complicated, it describes the structure of the object.

4.1 Integers and Numerics

Integers and numerics are exactly what they sound like. Integers can take on whole number values, while numerics can take on any decimal value. The reason that there are two separate data types is that integers require less memory to store than numerics. For most users, the distinction can be ignored.

```
x <- c(1,2,1,2,1)
# show that x is of type 'numeric'
str(x)   # the str() command show the STRucture of the object
```

```
##  num [1:5] 1 2 1 2 1
```

4.2 Character Strings

In R, we can think of collections of letters and numbers as a single entity called a string. Other programming languages think of strings as vectors of letters, but R does not so you can't just pull off the first character using vector tricks. In practice, there are no limits as to how long string can be.

```
x <- "Goodnight Moon"

# Notice x is of type character (chr)
str(x)
```

```
##  chr "Goodnight Moon"
```

```
# R doesn't care if I use single quotes or double quotes, but don't mix them...
y <- 'Hop on Pop!'

# we can make a vector of character strings
Books <- c(x, y, 'Where the Wild Things Are')
Books
```

```
## [1] "Goodnight Moon"            "Hop on Pop!"
## [3] "Where the Wild Things Are"
```

Character strings can also contain numbers and if the character string is in the correct format for a number, we can convert it to a number.

```
x <- '5.2'
str(x)     # x really is a character string
```

```
##  chr "5.2"
```

```
x
```

```
## [1] "5.2"
```

```
as.numeric(x)
```

```
## [1] 5.2
```

If we try an operation that only makes sense on numeric types (like addition) then R complain unless we first convert it. There are places where R will try to coerce an object to another data type but it happens inconsistently and you should just do the conversion yourself

```
x+1
```

```
## Error in x + 1: non-numeric argument to binary operator
```

```
as.numeric(x) + 1
```

```
## [1] 6.2
```

4.3 Factors

Factors are how R keeps track of categorical variables. R does this in a two step pattern. First it figures out how many categories there are and remembers which category an observation belongs two and second, it keeps a vector character strings that correspond to the names of each of the categories.

```
# A charater vector
y <- c('B','B','A','A','C')
y
```

```
## [1] "B" "B" "A" "A" "C"
```

```
# convert the vector of characters into a vector of factors
z <- factor(y)
str(z)
```

```
##  Factor w/ 3 levels "A","B","C": 2 2 1 1 3
```

Notice that the vector `z` is actually the combination of group assignment vector `2,2,1,1,3` and the group names vector `"A","B","C"`. So we could convert z to a vector of numerics or to a vector of character strings.

```
as.numeric(z)
```

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

```
as.character(z)
```

```
## [1] "B" "B" "A" "A" "C"
```

Often we need to know what possible groups there are, and this is done using the `levels()` command.

```
levels(z)
```

```
## [1] "A" "B" "C"
```

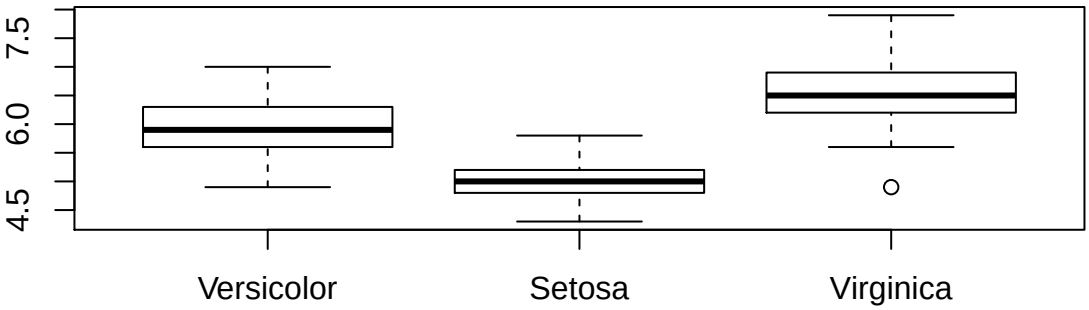
Notice that the order of the group names was done alphabetically, which we did not chose. This ordering of the levels has implications when we do an analysis or make a plot and R will always display information about the factor levels using this order. It would be nice to be able to change the order. Also it would be really nice to give more descriptive names to the groups rather than just the group code in my raw data. I find it is usually easiest to just convert the vector to a character vector, and then convert it back using the `levels=` argument to define the order of the groups, and labels to define the modified names.

```
z <- factor(z,                        # vector of data levels to convert
            levels=c('B','A','C'),    # Order of the levels
            labels=c("B Group", "A Group", "C Group")) # Pretty labels to use
z
```

```
## [1] B Group B Group A Group A Group C Group
## Levels: B Group A Group C Group
```

For the Iris data, the species are ordered alphatetically. We might want to re-order how they appear in a graphs to place Versicolor first. The `Species` names are not capitalized, and perhaps I would like them to begin with a capital letter.

```
iris$Species <- factor( iris$Species,
                        levels = c('versicolor','setosa','virginica'),
                        labels = c('Versicolor','Setosa','Virginica'))
boxplot( Sepal.Length ~ Species, data=iris)
```



Often we wish to take a continuous numerical vector and transform it into a factor. The function cut() takes a vector of 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
# Notice that the the outside breakpoints must include all the data points.
# That is, the smallest break must be smaller than all the data, and the largest
# must be larger (or equal) to all the data.
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]
```

```
# 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
```

4.4 Logicals

Often I wish to know which elements of a vector are equal to some value, or are greater than something. R allows us to make those tests at the vector level.

Very often we need to make a comparison and test if something is equal to something else, or if one thing is bigger than another. To test these, we will use the `<`, `<=`, `==`, `>=`, `>`, and `!=` operators. These can be used similarly to

```
6 < 10    # 6 less than 10?
```

```
## [1] TRUE
```

```
6 == 10   # 6 equal to 10?
```

```
## [1] FALSE
```

```
6 != 10   # 6 not equal to 10?
```

```
## [1] TRUE
```

where we used 6 and 10 just for clarity. The result of each of these is a logical value (a `TRUE` or `FALSE`). In most cases these would be variables you had previously created and were using.

Suppose I have a vector of numbers and I want to get all the values greater than 16. Using the $>$ comparison, I can create a vector of logical values that tells me if the specified value is greater than 16. The `which()` takes a vector of logicals and returns the indices that are true.

```
x <- -10:10     # a vector of 20 values, (11th element is the 0)
x
```

```
##  [1] -10  -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1   2   3   4   5   6
## [18]   7   8   9  10
```

```
x > 0           # a vector of 20 logicals
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
which( x > 0 )  # which vector elements are > 0
```

```
##  [1] 12 13 14 15 16 17 18 19 20 21
```

```
x[ which(x>0) ] # Grab the elements > 0
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

On function I find to be occasionally useful is the `is.element(el, set)' function which allows me to figure out which elements of a vector are one of a set of possibilities. For example, I might want to know which elements of the`letters‘ vector are vowels.

```
letters  # this is all 26 english lowercase letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
vowels <- c('a','e','i','o','u')
which( is.element(letters, vowels) )
```

```
## [1]  1  5  9 15 21
```

This shows me the vowels occur at the 1st, 5th, 9th, 15th, and 21st elements of the alphabet.

Often I want to make multiple comparisons. For example given a bunch of students and a vector of their GPAs and another vector of their major, maybe I want to find all undergraduate Forestry majors with a GPA greater than 3.0. Then, given my set of university students, I want ask two questions: Is their major Forestry, and is their GPA greater than 3.0. So I need to combine those two logical results into a single logical that is true if both questions are true.

The command & means “and” and | means “or”. We can combine two logical values using these two similarly:

```
TRUE  & TRUE     # both are true so combo so result is true
```

```
## [1] TRUE
```

```
TRUE  & FALSE    # one true and one false so result is false
```

```
## [1] FALSE
```

```
FALSE & FALSE    # both are false so the result is false
```

```
## [1] FALSE
```

```
TRUE  | TRUE     # at least one is true -> TRUE
```

```
## [1] TRUE
```

```
TRUE  | FALSE    # at least one is true -> TRUE
```

```
## [1] TRUE
```

```
FALSE | FALSE    # neither is true -> FALSE
```

```
## [1] FALSE
```

4.5 Exercises

1. Create a vector of character strings with six elements

```
test <- c('red','red','blue','yellow','blue','green')
```

and then

a. Transform the `test` vector just you created into a factor.
b. Use the `levels()` command to determine the levels (and order) of the factor you just created.
c. Transform the factor you just created into integers. Comment on the relationship between the integers and the order of the levels you found in part (b).
d. Use some sort of comparison to create a vector that identifies which factor elements are the red group.

2. Given the vector of ages,

```
ages <- c(17, 18, 16, 20, 22, 23)
```

create a factor that has levels `Minor` or `Adult` where any observation greater than or equal to 18 qualifies as an adult. Also, make sure that the order of the levels is `Minor` first and `Adult` second.

3. Suppose we vectors that give a students name, their GPA, and their major. We want to come up with a list of forestry students with a GPA of greater than 3.0.

```
Name <- c('Adam','Benjamin','Caleb','Daniel','Ephriam', 'Frank','Gideon')
GPA <- c(3.2, 3.8, 2.6, 2.3, 3.4, 3.7, 4.0)
Major <- c('Math','Forestry','Biology','Forestry','Forestry','Math','Forestry')
```

a) Create a vector of TRUE/FALSE values that indicate whether the students GPA is greater than 3.0.
b) Create a vector of TRUE/FALSE values that indicate whether the students' major is forestry.
c) Create a vector of TRUE/FALSE values that indicates if a student has a GPA greater than 3.0 and is a forestry major.
d) Convert the vector of TRUE/FALSE values in part (c) to integer values using the `as.numeric()` function. Which numeric value corresponds to TRUE?
e) Sum (using the `sum()` function) the vector you created to count the number of students with GPA > 3.0 and are a forestry major.

4. Make two variables, and call them `a` and `b` where `a=2` and `b=10`. I want to think of these as defining an interval.

a. Define the vector `x <- c(-1, 5, 12)`
b. Using the `&`, come up with a comparison that will test if the value of `x` is in the interval $[a, b]$. (We want the test to return `TRUE` if $a \leq x \leq b$). That is, test if `a` is less than `x` and if `x` is less than `b`. Confirm that for x defined above you get the correct vector of logical values.
c. Similarly make a comparison that tests if `x` is outside the interval $[a, b]$ using the `|` operator. That is, test if `x < a` or `x > b`. I want the test to return TRUE is x is less than a or if x is greater than b. Confirm that for x defined above you get the correct vector of logical values.

Chapter 5

Basic Graphing

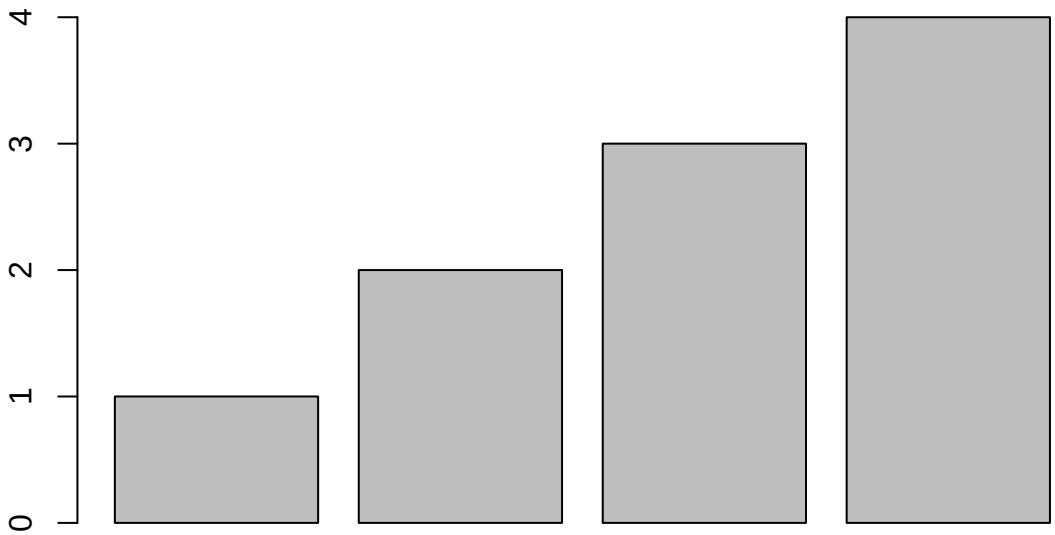
One of the most important things we can do in exploratory statistics is to just look at the data in graphical form so that we can more easily see trends. Most of the graph functions have similar options for setting the main title, x-label, y-labels, etc.

5.1 Univariate Graphs

6.1.1 Barcharts

This is chart that is intended to show the relative differences between a number of groups. To create this graph, all we need is the heights of the individual bars.

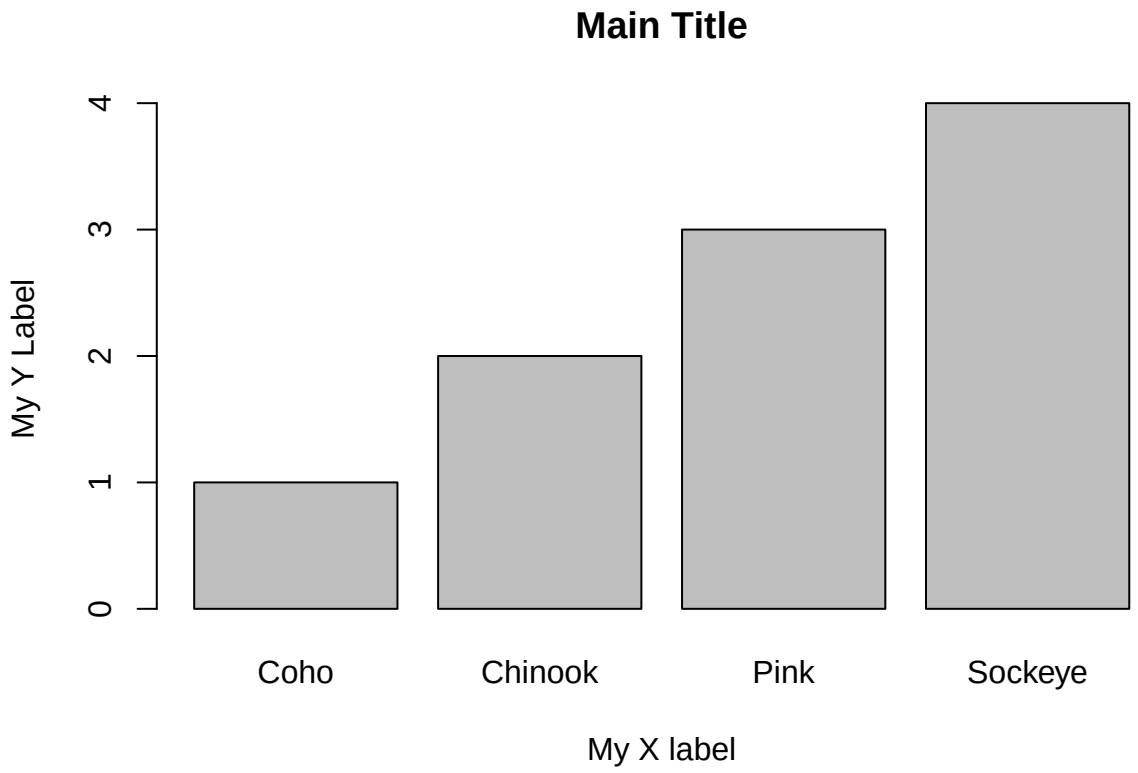
```
x <- 1:4      # The vector 1,2,3,4
barplot(x)    # First bar has height 1, next is 2, etc
```



This is extremely basic, but is a bit ugly. We'll add a x and y labels and names for each bar.

```
x <- 1:4
barplot(x,
        xlab='My X label',
        ylab='My Y Label',
        main='Main Title',
```

```
          names.arg=c('Coho','Chinook','Pink','Sockeye')
        )
```

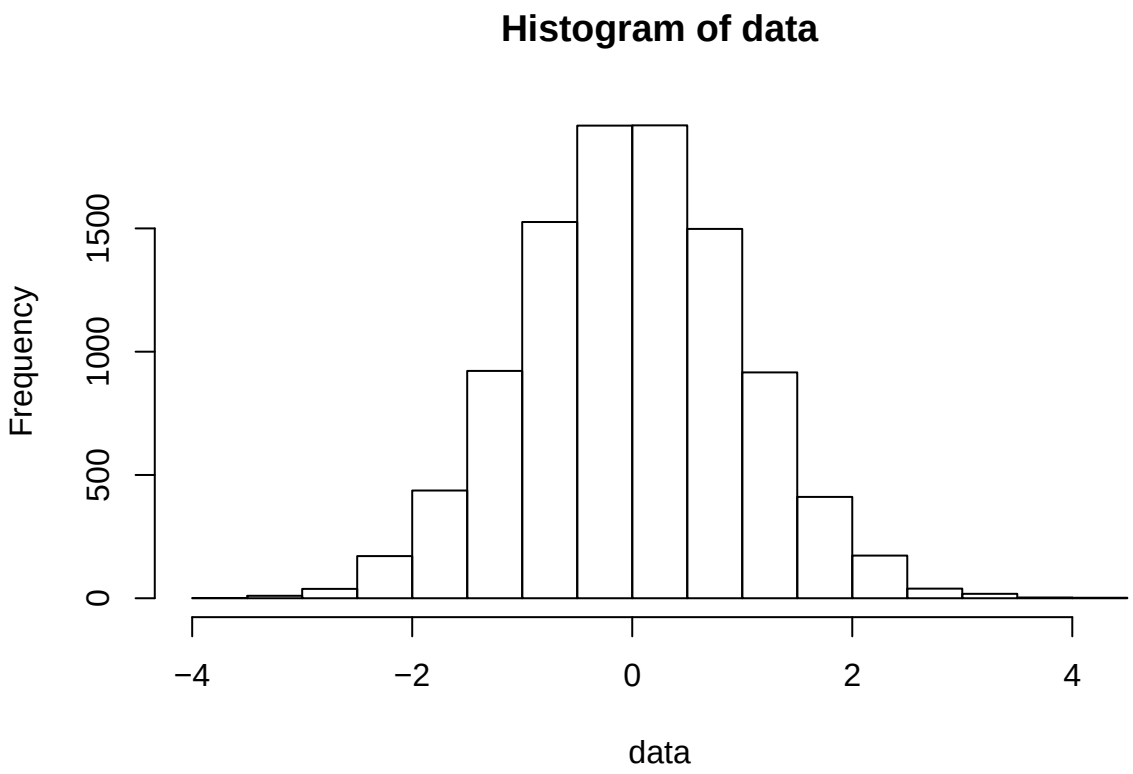


5.1.1 Histograms

Histograms are way of summarizing a large number of observations. It allows the reader to see what values are very common and along with the range of the data. The primary aspects of this plot that you might want to change are the number of histogram bins and where the breaks between bins occurs.

