MATH 3070 R Lecture Notes

Curtis Miller

2018-08-17

Contents

Preface **5**

Lecture 1 **7**
R Basics . . . 7

Lecture 2 **15**
Univariate Data Analysis . . . 15

Lecture 3 **35**
R Data Structures . . . 35

Lecture 4 **41**
Working with Data Frames . . . 41
Applying a Function Over a Collection . . . 42
Using External Data . . . 46

Lecture 5 **49**
Multivariate Visualization . . . 49
Base R Plotting . . . 49

Lecture 6 **57**
lattice Plotting . . . 57
ggplot2 Plotting . . . 62

Lecture 7 **85**
Distribution Comparison . . . 85
Model Formula Interface . . . 87
Splitting and Stacking Data . . . 90

Lecture 8 **95**
Paired Data . . . 95
Categorical Bivariate Data . . . 103

Lecture 9 **109**
Random Number Generation . . . 109
Families of Distributions . . . 110
Central Limit Theorem . . . 123

Lecture 10 **127**
magrittr and the Pipe Operator . . . 127
dplyr: Subsetting With Power . . . 130
reshape2: Melting and casting data . . . 136

Lecture 11 **141**
Inferential Statistics via Computational Methods 141
Simulation 141
Significance Testing 147

Lecture 12 **151**
Bootstrapping 151
Bayesian Statistics 152

Lecture 13 **157**
Confidence Interval Basics 157
Confidence Intervals "By Hand" 158
R Functions for Confidence Intervals 159

Lecture 14 **171**
Hypothesis Testing Basics 171
Hypothesis Testing "By Hand" 172
R Functions for Statistical Testing 173
Power Analysis 181
p-Hacking 183

Conclusion **185**

Preface

These lecture notes were written by me to accompany John Verzani's *Using R for Introductory Statistics (2nd ed.)*, to be delivered in lectures teaching students how to program with R in the programming lab accompanying a lecture section focusing on the statistical methods themselves. No knowledge of programming is assumed; my objective was to teach basic R programming well enough to use R for statistical analyses.

These notes are not intended to stand alone; I like Verzani's book and I believe that these notes should supplement it, not replace it. For those taking the programming lab for the University of Utah's Mathematics Department statistics courses, I would *insist* on reading Verzani's book in addition to these lecture notes. However, these notes could serve as a light weight introduction to R and statistical programming.

These lecture notes were adapted from lecture notes written for an eight-week intensive course covering the same topics that I also wrote. Most of the notes are an exact duplication, but in order to accomodate the instructors of the lecture sections (none of which I was teaching at the time) I added and rearranged lectures to slow down the lab's pace.

Lectures 1 through 4 cover R basics. Lectures 5 and 6 cover plotting. Lectures 7 and 8 cover multivariate analysis (lightly; this is a topic covered in greater depth in another course). Lecture 9 discusses probability models in R. Lecture 10 dives into the "tidyverse", discussing **dplyr**, **magrittr**, and **reshape2** for data manipulation (this does *not* correspond to material in Verzani's book). Lecture 11 discusses computer-intensive methods for hypothesis testing (based on resampling methods). Lecture 12 discusses bootstrapping and Bayesian statistics (*very light* Bayesian statistics). Chapter 13 discusses confidence intervals, and Chapter 14 covers hypothesis testing.

I hope that you find these notes useful, and wish you the best of luck.

Curtis Miller

Lecture 1

R Basics

Using R as a Calculator

Lots of work in R is done via the interpreter. You can use the interpreter as a calculator, and thus can do some simple calculations.

```
# Some basic arithmetic
2 + 2
```

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

```
1 - 7
```

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

```
2 * 3
```

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

```
(1 + 1) * 3
```

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

```
2^3
```

```
## [1] 8
```

```
10/2
```

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

```
# One can also use functions for more advanced calculations
sqrt(9)
```

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

```
exp(3)
```

```
## [1] 20.08554
```

```
# A familiar built-in value
pi
```

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

When using a computer as a calculator, though, you may observe behavior that seems... strange.

```
# Be aware that any time you use computers for calculation you may get the
# 'wrong' answer. This is due to floating-point arithmetic and how computers
```

```
# see numbers.
sqrt(2)^2
```

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

```
# What's this?
sqrt(2)^2 - 2
```

```
## [1] 4.440892e-16
```

(For an explanation of this phenomenon, watch this video by Computerphile on floating-point numbers.)

Obviously, though, we want to use R for more than a glorified calculator.

Variables

As with any programming language, R can store information in **variables**. R has a few ways to store variables, with the `<-`, `->`, and `=` operators (there are a few others that I won't discuss here).

```
# One way to create a variable
var1 <- 1
# Another way to create a variable
var2 = 2
# A third way, this time where the variable is on the RIGHT side of the
# assignment operator!
var3 <- 3
# What do you get?
var1 + var2
```

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

```
var2 * var3
```

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

There are rules as to how variables can be named. In addition to not being reserved words, they can only use alpha-numeric characters (letters and numbers), or the `_` or `.` characters. Also, they cannot begin with a number (so while `var1` is legal, `1var` is not). Variables are case-sensitive; `var1` is not the same as `Var1` or `VAR1`.

```
# These are all valid names for variables
var1 <- 10
variable_number_2 <- 20
variable.number.3 <- 30  # <---- I would suggest avoiding this style of variable naming, though; use '_
variableNumber4 <- 40

# None of these are the same
var1 <- 10
Var1 <- 230
VAR1 <- -5


var1 + 1
```

```
## [1] 11
```

```
Var1 + 1
```

```
## [1] 231
```

```
VAR1 + 1
```

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

As a side note, notice that a line break will usually begin a new command. **This is not always true!** If a command is not finished from R's perspective, it will see the contents of the next line as being part of the same command. This can sometimes be confusing, while other times it can be used to your advantage. Furthermore, you can use the `;` character to separate commands on the same line, or explicitly end a command like in C-based programming languages like C, C++, or Java (though this is usually not done). Aside from this, R generally ignores white space (space, tabs, new lines, etc.).

```
# Because there is nothing after the '+' on the first line, R looks for the
# rest of the command on the second line.
5 +
5
```

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

```
# It could even be a few lines down! (DON'T EVER DO THIS!!!!)
10 +



1
```

```
## [1] 11
```

R variables are untyped, but that does not mean that the object they reference is untyped. Some basic types you will encounter early (and there are *many, many others*) include:

- **Numeric**, like `124` or `3.14159`
- **Character**, like `"hello bobby"` or `"124"` or `'cmiller@math.utah.edu'`
- **Boolean**, either `TRUE` or `FALSE`
- **Vectors**, which are a collection of objects of all the same type
- **Functions**, which you can think of as being a "mini program", like `log`, `sqrt`, `mean`, or `help`
- **Data frames**, which store datasets in memory in a tabular format

```
# A numeric variable
num_var <- 124
# Character data
char_var <- "hello"
# Boolean
bool_var <- TRUE
# A vector of data
data_vec <- c(1, 20, 6, 2)
# Another vector of data
data_vec2 <- c("hello", "world")

# There are functions for type checking
not_a_number <- "124"
is.numeric(not_a_number)
```

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

```
is.character(not_a_number)
```

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

```
# Sometimes you can force a variable of one type to be another type.
is_a_number <- as.numeric(not_a_number)
is.numeric(is_a_number)
```

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

Packages

The true power of R is in its packages. R has an ever-growing community of users and developers, many of whom write free packages to extend R's functionality. These packages are made available to all R users on websites like CRAN (the Comprehensive R Archive Network) or GitHub.

Packages from CRAN are easily downloaded and installed using the `install.packages()` function, like so:

```
# Install the "UsingR" package for Verzani's book
install.packages("UsingR")
```

Once you install a new package, you can load it into the R environment using `require()` or `library()`.

```
library(UsingR)
# Alternatively, you could use require("UsingR")
```

Today there are well over 7000 packages on CRAN alone, and this growth is likely to continue.

Data sets

Datasets in R are usually stored in vectors (for univariate data) or data frames (for multivariate data). For now, we will work with built-in data sets or those included in packages. If a dataset isn't already in the R environment but does exist in some package, it can be brought in using the `data()` function. The `head()` function allows us to view only a few observations from a dataset (in order to prevent our screen from being flooded with all the data), and the `str()` function gives us a further description of the dataset.

```
# Let's load in a univeriate dataset containing the lengths of major North
# American rivers
data(rivers)
# We can see a few of the observations to get a glimpse at the nature of the
# data.
head(rivers)
```

```
## [1] 735 320 325 392 524 450
```

```
# We can see more information via str()
str(rivers)
```

```
##  num [1:141] 735 320 325 392 524 ...
```

```
# Just for fun, what is the combined length of all these rivers?
sum(rivers)
```

```
## [1] 83357
```

```
# Let's have even more fun by seeing what proportion of this total length
# each river accounts for.
rivers/sum(rivers)
```

```
##   [1] 0.008817496 0.003838910 0.003898893 0.004702664 0.006286215
##   [6] 0.005398467 0.017503029 0.001619540 0.005578416 0.007197956
##  [11] 0.003958876 0.004030855 0.003359046 0.003778927 0.010437036
```

```
##  [16] 0.010868913 0.002423312 0.003946879 0.003479012 0.011996593
##  [21] 0.007197956 0.006058279 0.017395060 0.010077138 0.014911765
##  [26] 0.010676968 0.004198808 0.004882613 0.003431026 0.003359046
##  [31] 0.006298211 0.008637547 0.004678671 0.002999148 0.003922886
##  [36] 0.002759216 0.003179097 0.010197104 0.002519285 0.007557854
##  [41] 0.003119114 0.002759216 0.004318773 0.008757513 0.007197956
##  [46] 0.003670957 0.004678671 0.005038569 0.003491009 0.008517581
##  [51] 0.004078842 0.002603261 0.003371043 0.004222801 0.003107118
##  [56] 0.002999148 0.005638399 0.008157683 0.006838058 0.004198808
##  [61] 0.003598978 0.006718092 0.010796934 0.007497871 0.003982869
##  [66] 0.028168000 0.014048010 0.044507360 0.027772113 0.030387370
##  [71] 0.009357343 0.003359046 0.004918603 0.005518433 0.003119114
##  [76] 0.003059131 0.005170532 0.004198808 0.009117411 0.007413894
##  [81] 0.004054848 0.011768658 0.015667550 0.005998296 0.008349629
##  [86] 0.007257939 0.002999148 0.004930600 0.012644409 0.008817496
##  [91] 0.002795206 0.005218518 0.005878331 0.003718944 0.005518433
##  [96] 0.004594695 0.004498722 0.015235673 0.006538143 0.005338484
## [101] 0.022613578 0.004558705 0.003598978 0.004558705 0.004522716
## [106] 0.005098552 0.003311060 0.002519285 0.009597274 0.005038569
## [111] 0.004198808 0.004318773 0.006454167 0.013196252 0.014455895
## [116] 0.003766930 0.002843193 0.007317922 0.004318773 0.006478160
## [121] 0.012452464 0.005086555 0.003718944 0.003598978 0.005326487
## [126] 0.003610974 0.003215087 0.007437888 0.002579267 0.007821779
## [131] 0.010796934 0.006298211 0.002951162 0.004318773 0.006346198
## [136] 0.005998296 0.008637547 0.003239080 0.005158535 0.008049714
## [141] 0.021233970
```

```
# Let's look at our first ever data frame, the famous iris dataset
head(iris)
```

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

```
str(iris)
```

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

```
# You can think of this as being a table or matrix. In fact, it's a
# combination of equal-length vectors. We can get the Sepal.Length vector
# using the $ operator
iris$Sepal.Length
```

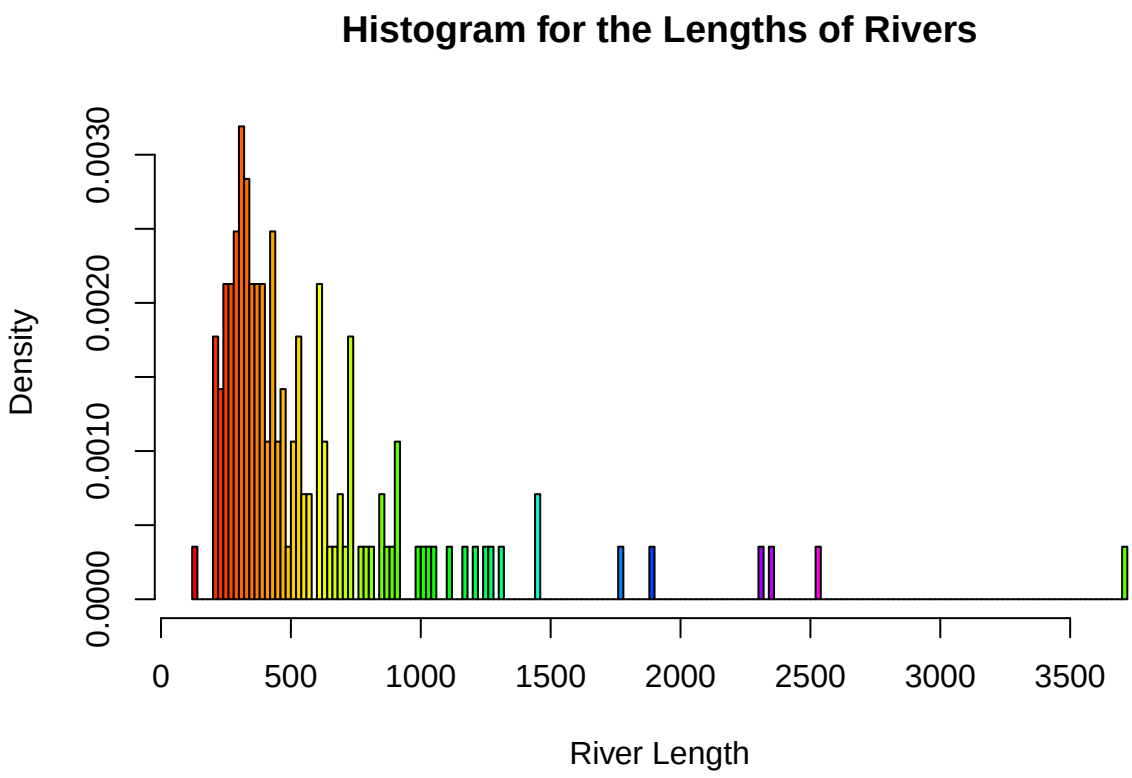
```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
##  [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
##  [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
##  [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8
```

```
##  [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4
##  [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

Comments

You may have noticed already the `#` symbol in the code. This is called a **comment**. The R interpreter completely ignores comments. They are used for annotating code. *Commenting is not optional; it's essential to understanding code when it is read (and I promise you, it will be). You can't write good code without comments. Consequently, I require you to write good, extensive comments in your homework!*

```
# Hi! I'm a comment! R doesn't care about me, but that's okay. I'm very
# useful for understanding code, and if you forget to add me, an angry
# programmer will come to your house and punch you in the face!

# Here, I'm saying that the following code makes a histogram of the rivers
# data, using relative frequency, a custom main title, a custom x-axis
# label, a bin count based on the square root of the size of the dataset,
# right-inclusive classes, and a beautiful rainbow fill because we're classy
# people who thinks this looks good (even though it doesn't)
hist(rivers, breaks = ceiling(length(rivers)), freq = FALSE, right = FALSE,
    main = "Histogram for the Lengths of Rivers", xlab = "River Length", col = rainbow(ceiling(length(r
```



Help

R has many, many functions. In 2014, there were approximately 182,393 R functions that could be used (most of them in packages). There is no way anyone could remember all of them.

Fortunately, it's easy to access documentation in R, especially if you are using RStudio. The `help()` function will look up documentation for a string entered. This is made even easier by just typing `?` and the name of the package/function/dataset you want to see documentation for.

```
# Here's how to access the documentation of the mean() function
help("mean")
# Or even easier:
?mean
```

Lecture 2

Univariate Data Analysis

Vectors

Univariate data is usually stored as vectors. The most basic function for creating a vector is the `c()` function, where the arguments of the function (separated by commas) become the contents of the vector. It generally doesn't matter what the type of data the contents of the vector are *so long as they are all the same.*

```
# A numeric vector of fictitious data
num_vec <- c(10, 13, -1, 0.02, 0, -3.31)
# A vector of character data
char_vec <- c("joe", "phil", "jan", "denise", "tom")
# A vector of boolean values
bool_vec <- c(TRUE, TRUE, FALSE)
# A vector of functions? WHAT IS THIS MADNESS?
func_vec <- c(mean, sum, sd)
# But this does not create a vector of vectors; all vectors are flattened into one vector (it is possib
not_a_vec_of_vecs <- c(c(1,2,3),c(4,54),c(10,2,-6))

# If I want to see the contents of a vector, just type its name into the interpreter
not_a_vec_of_vecs
```

```
## [1]  1  2  3  4 54 10  2 -6
```

When numbers, strings, boolean values, and other similar objects are assigned to a variable, that variable is interpreted as being a vector.

```
im_a_vector <- 5
is.vector(im_a_vector)
```

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

To access the contents of a vector, you can use `[]` notation, like `vec[x]`. `x` identifies the elements of the vector you want. This is called **indexing**.

Some notes about `x`:

- Items in a vector can always be found via an integer. `vec[1]` will get the first element of the vector, and `vec[5]` the fifth. Also, instead of specifying what indices you *do* want, you can specify the indices you *don't* want with negative integers. For example, `vec[-1]` is `vec` with all *except for* `vec[1]`. `vec[0]` is an empty vector that is of the same type as `vec`.
- Some vectors have named elements. In that case, you can access elements of the vector by name, using a character string. For example, if `cities` is a vector of city populations which is indexed with city names, `cities["Salt Lake City"]` will find the population of Salt Lake City in the vector. If you

want to see the names of the elements, use the `names()` function. (Not surprisingly, the object returned by `names()` is also a vector, specifically a character vector.)

- `x` can be another vector, and thus you can index a vector with another vector so long as the values contained in `x` are a valid means of indexing. We could index a vector with `vec[c(1,2,3)]` or `cities[c("Salt Lake City", "Provo")]`.
- `x` could be a vector of booleans. Every `TRUE` in `x` will lead to the element in `vec` corresponding to where the `TRUE` is located in `x` will be included, and every element in `vec` where `x` is `FALSE` will not be included. (While this implies that `x` must be the same length as `vec`, this is not necessarily true; if `x` is shorter, the contents of `x` will be recycled until R has made a decition for each element of `vec` whether to include it or not. I discuss recycling more later.)
- `vec[]` will return the entire vector `vec`. This is sometimes useful.