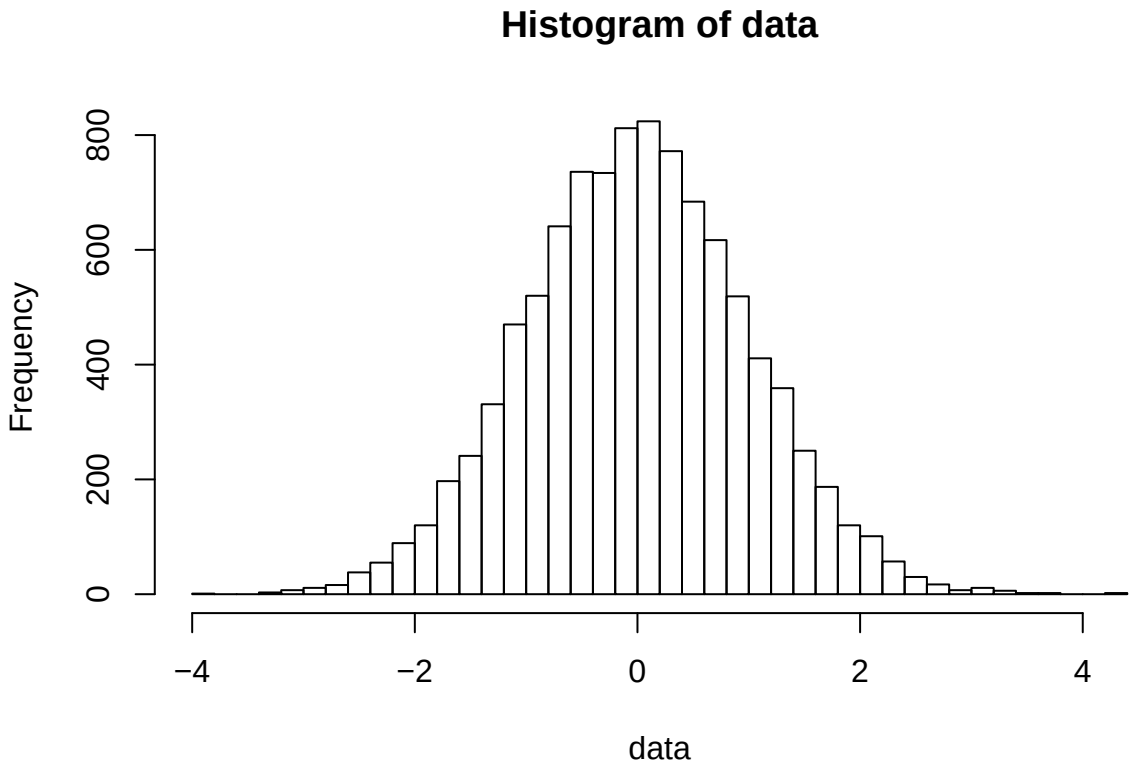
To generate data for the histogram, we'll use the function `rnorm(n, mean=0, sd=1)` which generates n random variables from a normal distribution with mean `mean` and standard deviation `sd`. Notice that rnorm will default to giving random variables from a standard normal distribution.

```
data <- rnorm(10000)
hist(data)
```



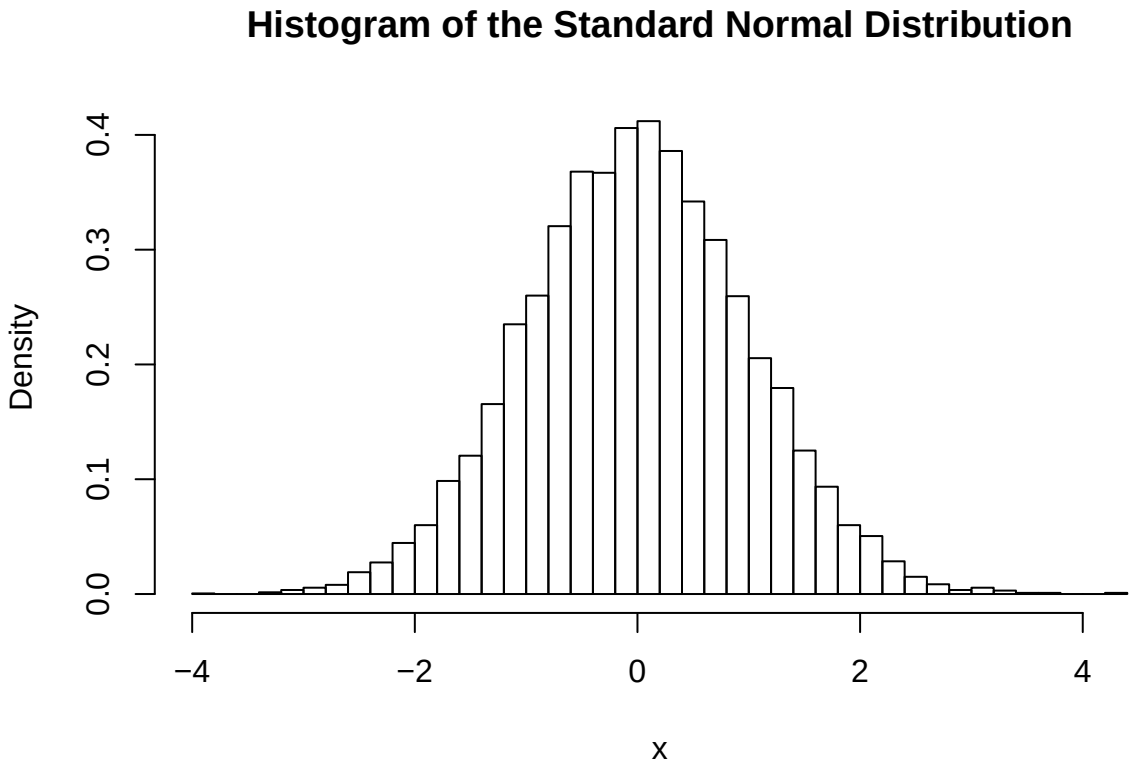
R does a decent job of picking reasonable defaults for the number of bins and breakpoints, but with 10,000 observations, I think we should have more bins. To do this, I'll use the optional breaks argument to specify where to split the bins.

```
hist(data, breaks=30)
```



Finally I might want the y-axis to not be the number of observations within a bin, but rather the density of an observation. Density is calculated by taking the number of observations in a bin and then dividing by the total number of observations and then dividing by width of the bin. This forces the sum of the area in all of the bins to be 1. To do this, we'll use the freq argument and set it to be FALSE.

```
hist(data, breaks=30, freq=FALSE, xlab='x',
     main='Histogram of the Standard Normal Distribution' )
```

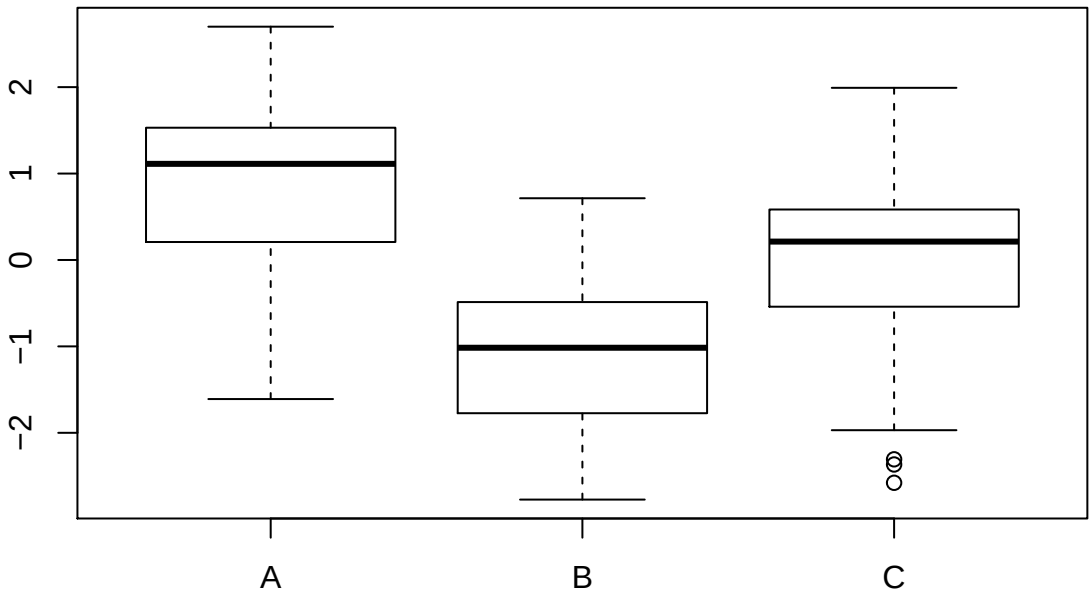


5.2 Bivariate Plots

5.2.1 Boxplots

Boxplots are designed to compare distributions among multiple groups.

```
y <- c( rnorm(50, mean=1),    # Group A is centered at 1
        rnorm(50, mean=-1),   # Group B is centered at -1
        rnorm(50, mean=0))    # Group C is centered at 0
group <- c( rep('A', 50), rep('B',50), rep('C',50) )
boxplot(y ~ group)
```



When calling boxplot() the main argument is a formula that describes a relationship between two variables. Formulas in R are always in the format `y.variable ~ x.variable` where I think of this as saying my y-variable is a function of the x-variable.

You can get the boxplot to separate on more than one variable by using `y.variable ~ x1.variable + x2.variable`.

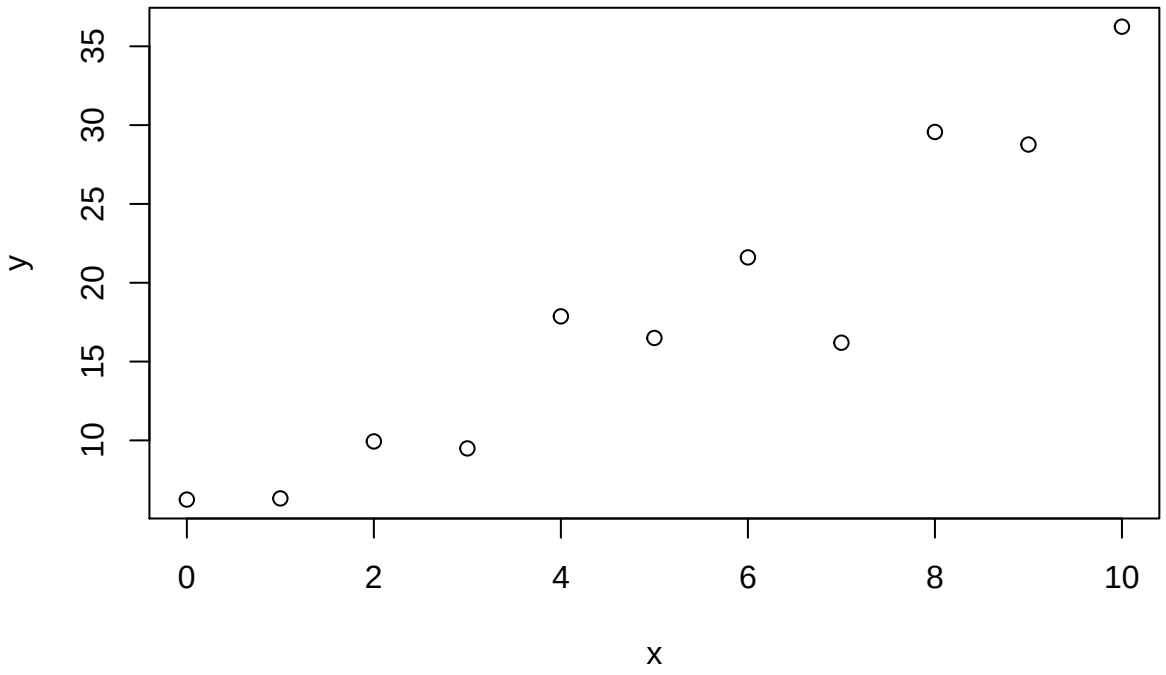
5.2.2 Scatterplots

Scatterplots are a way to explore the relationship between two continuous variables.

```
x <- seq(0,10,by=1)
y <- 2 + 3*x + rnorm(10, sd=4)
```

```
## Warning in 2 + 3 * x + rnorm(10, sd = 4): longer object length is not a
## multiple of shorter object length
```

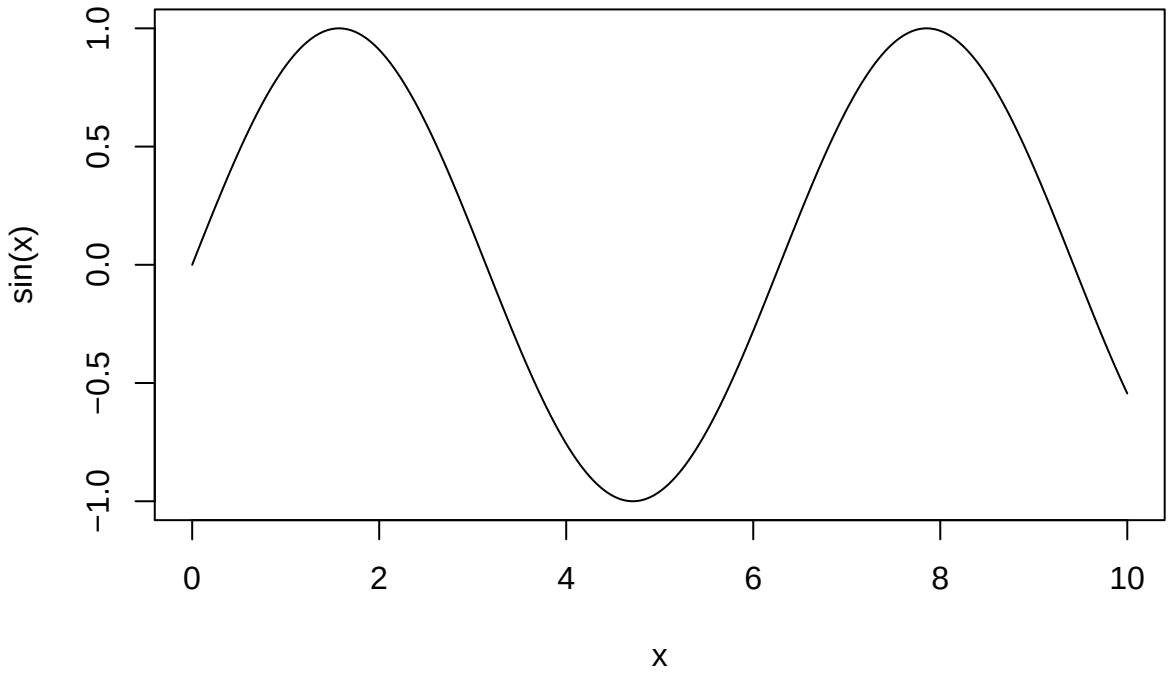
```
plot(x, y)
```



5.2.3 Arbitrary Curves

Often I want to plot a curve. To do that, I'll make a scatterplot, but tell R that I don't want points, but rather I want it to connect the dots into a curve. To do this I will change the type argument to plot and tell it to make a line.

```
x <- seq(0,10,by=.01)
plot(x, sin(x), type='l')
```



5.3 Advanced tricks

R has a very powerful graphing system that is highly customizable. When a standard plot isn't sufficient for my needs I will either start with a blank plot or just a piece of a plot and then add elements as necessary.

All of the graphical parameters that can be modified are available in `par()` and the help page for par is useful for finding out what things you can modify.

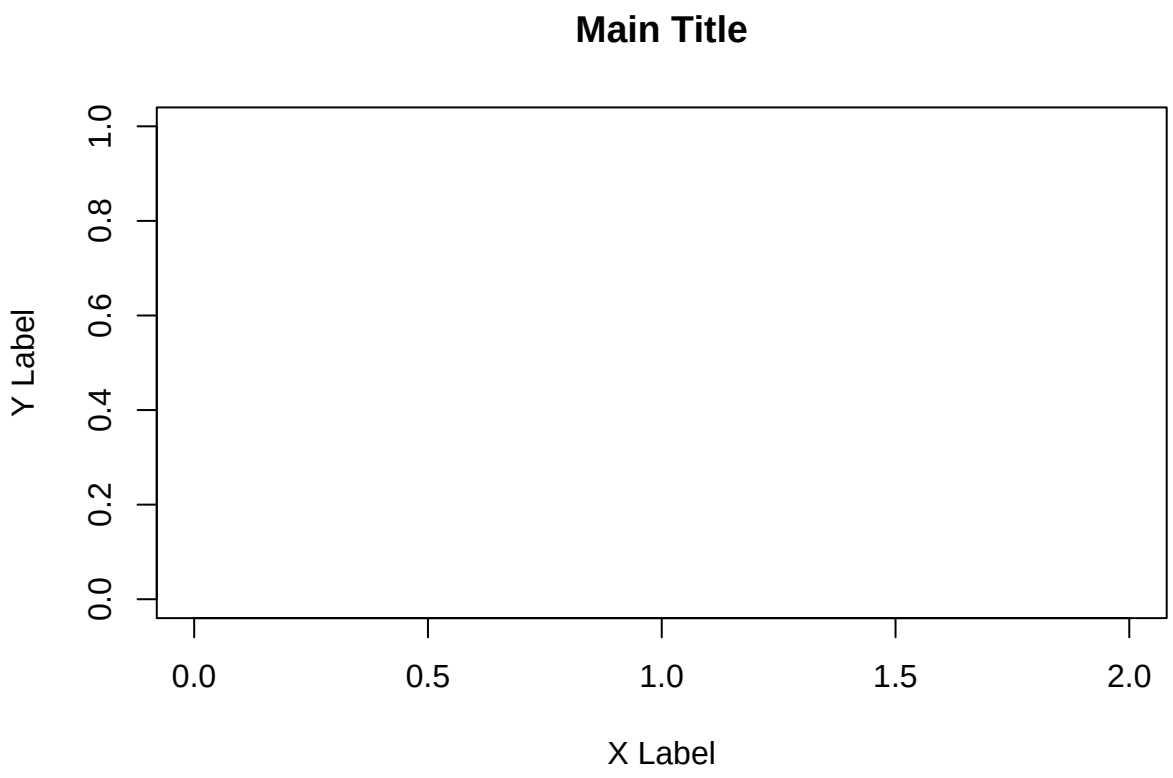
5.3.1 Blank plots

The plot function has an argument type that defines the behavior of the scatter plot. The default is to plot points, but another option we have used is to connect the points by lines.

type	Result
p	Points
l	Line (connect the dots)
b	Both points and lines
n	None (set up the plot box, but don't graph anything)

Another useful option to plot is the arguments xlim and ylim which take vectors that define the extent of the two axes.

```
plot(NA, NA, xlim=c(0,2), ylim=c(0,1), type='n',
    xlab='X Label', ylab='Y Label', main='Main Title')
```

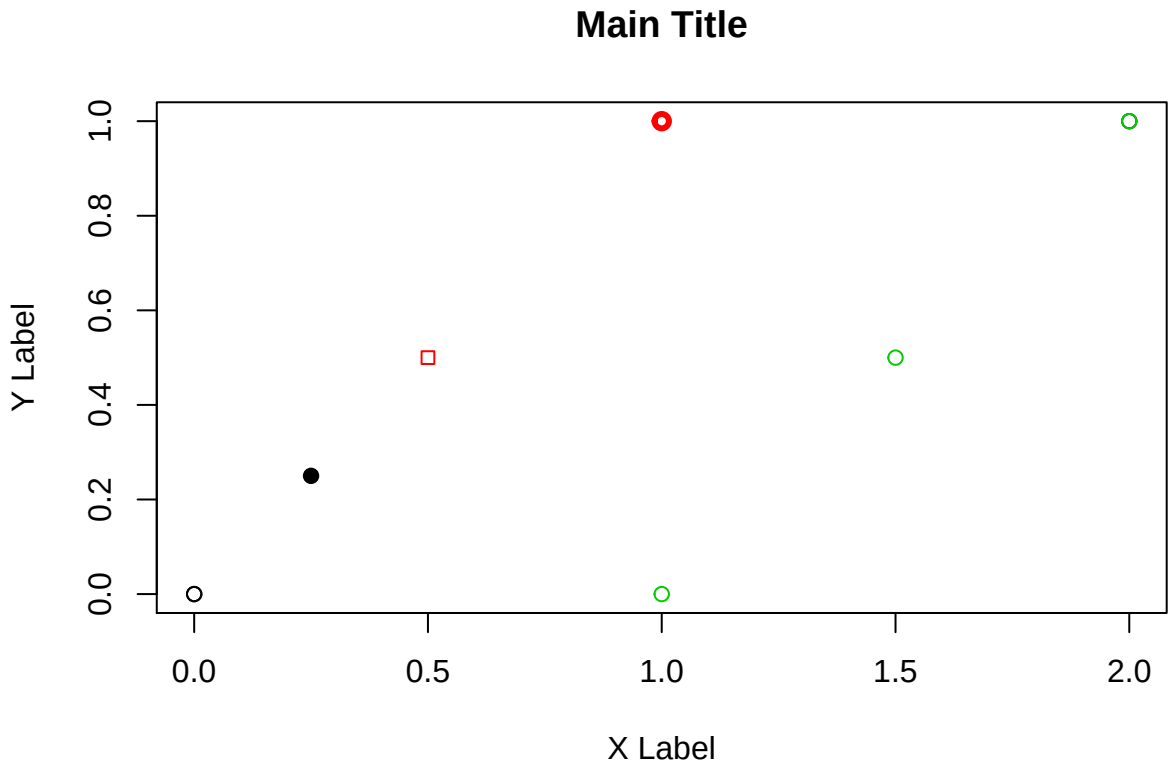


5.3.2 Points

Adding points to a graph is done by the function `points(x, y)` which takes arguments x and y which are numerical vectors of equal length. Arguments that affect the display of points are given below.

- `pch` - point character. This can either be a single character or an integer code for one of a set of graphics symbols. Unfortunately the integer codes start at 19. For a description of which code maps to what character, see the help documention `?points`.
- `col` - Color of point. This can be a character string or an integer code. The default is black which is also #1. #2 is red and #3 is green.
- `lwd` - Line width for drawing the symbols

```
plot(x=c(0,2),y=c(0,1), #type='n',
    xlab='X Label', ylab='Y Label', main='Main Title')
points( 0, 0 )             # open circle
points(.25, .25, pch=19)  # closed black dot
points(.50, .50, pch=22, col='red')  # open red square
points(1.0, 1.0, col=2, lwd=3)       # thick open red circle
points(c(1,1.5,2), c(0,.5,1), col=3) # green circles
```



For more details about the different point types, refer to the help page for points().

5.3.3 Lines

Adding lines to a graph often done by the function `abline(intercept, slope)` which defines a line. Alternately it supports arguments to draw a horizontal (`h=`) or vertical line (`v=`).

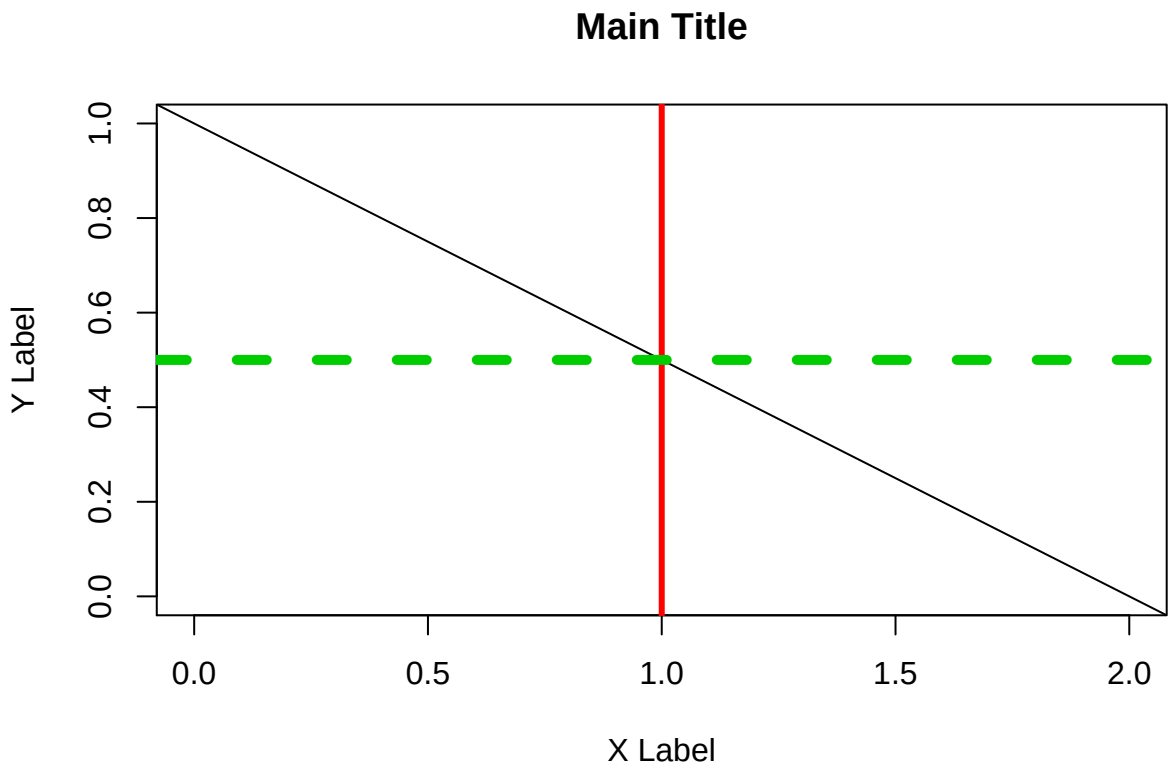
Arguments that affect the line characteristics are

- `col` - Color of point. This can be a character string or an integer code. 1-4 correspond to black, red, green, and blue.

- `lwd` - Line width, defaults to 1.

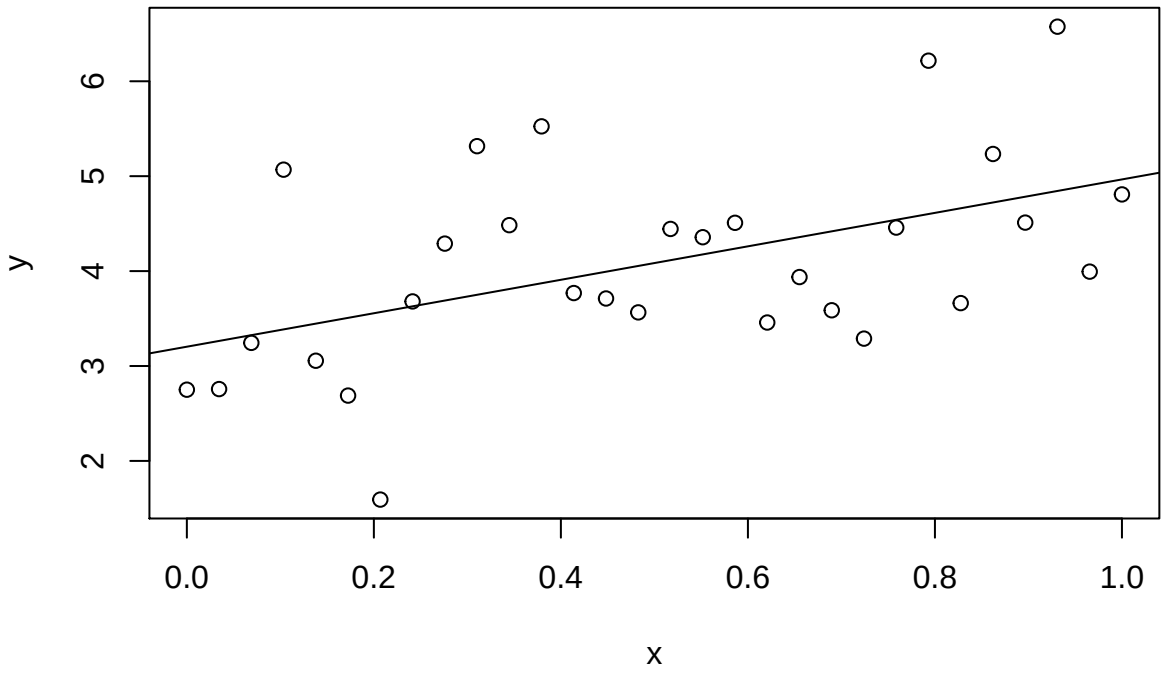
- `lty` - Line type. 1 is a solid line, 2 is dashed, 3 is dotted.

```
plot(NA, NA, xlim=c(0,2), ylim=c(0,1), type='n',        # Set up a blank plot...
    xlab='X Label', ylab='Y Label', main='Main Title')
abline(1, -1/2)                    # Black line y-intercept=1, slope=-1/2
abline(v=1, col=2, lwd=3)          # Vertical solid red line
abline(h=.5, col=3, lty=2, lwd=5) # Horizonal green dashed line
```



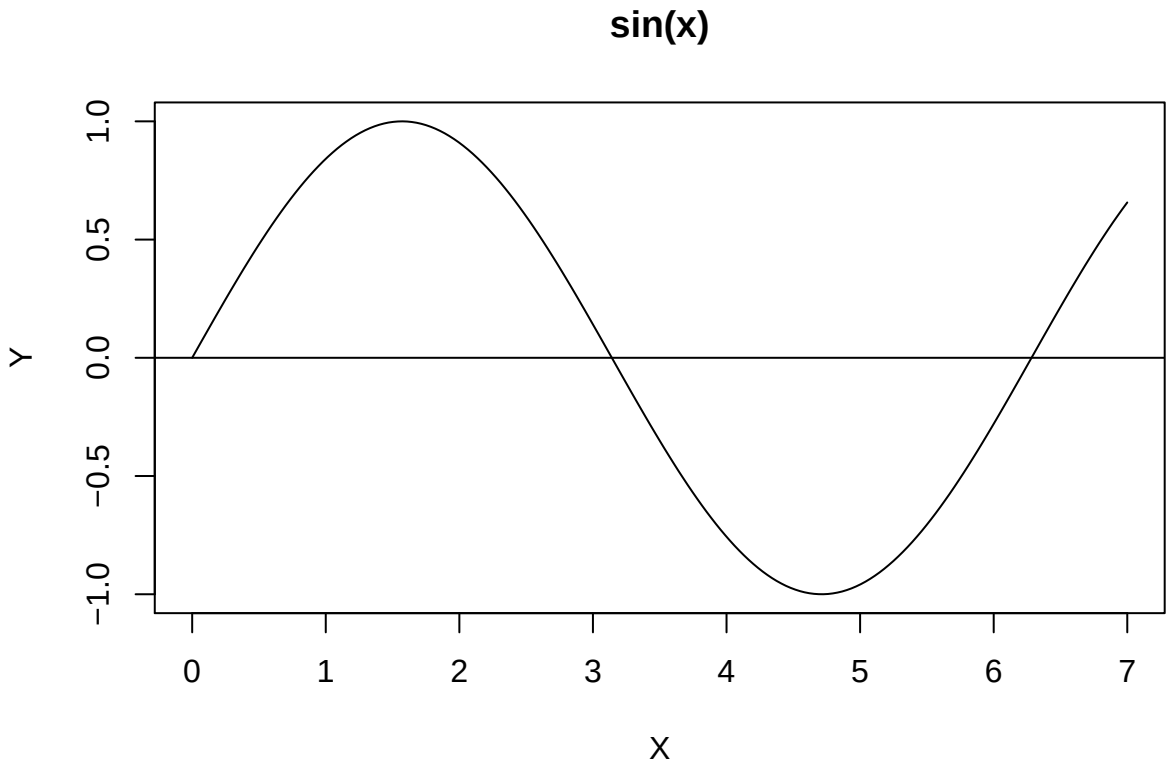
The way that `abline()` is defined, it is very convenient to add the least-squares regression line to a plot of data points.

```
x <- seq(0,1, length.out=30)
y <- 3 + 2*x + rnorm(30)
plot(x,y)
abline( coef(lm(y~x)) )     # add the regression line to the plot
```



The second way to add a line to a plot is using the `lines()` function which allows you to draw an arbitrary curve. It requires vectors of `x` and `y` values.

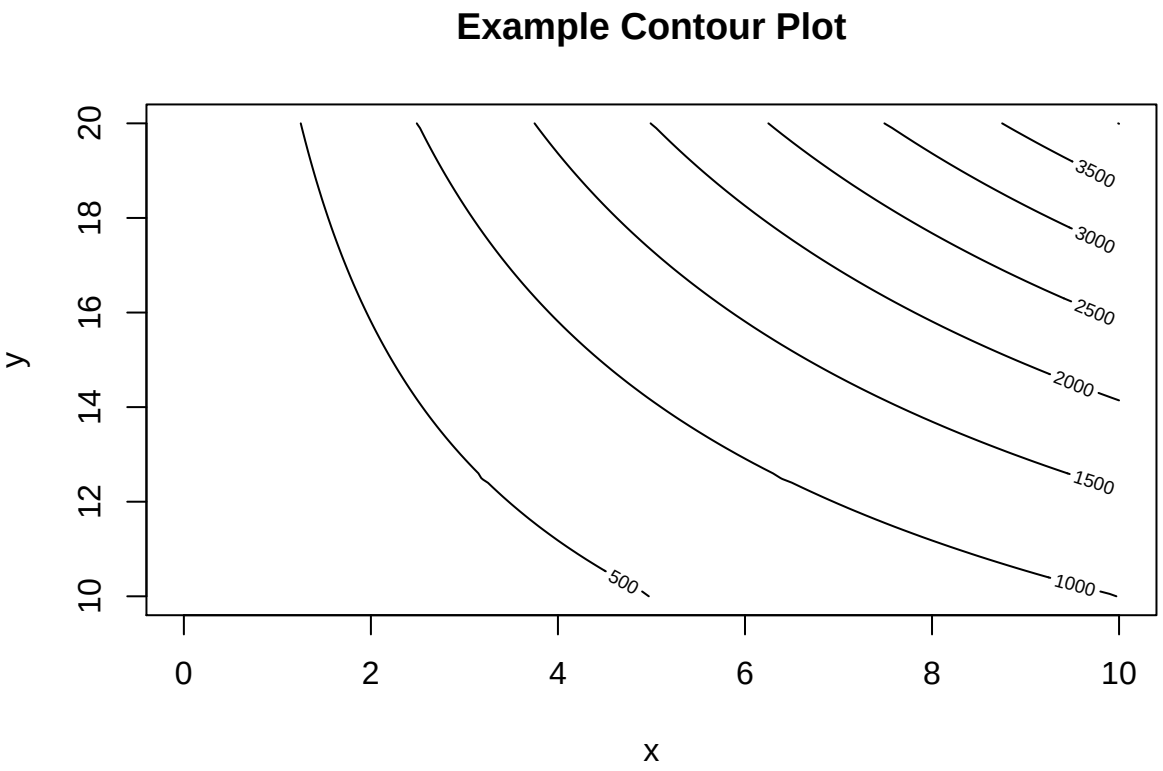
```
plot(NA, NA, xlim=c(0,7), ylim=c(-1,1), type='n',    # again a blank plot
    xlab='X', ylab='Y', main='sin(x)')
x <- seq(0,7, length=201)                            # 201 values from 0 to 7
lines(x, sin(x))                                     # sin(x)
abline(h=0)                                          # Horizontal line at 0
```



5.3.4 Contour Plots

R can make more advanced plots. One common three dimensional plot is a contour plot, basically a topographical map. To produce this map, we need vectors of `x` and `y` coordinates, and a matrix of `z` values (the elevations in a topo map).

```
x <- seq( 0, 10, length=101)
y <- seq(10, 20, length=101)
z <- matrix(NA, ncol=101, nrow=101)
for(i in 1:101){
    for(j in 1:101){
        z[i,j] <- x[i] * y[j]^2
    }
}
contour(x,y,z, xlab='x', ylab='y', main='Example Contour Plot')
```



5.4 Exercises

For this set of exercises, we'll use a dataset that contains information about the birthweight of infants and how it relates to the mother's characteristics. First we'll load the package the data lives in and update two factor variables to have more intelligent labels.

```
library(MASS)
library(dplyr)
birthwt <- birthwt %>% mutate(
  race  = factor(race,  labels=c('White','Black','Other')),
  smoke = factor(smoke, labels=c('No Smoke', 'Smoke')))
```

1. Create a histogram of the birthweights (The column is `bwt`).

2. Make a set of boxplots of `bwt` relative to the mothers `race`.
3. Modify the boxplot command so that the we include both `race` and `smoke` by using the formula `bwt~race+smoke`.
4. Create a scatterplot of the baby's `bwt` relative to the mother's weight `lwt`.
   a. First create the simple scatterplot using the `plot` command. You could using either of the following commands to do this.

```
plot( birthwt$lwt, birthwt$bwt )
plot( bwt ~ lwt, data=birthwt)
```

   b. Add the mothers race by adding `col=birthwt$race` or `col=race` argument to the plot command.
   c. Similarly add the mothers smoking status to the plot by using the character type argument `pch=`. Unfortunately, we cannot just say `pch=smoke` because the factor has two levels (coded as 1 and 2), but we need to use point types that are 19 or above. So I want these to be point type 21 (a filled circle) and point type 22 (a filled square). So annoyingly I need to add 20 to the integer value of the `smoke` variable.

```
plot( bwt ~ lwt, data=birthwt,
      col=race,
      pch=as.integer(smoke)+20)
```

Chapter 6

Matrices, Data Frames, and Lists

6.1 Matrices

We often want to store numerical data in a square or rectangular format and mathematicians will call these "matrices". These will have two dimensions, rows and columns. To create a matrix in R we can create it directly using the `matrix()` command which requires the data to fill the matrix with, and optionally, some information about the number of rows and columns:

```
W <- matrix( c(1,2,3,4,5,6), nrow=2, ncol=3 )
W
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Notice that because we only gave it six values, the information the number of columns is redundant and could be left off and R would figure out how many columns are needed. Next notice that the order that R chose to fill in the matrix was to fill in the first column then the second, and then the third. If we wanted to fill the matrix in order of the rows first, then we'd use the optional `byrow=TRUE` argument.