```
# You can assign names to elements using name = value notation in c()
friendly_vector <- c(jon = 1, tony = 4, janet = 5)
names(friendly_vector)
```

```
## [1] "jon"   "tony"  "janet"
```

```
friendly_vector[1]
```

```
## jon
##   1
```

```
friendly_vector[c(2,3)]
```

```
##  tony janet
##     4     5
```

```
friendly_vector[-2]
```

```
##   jon janet
##     1     5
```

```
friendly_vector["tony"]
```

```
## tony
##    4
```

```
friendly_vector[c("jon", "tony")]
```

```
##  jon tony
##    1    4
```

```
friendly_vector[c(TRUE, TRUE, FALSE)]
```

```
##  jon tony
##    1    4
```

```
# You can rename elements in a vector like so:
names(friendly_vector) <- c("jack", "jill", "dick")
friendly_vector["jack"]
```

```
## jack
##    1
```

You can also use indexing to change the values of a vector, or even expand it. `vec[x] <- val` will replace the contents of `vec` at `x` with `val` if `val` is the same type as the rest of the data in `vec` and `x` is any valid means of indexing `vec`. `x` could consist of indices that do not already exist in `vec`, in which case those indices will be added to `vec`, thus expanding it. If `x` consists of integer indices outside the range of existing integer indices, these other indices will be created as well and filled with `NA`'s.

You can also delete values from the vector with `vec[x] <- NULL`

```
# friendly_vector was defined in an earlier code block
# Change existing values
friendly_vector["jack"] <- 16
friendly_vector[2] <- 19
friendly_vector
```

```
## jack jill dick
##   16   19    5
```

```
# Adding new values
friendly_vector[4] <- 21
friendly_vector
```

```
## jack jill dick
##   16   19    5   21
```

```
friendly_vector["danielle"] <- 0
friendly_vector
```

```
##     jack     jill     dick          danielle
##       16       19        5       21        0
```

```
friendly_vector[c(6, 7, 8)] <- 12
friendly_vector
```

```
##     jack     jill     dick          danielle
##       16       19        5       21        0       12       12       12
```

```
friendly_vector[20] <- 18
```

While vectors must contain data of one type, there is a special type that can be included in any vector: `NA`. This value represents "missing information". `NA` isn't like other values and needs to be handled with care. The function `is.na()` identifies these values in vectors.

```
not_finished <- c(1, 4, 5, NA, 2, 2)
not_finished
```

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

```
# If I want to access the non-NA parts of the vector, I can do so like this
not_finished[!is.na(not_finished)]
```

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

There are other special types in R resembling but dfferent from `NA`. `NULL` is a lot like `NA` but usually means that something in R is unavailable (whereas `NA` is more akin to missing data in a dataset). `Inf` and `-Inf` are special values denoting infinity and negative infinity respectively. These are, in some sense, numeric, and represent values that are very large (or very small, in the case of `-Inf`), and can occur when dividing non-zero numbers by zero. `NaN` effectively means "not a number", and occurs when some numerical error occurs, like dividing zero by zero. Again, these cases must be handled with care, and there are special functions like `is.na()` for detecting them in vectors.

There are other ways to create vectors other than with the `c()` function. Some common ways are listed below:

- The `:` constructor can be used to create sequences. For example, `1:10` will create a vector of numbers from 1 to 10, incrementing by 1. `10:0` creates a vector starting at 10 and ending at 0, decrementing by 1. You can also replace the endpoints with variables, or expressions in parentheses, like `1:n` or `1:(2 + 2)`.

- Sometimes you may want to create a sequence but want more control over incrementation, or how many elements in the vector you want. In this case, use the `seq()` function. You can make a sequence of numbers incrementing by 2 going from 1 to 100 with `seq(1, 100, by = 2)`, or a sequence of numbers between 0 and 1 with length 100 with `seq(0, 1, length = 100)`. (See `help("seq")` to see all the many ways to create sequences with `seq()`.)
- The `rep()` function can make vectors with repeating elements. Let's say I want to repeat the character values ""a", "b", and "c" three times total. I can do so with `rep(c("a", "b", "c"), times = 3)`. Alternatively, if I wanted to repeat these values *each* three times, I would do so with `rep(c("a", "b", "c"), each = 3)`. (See `help("rep")` to see all the many ways to create sequences with `rep()`.)
- Sometimes I want to create character vectors where I have pasted together strings from separate vectors. For example, if I want a character vector containing the names of one hundred treatments, it may be tedious to type `c("Treatment 1", "Treatment 2", ..., "Treatment 100")`. The `paste()` function makes this much easier. I can create such a vector with `paste("Treatment", 1:100)`; each of the elements from both vectors will be pasted together into a new vector. By default, these elements will be separated with a space character, but I can change this behavior by specifying the `sep` parameter in `paste()`. For example, if I want `"Treatment_1"` rather than `"Treatment 1"`, I can do so with `paste("Treatment", 1:100, sep = "_")`.

I demonstrate these techniques below.

```
# Create a vector of numbers from 1 to 10
1:10
```

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

```
# In reverse
10:1
```

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

```
# Getting creative
1:(2 + 2)
```

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

```
# Another way to make a sequence
seq(-1, 1, by = 0.5)
```

```
## [1] -1.0 -0.5  0.0  0.5  1.0
```

```
# Yet another way to make a sequence
seq(0, 20, length = 3)
```

```
## [1]  0 10 20
```

```
# A repeating sequence of letters
rep(c("a", "b", "c"), times = 3)
```

```
## [1] "a" "b" "c" "a" "b" "c" "a" "b" "c"
```

```
# A sequence of repeating letters
rep(c("a", "b", "c"), each = 3)
```

```
## [1] "a" "a" "a" "b" "b" "b" "c" "c" "c"
```

```
# A quick way to make a character vector
paste("Treatment", 1:5)
```

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

You can do mathematical operations with vectors, such as `+`, `-`, `*`, `/`, `^`, and others. Operations are often applied component-wise, using R's recycling behavior. **Recycling** occurs when two vectors are not of equal

length. Let's say you have a vector `vec1` that is longer than `vec2`, and you do an operation such as `vec1 + vec2`. Let's say `length(vec1) == 12` and `length(vec2) == 4`. At first, the resulting vector will add, component-wise, each element from each vector; think `vec1[1] + vec2[1]`, `vec1[2] + vec2[2]`, `vec1[3] + vec2[3]`, and `vec1[4] + vec2[4]`. After the fourth index, though, there are no more elements in `vec2`, so R will start from the beginning of `vec2` and keep going, resulting in `vec1[5] + vec2[1]`, `vec1[6] + vec2[2]`, and so on. R will continue to do this until it has reached the end of `vec1`. You would get the same result for `vec2 + vec1`; it doesn't matter which side of `+` has the longer vector. If the longer vector is not a multiple of the shorter vector, though, R will throw a warning message.

```
vec1 <- c(0, 0, 10, -1, 5, 6)
# R treats "1" as a vector of length 1, and thus recycles
vec1 + 1
```

```
## [1]  1  1 11  0  6  7
```

```
vec1 * 2
```

```
## [1]  0  0 20 -2 10 12
```

```
vec1 ^ 2
```

```
## [1]   0   0 100   1  25  36
```

```
vec2 <- 1:2
# Another case of recycling behavior
vec1 + vec2
```

```
## [1]  1  2 11  1  6  8
```

```
# Same result if I switch the order
vec2 + vec1
```

```
## [1]  1  2 11  1  6  8
```

```
# R will complain if the length of the longer vector is not a multiple of the length of the shorter obj
vec1 / 1:4
```

```
## Warning in vec1/1:4: longer object length is not a multiple of shorter
## object length
```

```
## [1]  0.000000  0.000000  3.333333 -0.250000  5.000000  3.000000
```

```
vec1 ^ vec2
```

```
## [1]  0  0 10  1  5 36
```

```
# Does NOT get same result because ^ is not commutative
vec2 ^ vec1
```

```
## [1]  1.0  1.0  1.0  0.5  1.0 64.0
```

Of course, when working with vectors of the same length, recycling isn't a concern.

```
# These vectors are the same length, so no need to worry about recycling
vec1 <- 1:3
vec2 <- 10:12
vec1 + vec2
```

```
## [1] 11 13 15
```

```
vec1 * vec2
```

```
## [1] 10 22 36
```

```
vec1 ^ vec2
```

```
## [1]      1   2048 531441
```

```
# That said, it's not hard to guess what will happen when one of the objects is length one (like a vect
(1:10) ^ 2 # First ten squares
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

```
2 ^ (1:10) # First ten powers of two
```

```
##  [1]    2    4    8   16   32   64  128  256  512 1024
```

Recycling occurs elsewhere in R. We saw recycling behavior when indexing a vector with a vector of booleans earlier. There are other instances in R as well.

Some functions are vector-valued. For example, when passed a vector, `sqrt()` will take the square root of every element in the vector.

```
vec <- 1:5
sqrt(vec)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

```
exp(vec)
```

```
## [1]   2.718282   7.389056  20.085537  54.598150 148.413159
```

```
vec[3] <- NA
# Creates a vector of booleans
is.na(vec)
```

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

Another important set of operators are logical operators, which return boolean data. Such operators include:

- `==`: This detects equality (notice that this is *not* `=`, which is an assignment operator). If or when the object on the left equals the object on the right, the result will be `TRUE`; otherwise, it will be `FALSE`. For vectors, this does not return whether the two vectors are identical, but when one vector is equal to the other component-wise.
- `<` and `>`: These detects "less" or "greater than", like in mathematics. This is intended for numeric data, but can be used for other types of data as well (although rarely, and probably not well).
- `<=` and `>=`: These detect "less than or equal to" or "greater than or equal to".
- `!=`: This detects "not equal to", and is the opposite of `==`.
- `&`: This is logical "and", and is true when the boolean or statment on the left is true *and* the boolean or statement on the right is true. Thus, `(1 == 1) & (2 >= 1) == TRUE` and `(1 == 2) & (2 >= 1) == FALSE`.
- `|`: This is logical "or", and is true when the boolean or statement on the left is true *or* the boolean or statement on the right is true. Thus, `(1 == 1) | (2 >= 1) == TRUE` and `(1 == 2) | (2 >= 1) == TRUE`.
- `!`: This is logical "not", negating any truth statement. So `!TRUE == FALSE` and `!(1 == 2) == TRUE`.
- `%in%`: This logical operator is unique compared to the others considered here, since this is actually a function. The argument on the right-hand side of this operator must be a vector, and this operator whether elements on the left-hand side are in the vector on the right-hand side (in logic, this is $x \in S$ where S is a set). Thus, `3 %in% c(1,2,3) == TRUE` and `3 %in% c(1, 2) == FALSE`.

Like other operators, these utilize recycling and may return vectors. Examples are shown below.

```
vec1 <- c(1, 4, 21, 22, -5)
vec2 <- c(1, 2, 3, 4, -5)
```

```
# True only in the first and last positions
vec1 == vec2
```

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

```
vec1 < 4  # True for first and last elements
```

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

```
vec2 <= 4  # True in first, second, and last elements
```

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

```
!(vec1 <= 4)  # Inverse of above
```

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

```
# %% is the modulus operator, returning the remainder when the number on the
# left is divided by the number on the right. vec1 %% 2 == 0 is a way to
# detect even numbers (since the remainder when dividing an even number by 2
# must be zero).
(vec1 > 20) & !(vec1%%2 == 0)  # Only true in third position
```

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

```
(vec1 > 20) | !(vec1%%2 == 0)  # Not true in second or fourth
```

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

```
1:4 %in% vec1  # One and four are in vec1
```

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

```
# Some useful functions are the any() and all() functions, which take
# boolean vectors as arguments and return whether anywhere the vector is
# true or whether the vector is true everywhere, respectively. In other
# words, any() will 'or' all elements of the vector, and all() will 'and'
# all elements in the vector.
any(1:4 %in% vec1)  # Is there a number between 1 and 4 in vec1?
```

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

```
all(1:4 %in% vec1)  # Are all numbers between 1 and 4 in vec1?
```

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

Many other functions return logical data, notably the `is.object()` family of functions like `is.vector()` or `is.na()`.

While a vector of booleans for when a condition is true is nice, sometimes you may not want whether a statement is true for each element of a vector, but for which elements of a vector a statement is true. In this case, the `which()` function will tell you the indices of where an input vector is `TRUE`.

```
which(c(TRUE, FALSE, FALSE, TRUE, FALSE))
```

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

```
which(15:25 > 20)
```

```
## [1]  7  8  9 10 11
```

```
# I'm going to create a dataset with NA's, and use which() to find the NA's
# and also the 'good' data
```

```
data_vec <- 1:100
data_vec[c(21, 33, 49, 61, 62)] <- NA
which(is.na(data_vec))  # Where are the NA's
```

```
## [1] 21 33 49 61 62
```

```
which(!is.na(data_vec))  # Where is the non-NA data?
```

```
##  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
## [18]  18  19  20  22  23  24  25  26  27  28  29  30  31  32  34  35  36
## [35]  37  38  39  40  41  42  43  44  45  46  47  48  50  51  52  53  54
## [52]  55  56  57  58  59  60  63  64  65  66  67  68  69  70  71  72  73
## [69]  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
## [86]  91  92  93  94  95  96  97  98  99 100
```

```
# How much data is NA?
length(which(is.na(data_vec)))
```

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

```
# Alternatively, I can sum a boolean vector to find the same answer (FALSE
# is treated as 0 and TRUE as 1)
sum(is.na(data_vec))
```

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

A special type of vector not discussed earlier is a **factor** vector, which is similar to a character vector but requires that each value of the vector be in a list of **levels**, which describe the values a factor vector can take. (R also views vactors differently internally than character vectors.) Factor vectors are thus used for storing categorical data. You can create a factor with the `factor()` function.

When manipulating factor data, there is an additional restriction to the ones discussed already: you cannot add a value that is not in the levels of the factor (aside from `NA`). You would have to add the value to the levels of the factor first, then add the value. Also, beware that changing the levels will also change the values stored in the factor. You can set the levels of a factor with `levels(x) <- y`, where `x` is the factor vector and `y` a vector representing the new levels of the factor.

```
# Create a factor of color data
color_data <- c("blue", "red", "blue", "blue", "blue", "red", "red", "blue")
# Create the factor, declaring the levels; notice that I included a category
# that is not in the data vector
colcat1 <- factor(color_data, levels = c("red", "green", "blue"))
colcat1
```

```
## [1] blue red  blue blue blue red  red  blue
## Levels: red green blue
```

```
# If I do not declare the levels, R will use the values stored in the vector
# to guess what they are
colcat2 <- factor(color_data)
# Now I change data; for the first, no complaint if I add 'green'
colcat1[1] <- "green"
colcat1
```

```
## [1] green red   blue  blue  blue  red   red   blue
## Levels: red green blue
```

```
# But 'green' is not in the levels of colcat2, so an warning is issued and
# NA is added instead
colcat2[1] <- "green"
```

```
## Warning in `[<-.factor`(`*tmp*`, 1, value = "green"): invalid factor level,
## NA generated
```

```
colcat2
```

```
## [1] <NA> red  blue blue blue red  red  blue
## Levels: blue red
```

```
# I can see the levels of these with levels()
levels(colcat1)
```

```
## [1] "red"   "green" "blue"
```

```
levels(colcat2)
```

```
## [1] "blue" "red"
```

```
# I can rearrange all the categories by changing the levels
levels(colcat1) <- c("blue", "red", "green")
colcat1
```

```
## [1] red   blue  green green green blue  blue  green
## Levels: blue red green
```

```
# Here I add a new level to those specified for colcat2
levels(colcat2)[3] <- "green"
# Now I can add 'green' to colcat2 without complaint
colcat2[1] <- "green"
colcat2
```

```
## [1] green red   blue  blue  blue  red   red   blue
## Levels: blue red green
```

Functions

Functions are one of the most important objects in R. In fact, R follows a programming paradigm called **functional programming**, where most operations are seen as the evaluation of functions. You can think of **functions** as miniature programs for performing certain tasks.

In R, functions have two parts:

- **Arguments** are the values passed to the function for evaluation. In the statement `f(x, y)`, the variables between the parentheses, `x` and `y`, are the arguments of the function. A function will typically expect an argument to be of a particular type and may throw an error when that type is not received, but the expected type could be anything from a vector to a data frame to even another function. Some functions have many arguments but have default values assigned to most of them so that the user specifies those arguments only if they wish to change the function's default behavior.
- The **body** of a function is the part of the function that performs computations (involving the parameters) and possibly returning an output. The output of the function is either the last value computed (and not assigned to a variable) or a value returned by the `return()` function.

To create a named function, use the syntax `my_func <- function(arguments) { function body }`. Below I create a function.

```
# Let's create a function that returns the length of the longest vector
# passed to it. It will take two vectors x and y as arguments, and return
# the length of the longer vector
longest_length <- function(x, y) {
```

```
    l1 <- length(x)  # This is a local variable that will be visible only in the function, not the rest
    l2 <- length(y)
    max(l1, l2)  # This is the last unevaluated computation and thus is returned by the function
}
# Note that longest_length is a variable storing a function, and
# longest_length() is a function call

# Testing it out
vec1 <- c("bob", "jim", "margaret", "danny")
vec2 <- c(22, -9)
longest_length(vec1, vec2)
```

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

```
# Functions will check the position of objects passed and assign them to the
# arguments in the same position in the function definition. Alternatively
# (and very useful when there are many arguments not all of which are
# specified), you can use = to set specific arguments by name.
longest_length(y = vec2, x = vec1)
```

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

Functions are extremely important objects in R and the key to R programming. Appendix A of the Verzani textbook discusses function programming in more detail.

Visually Exploring a Dataset

R has many techniques built-in for visually analyzing datasets, and many packages that do so even better, such as the very popular **ggplot2** package (I used **ggplot2** for all the graphics for my first report on Utah's gender gap in wages written for Voices for Utah Children, which you can read here). Here I will discuss how to make a few basic graphics for visually analyzing a dataset.

Stem-and-leaf plot

Use the `stem()` function to create a stem-and-leaf plot.