```
W <- matrix( c(1,2,3,4,5,6), nrow=2, byrow=TRUE )
W
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

The alternative to the `matrix()` command is we could create two columns as individual vectors and just push them together. Or we could have made three rows and lump them by rows instead. To do this we'll use a group of functions that bind vectors together. To join two column vectors together, we'll use `cbind` and to bind rows together we'll use the `rbind` function

```
a <- c(1,2,3)
b <- c(4,5,6)
cbind(a,b)  # Column Bind:  a,b are columns in resultant matrix
```

```
##      a b
## [1,] 1 4
## [2,] 2 5
## [3,] 3 6
```

```
rbind(a,b)  # Row Bind:     a,b are   rows  in resultant matrix
```

```
##   [,1] [,2] [,3]
## a    1    2    3
## b    4    5    6
```

Notice that doing this has provided R with some names for the individual rows and columns. I can change these using the commands `colnames()` and `rownames()`.

```
M <- matrix(1:6, nrow=3, ncol=2, byrow=TRUE)
colnames(M) <- c('Column1', 'Column2')       # set column labels
rownames(M) <- c('Row1', 'Row2','Row3')      # set row labels
M
```

```
##      Column1 Column2
## Row1       1       2
## Row2       3       4
## Row3       5       6
```

This is actually a pretty peculiar way of setting the *attributes* of the object `M` because it looks like we are evaluating a function and assigning some value to the function output. Yes it is weird, but R was developed in the 70s and it seemed like a good idea at the time.

Accessing a particular element of a matrix is done in a similar manner as with vectors, using the `[ ]` notation, but this time we must specify which row and which column. Notice that this scheme always is `[row, col]`.

```
M1 <- matrix(1:6, nrow=3, ncol=2)
M1
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
M1[1,2]     # Grab row 1, column 2 value
```

```
## [1] 4
```

```
M1[1, 1:2]  # Grab row 1, and columns 1 and 2.
```

```
## [1] 1 4
```

I might want to grab a single row or a single column out of a matrix, which is sometimes referred to as taking a slice of the matrix. I could figure out how long that vector is, but often I'm too lazy. Instead I can just specify the specify the particular row or column I want.

```
M1
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
M1[1, ]    # grab the 1st row
```

```
## [1] 1 4
```

```
M1[ ,2]    # grab second column (the spaces are optional...)
```

```
## [1] 4 5 6
```

6.2 Data Frames

Matrices are great for mathematical operations, but I also want to be able to store data that is numerical. For example I might want to store a categorical variable such as manufacturer brand. To generalize our concept of a matrix to include these types of data, we will create a structure called a `data.frame`. These are very much like a simple Excel spreadsheet where each column represents a different trait or measurement type and each row will represent an individual.

Perhaps the easiest way to create a data frame is to just type the columns of data

```
data <- data.frame(
  Name  = c('Bob','Jeff','Mary'),
  Score = c(90, 75, 92)
)
# Show the data.frame
data
```

```
##   Name Score
## 1  Bob    90
## 2 Jeff    75
## 3 Mary    92
```

Because a data frame feels like a matrix, R also allows matrix notation for accessing particular values.

Format	Result
`[a,b]`	Element in row `a` and column `b`
`[a,]`	All of row `a`
`[,b]`	All of column `b`

Because the columns have meaning and we have given them column names, it is desirable to want to access an element by the name of the column as opposed to the column number.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 be done with it is much nicer to work with and I make fewer dumb mistakes.

```
data$Name       # The $-sign means to reference a column by its label
```

```
## [1] Bob  Jeff Mary
## Levels: Bob Jeff Mary
```

```
data$Name[2]    # Notice that data$Name results in a vector, which I can manipulate
```

```
## [1] Jeff
## Levels: Bob Jeff Mary
```

I can mix the `[ ]` notation with the column names. The following is also acceptable:

```
data[, 'Name']   # Grab the column labeled 'Name'
```

```
## [1] Bob  Jeff Mary
## Levels: Bob Jeff Mary
```

The next thing we might wish to do is add a new column to a preexisting data frame. There are two ways to do this. First, we could use the `cbind()` function to bind two data frames together. Second we could reference a new column name and assign values to it.

```
Second.score <- data.frame(Score2=c(41,42,43))  # another data.frame
data <-  cbind( data, Second.score )            # squish them together
data
```

```
##   Name Score Score2
## 1  Bob    90     41
## 2 Jeff    75     42
## 3 Mary    92     43
```

```
# if you assign a value to a column that doesn't exist, R will create it
data$Score3 <- c(61,62,63) # the Score3 column will created
data
```

```
##   Name Score Score2 Score3
## 1  Bob    90     41     61
## 2 Jeff    75     42     62
## 3 Mary    92     43     63
```

Data frames are very commonly used and many commonly used functions will take a `data=` argument and all other arguments are assumed to be in the given data frame. Unfortunately this is not universally supported by all functions and you must look at the help file for the function you are interested in.

Data frames are also very restricive in that the shape of the data must be rectangular. If I try to create a new column that doesn't have enough rows, R will complain.

```
data$Score4 <- c(1,2)
```

```
## Error in `$<-.data.frame`(`*tmp*`, "Score4", value = c(1, 2)): replacement has 2 rows, data has 3
```

6.3 Lists

Data frames are quite useful for storing data but sometimes we'll need to store a bunch of different pieces of information and it won't fit neatly as a data frame. The most general form of a data structure is called a list. This can be thought of as a vector of objects where there is no requirement for each element to be the same type of object.

Consider that I might need to store information about person. For example, suppose that I want make an object that holds information about my immediate family. This object should have my spouse's name (just one name) as well as my siblings. But because I have many siblings, I want the siblings to be a vector of names. Likewise I might also include my pets, but we don't want any requirement that the number of pets is the same as the number of siblings (or spouses!).

```
wife <- 'Aubrey'
sibs <- c('Tina','Caroline','Brandon','John')
pets <- c('Beau','Tess','Kaylee')
Derek <- list(Spouse=wife, Siblings=sibs, Pets=pets) # Create the list
str(Derek) # show the structure of object
```

```
## List of 3
##  $ Spouse  : chr "Aubrey"
##  $ Siblings: chr [1:4] "Tina" "Caroline" "Brandon" "John"
##  $ Pets    : chr [1:3] "Beau" "Tess" "Kaylee"
```

Notice that the object `Derek` is a list of three elements. The first is the single string containing my wife's name. The next is a vector of my siblings' names and it is a vector of length four. Finally the vector of pets' names is only of length three.

To access any element of this list we can use an indexing scheme similar to matrices and vectors. The only difference is that we'll use two square brackets instead of one.

```
Derek[[ 1 ]]    # First element of the list is Spouse!
```

```
## [1] "Aubrey"
```

```
Derek[[ 3 ]]    # Third element of the list is the vector of pets
```

```
## [1] "Beau"   "Tess"   "Kaylee"
```

There is a second way I can access elements. For data frames it was convenient to use the notation `DataFrame$ColumnName` and we will use the same convention for lists. Actually a data frame is just a list with the requirement that each list element is a vector and all vectors are of the same length. To access my pets names we can use the following notation:

```
Derek$Pets         # Using the '$' notation
```

```
## [1] "Beau"   "Tess"   "Kaylee"
```

```
Derek[[ 'Pets' ]]  # Using the '[[ ]]' notation
```

```
## [1] "Beau"   "Tess"   "Kaylee"
```

To add something new to the list object, we can just make an assignment in a similar fashion as we did for `data.frame` and just assign a value to a slot that doesn't (yet!) exist.

```
Derek$Spawn <- c('Elise', 'Casey')
```

We can also add extremely complicated items to my list. Here we'll add a `data.frame` as another list element.

```
# Recall that we previous had defined a data.frame called "data"
Derek$RandomDataFrame <- data  # Assign it to be a list element
str(Derek)
```

```
## List of 5
##  $ Spouse         : chr "Aubrey"
##  $ Siblings       : chr [1:4] "Tina" "Caroline" "Brandon" "John"
##  $ Pets           : chr [1:3] "Beau" "Tess" "Kaylee"
##  $ Spawn          : chr [1:2] "Elise" "Casey"
##  $ RandomDataFrame:'data.frame': 3 obs. of  4 variables:
##   ..$ Name  : Factor w/ 3 levels "Bob","Jeff","Mary": 1 2 3
##   ..$ Score : num [1:3] 90 75 92
##   ..$ Score2: num [1:3] 41 42 43
##   ..$ Score3: num [1:3] 61 62 63
```

Now we see that the list `Derek` has five elements and some of those elements are pretty complicated. In fact, I could happily have lists of lists and have a very complicated nesting structure.

The place that most users will run into lists is that the output of many statistical procedures will return the results in a list object. When a user asks R to perform a regression, the output returned is a list object, and we'll need to grab particular information from that object afterwards. For example, the output from a t-test in R is a list:

```
x <- c(5.1, 4.9, 5.6, 4.2, 4.8, 4.5, 5.3, 5.2)   # some toy data
results <- t.test(x, alternative='less', mu=5)   # do a t-test
str(results)                                     # examine the resulting object
```

```
## List of 9
##  $ statistic  : Named num -0.314
##   ..- attr(*, "names")= chr "t"
##  $ parameter  : Named num 7
##   ..- attr(*, "names")= chr "df"
##  $ p.value    : num 0.381
##  $ conf.int   : atomic [1:2] -Inf 5.25
##   ..- attr(*, "conf.level")= num 0.95
##  $ estimate   : Named num 4.95
```

```
##   ..- attr(*, "names")= chr "mean of x"
##  $ null.value : Named num 5
##   ..- attr(*, "names")= chr "mean"
##  $ alternative: chr "less"
##  $ method     : chr "One Sample t-test"
##  $ data.name  : chr "c(5.1, 4.9, 5.6, 4.2, 4.8, 4.5, 5.3, 5.2)"
##  - attr(*, "class")= chr "htest"
```

We see that result is actually a list with 9 elements in it. To access the p-value we could use:

```
results$p.value
```

```
## [1] 0.3813385
```

If I ask R to print the object `results`, it will hide the structure from you and print it in a "pretty" fashion because there is a `print` function defined specifically for objects created by the `t.test()` function.

```
results
```

```
##
##  One Sample t-test
##
## data:  c(5.1, 4.9, 5.6, 4.2, 4.8, 4.5, 5.3, 5.2)
## t = -0.31399, df = 7, p-value = 0.3813
## alternative hypothesis: true mean is less than 5
## 95 percent confidence interval:
##      -Inf 5.251691
## sample estimates:
## mean of x
##      4.95
```

6.4 Exercises

1. In this problem, we will work with the matrix

$$\begin{bmatrix} 2 & 4 & 6 & 8 & 10 \\ 12 & 14 & 16 & 18 & 20 \\ 22 & 24 & 26 & 28 & 30 \end{bmatrix}$$

   a) Create the matrix in two ways and save the resulting matrix as `M`.
      i. Create the matrix using some combination of the `seq()` and `matrix()` commands.
      ii. Create the same matrix by some combination of multiple `seq()` commands and either the `rbind()` or `cbind()` command.
   b) Extract the second row out of `M`.
   c) Extract the element in the third row and second column of `M`.

2. Create and manipulate a data frame.

   a) Create a `data.frame` named `my.trees` that has the following columns:
      - Girth = c(8.3, 8.6, 8.8, 10.5, 10.7, 10.8, 11.0)
      - Height= c(70, 65, 63, 72, 81, 83, 66)
      - Volume= c(10.3, 10.3, 10.2, 16.4, 18.8, 19.7, 15.6)
   b) Extract the third observation (i.e. the third row)
   c) Extract the Girth column referring to it by name (don't use whatever order you placed the columns in).
   d) Print out a data frame of all the observations *except* for the fourth observation. (i.e. Remove the fourth observation/row.)

3. Create and manipulate a list.
   a) Create a list named my.test with elements
      - x = c(4,5,6,7,8,9,10)
      - y = c(34,35,41,40,45,47,51)
      - slope = 2.82
      - p.value = 0.000131
   b) Extract the second element in the list.
   c) Extract the element named `p.value` from the list.
4. The function `lm()` creates a linear model, which is a general class of model that includes both regression and ANOVA. We will call this on a data frame and examine the results. For this problem, there isn't much to figure out, but rather the goal is to recognize the data structures being used in common analysis functions.
   a) There are many data sets that are included with R and its packages. One of which is the `trees` data which is a data set of $n = 31$ cherry trees. Load this dataset into your current workspace using the command:

```
data(trees)     # load trees data.frame
```

   b) Examine the data frame using the `str()` command. Look at the help file for the data using the command `help(trees)` or `?trees`.
   c) Perform a regression relating the volume of lumber produced to the girth and height of the tree using the following command

```
m <- lm( Volume ~ Girth + Height, data=trees)
```

   d) Use the str() command to inspect `m`. Extract the model coefficients from this list.
   e) The list m can be passed to other functions. For example, the function `summary()` will take the list and recognize that it was produced by the `lm()` function and produce a summary table in the manner that we are used to seeing. Produce that summary table using the command

```
summary(m)
```

```
##
## Call:
## lm(formula = Volume ~ Girth + Height, data = trees)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -6.4065 -2.6493 -0.2876  2.2003  8.4847
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -57.9877     8.6382  -6.713 2.75e-07 ***
## Girth         4.7082     0.2643  17.816  < 2e-16 ***
## Height        0.3393     0.1302   2.607   0.0145 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.882 on 28 degrees of freedom
## Multiple R-squared:  0.948,  Adjusted R-squared:  0.9442
## F-statistic:   255 on 2 and 28 DF,  p-value: < 2.2e-16
```

Chapter 7

Importing Data

Reading data from external sources is necessary. It is most common for data to be in a data-frame like storage, such as a MS Excel workbook, so we will concentrate on reading data into a data.frame.

In the typical way data is organized, we think of each column of data representing some trait or variable that we might be interested in. In general, we might wish to investigate the relationship between variables. In contrast, the rows of our data represent a single object on which the column traits are measured. For example, in a grade book for recording students scores throughout the semester, their is one row for every student and columns for each assignment. A greenhouse experiment dataset will have a row for every plant and columns for treatment type and biomass.

One concept that will be important is to recognize that every time you start up RStudio, it picks an appropriate working directory. This is the directory where it will first look for script files or data files. So often I will set the working directory to be the same directory as the Rmarkdown file that I'm working on (because I typically will have a couple of MS Excel files, an Rmarkdown file that does an analysis, and the final output file all stored in the same location). If you click on an Rmarkdown file to open RStudio, then the working directory will automatically be set appropriately. To set the working directory explicitly, you can use the GUI tools `Session -> Set Working Directory...`.

When you knit your Rmarkdown file, it always starts up a new version of R with the working directory set to the location of the Rmarkdown file. So whenever you specify a file location, it should be relative to the location of the Rmarkdown file you are working on!

7.1 Comma Separated Data

To consider how data might be stored, we first consider the simplest file format… the comma separated values file. In this file time, each of the "cells" of data are separated by a comma. For example, the data file storing scores for three students might be as follows:

```
Able, Dave, 98, 92, 94
Bowles, Jason, 85, 89, 91
Carr, Jasmine, 81, 96, 97
```

Typically when you open up such a file on a computer with Microsoft Excel installed, Excel will open up the file assuming it is a spreadsheet and put each element in its own cell. However, you can also open the file using a more primitive program (say Notepad in Windows, TextEdit on a Mac) you'll see the raw form of the data.

Having just the raw data without any sort of column header is problematic (which of the three exams was the final??). Ideally we would have column headers that store the name of the column.

```
LastName, FirstName, Exam1, Exam2, FinalExam
Able, Dave, 98, 92, 94
Bowles, Jason, 85, 89, 91
Carr, Jasmine, 81, 96, 97
```

To see another example, open the "Body Fat" dataset from the Lock[5] introductory text book at the website [http://www.lock5stat.com/datasets/BodyFat.csv]. The first few rows of the file are as follows:

```
Bodyfat,Age,Weight,Height,Neck,Chest,Abdomen,Ankle,Biceps,Wrist
32.3,41,247.25,73.5,42.1,117,115.6,26.3,37.3,19.7
22.5,31,177.25,71.5,36.2,101.1,92.4,24.6,30.1,18.2
22,42,156.25,69,35.5,97.8,86,24,31.2,17.4
12.3,23,154.25,67.75,36.2,93.1,85.2,21.9,32,17.1
20.5,46,177,70,37.2,99.7,95.6,22.5,29.1,17.7
```

To make R read in the data arranged in this format, we need to tell R three things:

1. Where does the data live? Often this will be the name of a file on your computer, but the file could just as easily live on the internet (provided your computer has internet access).
2. Is the first row data or is it the column names?
3. What character separates the data? Some programs store data using tabs to distinguish between elements, some others use white space. R's mechanism for reading in data is flexible enough to allow you to specify what the separator is.