```
stem(rivers)
```

```
##
##   The decimal point is 2 digit(s) to the right of the |
##
##    0 | 4
##    2 | 011223334555566667778888899900001111223333344455555666688888999
##    4 | 1112223334455667790012333445 67
##    6 | 000112233578012234468
##    8 | 045790018
##   10 | 04507
##   12 | 1471
##   14 | 56
##   16 | 7
##   18 | 9
##   20 |
##   22 | 25
##   24 | 3
```

```
##   26 |
##   28 |
##   30 |
##   32 |
##   34 |
##   36 | 1
```

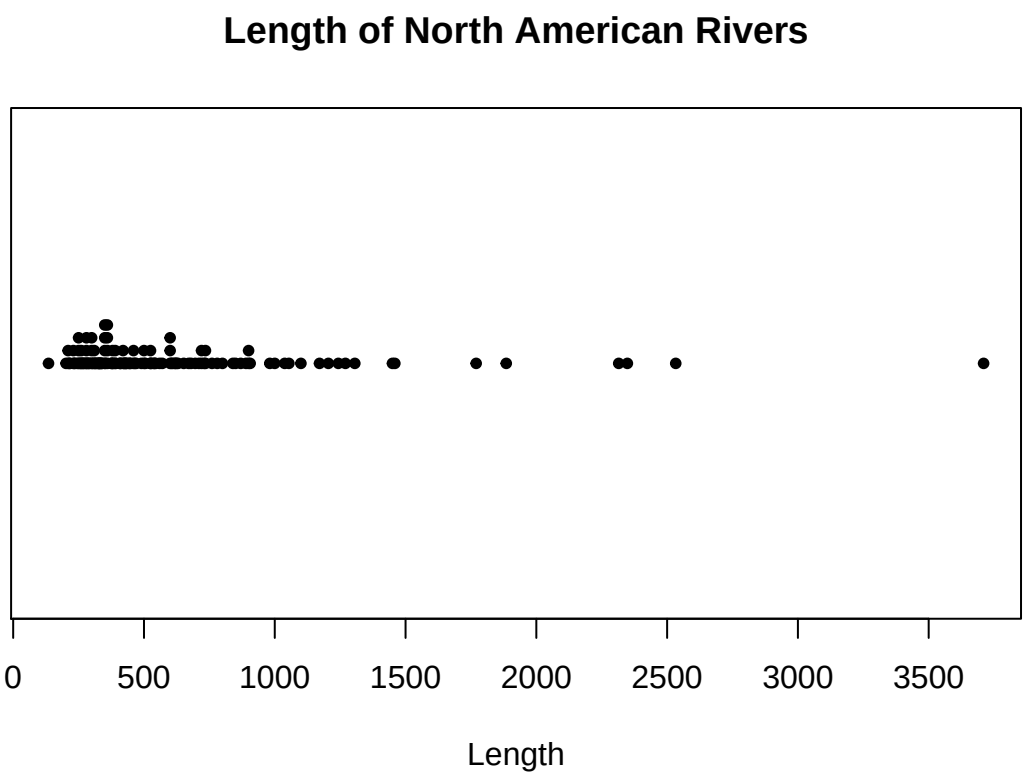
The display suggests that the `rivers` dataset is very right-skewed. Let's see if this agrees with other plots.

Dotplot

You can use `stripchart()` to make dotplots. The default behavior of this function doesn't produce a very useful plot (truth be told, plotting functions in other packages make better dotplots in general than `stripchart()`), so set `method = "stack"` to enable stacking. I also set `pch = 20` to change the plotting character used from the default square to a filled-in circle, which is easier to read.

Many R plotting functions have parameters `main`, `xlab`, and `ylab`. These are so you set the title of the plot, and the names of the labels. I do so in the plot I create as well.

```
# Make a dotplot of the rivers data
stripchart(rivers, method="stack", pch = 20,
           # Adding axis labels
           main = "Length of North American Rivers",
           xlab = "Length")
```

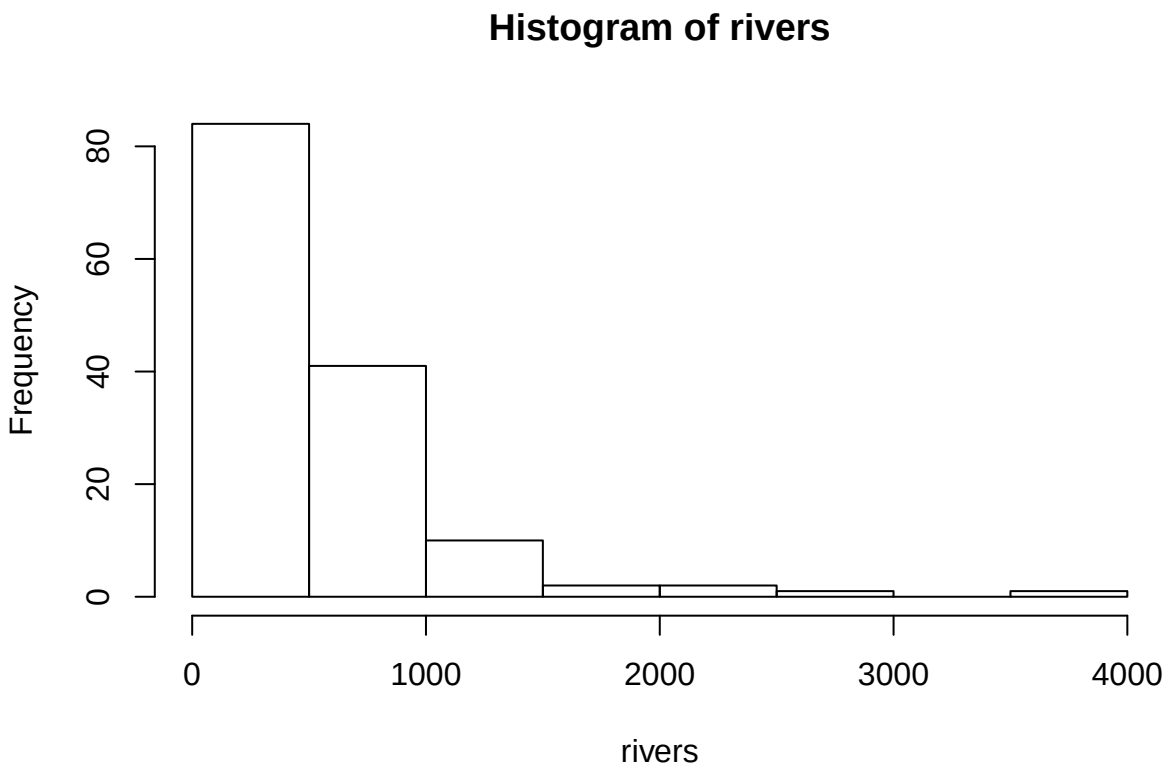


The `rivers` dataset is slightly large, with 141 observations. The plots used so far may not result in very good graphics for datasets like this. I next consider graphics that may handle larger datasets better.

Histogram

The `hist()` function creates histograms in R. Calling `hist(x)` for some dataset `x` will create a histogram R thinks is appropriate. R will automatically choose classes and the number of classes to used based on built-in algorithms. I show an example below:

```
hist(rivers)
```



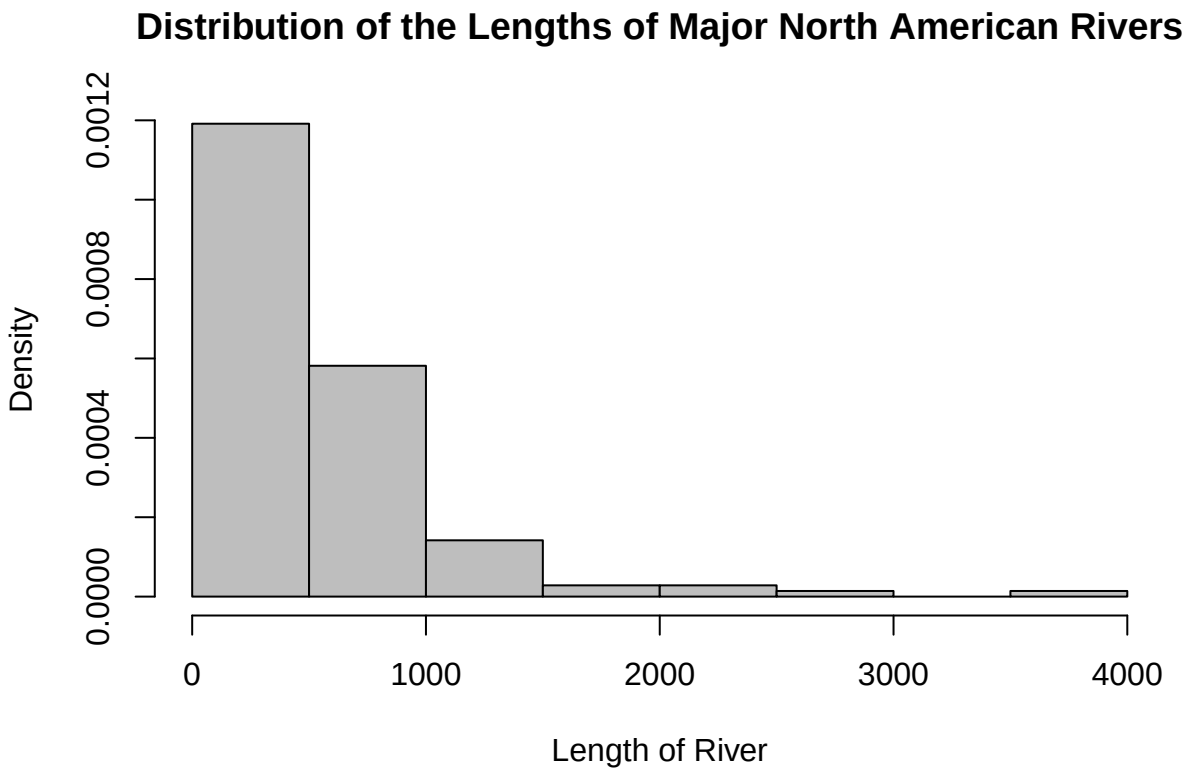
We can change the parameters of the `hist()` function to have more control over the result. For example:

- R automatically chooses axis names and the main title, which usually are not very good names. We can change these defaults to reasonable labels by setting the `main`, `xlab`, and `ylab` parameters.
- By default, R will plot the frequency rather than the relative frequency of each class. The result is indistinguishable until you wish to overlay a histogram with a smooth curve or otherwise be more creative with the chart. Set `freq = FALSE` to show relative frequencies, or probabilities.
- R creates left-inclusive histograms by default. If we want right-inclusive histograms, set `right = TRUE`.
- The `breaks` parameters is used for setting where the break points are located. If we set `breaks` to an integer, this will tell R how many classes to use. For example, if we want $\sqrt{n}$ classes for the `rivers` dataset, we can do so by setting `breaks = round(sqrt(length(x)))`. (That said, the method R uses for determining how many classes to use is usually better than the $\sqrt{n}$ rule.)
- Do we really like white bars in our histogram? The `col` parameter can change their color. For example, set `col = "gray"` for gray bars.

There are many more parameters for the `hist()` function than this; I invite you to read the function's documentation for more details. Here is another histogram for the `rivers` dataset, this one changing some parameters.

```
hist(rivers, main = "Distribution of the Lengths of Major North American Rivers",
     xlab = "Length of River",
     freq = FALSE,
```

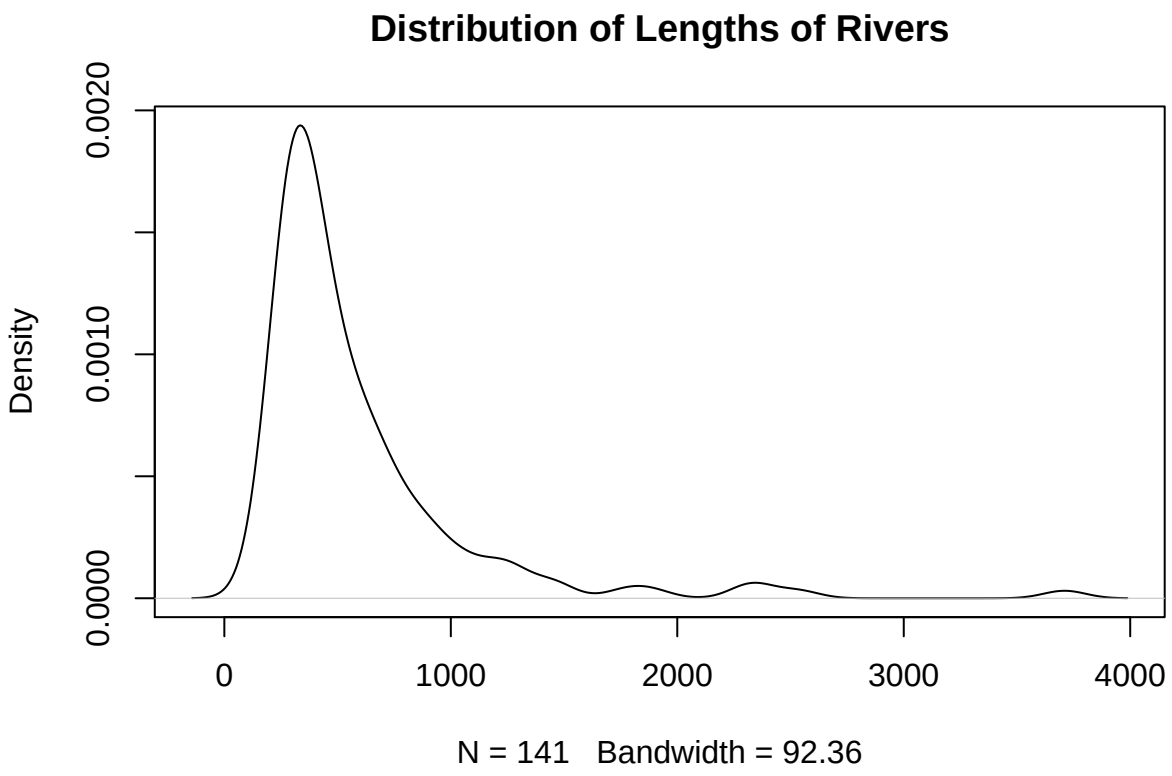
```
    right = TRUE,
    breaks = round(sqrt(length(rivers))), # Using the sqrt(n) rule
    col = "gray")
```



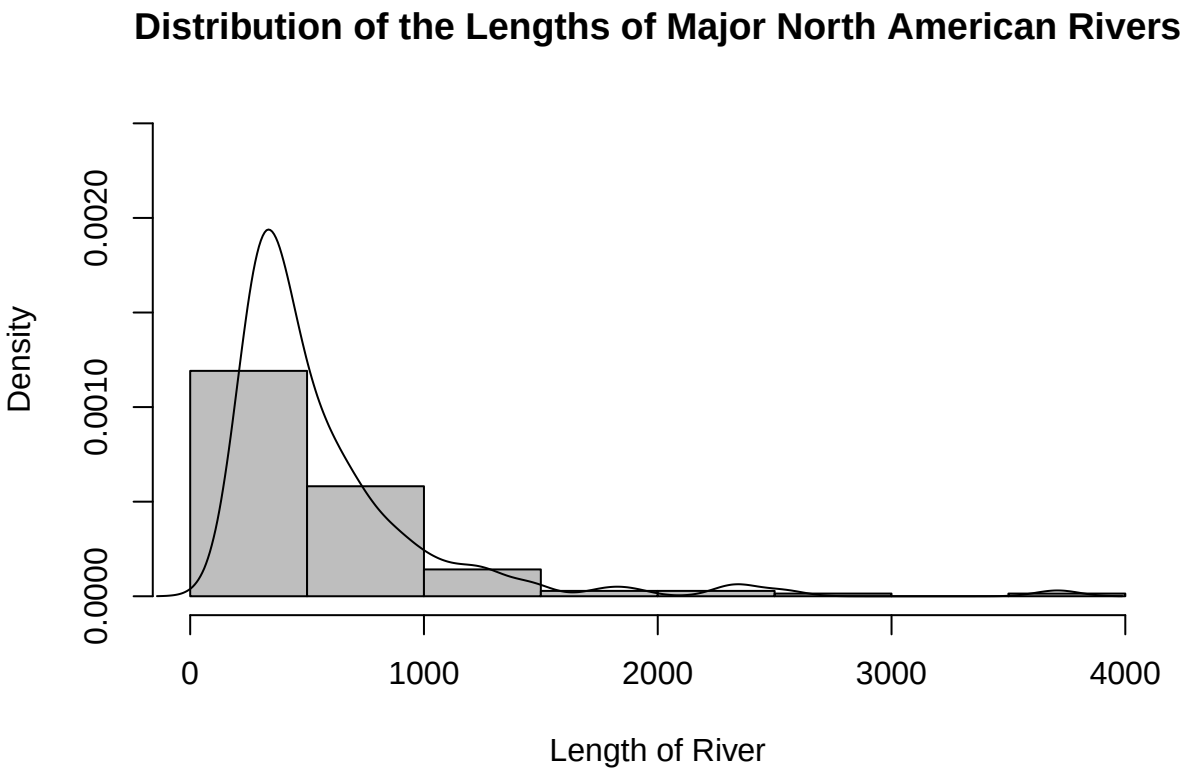
Density Curve

A density curve is another way to view a distribution where the end result is a smooth curve, It can be interpreted like a histogram, but it avoids disadvantages that come with choosing discrete classes. You can create a density curve with `plot(density(x))`.

```
plot(density(rivers), main = "Distribution of Lengths of Rivers")
```



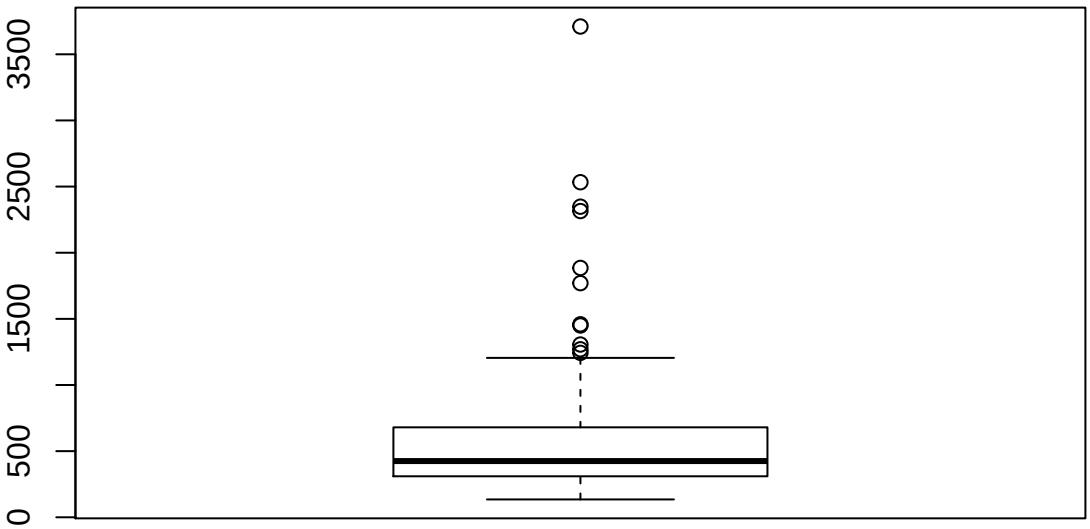

```r
# It is possible to superimpose a density plot on top of a histogram to see the relationship. Just be s
hist(rivers, main = "Distribution of the Lengths of Major North American Rivers",
     xlab = "Length of River",
     freq = FALSE,
     right = TRUE,
     breaks = round(sqrt(length(rivers))), # Using the sqrt(n) rule
     col = "gray",
     ylim = c(0, 0.0025)) # ylim sets the limits of the vertical axis
lines(density(rivers))
```



Boxplot

You can create a boxplot in R with the `boxplot()` function.

```
boxplot(rivers)
```



One of the major advantages of boxplots is being able to compare distributions of different samples. To do so, issue a call to `boxplot()` of the form `boxplot(x ~ y)`, where `x` is a data vector with all samples combined, and `y` is a character or factor vector identifying the sample each element of `x` belongs to.

```
# I will be working with the iris dataset for this example
# Looking at the data
str(iris)
```

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

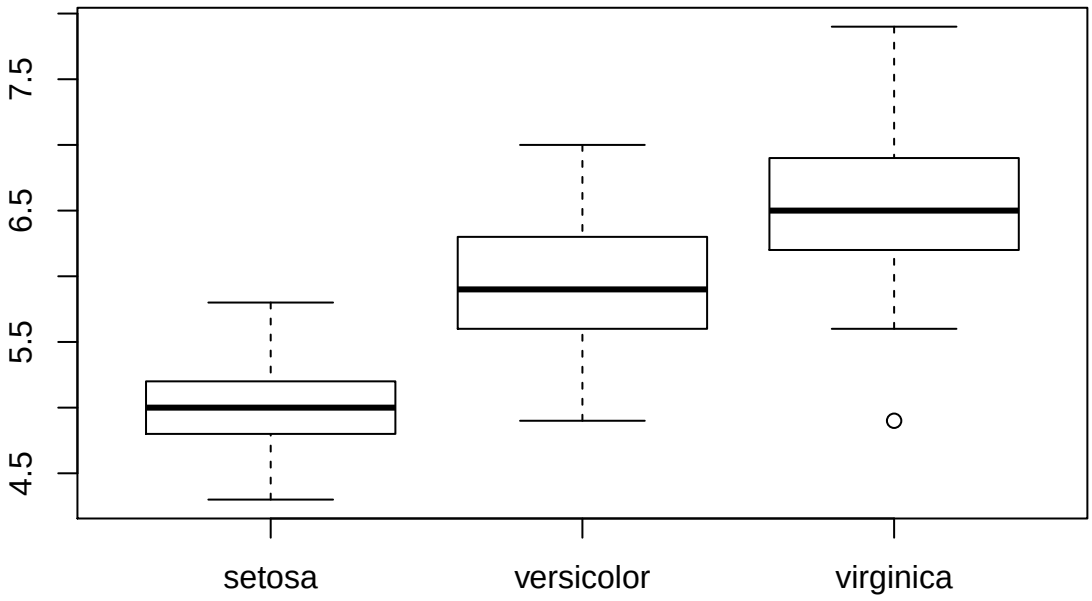
```
iris$Sepal.Length
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
##  [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
##  [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
##  [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8
##  [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4
##  [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

```
iris$Species
```

```
##   [1] setosa     setosa     setosa     setosa     setosa     setosa
##   [7] setosa     setosa     setosa     setosa     setosa     setosa
##  [13] setosa     setosa     setosa     setosa     setosa     setosa
##  [19] setosa     setosa     setosa     setosa     setosa     setosa
##  [25] setosa     setosa     setosa     setosa     setosa     setosa
##  [31] setosa     setosa     setosa     setosa     setosa     setosa
##  [37] setosa     setosa     setosa     setosa     setosa     setosa
##  [43] setosa     setosa     setosa     setosa     setosa     setosa
##  [49] setosa     setosa     versicolor versicolor versicolor versicolor
##  [55] versicolor versicolor versicolor versicolor versicolor versicolor
##  [61] versicolor versicolor versicolor versicolor versicolor versicolor
##  [67] versicolor versicolor versicolor versicolor versicolor versicolor
##  [73] versicolor versicolor versicolor versicolor versicolor versicolor
##  [79] versicolor versicolor versicolor versicolor versicolor versicolor
##  [85] versicolor versicolor versicolor versicolor versicolor versicolor
##  [91] versicolor versicolor versicolor versicolor versicolor versicolor
##  [97] versicolor versicolor versicolor versicolor virginica  virginica
## [103] virginica  virginica  virginica  virginica  virginica  virginica
## [109] virginica  virginica  virginica  virginica  virginica  virginica
## [115] virginica  virginica  virginica  virginica  virginica  virginica
## [121] virginica  virginica  virginica  virginica  virginica  virginica
## [127] virginica  virginica  virginica  virginica  virginica  virginica
## [133] virginica  virginica  virginica  virginica  virginica  virginica
## [139] virginica  virginica  virginica  virginica  virginica  virginica
## [145] virginica  virginica  virginica  virginica  virginica  virginica
## Levels: setosa versicolor virginica
```

```
boxplot(iris$Sepal.Length ~ iris$Species)
```



Bar Chart

For a categorical dataset, we can create a bar chart using the `barplot()` function, along with the `summary()` function. This is a natural way to visualize categorical data.

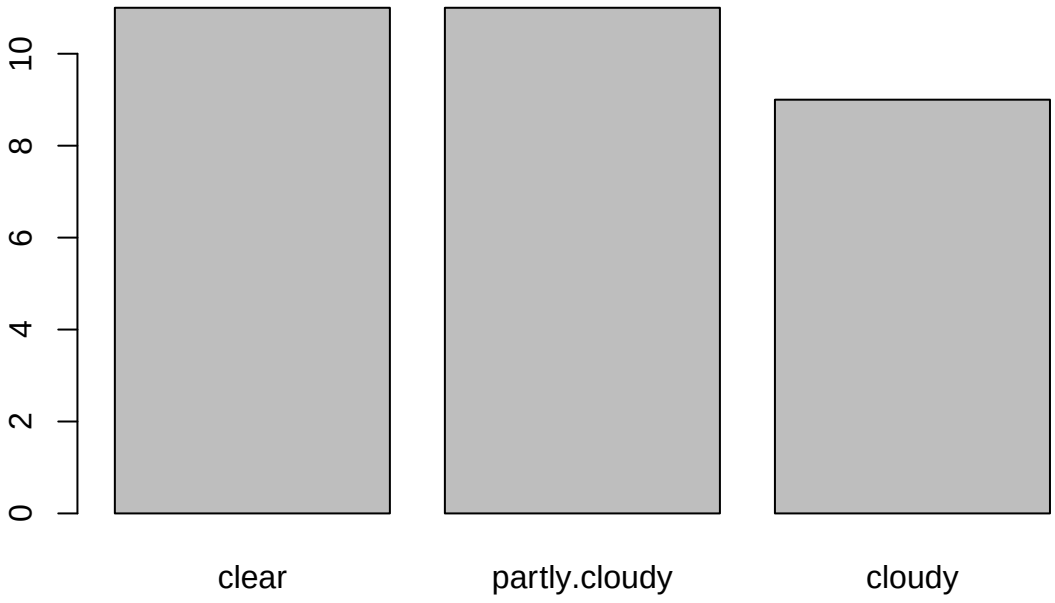
```
library(UsingR)
central.park.cloud  # Looking at the data
```

```
##  [1] partly.cloudy partly.cloudy partly.cloudy clear         partly.cloudy
##  [6] partly.cloudy clear         cloudy        partly.cloudy clear
## [11] cloudy        partly.cloudy cloudy        cloudy        clear
## [16] partly.cloudy partly.cloudy clear         clear         clear
## [21] clear         cloudy        cloudy        cloudy        cloudy
## [26] cloudy        clear         partly.cloudy clear         clear
## [31] partly.cloudy
## Levels: clear partly.cloudy cloudy
```

```
# For this object (a factor vector), summary will give the counts of each
# category.
summary(central.park.cloud)
```

```
##         clear partly.cloudy        cloudy
##            11            11             9
```

```
# Shows the cloudiness of different days in Central Park
barplot(summary(central.park.cloud))
```



Numerical Summaries

A numerical summary tries to describe some aspect of a dataset using numbers. Two classes of numerical summaries include measures of location and measures of spread. There are many other numerical summaries (the textbook used in this lab describes measures of skewness and kurtosis in addition to measures of location and spread), but we will focus on these two.

First, a great way to obtain appropriate numerical summaries for a dataset is with the `summary()` function. This is a generic function that changes its behavior depending on the object passed to it. If `x` is a numeric vector, `summary(x)` will compute the **five-number summary** of `x` (minimum, first quartile, median, third quartile, maximum) in addition to the mean of `x`. On the other hand, if `x` is a factor or character vector, `summary(x)` will compute the frequency of each category in `x`. Thus, if you have a dataset you are unfamiliar with, `summary()` is a good way to see some basic information about it.

```
# Some basic information about rivers, a numeric dataset
summary(rivers)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   135.0   310.0   425.0   591.2   680.0  3710.0
```

```
# Frequencies for cloud conditions in central.park.cloud, a factor vector
summary(central.park.cloud)
```

```
##         clear partly.cloudy        cloudy
##            11            11             9
```

Of course it is possible to compute specific numeric summaries.

The **mean** of a dataset is defined as:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

where x_i is the i^{th} observation and n the size of the dataset. The mean can be computed in R using the `mean()` function. To obtain the trimmed mean, you can set the `trim` parameter to a value between `0` and `0.5`; this sets how much of the data to "trim" from either end of the ordered dataset. If `trim = 0` (the

default), no data is trimmed from either end, but if `trim = 0.5`, as much data is possible is trimmed from either end and the median is computed.

```
# The mean executive pay (in $10,000's)
mean(exec.pay)
```

```
## [1] 59.88945
```

```
# There are very large outliers, though; what happens to the mean if we trim
# 10% of the data from either side of the (ordered) dataset?
mean(exec.pay, trim = 0.1)
```

```
## [1] 29.96894
```

The **median** is the number that splits the dataset in half after being ordered. You can compute the median in R using the `median()` function.

```
# The median executive pay; compare to the mean or trimmed mean
median(exec.pay)
```

```
## [1] 27
```

The $(100\alpha)^{\text{th}}$ **percentile** is the number such that $100\alpha\%$ of the dataset lies below and $100(1-\alpha)\%$ above the number. The `quantile()` function in R computes percentiles (also referred to as quantiles). `quantile(x)` will effectively find the five-number summary of the dataset `x`, including the first and third quartiles. Alternatively, one can call `quantile(x, p)`, where `p` is a vector (or perhaps just a number that will be interpreted as a vector) specifying with percentiles are wanted (these are numbers between 0 and 1).

The `fivenum()` function will find five-number summaries outright, but one may as well call `quantile()` with the default parameters (the presentation is better anyway).

```
# The five-number summary using fivenum
fivenum(exec.pay)
```

```
## [1]    0.0   14.0   27.0   41.5 2510.0
```

```
# The same information using quantile
quantile(exec.pay)
```

```
##     0%    25%    50%    75%   100%
##    0.0   14.0   27.0   41.5 2510.0
```

```
# What is the 99th percentile of executive pay?
quantile(exec.pay, .99)
```

```
##    99%
## 906.62
```

```
# The deciles of exec.pay, breaking the dataset into 10 parts
quantile(exec.pay, seq(0, 1, by = .1))
```

```
##     0%    10%    20%    30%    40%    50%    60%    70%    80%    90%
##    0.0    9.0   12.6   16.0   22.0   27.0   31.0   38.0   48.0   91.4
##   100%
## 2510.0
```

Now let's discuss measures of spread. The **range** of a dataset is the difference between the largest and smallest values:

$$\text{range} = \max_i x_i - \min_i x_i$$

R's `range()` function will find the maximum and minimum of the dataset, but won't difference them. This is not a problem, though; simply call `diff()` on the result of range to subtract the minimum from the maximum, like `diff(range(x))`, where `x` is the dataset.

```
# The largest and smallest executive pay
range(exec.pay)
```

```
## [1]    0 2510
```

```
# The range
diff(range(exec.pay))
```

```
## [1] 2510
```

Another (more common) means for numerically describing the spread of a dataset is with the **standard deviation** or its square, the **variance**. The variance is defined as follows:

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

The standard deviation is merely the square root of the variance.

$$s = \sqrt{s^2}$$

In R, the function `var()` finds a dataset's variance, and `sd()` finds the standard deviation.

```
# The variance of exec.pay
var(exec.pay)
```

```
## [1] 42867.03
```

```
# We could take the square root of the variance to get the standard
# deviation
sqrt(var(exec.pay))
```

```
## [1] 207.0435
```

```
# Or we could just use sd
sd(exec.pay)
```

```
## [1] 207.0435
```

For categorical data, the simplest way to numerically summarize the data is with a table. We can create one with the `table()` function

```
table(central.park.cloud)
```

```
## central.park.cloud
##         clear partly.cloudy        cloudy
##            11            11             9
```

```
# Create a frequency table by dividing a table by the sample size (i.e. the
# length of the data vector)
table(central.park.cloud)/length(central.park.cloud)
```

```
## central.park.cloud
##         clear partly.cloudy        cloudy
##     0.3548387     0.3548387     0.2903226
```

Lecture 3

R Data Structures

As mentioned before, vectors are hardly the only data structure in R. There are other very important data structures R uses.

Lists

A **list** is a generalized vector in R. An R vector requres that all data saved stored in the vector be of the same type. A list has no such requirement. You can easily create lists with numbers, strings, vectors, functions, and other lists all in one object. Lists in R are created with the `list()` function, where each element of the list is separated with a `,` (note that lists don't flatten vectors like `c()` does; every item separated by a comma gets its own index in the list).

```
# Let's make a list of mixed type!
l1 <- list(1, "fraggle rock", c("henry", "margaret", "donna"), list(1:2, paste("Test",
    1:10)))
l1
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "fraggle rock"
##
## [[3]]
## [1] "henry"    "margaret" "donna"
##
## [[4]]
## [[4]][[1]]
## [1] 1 2
##
## [[4]][[2]]
##  [1] "Test 1"  "Test 2"  "Test 3"  "Test 4"  "Test 5"  "Test 6"  "Test 7"
##  [8] "Test 8"  "Test 9"  "Test 10"
```

```
# This list has no names for its elements; we could specify some using
# names()
names(l1) <- c("num", "char", "vec", "inner_list")
# We can also assign names when we create the list
l2 <- list(char = "monday", vec = c("and", "but", "or"))
l2
```

```
## $char
## [1] "monday"
##
## $vec
## [1] "and" "but" "or"
```

How do we reference the objects stored in a list? We have a few options:

- If we wish that the object returned by the reference also be a list, we can use single-bracket notation like we did with vectors, like `li[x]` where `x` is any means for selecting elements of the list (number, string, vector, boolean vector, etc.).
- If we *the object stored at x*, we can use double bracket notation, like `l1[[x]]` where `x` is either a number or a string (`x` cannot be a vector in this case). The difference between `l1[x]` and `li[[x]]` may be subtle, but it's very important. `li[x]` *is a list, and* `l1[[x]]` *is an object stored in a list.* (This difference is also true for vectors; `vec[x]` is a vector, and `vec[[x]]` is an object stored in a vector. Rarely does this make a difference, but sometimes it does, like when the vector is a vector of functions.)
- If the elements of the list are named, instead of referencing them with `l1[["x"]]` (`x` is the name of the element), we can use `$` notation, like `l1$x`. This is usually how named elements are referenced.

```
# This is a list
l1[1:3]
```

```
## $num
## [1] 1
##
## $char
## [1] "fraggle rock"
##
## $vec
## [1] "henry"    "margaret" "donna"
```

```
is.list(l1[1:3])
```

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

```
# This is item stored in the third position of the list
l1[[3]]
```

```
## [1] "henry"    "margaret" "donna"
```

```
# This is not a list
is.list(l1[[3]])
```

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

```
# Notice the difference
l1[3]
```

```
## $vec
## [1] "henry"    "margaret" "donna"
```

```
# We can also reference by name
l2["vec"]
```

```
## $vec
## [1] "and" "but" "or"
```

```
l2[["vec"]]
```

```
## [1] "and" "but" "or"
```

```
# An alternative way to reference the contents of an element by name
l2$vec
```

```
## [1] "and" "but" "or"
```

More complex objects in R are often simply lists with a specific structure, thus making lists very important.

Matrices

An R **matrix** is much like an R vector (in fact, internally they are the same, with matrices having additionaly attributes for dimension). A matrix is two-dimensional, with a row and column dimension. Like a vector, matrices only allow data of a single type. There are a few ways to make matrices in R:

- The `rbind()` function takes an arbitrary number of vectors as inputs (all of equal length), and creates a matrix where each input vector is a row of the matrix. `cbind()` is exactly like `rbind()` except that the vectors become columns rather than rows.
- The `matrix()` function takes a single vector input and turns that vector into a matrix. You can set either the `nrow` parameter or the `ncol` parameter to the number of rows or columns respectively that you desire your matrix to have (it is not necessary to specify both, though not illegal either so long as the product of the dimensions equals the length of the input vector). By default, R will fill the matrix by column; this means that it will fill the first column with the first contents of your input vector in sequence, then the next column with remaining elements, and so on until the matrix is filled and the contents of the input vector "exhausted."" Changing the `byrow` parameter to `byrow = TRUE` changes this behavior, and R will fill the matrix by rows rather than columns.

Both the rows and the columns of a matrix can be named, though you don't use the `names()` function for seeing or changing these names. Instead, use the `rownames()` or `colnames()` function for accessing or modifying the row names and column names, respectively.