The primary function that we'll use to read data from a file and into R is the function `read.table()`. This function has many optional arguments but the most commonly used ones are outlined in the table below.

Argument	Default	What it does
`file`		A character string denoting the file location
`header`	FALSE	Is the first line column headers?
`sep`	" "	What character separates columns. " " == any whitespace
`skip`	0	The number of lines to skip before reading data. This is useful when there are lines of text that describe the data or aren't actual data
`na.strings`	'NA'	What values represent missing data. Can have multiple. E.g. `c('NA', -9999)`
`quote`	" and '	For character strings, what characters represent quotes.

To read in the "Body Fat" dataset we could run the R command:

```
BodyFat <- read.table(
  file   = 'http://www.lock5stat.com/datasets/BodyFat.csv',  # where the data lives
  header = TRUE,                                             # first line is column names
  sep    = ',' )                                             # Data is sparated by commas

str(BodyFat)
```

```
## 'data.frame':    100 obs. of  10 variables:
##  $ Bodyfat: num  32.3 22.5 22 12.3 20.5 22.6 28.7 21.3 29.9 21.3 ...
##  $ Age    : int  41 31 42 23 46 54 43 42 37 41 ...
##  $ Weight : num  247 177 156 154 177 ...
##  $ Height : num  73.5 71.5 69 67.8 70 ...
##  $ Neck   : num  42.1 36.2 35.5 36.2 37.2 39.9 37.9 35.3 42.1 39.8 ...
##  $ Chest  : num  117 101.1 97.8 93.1 99.7 ...
##  $ Abdomen: num  115.6 92.4 86 85.2 95.6 ...
##  $ Ankle  : num  26.3 24.6 24 21.9 22.5 22 23.7 21.9 24.8 25.2 ...
```

```
##  $ Biceps : num  37.3 30.1 31.2 32 29.1 35.9 32.1 30.7 34.4 37.5 ...
##  $ Wrist  : num  19.7 18.2 17.4 17.1 17.7 18.9 18.7 17.4 18.4 18.7 ...
```

Looking at the help file for `read.table()` we see that there are variants such as `read.csv()` that sets the default arguments to header and sep more intelligently. Also, there are many options to customize how R responds to different input.

7.2 MS Excel

Commonly our data is stored as a MS Excel file. There are two approaches you could use to import the data into R.

1. From within Excel, export the worksheet that contains your data as a comma separated values (.csv) file and proceed using the tools in the previous section.
2. Use functions within R that automatically convert the worksheet into a .csv file and read it in. One package that works nicely for this is the `readxl` package.

I generally prefer using option 2 because all of my collaborators can't live without Excel and I've resigned myself to this. However if you have complicated formulas in your Excel file, it is often times safer to export it as a .csv file to guarantee the data imported into R is correct. Furthermore, other spreadsheet applications (such as Google sheets) requires you to export the data as a .csv file so it is good to know both paths.

Because R can only import a complete worksheet, the desired data worksheet must be free of notes to yourself about how the data was collected, preliminary graphics, or other stuff that isn't the data. I find it very helpful to have a worksheet in which I describe the sampling procedure and describe what each column means (and give the units!), then a second worksheet where the actual data is, and finally a third worksheet where my "Excel Only" collaborators have created whatever plots and summary statistics they need.

The simplest package for importing Excel files seems to be the package `readxl`. Another package that does this is the XLConnect which does the Excel -> .csv conversion using Java. Another package the works well is the `xlsx` package, but it also requires Java to be installed. The nice thing about these two packages is that they also allow you to write Excel files as well. The RODBC package allows R to connect to various databases and it is possible to make it consider an Excel file as an extremely crude database.

The `readxl` package provides a function `read_exel()` that allows us to specify which sheet within the Excel file to read and what character specifies missing data (it assumes a blank cell is missing data if you don't specifying anything).

From GitHub, download the files `Example_1.xls`, `Example_2.xls`, `Example_3.xls` and `Example_4.xls` from the directory [https://github.com/dereksonderegger/STA_570L_Book/tree/master/data-raw]. Place these files in the same directory that you store your course work or make a subdirectory data to store the files in. Make sure that the working directory that RStudio is using is that same directory (Session -> Set Working Directory).

```
# load the library that has the read.xls function.
library(readxl)

# Where does the data live relative to my current working location?
#
# I made a subdirectory named 'data-raw' to store all the data files,
#   so the path to my data files will be './data-raw/Example_1.xls'
#   where "." indicates the current working directory
# If you stored the files in the same directory as your RMarkdown script,
#   then you would use './Example_1.xls' or just 'Example_1.xls'

# read the first worksheet of the Example_1 file
data.1 <- read_excel(  './data-raw/Example_1.xls'  )
```

```
# read the second worksheet where the second worksheet is named 'data'
data.2 <- read_excel('./data-raw/Example_2.xls', sheet=2     )
data.2 <- read_excel('./data-raw/Example_2.xls', sheet='data')
```

There is one additional problem that shows up while reading in Excel files. Blank columns often show up in Excel files because at some point there was some text in a cell that got deleted but a space remains and Excel still thinks there is data in the column. To fix this, you could find the cell with the space in it, or you can select a bunch of columns at the edge and delete the entire columns. Alternatively, you could remove the column after it is read into R.

Open up the file `Example_4.xls` in Excel and confirm that the data sheet has name columns out to carb. Read in the data frame using the following code:

```
data.4 <- read_excel('./data-raw/Example_4.xls', sheet='data')   # Extra Column Example
str(data.4)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    31 obs. of  13 variables:
##  $ model: chr  "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive" ...
##  $ mpg  : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl  : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp : num  160 160 108 258 360 ...
##  $ hp   : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat : num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt   : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec : num  16.5 17 18.6 19.4 17 ...
##  $ vs   : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am   : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear : num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb : num  4 4 1 1 2 1 4 2 2 4 ...
##  $ NA   : chr  NA NA " " NA ...
```

We notice that after reading in the data, there are an additional two columns (labeled `X` and `X.1` because R had to make up some column name so it chose that) that just has missing data (the NA stands for not available which means that the data is missing). Go back to the Excel file and go to row 4 column N and notice that the cell isn't actually blank, there is a space. Delete the space, save the file, and then reload the data into R. You should notice that the extra columns are now gone.

7.3 Exercises

1. Download from GitHub the data file Example_5.xls. Open it in Excel and figure out which sheet of data we should import into R. At the same time figure out how many initial rows need to be skipped. Import the data set into a data frame and show the structure of the imported data using the str() command. Make sure that your data has $n = 31$ observations and the three columns are appropriately named.

Chapter 8

Data Manipulation

Most of the time, our data is in the form of a data frame and we are interested in exploring the relationships. This chapter explores how to manipulate data frames and methods.

8.1 Classical functions for summarizing rows and columns

8.1.1 `summary()`

The first method is to calculate some basic summary statistics (minimum, 25th, 50th, 75th percentiles, maximum and mean) of each column. If a column is categorical, the summary function will return the number of observations in each category.

```
# use the iris data set which has both numerical and categorical variables
data( iris )
str(iris)    # recall what columns we have
```

```
## '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 ...
```

```
# display the summary for each column
summary( iris )
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
```

```
##
```

8.1.2 apply()

The summary function is convenient, but we want the ability to pick another function to apply to each column and possibly to each row. To demonstrate this, suppose we have data frame that contains students grades over the semester.

```
# make up some data
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) )
```

The `apply()` function will apply an arbitrary function to each row (or column) of a matrix or a data frame and then aggregate the results into a vector.

```
# Because I can't take the mean of the last names column,
# remove the name column
scores <- grades[,-1]
scores
```

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

```
# Summarize each column by calculating the mean.
apply( scores,      # what object do I want to apply the function to
       MARGIN=2,    # rows = 1, columns = 2, (same order as [rows, cols]
       FUN=mean     # what function do we want to apply
     )
```

```
## Exam1 Exam2 Final
## 83.00 83.75 88.50
```

To apply a function to the rows, we just change which margin we want. We might want to calculate the average exam score for person.

```
apply( scores,      # what object do I want to apply the function to
       MARGIN=1,    # rows = 1, columns = 2, (same order as [rows, cols]
       FUN=mean     # what function do we want to apply
     )
```

```
## [1] 95.66667 81.33333 81.00000 82.33333
```

This is useful, but it would be more useful to concatenate this as a new column in my grades data frame.

```
average <- apply(
  scores,      # what object do I want to apply the function to
  MARGIN=1,    # rows = 1, columns = 2, (same order as [rows, cols]
  FUN=mean     # what function do we want to apply
)
grades <- cbind( grades, average ) # squish together
grades
```

```
##   l.name Exam1 Exam2 Final  average
```

```
## 1    Cox    93    98    96 95.66667
## 2 Dorian    89    70    85 81.33333
## 3  Kelso    80    82    81 81.00000
## 4   Turk    70    85    92 82.33333
```

There are several variants of the `apply()` function, and the variant I use most often is the function `sapply()`, which will apply a function to each element of a list or vector and returns a corresponding list or vector of results.

8.2 Package `dplyr`

```
library(dplyr)   # load the dplyr package!
```

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()`.

Dr Wickham gave the following example of readability:

```
bopping(
  scooping_up(
    hopping_through(foo_foo),
    field_mice),
  head)
```

is more readably written:

```
foo_foo %>%
  hopping_through(forest) %>%
  scooping_up( field_mice) %>%
  bopping( head )
```

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.

8.2.1 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.

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

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

8.2.1.1.1 `select()`

Often you only want to work with a small number of columns of a data frame. It is relatively easy to do this using the standard `[,col.name]` notation, but is often pretty tedious.

```
# recall what the grades are
grades
```

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

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

```
grades %>% dplyr::select( Exam1, Exam2 )   # Exam1 and Exam2
```

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

```
grades %>% dplyr::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 %>% dplyr::select( -Exam1 )         # Negative indexing by name works
```

```
##   l.name Exam2 Final  average
## 1    Cox    98    96 95.66667
## 2 Dorian    70    85 81.33333
## 3  Kelso    82    81 81.00000
## 4   Turk    85    92 82.33333
```

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

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

8.2.1.1.2 `filter()`

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

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

```
##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2   Turk    70    85    92 82.33333
```

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`.

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

```
##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
```

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

```
##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
```

8.2.1.1.3 `slice()`

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

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

```
##   l.name Exam1 Exam2 Final  average
## 1    Cox    93    98    96 95.66667
## 2 Dorian    89    70    85 81.33333
```

8.2.1.2 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  average
## 1    Cox    93    98    96 95.66667
## 2 Dorian    89    70    85 81.33333
## 3  Kelso    80    82    81 81.00000
## 4   Turk    70    85    92 82.33333
```

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  average
## 1    Cox    93    98    96 95.66667
## 2   Turk    70    85    92 82.33333
## 3 Dorian    89    70    85 81.33333
## 4  Kelso    80    82    81 81.00000
```

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
```

8.2.1.3 mutate()

I often need to create a new column that is some function of the old columns. This was often cumbersome. Consider code to calculate the average grade in my grade book example.

```
grades$average <- (grades$Exam1 + grades$Exam2 + grades$Final) / 3
```

Instead, we could use the `mutate()` function and avoid all the `grades$` nonsense.

```
grades %>% mutate( average = (Exam1 + Exam2 + Final)/3 )
```

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

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
```

8.2.1.4 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
summarise( grades, 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(Exam2) )
```

```
##   mean.E1 stddev.E1
## 1      83      11.5
```

If we want to apply the same statistic to each column, we use the `summarise_each()` 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()`)).

```
# calculate the mean and stddev of each column - Cannot do this to Names!
grades %>%
  dplyr::select( Exam1:Final ) %>%
  summarise_each( funs(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
```

8.2.1.5 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 subsample 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.

8.2.2 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-dataframes, 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-dataframes. 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 one
# 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
```

```
## Source: local data frame [6 x 3]
## Groups: wool [?]
##
##     wool tension     n
##   <fctr>  <fctr> <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
```

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 tell the
             mean.breaks = mean(breaks),    # reader I am calculating several
             sd.breaks   = sd(breaks))      # statistics.
```

```
## Source: local data frame [6 x 5]
## Groups: wool [?]
##
##     wool tension     n mean.breaks sd.breaks
##   <fctr>  <fctr> <int>       <dbl>     <dbl>
## 1      A       L     9    44.55556 18.097729
## 2      A       M     9    24.00000  8.660254
## 3      A       H     9    24.55556 10.272671
## 4      B       L     9    28.22222  9.858724
## 5      B       M     9    28.77778  9.431036
## 6      B       H     9    18.77778  4.893306
```

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.

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

```
## Source: local data frame [6 x 4]
## Groups: wool, tension [1]
##
##   breaks   wool tension      resid
##    <dbl> <fctr>  <fctr>      <dbl>
## 1     26      A       L -18.555556
## 2     30      A       L -14.555556
## 3     54      A       L   9.444444
## 4     25      A       L -19.555556
## 5     70      A       L  25.444444
## 6     52      A       L   7.444444
```

8.2.3 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.

```
# input the initial data
race.results <- data.frame(
  name=c('Bob', 'Jeff', 'Rachel', 'Bonnie', 'Derek', 'April','Elise','David'),
  time=c(21.23, 19.51, 19.82, 23.45, 20.23, 24.22, 28.83, 15.73),
  gender=c('M','M','F','F','M','F','F','M'))
```

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

```
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
```

```
## Source: local data frame [8 x 4]
## Groups: gender [2]
##
##     name  time gender place
##   <fctr> <dbl> <fctr> <dbl>
## 1 Rachel 19.82      F     1
## 2 Bonnie 23.45      F     2
## 3  April 24.22      F     3
## 4  Elise 28.83      F     4
## 5  David 15.73      M     1
## 6   Jeff 19.51      M     2
## 7  Derek 20.23      M     3
## 8    Bob 21.23      M     4
```

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

Other (and slightly more readable) way to complete our task is to save the intermediate value of each step:

```
# how should I group?
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.

The way this is typically handled in R is to to just nest one command inside the next. The same set of commands could be run as follows:<<>>=This is extremely hard to read because the commands are separated from the arguments (e.g. the arrange function call was at the top, but the columns to arrange by are on the last line).

To get around this, the author of dplyr gives us an operator to combine these simple operations smoothly. The composition operation %>% takes the following A %>% f(B) and converts it to the statement f(A, B). This allows us to write the following code that does exactly what the above two code chunks did. <<>>=If I only wanted the top three finishers in each gender, we could simply add a filter command after the place column was calculated.<<>>=

8.3 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 the weights on day 10 and day 20.

d) Calculate the mean and standard deviation 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.

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. 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 and Age group.

Chapter 9

Data Reshaping

Most of the time, our data is in the form of a data frame and we are interested in exploring the relationships. However most procedures in R expect the data to show up in a 'long' format where each row is an observation and each column is a covariate. In practice, the data is often not stored like that and the data comes to us with repeated observations included on a single row. This is often done as a memory saving technique or because there is some structure in the data that makes the 'wide' format attractive. As a result, we need a way to convert data from 'wide' to 'long' and vice-versa.

9.1 `tidyr`

There is a common issue with obtaining data with many columns that you wish were organized as rows. For example, I might have data in a grade book that has several homework scores and I'd like to produce a nice graph that has assignment number on the x-axis and score on the y-axis. Unfortunately this is incredibly hard to do when the data is arranged in the following way:

```
grade.book <- rbind(
  data.frame(name='Alison',  HW.1=8, HW.2=5, HW.3=8),
  data.frame(name='Brandon', HW.1=5, HW.2=3, HW.3=6),
  data.frame(name='Charles', HW.1=9, HW.2=7, HW.3=9))
grade.book
```

```
##      name HW.1 HW.2 HW.3
## 1  Alison    8    5    8
## 2 Brandon    5    3    6
## 3 Charles    9    7    9
```

What we want to do is turn this data frame from a *wide* data frame into a *long* data frame. In MS Excel this is called pivoting. Essentially I'd like to create a data frame with three columns: `name`, `assignment`, and `score`. That is to say that each homework datum really has three pieces of information: who it came from, which homework it was, and what the score was. It doesn't conceptually matter if I store it as 3 columns or 3 rows so long as there is a way to identify how a student scored on a particular homework. So we want to reshape the HW1 to HW3 columns into two columns (assignment and score).

This package was built by the sample people that created dplyr and ggplot2 and there is a nice introduction at: [http://blog.rstudio.org/2014/07/22/introducing-tidyr/]

9.1.1 Verbs

As with the dplyr package, there are two main verbs to remember:

1. `gather` - Gather multiple columns that are related into two columns that contain the original column name and the value. For example for columns HW1, HW2, HW3 we would gather them into two column HomeworkNumber and Score. In this case, we refer to HomeworkNumber as the key column and Score as the value column. So for any key:value pair you know everything you need.

2. `spread` - This is the opposite of gather. This takes a key column (or columns) and a results column and forms a new column for each level of the key column(s).

```
library(tidyr)
# first we gather the score columns into columns we'll name Assesment and Score
tidy.scores <- grade.book %>%
  gather( key=Assessement,  # What should I call the key column
          value=Score,      # What should I call the values column
          HW.1:HW.3         # which columns to apply this to
          )
tidy.scores
```

```
##      name Assessement Score
## 1  Alison        HW.1     8
## 2 Brandon        HW.1     5
## 3 Charles        HW.1     9
## 4  Alison        HW.2     5
## 5 Brandon        HW.2     3
## 6 Charles        HW.2     7
## 7  Alison        HW.3     8
## 8 Brandon        HW.3     6
## 9 Charles        HW.3     9
```

To spread the key:value pairs out into a matrix, we use the spread() command.

```
# score columns into columns we'll name Assesment and Score
tidy.scores %>% spread( key=Assessement, value=Score )
```

```
##      name HW.1 HW.2 HW.3
## 1  Alison    8    5    8
## 2 Brandon    5    3    6
## 3 Charles    9    7    9
```

One way to keep straight which is the key column is that the key is the category, while value is the numerical value or response.