You can get the dimensions of a matrix with the `dim()` function. `nrow()` returns the number of rows of a matrix, and `ncol()` the number of columns. `length()` returns the number of elements in the matrix (so the product of the dimensions).

```
# Using rbind to make a matrix
mat1 <- rbind(c("jim bridger", "meadowbrook", "elwood"), c("copper hills", "kearns",
    "west jordan"), c("university of utah", "byu", "westminster"), c("slcc",
    "snow", "suu"))
# Likewise with cbind
mat2 <- cbind(c("jim bridger", "meadowbrook", "elwood"), c("copper hills", "kearns",
    "west jordan"), c("university of utah", "byu", "westminster"), c("slcc",
    "snow", "suu"))
mat1
```

```
##      [,1]                [,2]          [,3]
## [1,] "jim bridger"       "meadowbrook" "elwood"
## [2,] "copper hills"      "kearns"      "west jordan"
## [3,] "university of utah" "byu"         "westminster"
## [4,] "slcc"              "snow"        "suu"
```

```
mat2
```

```
##      [,1]          [,2]           [,3]                [,4]
## [1,] "jim bridger" "copper hills" "university of utah" "slcc"
## [2,] "meadowbrook" "kearns"       "byu"               "snow"
## [3,] "elwood"      "west jordan"  "westminster"       "suu"
```

```r
dim(mat1)  # The dimensions of mat1
```

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

```r
nrow(mat1)  # The number of rows of mat1
```

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

```r
ncol(mat1)  # The number of columns of mat1
```

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

```r
length(mat1)  # The number of elements stored in mat1
```

```
## [1] 12
```

```r
# Using matrix()
mat3 <- matrix(1:10, nrow = 2)
mat4 <- matrix(1:10, nrow = 2, byrow = FALSE)
mat3
```

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

```r
mat4
```

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

```r
# Naming matrix dimensions
rownames(mat3) <- c("odds", "evens")
colnames(mat3) <- c("first", "second", "third", "fourth", "fifth")
mat3
```

```
##       first second third fourth fifth
## odds      1      3     5      7     9
## evens     2      4     6      8    10
```

```r
# Internally, matrices are glorified vectors
as.vector(mat1)
```

```
##  [1] "jim bridger"        "copper hills"       "university of utah"
##  [4] "slcc"               "meadowbrook"        "kearns"
##  [7] "byu"                "snow"               "elwood"
## [10] "west jordan"        "westminster"        "suu"
```

To access the elements of the matrix, you *could* do so with `mat[x]`, where `x` is a vector. This will treat the matrix `mat` like a vector. Sometimes this is the behavior you want, but most of the time you probably wish to access the data using the matrix's rows and columns (otherwise you would have made a vector).

R uses the notation `[,]` for referencing elements in a matrix. Thus you can reference objects in a matrix with `mat[x,y]`, where `x` is a vector specifying the desired rows, and `y` a vector specifying the desired columns. All the rules for referencing elements of a vector apply to `x` and `y`, with the additional rule that leaving a dimension blank will lead to everything in that dimension being included. Thus, `mat[,y]` results in a matrix with all the rows of `mat` and columns determined by `y`, and `mat[x,]` a matrix with all the columns of `mat` and rows determined by `x`.

```r
# Get the (1,2) entry of mat1
mat1[1, 2]
```

```
## [1] "meadowbrook"
```

```
# The first row of mat1; notice that this is a vector
mat1[1, ]
```

```
## [1] "jim bridger" "meadowbrook" "elwood"
```

```
# The second column of mat1; notice that this is also a vector
mat1[, 2]
```

```
## [1] "meadowbrook" "kearns"      "byu"         "snow"
```

```
# We can preserve the matrix structure (in other words, not turn the result
# into a vector) by adding an additional comma and specifying the option
# drop=FALSE
mat1[1, , drop = FALSE]
```

```
##      [,1]           [,2]           [,3]
## [1,] "jim bridger" "meadowbrook" "elwood"
```

```
mat1[, 2, drop = FALSE]
```

```
##      [,1]
## [1,] "meadowbrook"
## [2,] "kearns"
## [3,] "byu"
## [4,] "snow"
```

```
# A small 2x3 submatrix of mat1
mat1[1:2, 1:3]
```

```
##      [,1]           [,2]           [,3]
## [1,] "jim bridger"  "meadowbrook" "elwood"
## [2,] "copper hills" "kearns"      "west jordan"
```

```
# The third odd number in 1 to 10
mat3["odds", "third"]
```

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

```
# The first and third even numbers in 1 to 10
mat3["evens", c("first", "third")]
```

```
## first third
##     2     6
```

Matrices generalize to arrays, and can have more than two dimensions. For example, if `arr` is a three-dimensional array, we may access an element in it with `arr[1, 4, 3]`. We will not discuss arrays any further than this.

Data Frames

An R **data frame** stores data in a tabular format. Technically, a data frame is a list of vectors of equal length, so a data frame is a list. But since each "column" of the data frame has equal length, it also looks like a matrix where each column can differ in type (so one column could be numeric data, another character data, yet another factor data, etc.). Thus we can reference the data in a data frame like it is a list or like it is a matrix.

- The matrix style of referencing data frame data is like `df[x,y]`, where `x` is the rows of the data frame and `y` the columns. All the rules for using this notation with matrices apply to data frames. The result

is another data frame.

- The list style for referencing a data frame references only the columns, not the rows. So `df[x]` will select the columns of `df` specified by `x`, and the result is another data frame. `df[[x]]` refers to the vector stored in `df[[x]]`; this is a vector, not a data frame. More commonly, though, we refer to a column of a data frame we want with the dollar notation; rather than use `df[["x"]]`, we use `df$x` to get the column vector `x` in `df`.

To create a data frame, we have options:

- We could use the `data.frame()` function, where each vector passed will become a column in the data frame.
- We could use the `as.data.frame()` function on an object easily coerced into a data frame, like a matrix or a list.

Some examples are shown below.

```
# Making a data frame with data.frame
df1 <- data.frame(numbers = 1:5, letters = c("a", "b", "c", "d", "e"))
df1
```

```
##   numbers letters
## 1       1       a
## 2       2       b
## 3       3       c
## 4       4       d
## 5       5       e
```

```
# Notice that the character vector was automatically made a factor vector!
str(df1)
```

```
## 'data.frame':    5 obs. of  2 variables:
##  $ numbers: int  1 2 3 4 5
##  $ letters: Factor w/ 5 levels "a","b","c","d",..: 1 2 3 4 5
```

```
colnames(mat2) <- c("elementary", "high school", "university", "local")
# Make a data frame out of a matrix If we don't want to turn character
# strings into factors, set stringsAsFactors to FALSE (this also works in
# data.frame)
df2 <- as.data.frame(mat2, stringsAsFactors = FALSE)
df2
```

```
##    elementary  high school         university local
## 1 jim bridger copper hills university of utah  slcc
## 2 meadowbrook       kearns                byu  snow
## 3      elwood  west jordan        westminster   suu
```

```
str(df2)
```

```
## 'data.frame':    3 obs. of  4 variables:
##  $ elementary : chr  "jim bridger" "meadowbrook" "elwood"
##  $ high school: chr  "copper hills" "kearns" "west jordan"
##  $ university : chr  "university of utah" "byu" "westminster"
##  $ local      : chr  "slcc" "snow" "suu"
```

```
newlist <- list(first = c("Tamara", "Danielle", "John", "Kent"), last = c("Garvey",
    "Wu", "Godfrey", "Morgan"))
# Making a data frame from a list
df3 <- as.data.frame(newlist, stringsAsFactors = FALSE)
```

Lecture 4

Working with Data Frames

Data frames are such a key tool for R users that packages are written solely for the accessing and manipulation of data in data frames. Thus they deserve more discussion.

Often we wish to work with multiple variables stored in a data frame, but while the `$` notation is convenient, even it can grow tiresome with complicated computations. The function `with()` can help simplify code. The first argument of `with()` is a data frame, and the second argument is a command to evaluate.

```
d <- mtcars[1:10, ]
# We wish to know which cars have mpg within the first and third quartile.
# Here's a first approach that is slightly cumbersome
d[d$mpg > quantile(d$mpg, 0.25) & d$mpg < quantile(d$mpg), ]
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

```
# We can use the with function to clean things up
d[with(d, mpg > quantile(mpg, 0.25) & mpg < quantile(mpg)), ]
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

Often users don't want all the data in a data frame, but only a subset of it. The `which()` could be used to get the desired rows and a vector the desired columns, but this can quickly become cumbersome. Alternatively, use the `subset()` function for this task. The data frame is the first argument passed to `subset()`. Next, pass information to the `subset` parameter to decide on what rows to include, or the `select` parameter to choose the columns. Names of variables in the data frame can be used in `subset()` like in `with()`; you don't need to use `$` notation to choose the variable from within the data frame. Additionally, unlike when selecting with vectors, you can use `:` to choose all columns between two *names*, not just numbers, and you can use `-` in front of a vector of names to declare columns you *don't* want.

```
names(mtcars)
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

```
# Notice that I do not list the names as strings
subset(mtcars, select = c(mpg, cyl), subset = mpg > quantile(mpg, 0.9))
```

```
##                 mpg cyl
## Fiat 128       32.4   4
## Honda Civic    30.4   4
```

```
## Toyota Corolla 33.9   4
## Lotus Europa   30.4   4
```

```
# Other ways to select columns Using : on column names selects columns
# between the names on either side
subset(mtcars, select = hp:qsec, subset = !is.na(mpg) & mpg > quantile(mpg,
    0.25) & mpg < quantile(mpg, 0.75) & cyl == 8)
```

```
##                   hp drat    wt  qsec
## Hornet Sportabout 175 3.15 3.440 17.02
## Merc 450SE        180 3.07 4.070 17.40
## Merc 450SL        180 3.07 3.730 17.60
## Dodge Challenger  150 2.76 3.520 16.87
## Pontiac Firebird  175 3.08 3.845 17.05
## Ford Pantera L    264 4.22 3.170 14.50
```

```
# Using - on a vector of names selects all columns except those in a vector
subset(mtcars, select = -c(drat, wt, qsec), subset = !is.na(mpg) & mpg > quantile(mpg,
    0.25) & mpg < quantile(mpg, 0.75) & cyl == 8)
```

```
##                   mpg cyl  disp  hp vs am gear carb
## Hornet Sportabout 18.7   8 360.0 175  0  0    3    2
## Merc 450SE        16.4   8 275.8 180  0  0    3    3
## Merc 450SL        17.3   8 275.8 180  0  0    3    3
## Dodge Challenger  15.5   8 318.0 150  0  0    3    2
## Pontiac Firebird  19.2   8 400.0 175  0  0    3    2
## Ford Pantera L    15.8   8 351.0 264  0  1    5    4
```

```
# Here is the above without using subset; notice how complicated the command
# is
mtcars[!is.na(mtcars$mpg) & mtcars$mpg > quantile(mtcars$mpg, 0.25) & mtcars$mpg <
    quantile(mtcars$mpg, 0.75) & mtcars$cyl == 8, !(names(mtcars) %in% c("drat",
    "wt", "qsec"))]
```

```
##                   mpg cyl  disp  hp vs am gear carb
## Hornet Sportabout 18.7   8 360.0 175  0  0    3    2
## Merc 450SE        16.4   8 275.8 180  0  0    3    3
## Merc 450SL        17.3   8 275.8 180  0  0    3    3
## Dodge Challenger  15.5   8 318.0 150  0  0    3    2
## Pontiac Firebird  19.2   8 400.0 175  0  0    3    2
## Ford Pantera L    15.8   8 351.0 264  0  1    5    4
```

There are many other details about working with data frames that are common parts of an analysts workflow, such as reshaping a data frame (keeping the same information stored in a data frame but changing the data frame's structure) and merging (combining information in two data frames). Read the textbook for more information and examples of these very important ideas. The entire process of bringing data into a workable format is called **data cleaning**, a significant and often underappreciated part of an analyst's job.

Applying a Function Over a Collection

Often we wish to apply a function not to a single object or variable but instead a collection so we can get multiple values. For example, if we want all powers of two from one to ten, we could do so with the following:

```
2^1:10
```

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

A similar idea is that we could take the square root of numbers between 0 and 1 with:

```
sqrt(seq(0, 1, by = 0.1))
```

```
##  [1] 0.0000000 0.3162278 0.4472136 0.5477226 0.6324555 0.7071068 0.7745967
##  [8] 0.8366600 0.8944272 0.9486833 1.0000000
```

It may not be this simple though. For example, suppose we have a data frame, which I construct below:

```
library(MASS)
cdat <- subset(Cars93, select = c(Min.Price, Price, Max.Price, MPG.city, MPG.highway,
    EngineSize, Horsepower, RPM))
head(cdat)
```

```
##   Min.Price Price Max.Price MPG.city MPG.highway EngineSize Horsepower
## 1      12.9  15.9      18.8       25          31        1.8        140
## 2      29.2  33.9      38.7       18          25        3.2        200
## 3      25.9  29.1      32.3       20          26        2.8        172
## 4      30.8  37.7      44.6       19          26        2.8        172
## 5      23.7  30.0      36.2       22          30        3.5        208
## 6      14.2  15.7      17.3       22          31        2.2        110
##    RPM
## 1 6300
## 2 5500
## 3 5500
## 4 5500
## 5 5700
## 6 5200
```

I want the mean of all the variables in `cdat`. `mean(cdat)` will not work; the `mean()` function does not know how to handle the different variables in a data frame.

We may instead try a `for` loop, like so:

```
# Make an empty vector
cdat_means <- c()
# This starts a for loop
for (vec in cdat) {
    # For ever vector in cdat (called vec in the body of the loop), the code in
    # the loop will be executed Compute the mean of vec, and add it to
    # cdat_means
    cdat_means <- c(cdat_means, mean(vec))
}
names(cdat_means) <- names(cdat)
cdat_means
```

```
##   Min.Price       Price   Max.Price    MPG.city MPG.highway  EngineSize
##    17.125806   19.509677   21.898925   22.365591   29.086022    2.667742
##  Horsepower         RPM
##  143.827957 5280.645161
```

A good R programmer will try to avoid `for` loops as much as possible. One reason is that `for` loops in R are slow, unlike in other languages. Since R is an interpreted language and also includes many features for interacting with R and writing code easier, R programs are going to be slower than in other languages. This is the price R pays for being interactive and much easier to write code for than compiled languages like C, C++, or Java. (A lot of R functions run fast because the function is actually an interface for a function written in C, C++, or FORTRAN.) Another reason R programmers avoid `for` loops is that there is often an alternative not using a loop that easier to both write and understand.

How could we rewrite the above code without using `for`? We could use the function `sapply()` and the call `sapply(v, f)`, where `v` is either a vector or list with the items you wish to iterate over, and `f` is a function to apply to each item. (Remember that a data frame is a list of vectors of equal length.) A vector is returned containing the result.

```
# A function to check if a number is even
even <- function(x) {
    # If x is divisible by 2 (the remainder is 0 when x is divided by 2), x is
    # even and the result is TRUE. Otherwise, the result is FALSE.
    x%%2 == 0
}

# Which numbers between 1 and 10 are even?
sapply(1:10, even)
```

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

```
# The means of the vectors in cdat (remember that a data frame is a list of
# equal length vectors)
sapply(cdat, mean)
```

```
##   Min.Price       Price   Max.Price    MPG.city MPG.highway  EngineSize
##   17.125806   19.509677   21.898925   22.365591   29.086022    2.667742
##  Horsepower         RPM
##  143.827957 5280.645161
```

```
# We can pass sapply an anonymous function, which is an unnamed function
# passed as an argument to some other function, used for some evaluation. I
# illustrate below by passing to sapply a function that computes the range
# of each of the variables in cdat.
sapply(cdat, function(vec) {
    diff(range(vec))
})
```

```
##   Min.Price       Price   Max.Price    MPG.city MPG.highway  EngineSize
##        38.7        54.5        72.1        31.0        30.0         4.7
##  Horsepower         RPM
##       245.0      2700.0
```

The `lapply()` function works exactly like the `sapply()` function, except `lapply()` returns a list rather than a vector.

Alternatively, if we have a function `f(x)` that knows how to work with an object `x`, we could **vectorize** `f` so it can work on a vector or list of objects like `x`. We can use the `Vectorize()` function for this task with a call like `vf <- Vectorize(f)`, where `f` is the function to vectorize, and `vf` is the new, vectorized version of `f`. The example below does what we did for `cdat` with both a `for` loop and `sapply()`, but now does so with a vectorized version of `mean()`.

```
vmean <- Vectorize(mean)
vmean(cdat)
```

```
##   Min.Price       Price   Max.Price    MPG.city MPG.highway  EngineSize
##   17.125806   19.509677   21.898925   22.365591   29.086022    2.667742
##  Horsepower         RPM
##  143.827957 5280.645161
```

Now suppose you have a data frame `d`, which contains information from different samples representing different populations. You wish to apply a function `f()` to data stored in `d$x`, and `d$y` determines which sample each row of the data frame (and thus, each entry of `d$x`) came from. You want `f()` to be applied

to the data in each sample, separately. You can do so with the `aggregate()` function in a call of the form `aggregate(x ~ y, data = d, f)`. I illustrate with the `iris` dataset below.

```
# The struture of iris
str(iris)
```

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

```
# The mean sepal length by species of iris
aggregate(Sepal.Length ~ Species, data = iris, mean)
```

```
##      Species Sepal.Length
## 1     setosa        5.006
## 2 versicolor        5.936
## 3  virginica        6.588
```

```
# The five-number summary of sepal length for each species of iris
aggregate(Sepal.Length ~ Species, data = iris, quantile)
```

```
##      Species Sepal.Length.0% Sepal.Length.25% Sepal.Length.50%
## 1     setosa           4.300            4.800            5.000
## 2 versicolor           4.900            5.600            5.900
## 3  virginica           4.900            6.225            6.500
##   Sepal.Length.75% Sepal.Length.100%
## 1            5.200             5.800
## 2            6.300             7.000
## 3            6.900             7.900
```

Let's now consider matrices. Perhaps we have a matrix and we wish to apply a function across the rows of the matrix or the columns of the matrix. The `apply()` function allows us to do just that in a call of the form `apply(mat, m, f)`, where `mat` is the matrix with data, `f` the function to apply, and `m` the margin to apply `f()` over. For matrices, a value of `1` for `m` will lead to the function being applied across rows, and a value of `2` across columns. I illustrate with a data set recording the ethnicity of selected Utah publich schools (to see how this data set was created, view the source code of this document).