9.2 Exercises

1. Suppose we are given information about the maximum daily temperature from a weather station in Flagstaff, AZ. The file is available at the GitHub site that this book is hosted on.

   ```
   FlagTemp <-  read.csv(
     'https://github.com/dereksonderegger/STA_570L_Book/raw/master/data-raw/FlagMaxTemp.csv',
      header=TRUE, sep=',')
   ```

   ```
   ## Warning in read.table(file = file, header = header, sep = sep, quote
   ## = quote, : incomplete final line found by readTableHeader on 'https://
   ## github.com/dereksonderegger/STA_570L_Book/raw/master/data-raw/
   ## FlagMaxTemp.csv'
   ```

   This file is in a wide format, where each row represents a month and the columns X1, X2, ..., X31 represent the day of the month the observation was made.

a. Convert data set to the long format where the data has only four columns: `Year`, `Month`, `Day`, `Tmax`.

b. Calculate the average monthly maximum temperature for each Month in the dataset (So there will be 360 mean maximum temperatures).

c. Convert the average month maximums back to a wide data format where each line represents a year and there are 12 columns of temperature data (one for each month) along with a column for the year.

Chapter 10

Graphing using ggplot2

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 elements to combine.

To make the most of ggplot2 it is important to wrap your mind around "The Grammar of Graphics". 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 exam.
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.

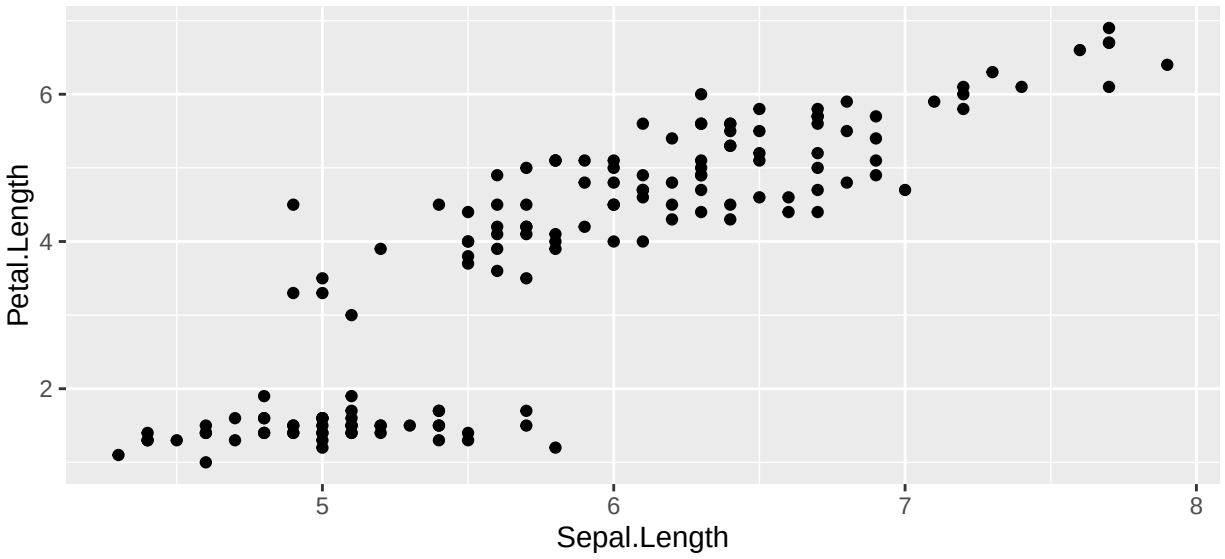
These notes are sufficient for graphing simple graphing, but are not intended to be exhaustive. There are many places online to get help with `ggplot2`. One very nice resource is the website [http://www.cookbook-r.com/Graphs/] which gives much of the information available in the book R Graphics Cookbook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

10.1 Basic Graphs

10.1.1 Scatterplots

To start with, we'll make a very simple scatterplot using the `iris` dataset that will make a scatterplot of `Sepal.Length` versus `Petal.Length`, which are two columns in my dataset.

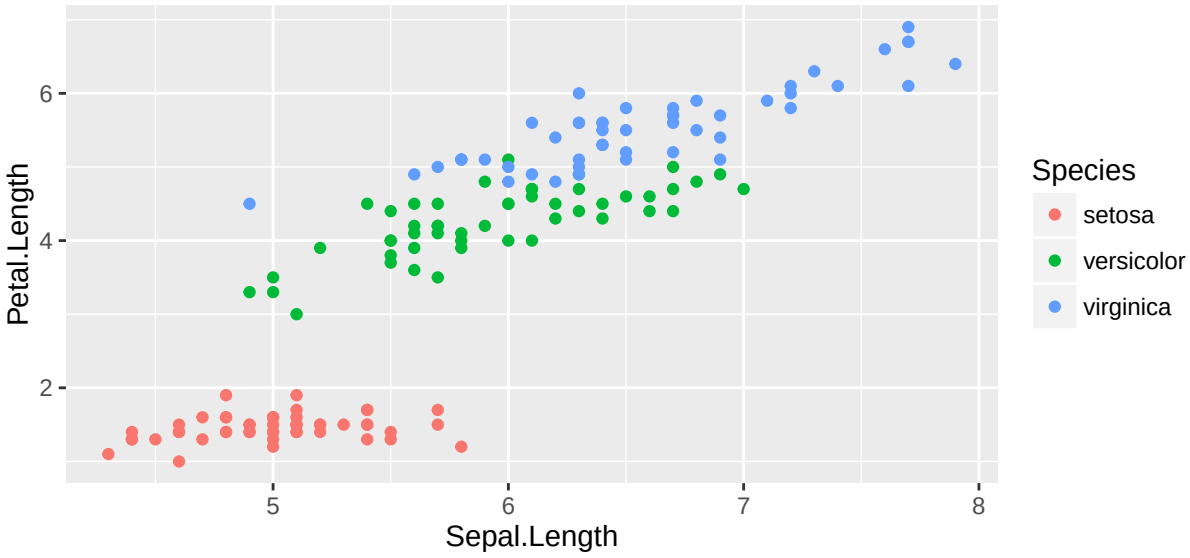
```
library(ggplot2)
ggplot( data=iris, aes(x=Sepal.Length, y=Petal.Length) ) +
    geom_point(  )
```



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.

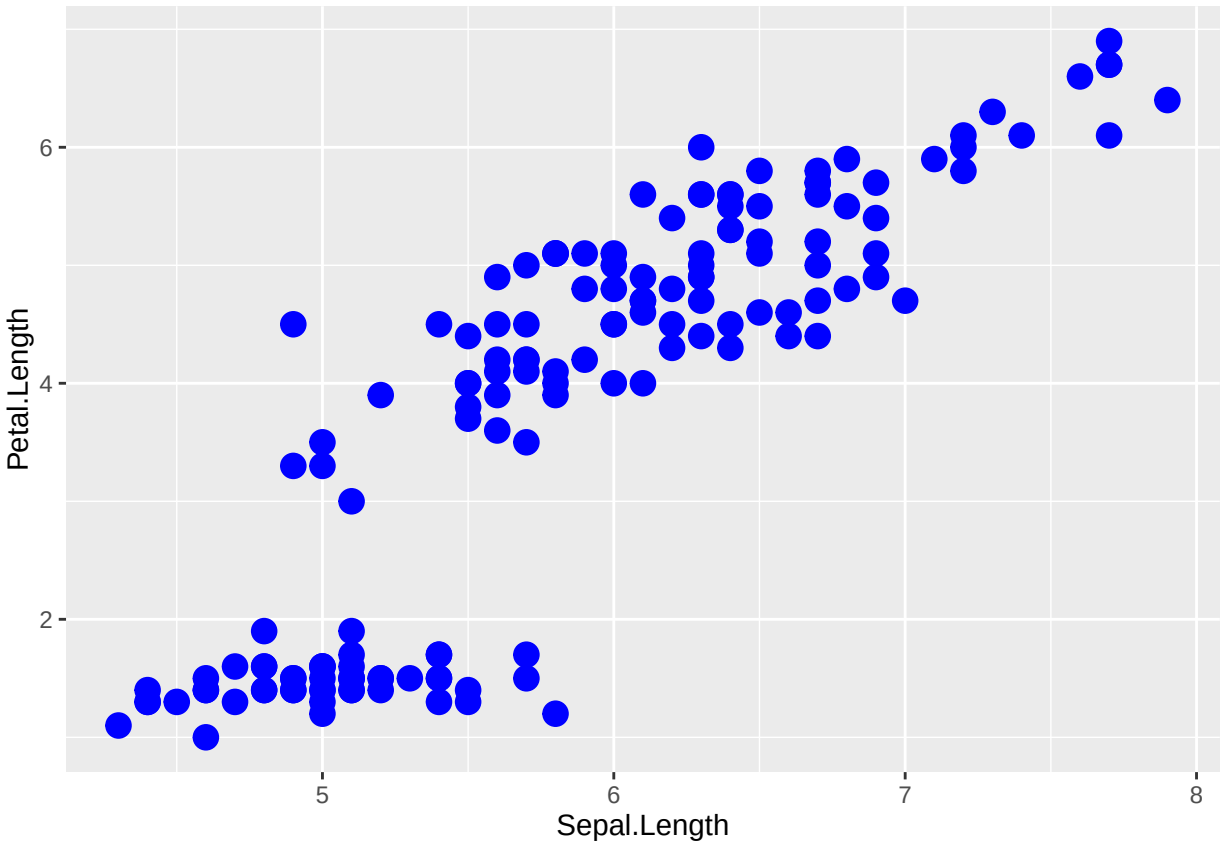
We can define other attributes that might reflect other aspects of the data. For example, we might want for the of the data point to change dynamically based on the species of iris.

```
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

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



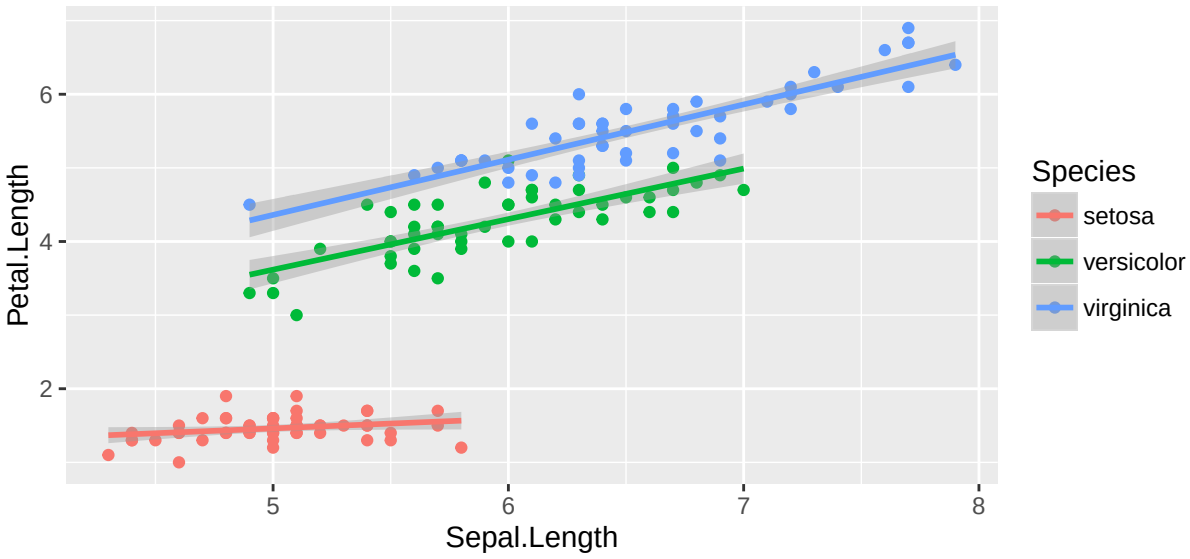
The important part isn't that color and size were defined in the geom_point() but that they were defined outside of an aes() function!

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.

Next, I suppose I want to add a regression line (the line that best summarizes the relationship between Sepal.Length and Petal.Length) to each of these groups. We can do this by adding another layer to the graph, in this case, a `smoother` layer. The `geom_smoother` is intended to take a scatterplot of points and draw the best-fitting curve to the data. There are several options for how it choses to do this, but I'll tell it to fit a regression line to each set of data. Below, we have a graph of data points, a regression line, and a confidence region for the line (I typically don't use this method because it has too many limitations as to how I fit the smoother. I prefer to fit a model to the data, calculate the predicted values along with whatever confidence intervals I want, and plot those directly using `geom_line()` and `geom_ribbon()`).

```
ggplot( data=iris, aes(x=Sepal.Length, y=Petal.Length, color=Species) ) +
  geom_point( ) +
  geom_smooth( method='lm' ) # fit a regression to each species
```



10.1.2 Bar Charts

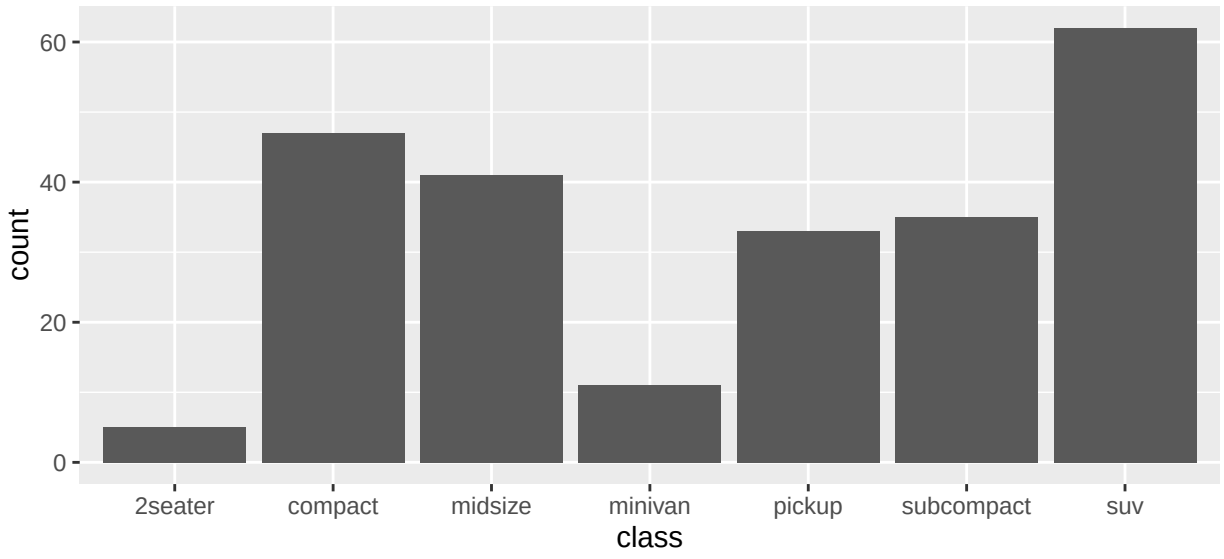
For displaying a categorical variable on the x-axis and a continuous variable on the y-axis, a bar chart is a good option. Here we consider a data set that gives the fuel efficiency of different classes of vehicles in two different years. This is a subset of data that the EPA makes available on [http://fueleconomy.gov]. 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`.

```
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" ...
```

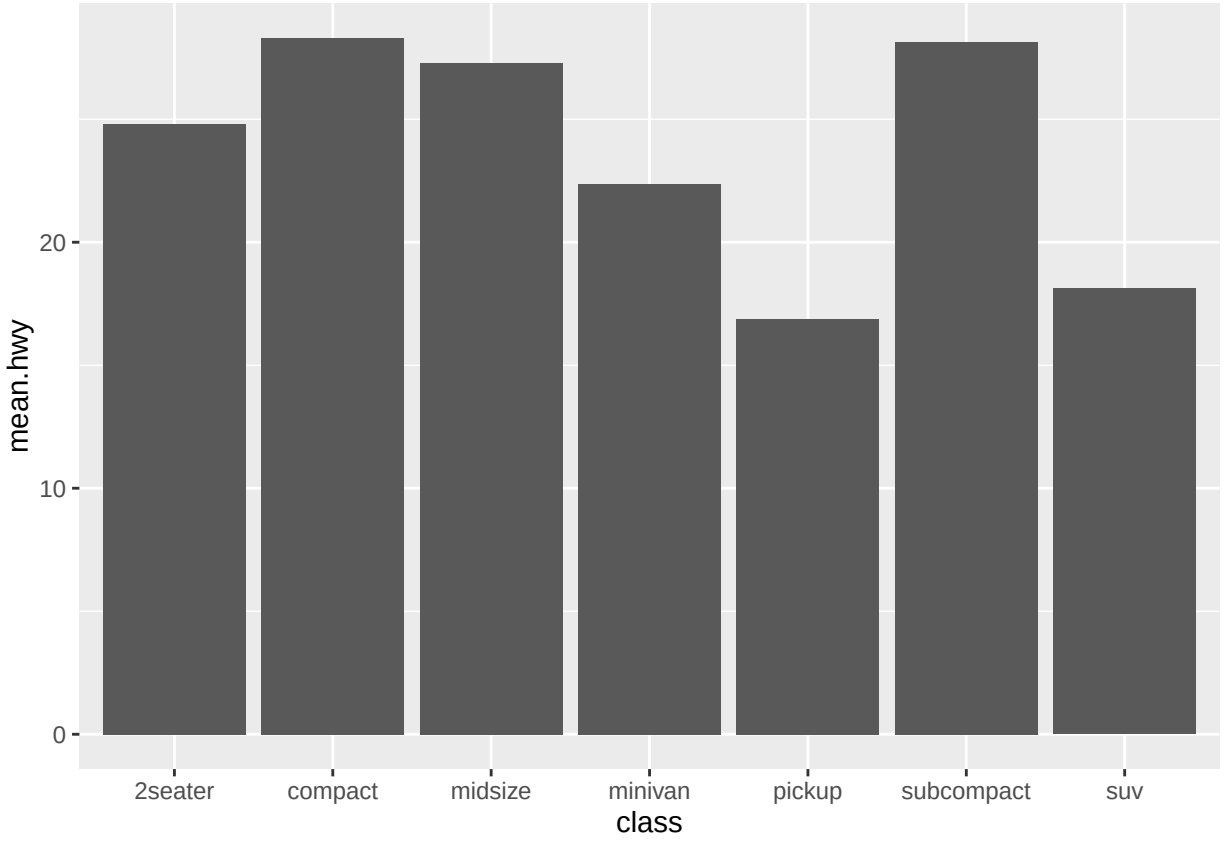
First we could summarize the data by how many of each model there are in the different classes.

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



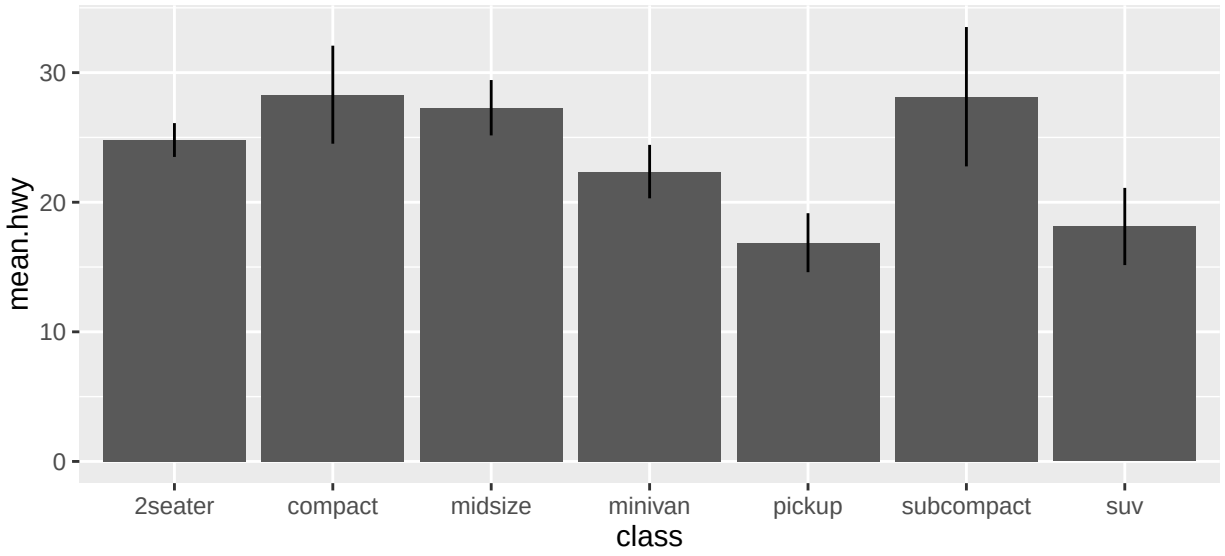
By default, the `geom_box()` just counts the number of cases and displays how many observations were in each class. If we were interested in knowing the mean highway fuel efficiency, we would have to summarize the data and calculate the mean for each class. Fortunately that is pretty easy to do.