```
## Loading required package: methods
```

```
school_race_dat
```

```
##                  Entheos Academy Kearns Entheos Academy Magna
## Native American                        0                     0
## Asian                                  4                     5
## Black                                  1                     5
## Hispanic                             145                   201
## Pacific Islander                      15                     3
## White                                334                   273
## Multiple Race                         23                    15
##                  Jim Bridger School Sunset Ridge Middle Copper Hills High
## Native American                   4                    5                9
## Asian                             6                   25               50
## Black                            12                   19               42
## Hispanic                        216                  322              551
## Pacific Islander                 12                   28               28
```

```
## White                            314                1124              1924
## Multiple Race                      7                  50               102
##                   Thomas Jefferson Jr High Kearns High
## Native American                         11          39
## Asian                                   13          49
## Black                                   17          53
## Hispanic                               260         937
## Pacific Islander                        42          99
## White                                  394        1138
## Multiple Race                            2          10
```

```
# Get row sums
apply(school_race_dat, 1, sum)
```

```
##  Native American            Asian            Black         Hispanic
##               68              152              149             2632
## Pacific Islander            White    Multiple Race
##              227             5501              209
```

```
# Column sums
apply(school_race_dat, 2, sum)
```

```
##   Entheos Academy Kearns    Entheos Academy Magna       Jim Bridger School
##                      522                      502                      571
##      Sunset Ridge Middle        Copper Hills High Thomas Jefferson Jr High
##                     1573                     2706                      739
##              Kearns High
##                     2325
```

```
# Row sums and column sums are actually used frequently, so there are
# specialized functions for these
rowSums(school_race_dat)
```

```
##  Native American            Asian            Black         Hispanic
##               68              152              149             2632
## Pacific Islander            White    Multiple Race
##              227             5501              209
```

```
colSums(school_race_dat)
```

```
##   Entheos Academy Kearns    Entheos Academy Magna       Jim Bridger School
##                      522                      502                      571
##      Sunset Ridge Middle        Copper Hills High Thomas Jefferson Jr High
##                     1573                     2706                      739
##              Kearns High
##                     2325
```

Using External Data

R would not be very useful if we had no way of loading in and saving data. R has means for reading data from spreadsheets such as `.xls` or `.xlsx` files made by Microsoft Excel. Functions for reading Excel files can be found in the **xlsx** or **gdata** packages.

Common plain-text formats for reading data include the comma-separated values format (`.csv`), tab-separated values format (`.tsv`), and the fixed-width format (`.fwf`). These files can be read in using the `read.csv()`, `read.table()`, and the `read.fwf()` functions (with `read.csv()` being merely a front-end for `read.table()`).

All of these functions parse a plain-text data file and return a data frame with the contents. Keep in mind that R will guess what type of data is stored in the file. Usually it makes a good guess, but this is not guaranteed and you may need to do some more data cleaning or give R more instructions on how to interpret the file.

In order to load a file, you must specify the location of the file. If the file is on your hard drive, there are a few ways to do so:

- You could use the `file.choose()` command to browse your system and locate the file. Once done, you will have a text string describing the location of the file on your system.
- Any R session has a **working directory**, which is where R looks first for files. You can see the current working directory with `getwd()`, and change the working directory with `setwd(path)`, where `path` is a string for the location of the directory you wish to set as the new working directory.

Let's assume we're loading in a `.csv` file (the approach is similar for other formats). The command `df <- read.csv("myfile.csv")` instructs R to read `myfile.csv` (which is presumably in the working directory, since we did not specify a full path; if it were not, we would either change the working directory or pass the full path to the function, which may look something like `read.csv("C:/path/to/myfile.csv")`, or `read.csv("/path/to/myfile.csv")`, depending on the system) and store the resulting data frame in `df`. Once done, `df` will now be ready for us to use.

Suppose that the data file is on the Internet. You can pass the url of the file to `read.csv()` and R will read the file online and make it available to you in your session. I demonstrate below:

```
# Total Primary Energy Consumption by country and region, for years 1980
# through 2008; in Quadrillion Btu (CSV Version). Dataset from data.gov,
# from the Department of Energy's dataset on total primary energy
# consumption.  Download and load in the dataset
energy <- read.csv("http://en.openei.org/doe-opendata/dataset/d9cd39c5-492e-4e82-8765-12e0657eeb4e/reso
    stringsAsFactors = FALSE)
# R did not parse everything correctly; turn some variables numeric
energy[2:30] <- lapply(energy[2:30], as.numeric)
# We want energy data for North American countries, from 2000 to 2008
us_energy <- subset(energy, select = X2000:X2008, subset = Country %in% c("Canada",
    "United States", "Mexico"))
us_energy
```

```
##      X2000    X2001    X2002    X2003     X2004     X2005    X2006
## 2 13.07669 12.87847 13.10786 13.52061  13.83128  14.16374 13.81736
## 4  6.37958  6.32931  6.32936  6.50563   6.48998   6.80188  7.36271
## 6 99.25385 96.53415 98.03879 98.31384 100.49743 100.60722 99.90566
##       X2007    X2008
## 2  14.07179 14.02923
## 4   7.27651  7.30898
## 6 101.67563 99.53011
```

Naturally you can export data frames into common formats as well. `write.csv()`, `write.table()`, and `write.fwf()` will write data into comma-separated value, tab-separated value, and fixed width formats. Their syntax is similar. To save a `.csv` file, issue the command `write.csv(df, file = "myfile.csv")`, where `df` is the data frame to save and `file` where to save it, which could be just a file name (resulting in the file being saved in the working directory), or an absolute path.

```
my_data <- data.frame(var1 = 1:10, var2 = paste("word", 1:10))
write.csv(my_data, file="my_data.csv")
```

There are other formats R can read and write to. The **foreign** package allows R to read data files created for other statistical software packages such as SAS or Stata. The **XML** package allows R to read XML and

HTML files. You can also read JSON files or data stored in Google Sheets. Refer to the textbook for more information.

Lecture 5

Multivariate Visualization

A picture is worth a thousand words. This may be especially true in statistics. While plotting univariate or even bivariate data is not too difficult, and we can learn much from such plots, multivariate data is much more difficult to visualize effectively. We often have a number of variables in a data set and we want to learn as much as we can about their relationships. We explore some techniques here.

One reason why many data analysts use R is because R can create graphics that are both informative and visually appealing without too much effort or time (if you use the right packages and know what you are doing). For the next two lectures, we will explore methods for visualizing multivariate data using three dominant plotting tools: base R plotting, plotting using the **lattice** package, and plotting using the **ggplot2** package. This first lecture discusses base R plotting.

Base R Plotting

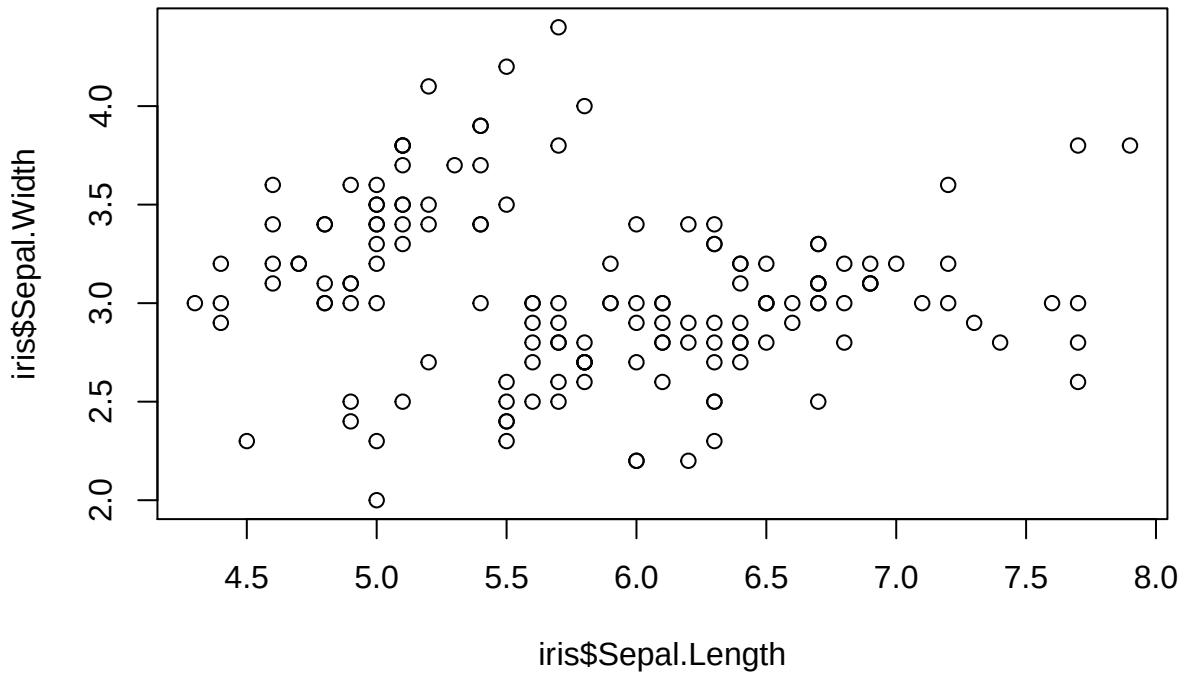
R comes built in with functions for plotting, the primary one being the `plot()` function, which we have seen before. It is a generic function that will often pick the right plot for the object passed to it. That said, we do have control over how to make a plot using base R.

A **scatterplot** plots each data point as an ordered (x, y) pair, with x being the value of the variable corresponding to x for each data point, and y the value for the corresponding y variable. Thus scatterplots are useful for visualizing bivariate data.

We can make scatterplots using base R with `plot(x, y)`, where `x` and `y` are numeric vectors containing the `x` and `y` coordinates to plot, respectively. The plot created by default will be a scatter plot, though there are many parameters for `plot()` that will change not only whether a scatterplot is drawn but other characteristics of the resulting plot, such as the extent of the axes, the character of the points, axis labels, and many others.

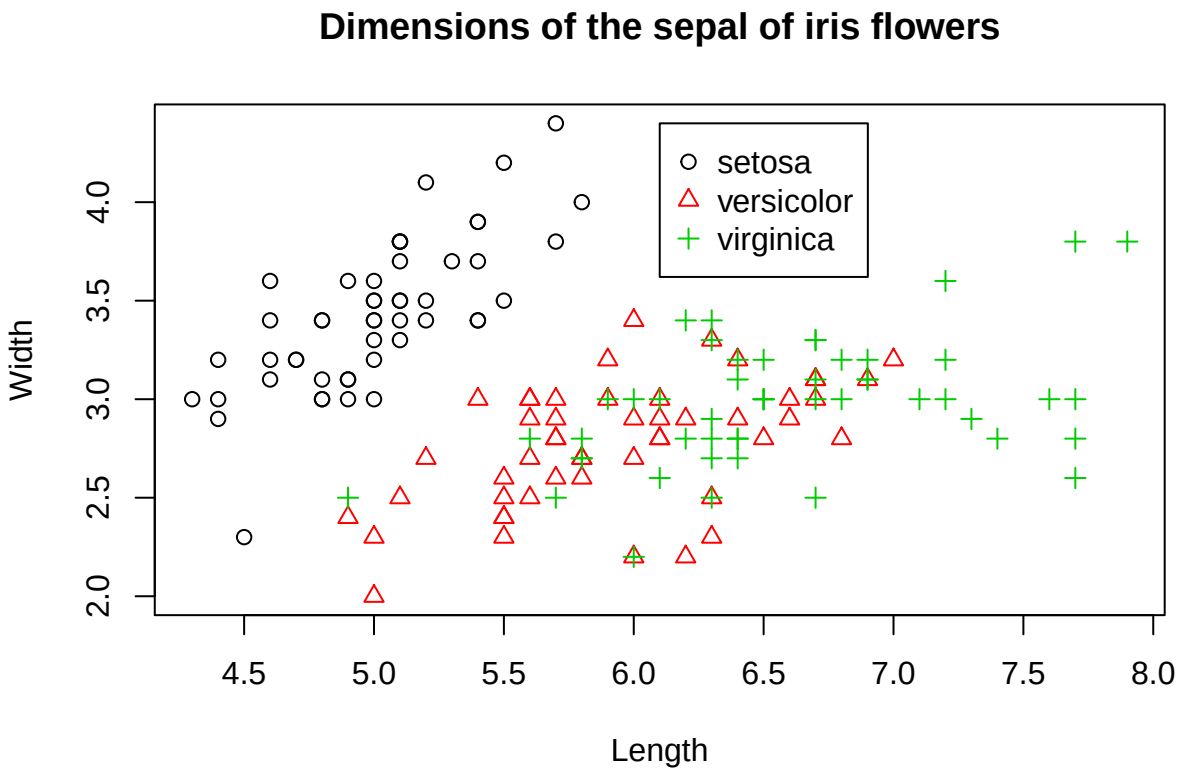
Suppose we wished to compare the sepal length of iris flowers to the petal length of iris flowers in the `iris` data set. We could do so in base R with the following:

```
plot(iris$Sepal.Length, iris$Sepal.Width)
```



This plot gives us some basic information about these variables, but a lot is not shown. In particular, we know that there are three species of iris included in the `iris` data set, and we would like to display this information on the chart. We could do so using the following:

```
with(iris,
     # First, the scatter plot
     plot(Sepal.Length, Sepal.Width,
          # Make the color depend on the species
          col = as.numeric(Species),
          # Also make the character depend on the species
          pch = as.numeric(Species),
          # Some labels to make the plot more informative
          xlab = "Length", ylab = "Width", main = "Dimensions of the sepal of iris flowers"
          )
     )
# Add a legend to the plot
legend(
  # The first two arguments are the coordinates of the legend on the plot
  6.1, 4.4,
  # The next argument is the species of flower to label
  c("setosa", "versicolor", "virginica"),
  # The colors used for encoding
  col = 1:3,
  # The character used for encoding
  pch = 1:3
)
```

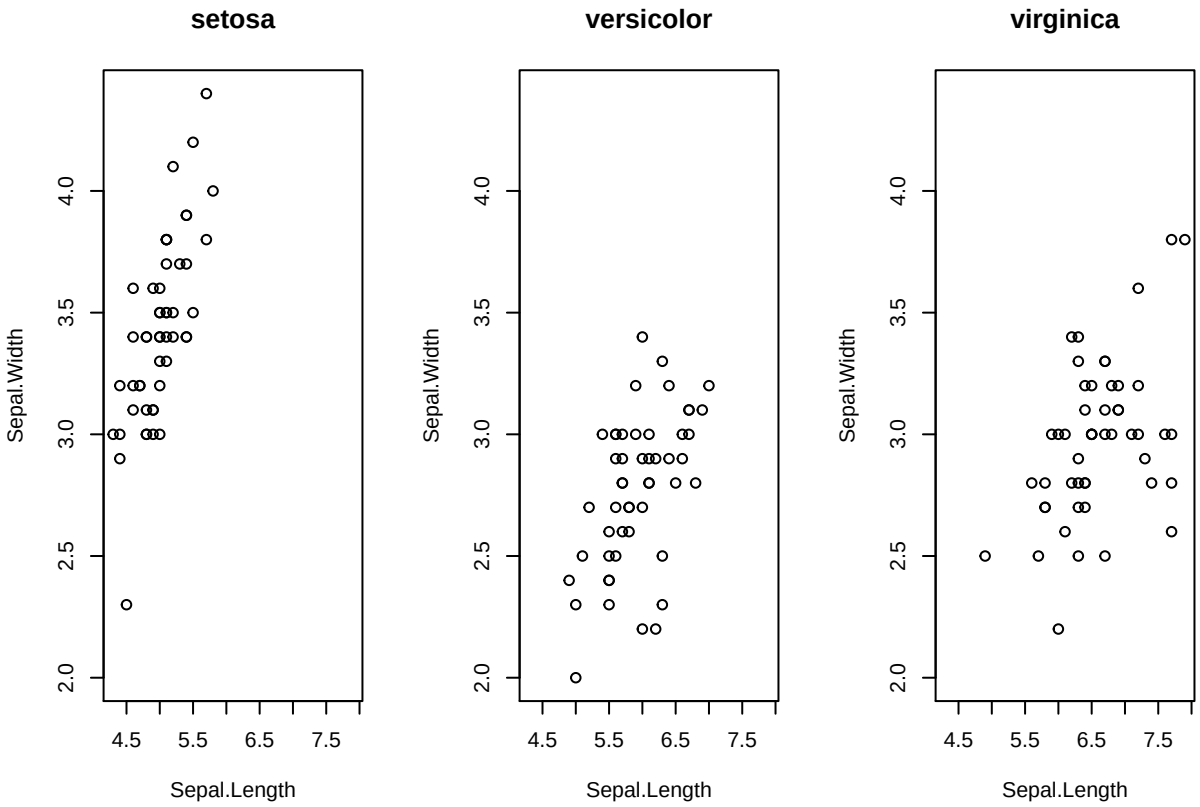


We kept the scatterplot, but distinguished different species by color. To further differentiate points, we also changed the shape of the points and made them depend on species. In other words, we doubly encoded the species information in both the shape and color of the points in the scatterplot.

Suppose we don't want all the species on the same plot. Versicolor and virginica flowers are intermixed, making them difficult to distinguish in the plot. We may try to create three plots side-by-side by changing the settings of a function called `par()`, which controls how plots are created. The following code does so:

```
width_range <- range(iris$Sepal.Width)
length_range <- range(iris$Sepal.Length)
# Manually split into three datasets
iris_setosa <- subset(iris, subset = Species == "setosa")
iris_versicolor <- subset(iris, subset = Species == "versicolor")
iris_virginica <- subset(iris, subset = Species == "virginica")
# The par function controls plotting parameters like margin size, borders,
# and many others. For this purpose, we would like to use margin to allow us
# to plot multiple plots, specifically in a 1x3 grid. The parameter we can
# set with par to control this is mfrow, which we set with a length 2 vector
# with the first coordinate being the number of rows, and the second
# coordinate being the number of columns. First, save the current par
# settings:
old_par <- par()
# Now, change the settings
par(mfrow = c(1, 3))
# After setting this way, we call plots as usual, but when they are made
# they will be added to the 1x3 graphic left-to-right, top-to-bottom.  I
# also use formula notation to create the plot here. To make a (x,y) plot,
# use y ~ x.
plot(Sepal.Width ~ Sepal.Length, data = iris_setosa, ylim = width_range, xlim = length_range,
    main = "setosa")
```

```
plot(Sepal.Width ~ Sepal.Length, data = iris_versicolor, ylim = width_range,
    xlim = length_range, main = "versicolor")
plot(Sepal.Width ~ Sepal.Length, data = iris_virginica, ylim = width_range,
    xlim = length_range, main = "virginica")
```



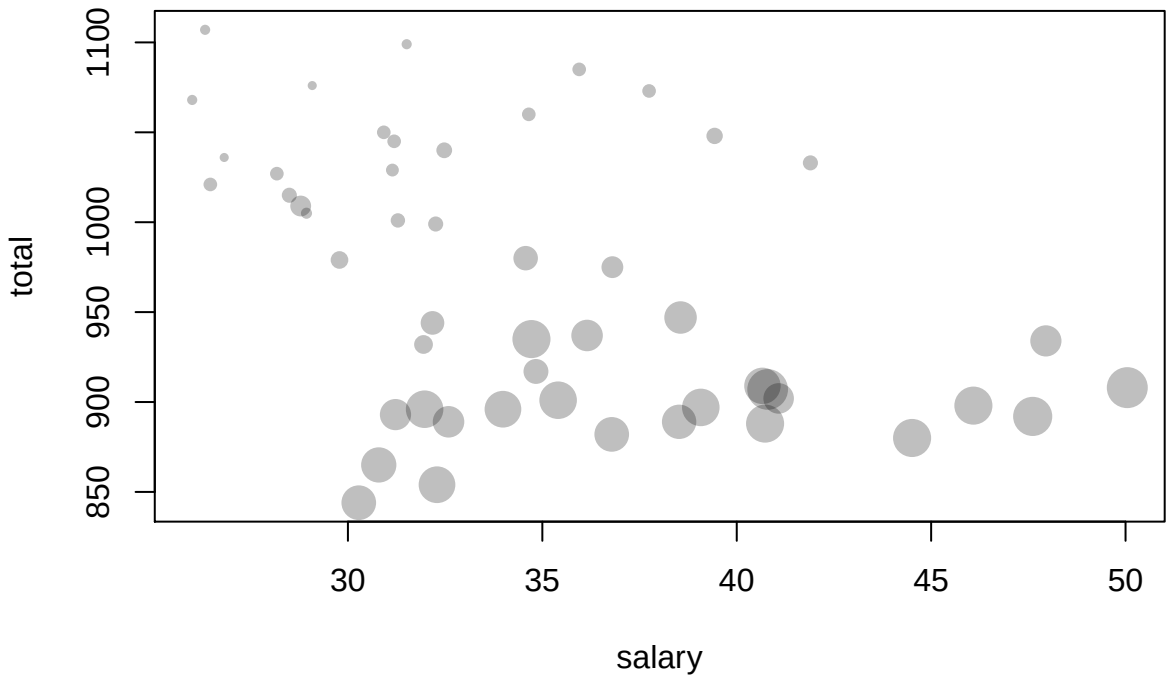

```
# You should reset par to its original settings once done, or you will
# continue to get plots using these settings, which may not be what you
# want.
par(old_par)
```

```
## Warning in par(old_par): graphical parameter "cin" cannot be set
## Warning in par(old_par): graphical parameter "cra" cannot be set
## Warning in par(old_par): graphical parameter "csi" cannot be set
## Warning in par(old_par): graphical parameter "cxy" cannot be set
## Warning in par(old_par): graphical parameter "din" cannot be set
## Warning in par(old_par): graphical parameter "page" cannot be set
```

Now, suppose that a third variable we wish to plot is numerical rather than categorical. We may use a **bubbleplot** to do so, where we create a scatterplot but each point in the scatterplot is a circle with area representing the value of a third variable. (Note that the third variable must be encoded by area, *not diameter!* This is because people percieve area, not diameter, and using diameter to encode information rather than area will create plots that are confusing and misleading.) We can create a bubble chart in R by making the `cex` parameter in `plot()`, which controls the size of points, dependant on one of the variables in the data set.

The following plot is a bubbleplot that shows states average teacher salary, average total SAT score, and encodes the percentage of SAT takers as the area of the bubbles.

```
# The SAT data is in UsingR
library(UsingR)
plot(total ~ salary, data = SAT,
     # filled circles
     pch = 16,
     # rgb is a function for creating colors via RGB values. The alpha parameter controls transparency.
     col=rgb(red = 0, green = 0, blue = 0, alpha = 0.250),
     # cex controls the size (length) of the points. The following makes the points' area dependant on
     cex = sqrt(perc/10))
```

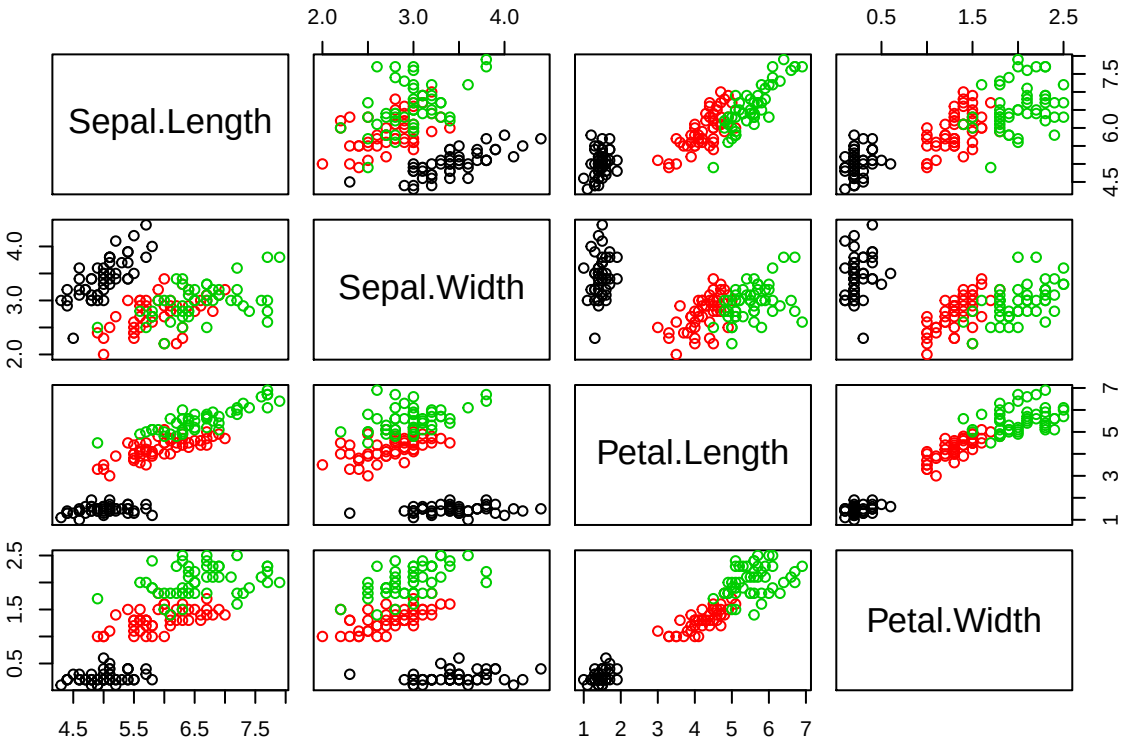


So far, we have seen ways to visualize three variables together, basing our graphics on scatterplots. What about data sets with more than three variables?

A **scatterplot matrix** creates multiple scatterplots in a grid (matrix), each showing a different combination of variables. The variables become the rows and columns of the matrix, and the plot in a particular row and column of the matrix represents a particular combination of the variables.

The `pairs()` function will create scatterplot matrices. The first argument passed to it is a data frame containing the data to be plotted, and other parameters can change details about how the plot is made. I create a scatterplot matrix for the `iris` dataset below.

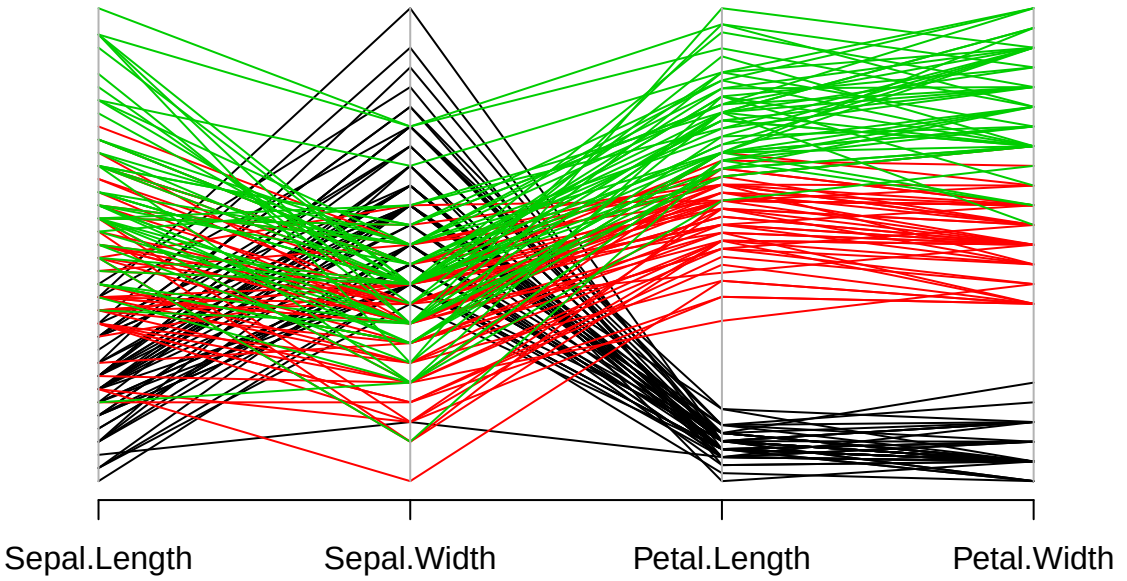
```
pairs(iris[c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")],
      # Make color depend on species
      col = iris$Species)
```



Another way to visualize multivariate data is with a **parallel coordinate plot**. This plot will display variables as vertical axis lines and data points as lines connecting the axes. The point where a data point line intersects an axis represents the value of that variable for that data point.

The `parcoord()` function (in the **MASS** package) creates parallel coordinate plots. The first argument is a data frame containing the data to be plotted, and other parameters control other features of the resulting plot. An example for the `iris` data is shown below.

```
library(MASS)
parcoord(iris[c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")],
    col = iris$Species)
```

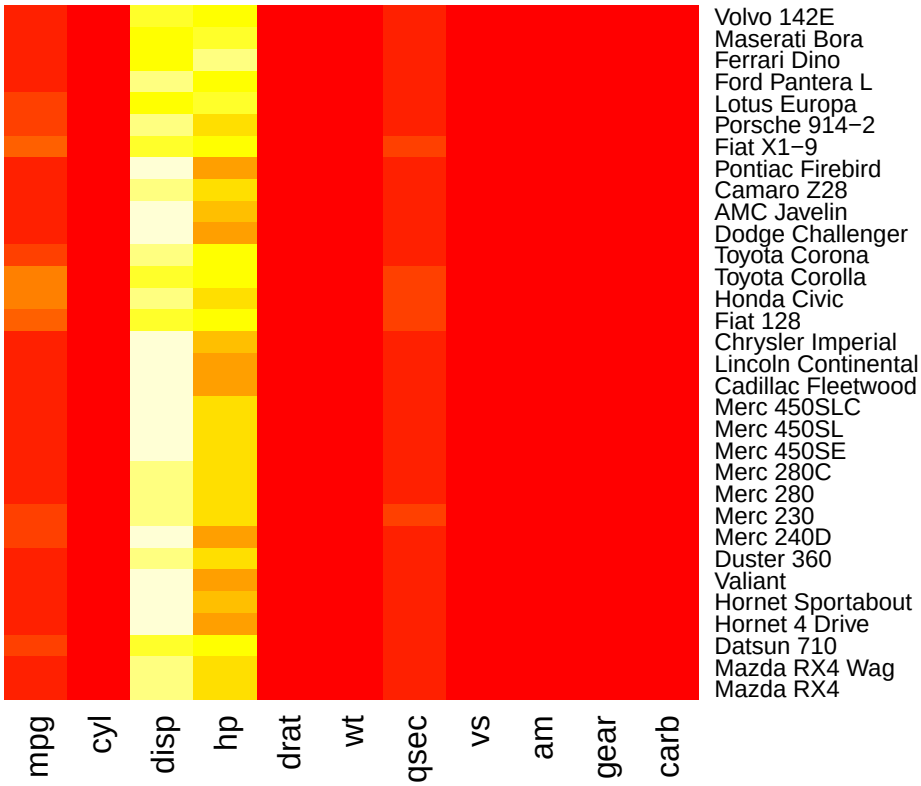


A **heatmap** is a matrix where the rows are data points, columns are variables, and in each cell of the matrix

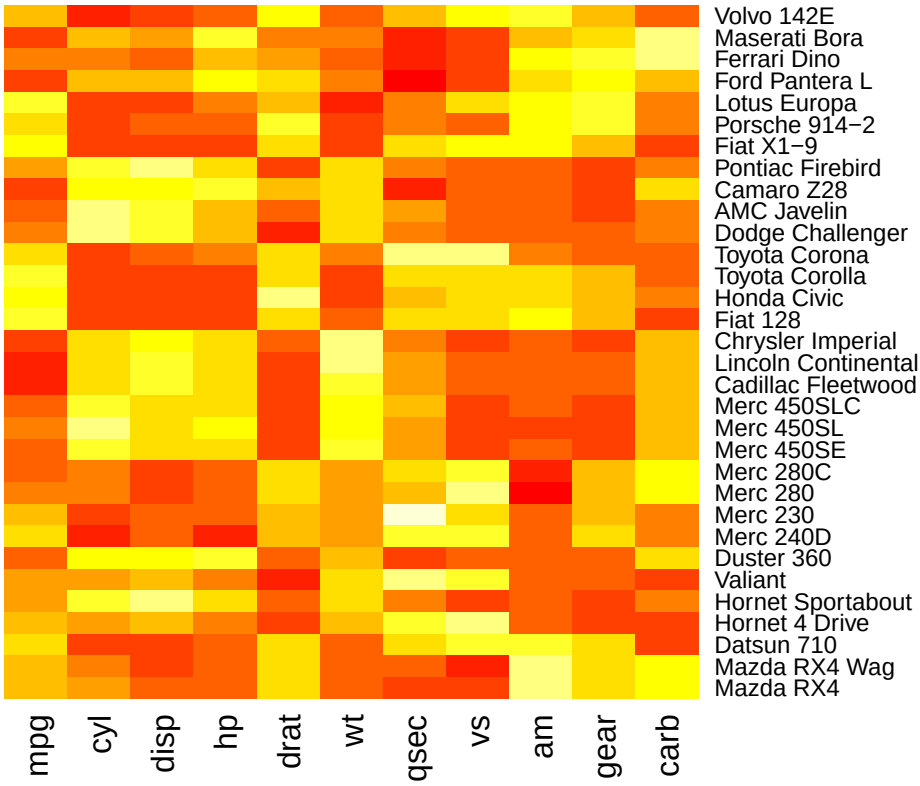
the value of a variable for an observation is represented in color. The hue or intensity of the color depends on the value of the variable.

The `heatmap()` function will create heatmaps in R. The first argument is the numeric matrix with the data to plot. I create a heatmap for the `mtcars` data set below.

```
heatmap(as.matrix(mtcars), Rowv = NA, Colv = NA)
```




```
# This is not a useful plot; the heatmap function thinks that all the
# observations are in the same units. What I will do is create a numeric
# matrix with the standardized value of each variable, where I subtract the
# mean of each variable from each observation, then divide by the standard
# deviation.
plotmat <- sapply(mtcars, scale)
rownames(plotmat) <- rownames(mtcars)
heatmap(plotmat, Rowv = NA, Colv = NA)
```



Plotting with base R has the advantage that all installations of R include the base R systems. All other plotting implementations are best seen as useful user interfaces for the existing base R system (particularly a plotting system called **grid**, which is the basis of **lattice** and **ggplot2**). Additionally, `plot()` is a generic function that can be programmed to create the most useful plot for the object passed to it.

That said, many data analysts prefer to use packages such as **lattice** or **ggplot2** for much of their graphical work, avoiding base R plotting. Unless there is a convenience function or `plot()` method for whatever plot you wish to make, it can be unacceptably tedious to make plots using only base R, especially if they are complex plots. The plot we created for the second `iris` scatterplot, for example, needed a legend, which we created manually in a manner both tedious and not very robust to changes in the plot or underlying data.

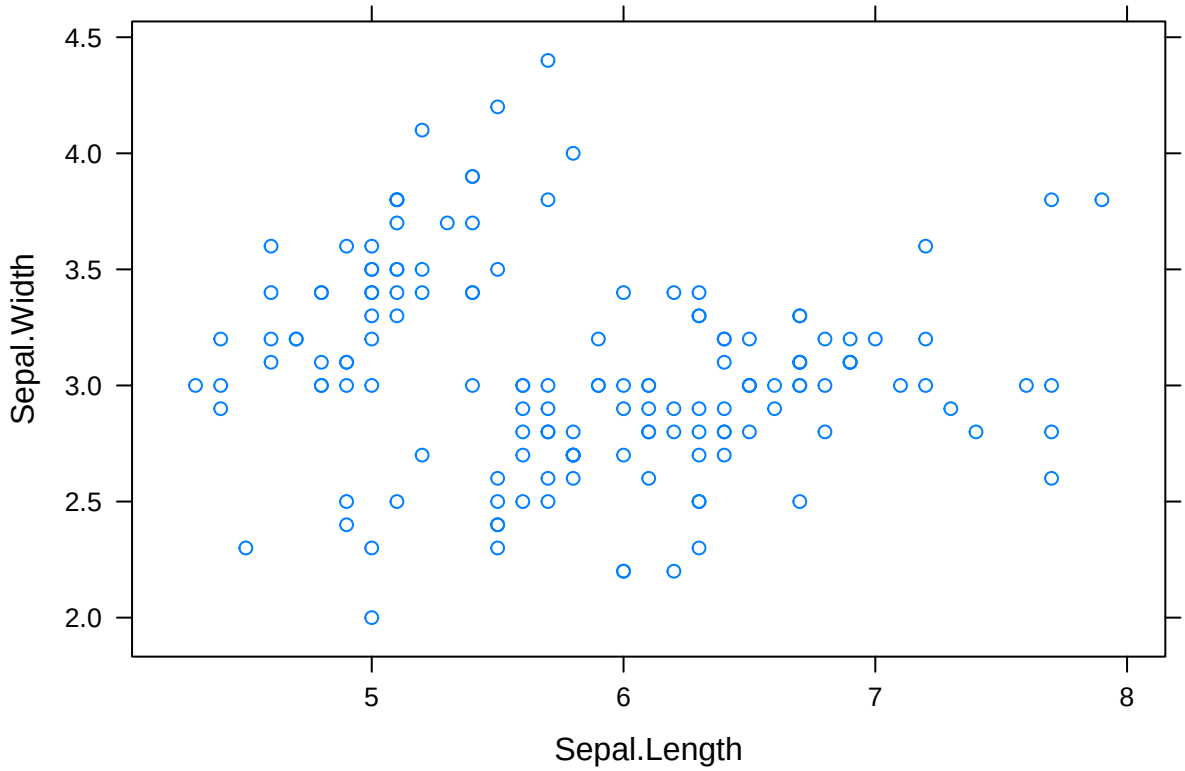
Lecture 6

lattice Plotting

One of the first packages made to make plotting in R easier was the **lattice** package, which is now included with standard R installations. **lattice** aims to make creating graphics for multivariate data easier, and relies on R's formula notation to do so.

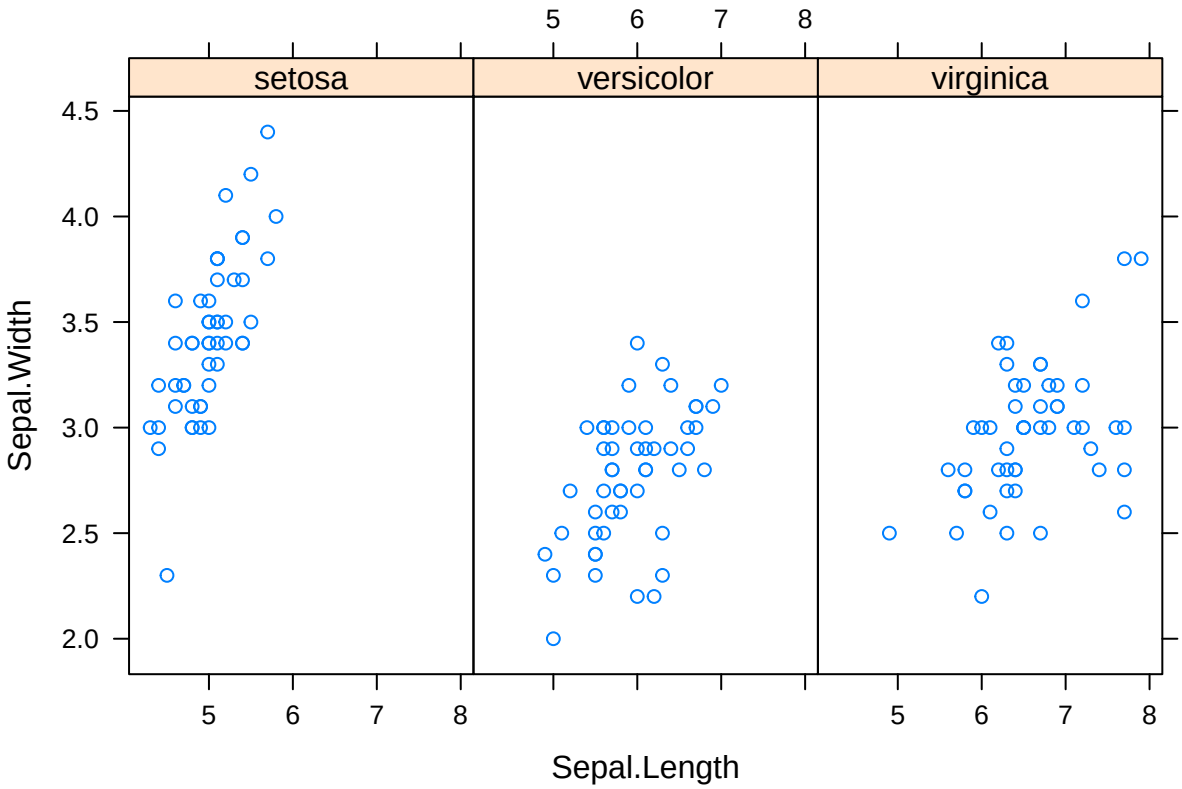
We can create a basic scatterplot in **lattice** using `xyplot(y ~ x, data = d)`, where `y` is the y-variable, `x` the x-variable, and `d` the data frame containing the data.