```
library(dplyr)  # for calculating the summary statistics
mpg.small <- mpg %>%
  group_by(class) %>%
  summarise(mean.hwy = mean(hwy),
            sd.hwy   = sd(hwy))

ggplot(mpg.small, aes(x=class, y=mean.hwy)) +
  geom_bar( stat = 'identity' )              # no further summarization!
```



The stat='identity' part is necessary to keep geom_bar() from doing any default summarization. We can add some error bars to show the standard deviation as well.

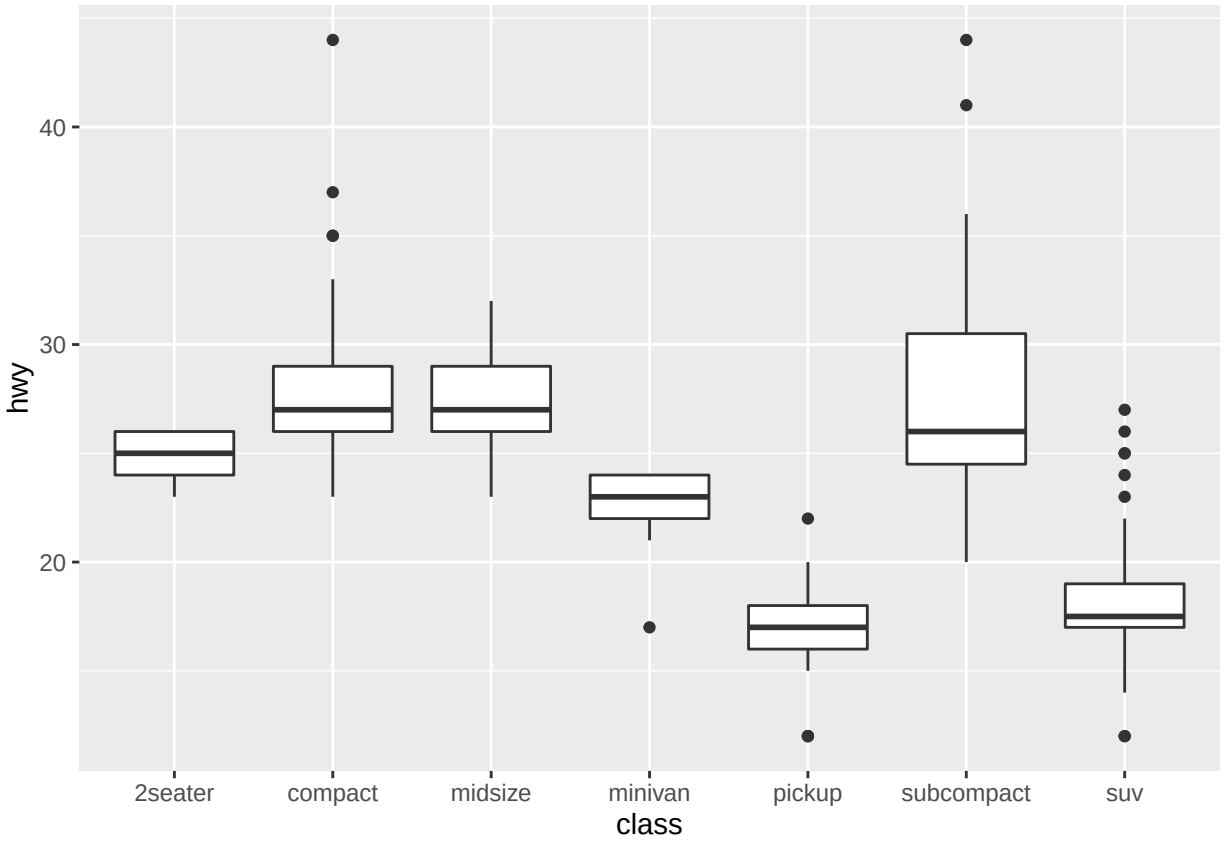
```
ggplot(mpg.small, aes(x=class, y=mean.hwy,
                      ymin = mean.hwy-sd.hwy,
                      ymax = mean.hwy+sd.hwy)) +
  geom_bar( stat = 'identity' ) +
  geom_linerange()
```



10.1.3 Box Plots

Boxplots are a common way to show a categorical variable on the x-axis and continuous on the y-axis. I actually prefer these over the barchart we did prior.

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



10.1.4 Geometries and Layers

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_bar`	A barplot	`x`
`geom_boxplot`	Boxplots	`x`
`geom_density`	A smoothed histogram	`x`
`geom_errorbar`	Error bars	`ymin, ymax`
`geom_histogram`	A histogram	`x`
`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	Description	Required Aesthetics
geom_ribbon	Enclose a region, and color the interior	ymin, ymax
geom_smooth	Add a ribbon that summarizes a scatterplot	x, y
geom_text	Add text to a graph	x, y, label

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!

- Can add layers with a +, or you can define two plots and add them together (second one over-writes anything that conflicts).

10.2 Getting Fancy

10.2.1 Bar Plot

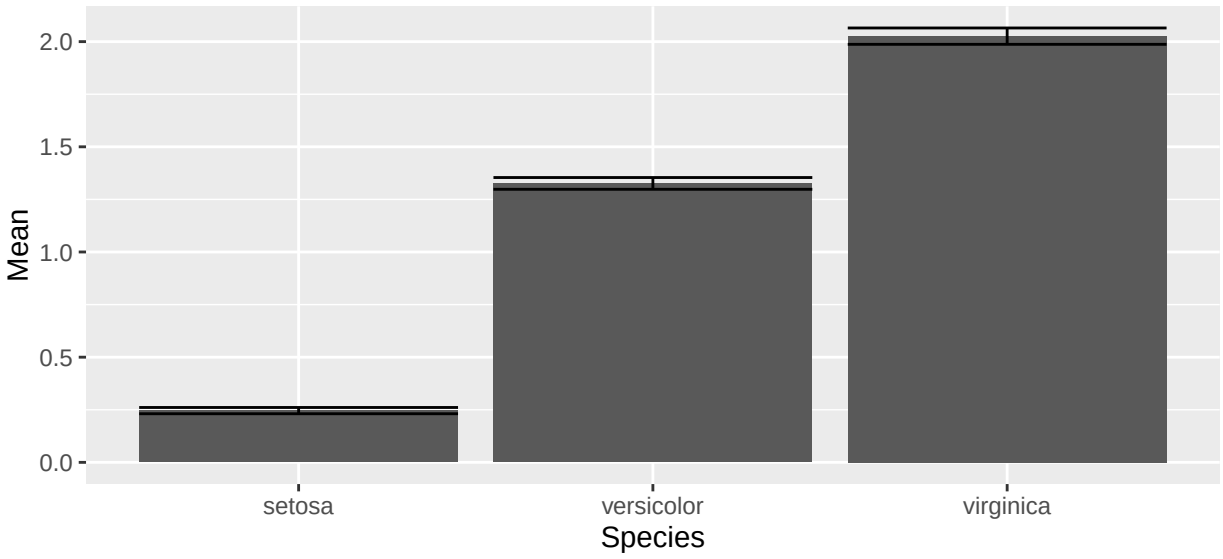
Suppose that you just want make some barplots and add ± S.E. bars. This should be really easy to do, but in the base graphics in R, it is a pain. Fortunately in ggplot2 this is easy. First, define a data frame with the bar heights you want to graph and the ± values you wish to use.

```
# Calculate the mean and sd of the Petal Widths for each species
library(dplyr)
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
##       <fctr> <dbl>      <dbl>     <dbl>     <dbl>
## 1     setosa 0.246 0.01490377 0.2310962 0.2609038
## 2 versicolor 1.326 0.02796645 1.2980335 1.3539665
## 3  virginica 2.026 0.03884138 1.9871586 2.0648414
```

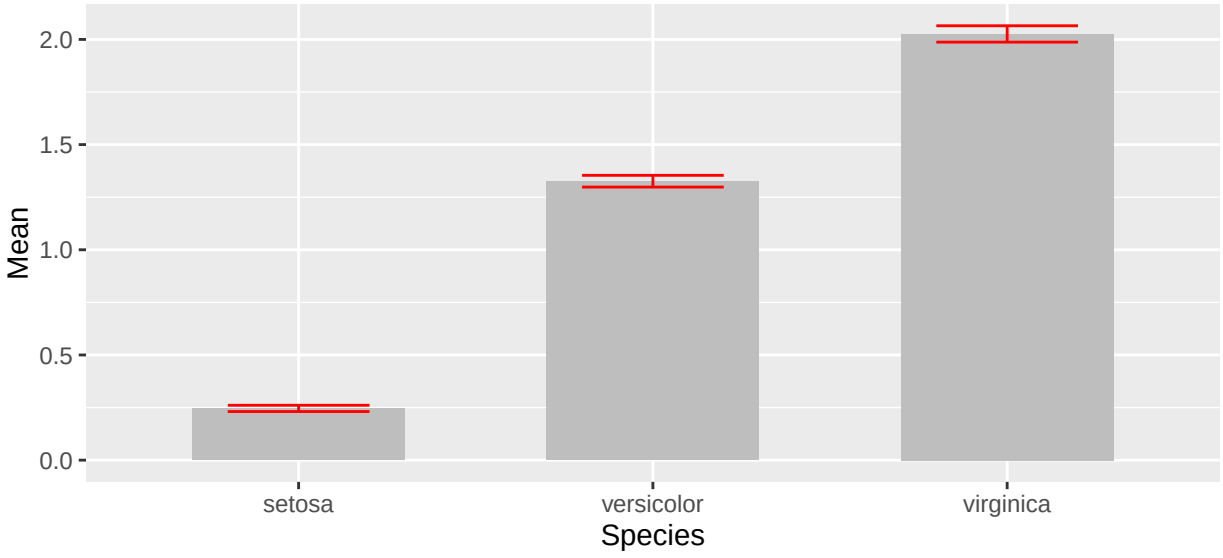
Next we take these summary statistics and define the following graph which makes a bar graph of the means and error bars that are ± 1 estimated standard deviation of the mean (usually referred to as the standard errors of the means). By default, geom_bar() tries to draw a bar plot based on how many observations 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.

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



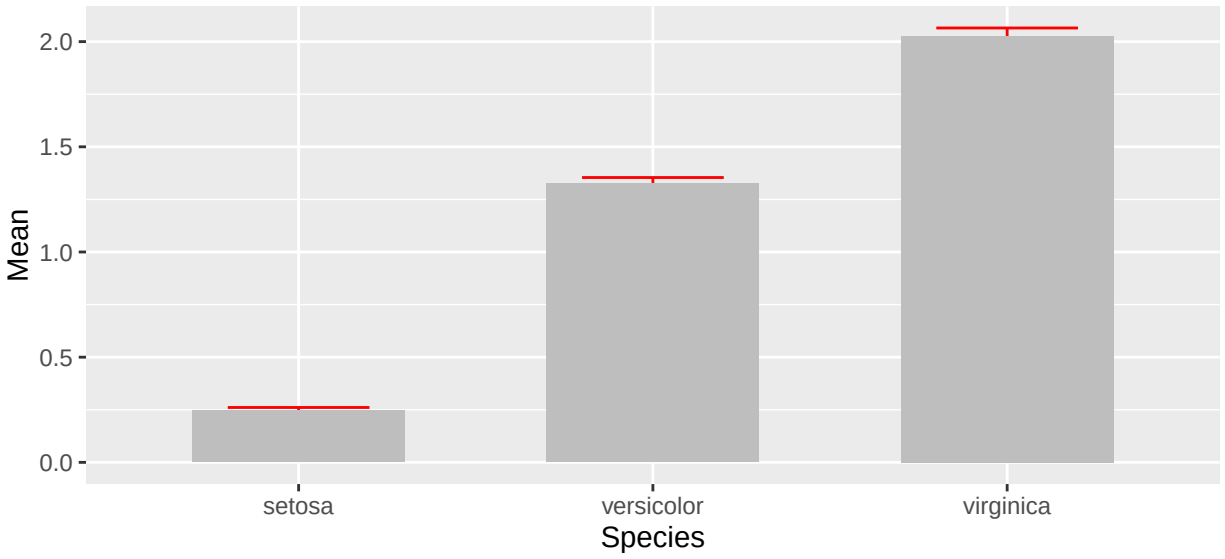
While this isn't too bad, we would like to make this a bit more pleasing to look at. Each of the bars is a little too wide and the error bars should be a tad narrower than then bar. Also, the fill color for the bars is too dark. So I'll change all of these, by setting those attributes *outside of an*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

```
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.

```
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)
```

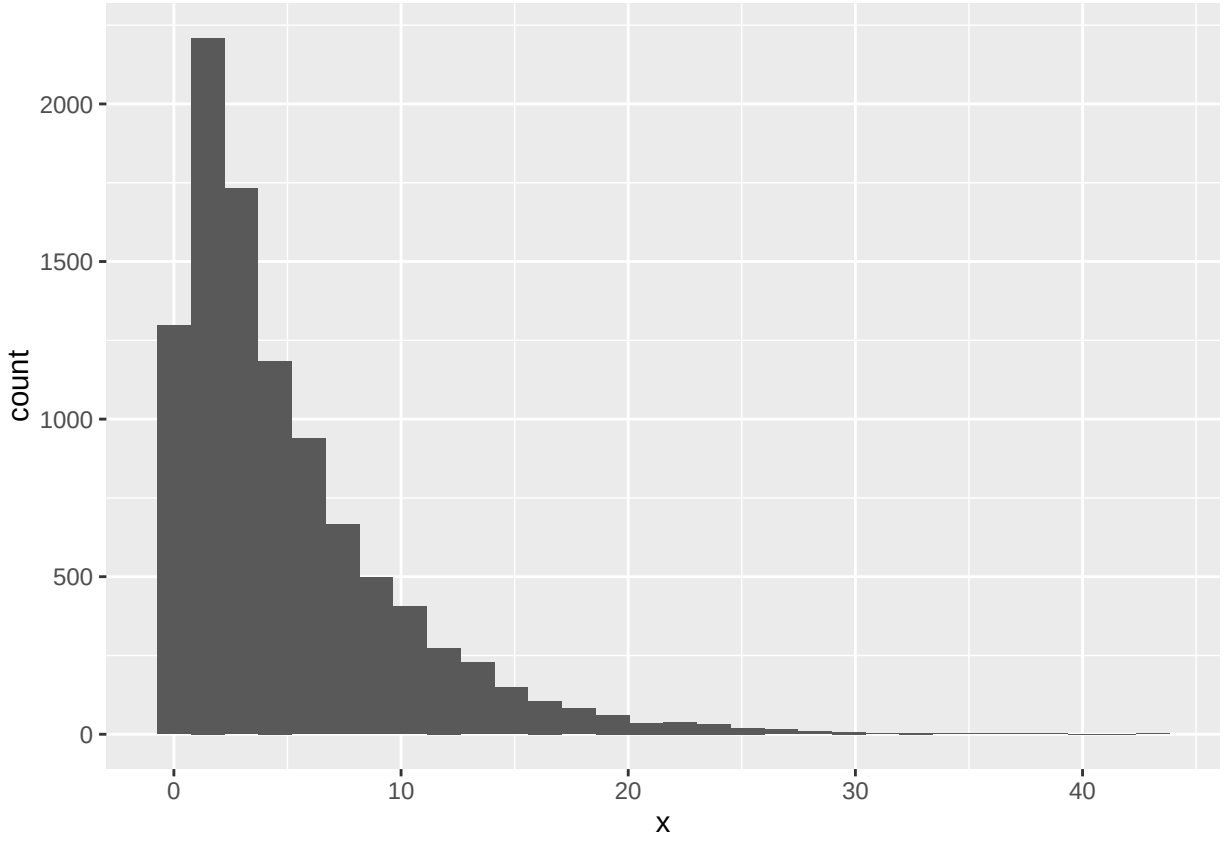


10.2.2 Histograms

Creating histograms of continuous data is a very common thing to do. The simplest way to do this in ggplot() is using the geom_histogram function. The simplest form is the the following:

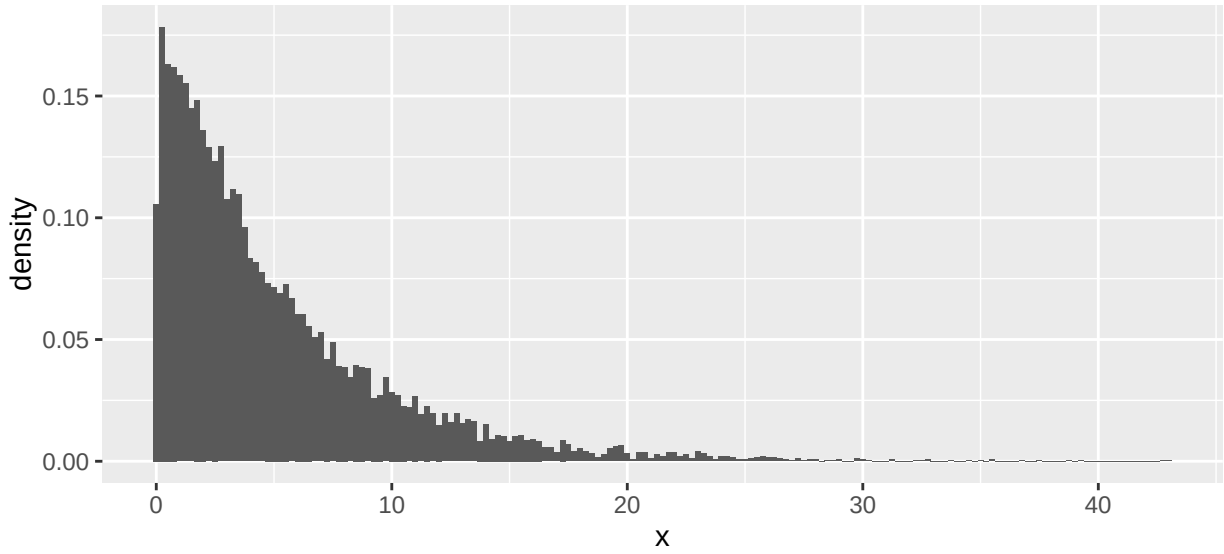
```
data <- data.frame(x=rexp(10000, rate=1/5))
ggplot(data, aes(x=x)) +
  geom_histogram()
```

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



Somewhat annoyingly, `ggplot2` does not by default use an intelligent choice for the number of bins. Instead we are stuck investigating different bin-widths by hand. To do this, we set the number of bins via the `binwidth` argument. Notice that the y-axis is the number of observations in each bin. If we want the y-axis to be density (so that the area shaded has area 1), we just need to tell geom_histogram to have `y=..density..` instead of the default.

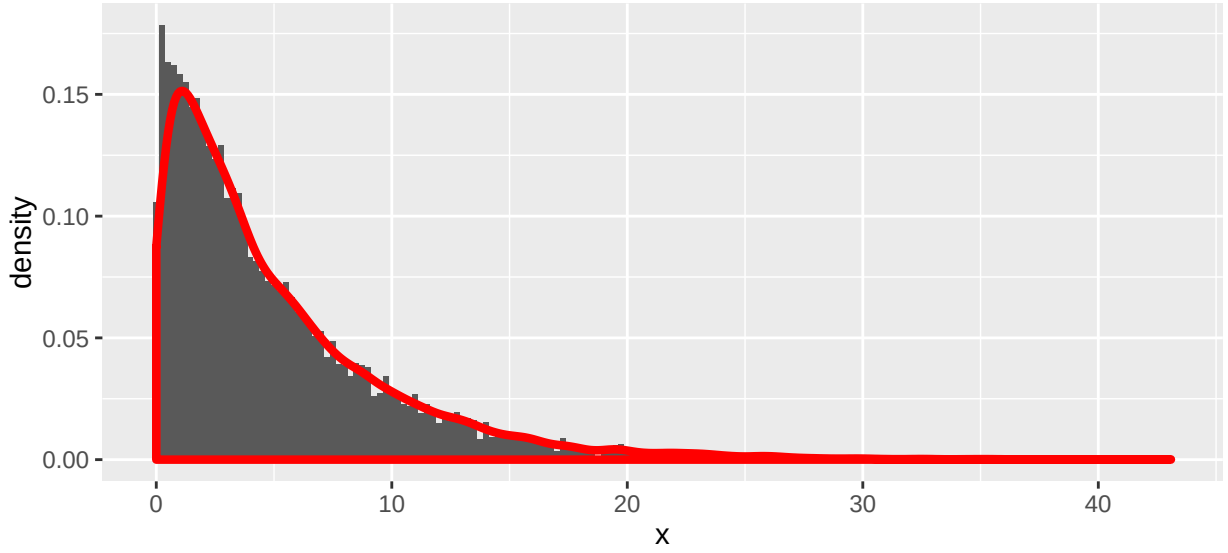
```
ggplot(data, aes(x=x, y=..density..)) +
  geom_histogram(binwidth=.25)
```



Often I want to also add some sort of smoothed density plot to my histogram. The geom to do that is

geom_density() which takes your x-values and creates a smoothed density function using kernel density algorithm with a normal kernel. To do this, we need both layers of my plot to have a y-axis of density.

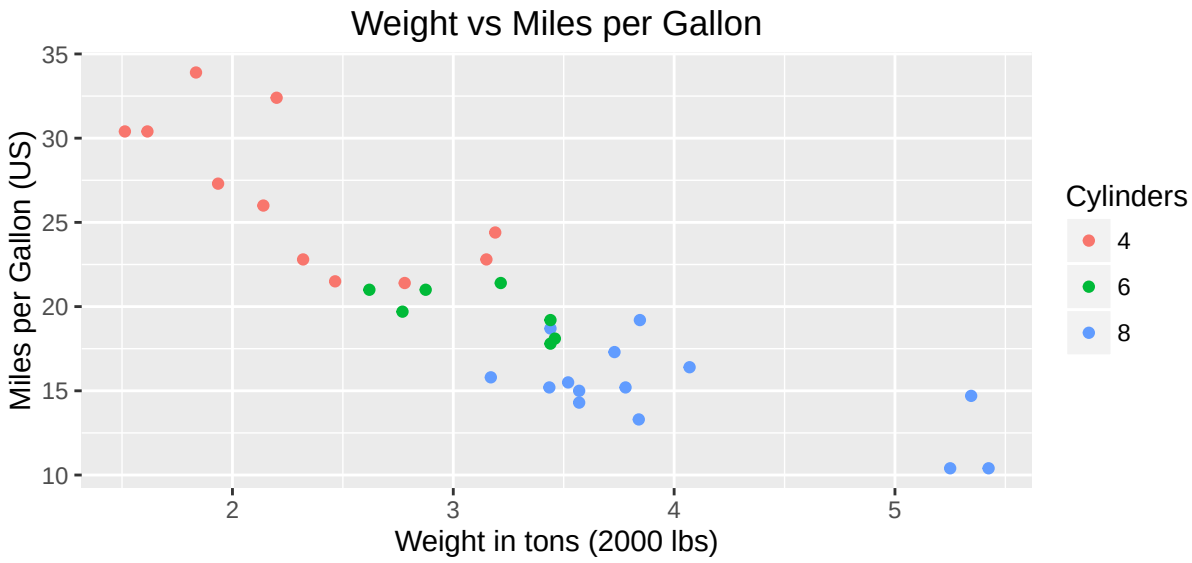
```
ggplot(data, aes(x=x, y=..density..)) +
  geom_histogram(binwidth=.25) +
  geom_density(color='red', size=1.5)
```



10.2.3 Adjusting 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)" ) +
  labs( y="Miles per Gallon (US)") +
  labs( color="Cylinders")
```



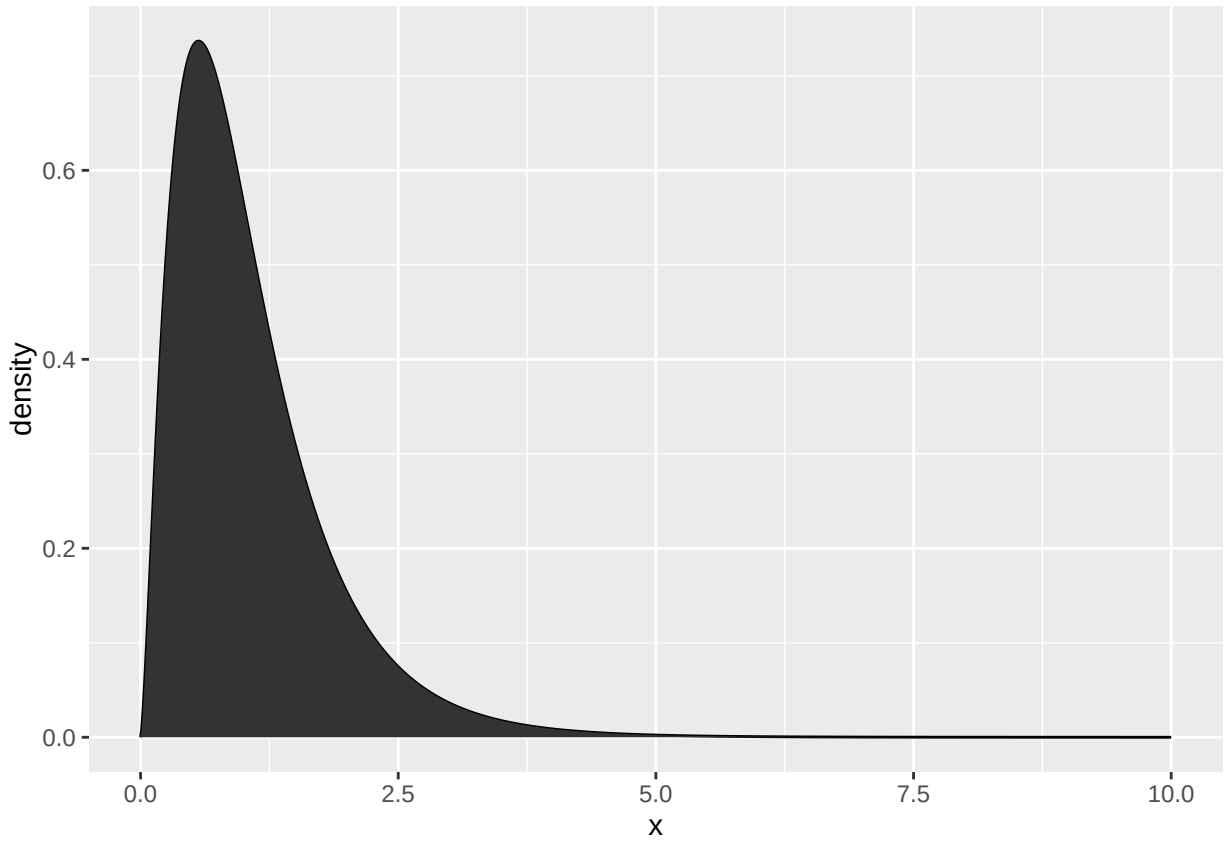
10.2.4 Plotting distributions

Often I need to plot a distribution and perhaps shade some area in. In this section we'll give a method for plotting continuous and discrete distributions using `ggplot2`.

10.2.4.1 Continuous distributions

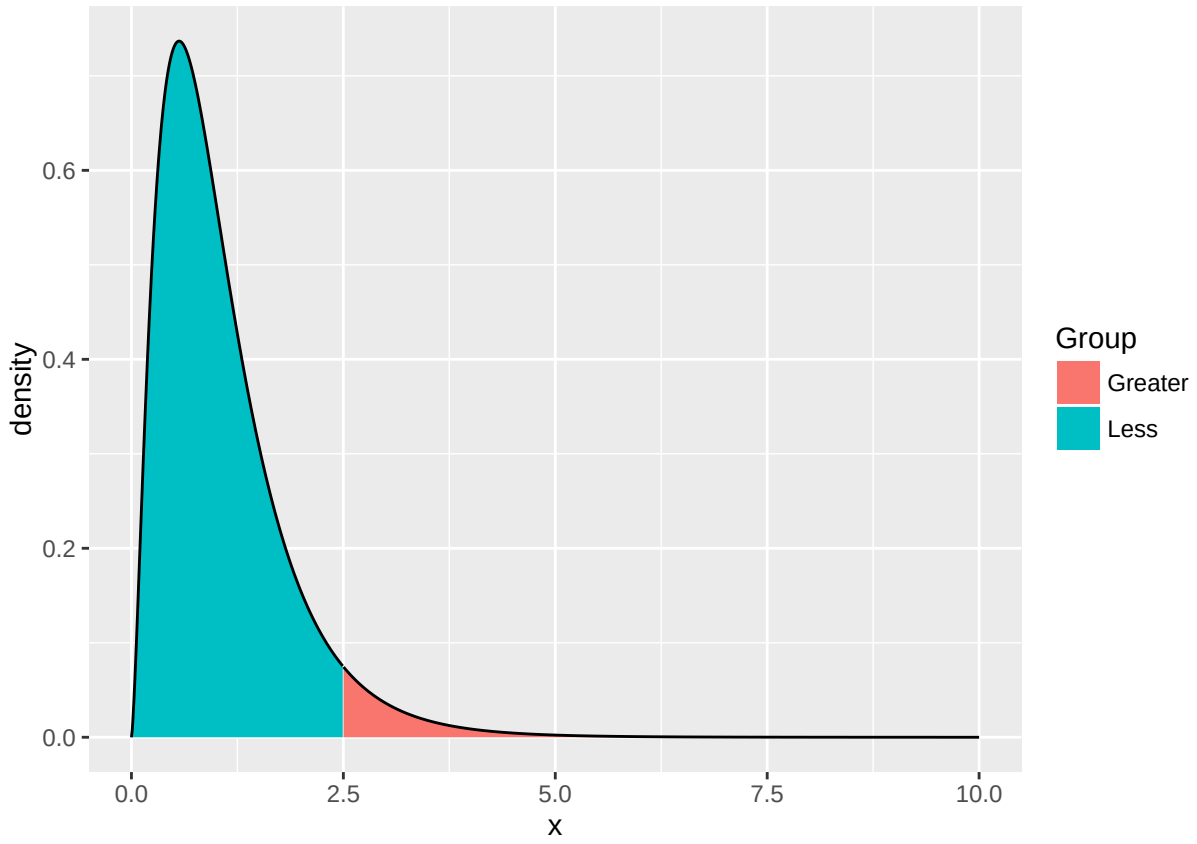
First we need to create a data.frame that contains a sequence of (x,y) pairs that we'll pass to our graphing program to draw the curve by connecting-the-dots, but because the dots will be very close together, the resulting curve looks smooth. For example, lets plot the F-distribution with parameters $\nu_1 = 5$ and $\nu_2 = 30$.

```
# define 1000 points to do a "connect-the-dots"
plot.data <- data.frame( x=seq(0,10, length=1000) ) %>%
  mutate( density = df(x, 5, 30) )

ggplot(plot.data, aes(x=x, y=density)) +
  geom_line()  +   # just a line
  geom_area()      # shade in the area under the line
```



This isn't too bad, but often we want to add some color to two different sections, perhaps we want different colors distinguishing between values ≥ 2.5 vs values < 2.5

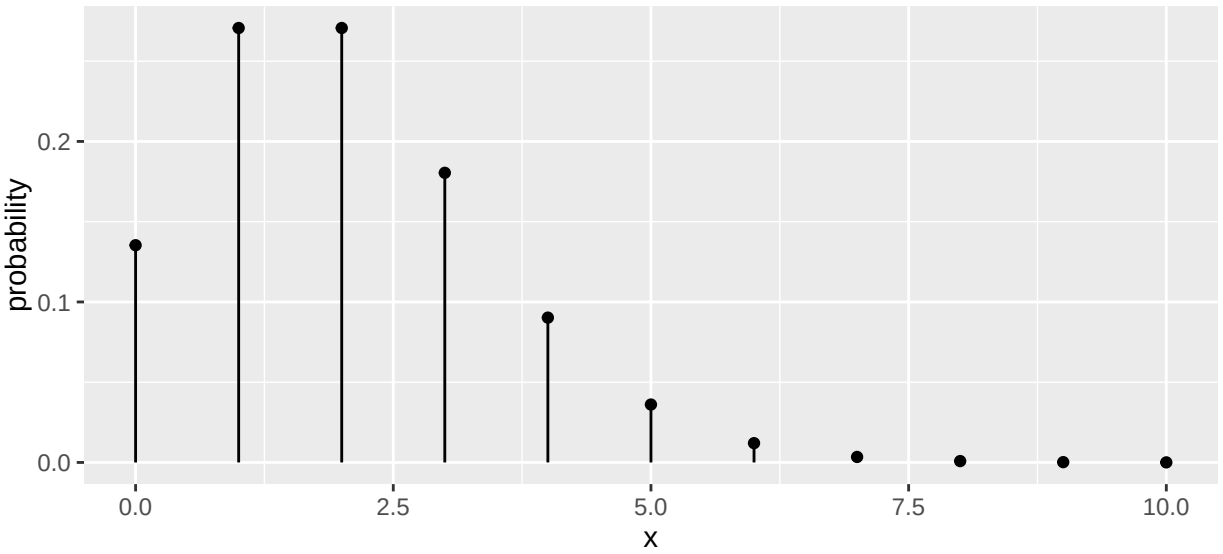
```
plot.data <- data.frame( x=seq(0,10, length=1000) ) %>%
  mutate( density = df(x, 5, 30),
          Group = ifelse(x <= 2.5, 'Less','Greater') )

ggplot(plot.data, aes(x=x, y=density, fill=Group)) +
  geom_area() +
  geom_line()
```



10.2.4.2 Discrete distributions

The idea for discrete distributions will be to draw points for the height and then add bars. Lets look at doing this for the poisson distribution with rate parameter $\lambda = 2$.

```
plot.data <- data.frame( x=seq(0,10) ) %>%
  mutate( probability = dpois(x, lambda=2) )

ggplot(plot.data, aes(x=x)) +
  geom_point( aes(y=probability) ) +
  geom_linerange(aes(ymax=probability), ymin=0)
```



The key trick here was to set the `ymin` value to always be zero.

10.3 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.

   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#prefix=&sw=both&exact=false]. 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 tricks.

Bibliography