```
library(lattice)
xyplot(Sepal.Width ~ Sepal.Length, data = iris)
```



What if we wish to create multiple plots, breaking up the plots by different categorical variables? We can do so with `xyplot(y ~ x | c, data = d)`, where all is as before but `c` is a categorical variable with which we break up the plots.

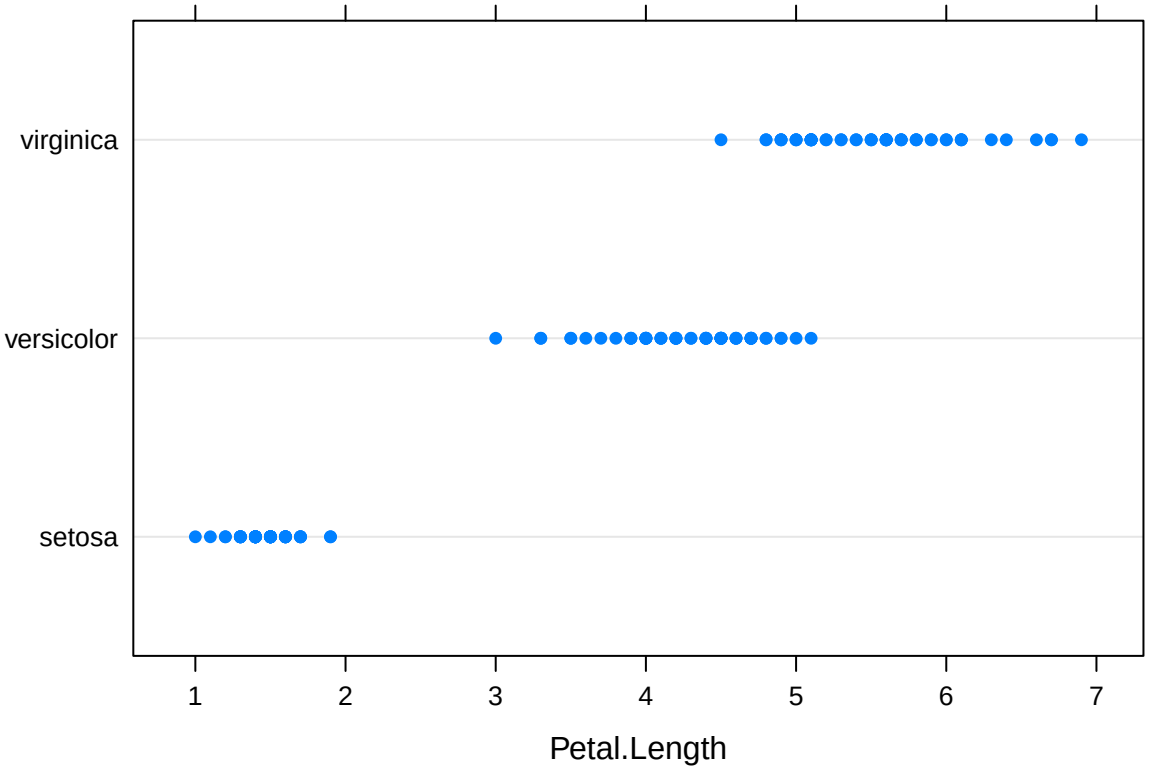
```
xyplot(Sepal.Width ~ Sepal.Length | Species, data = iris)
```



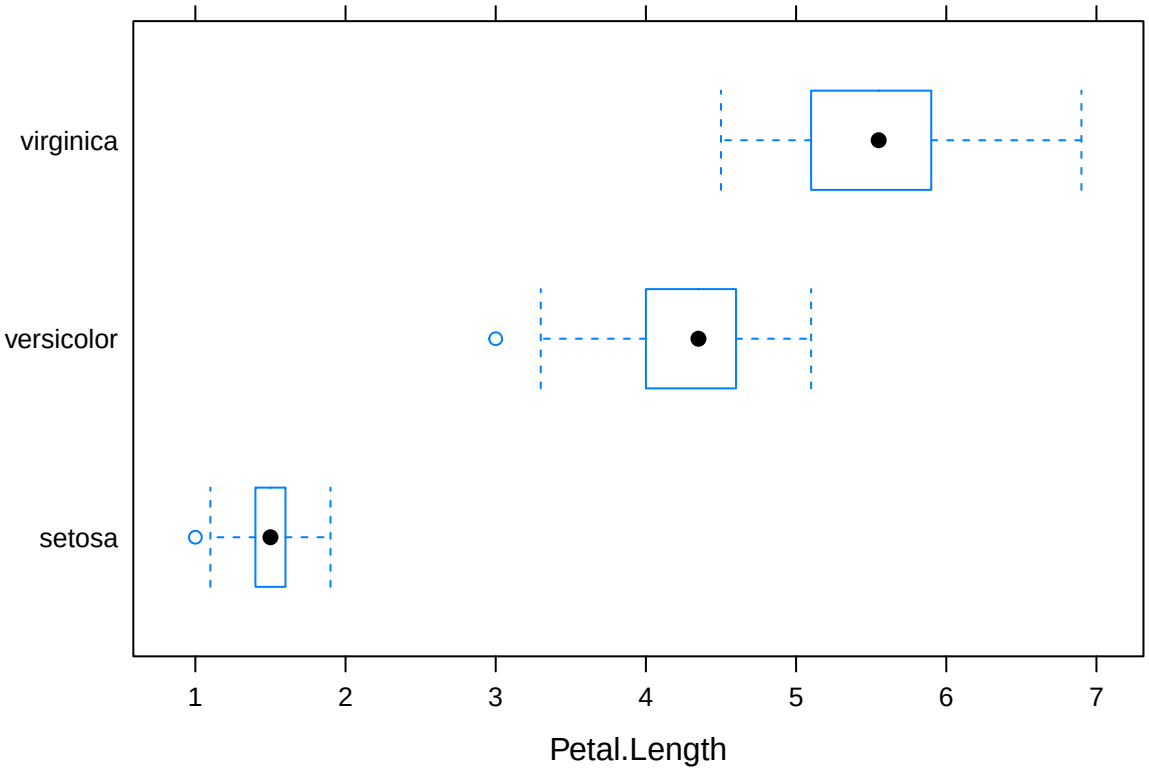

```
# Compare to the solution with base R
```

The aim of **lattice** is to create complex graphics using a single function call. Thus we can create other interesting graphics in lattice that we could make in base R. Some examples are shown below.

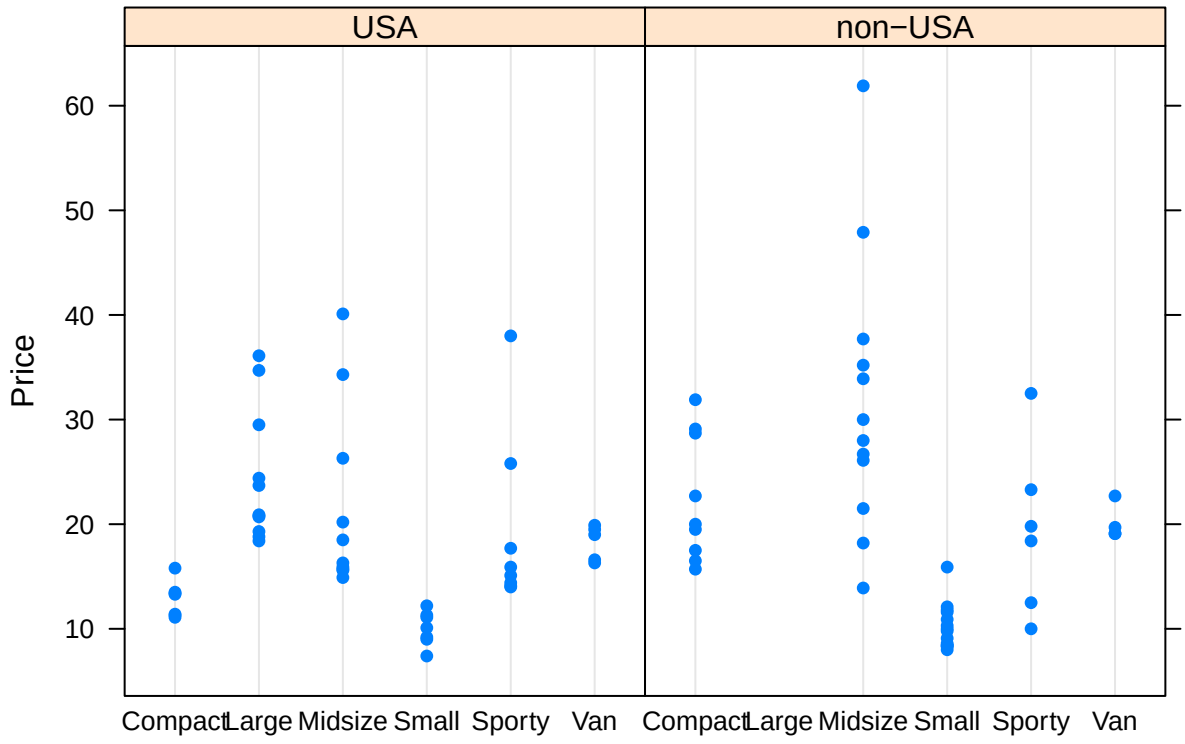
```
library(MASS)
# Lattice comparative dotplot of iris petal length
dotplot(Species ~ Petal.Length, data = iris)
```



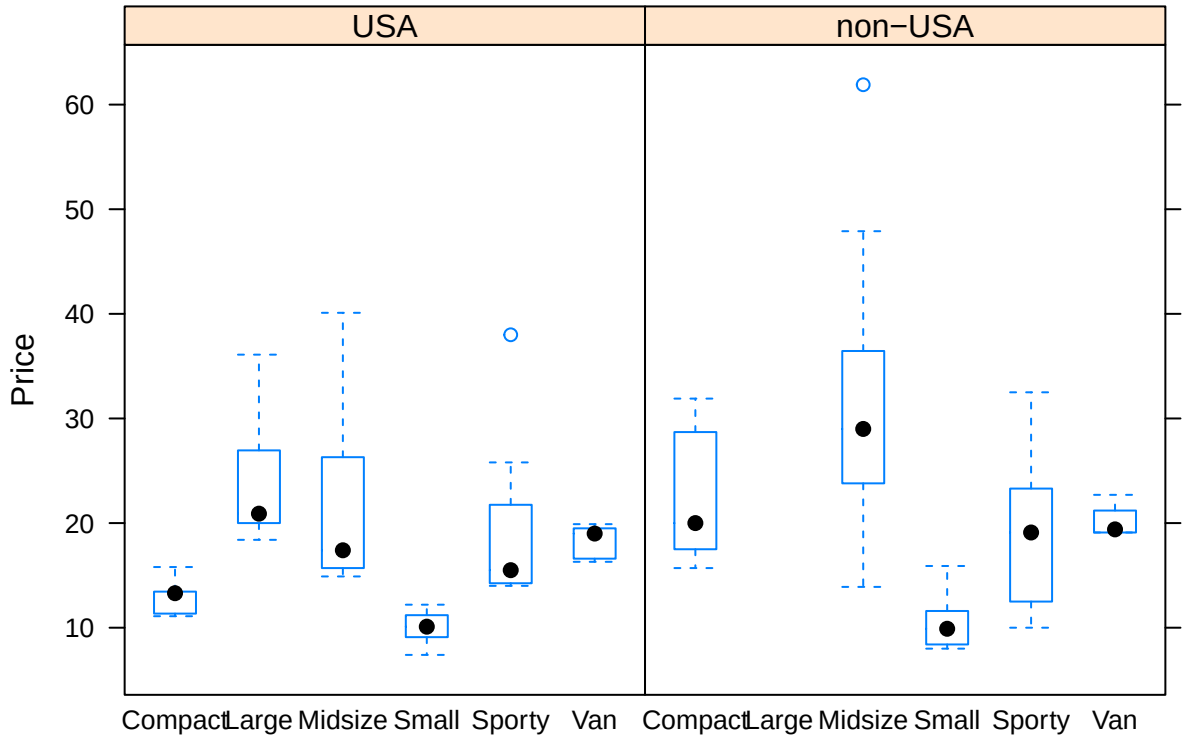

```
# Lattice comparative boxplot, which resembles the base R comparative
# boxplot in style
bwplot(Species ~ Petal.Length, data = iris)
```



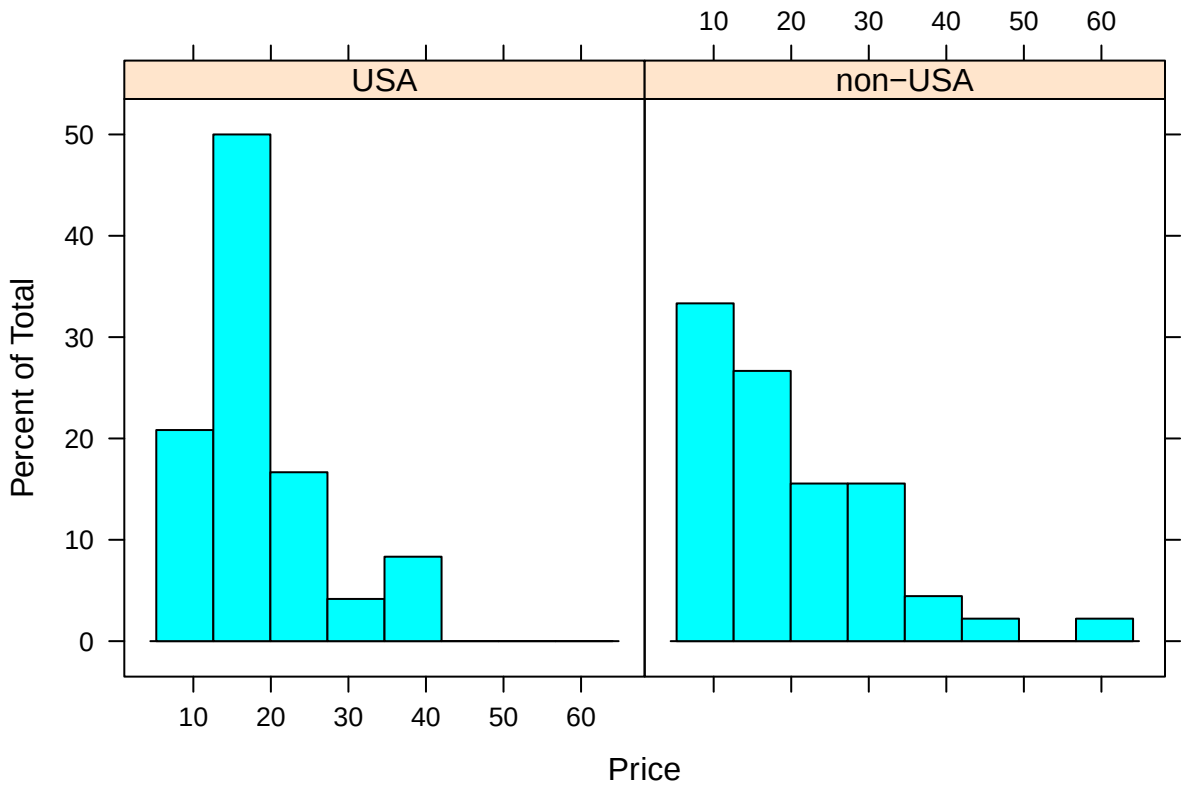
```
# For the Cars93 data set, let's look at price depending on the type of car
# and the origin of the car
dotplot(Price ~ Type | Origin, data = Cars93)
```



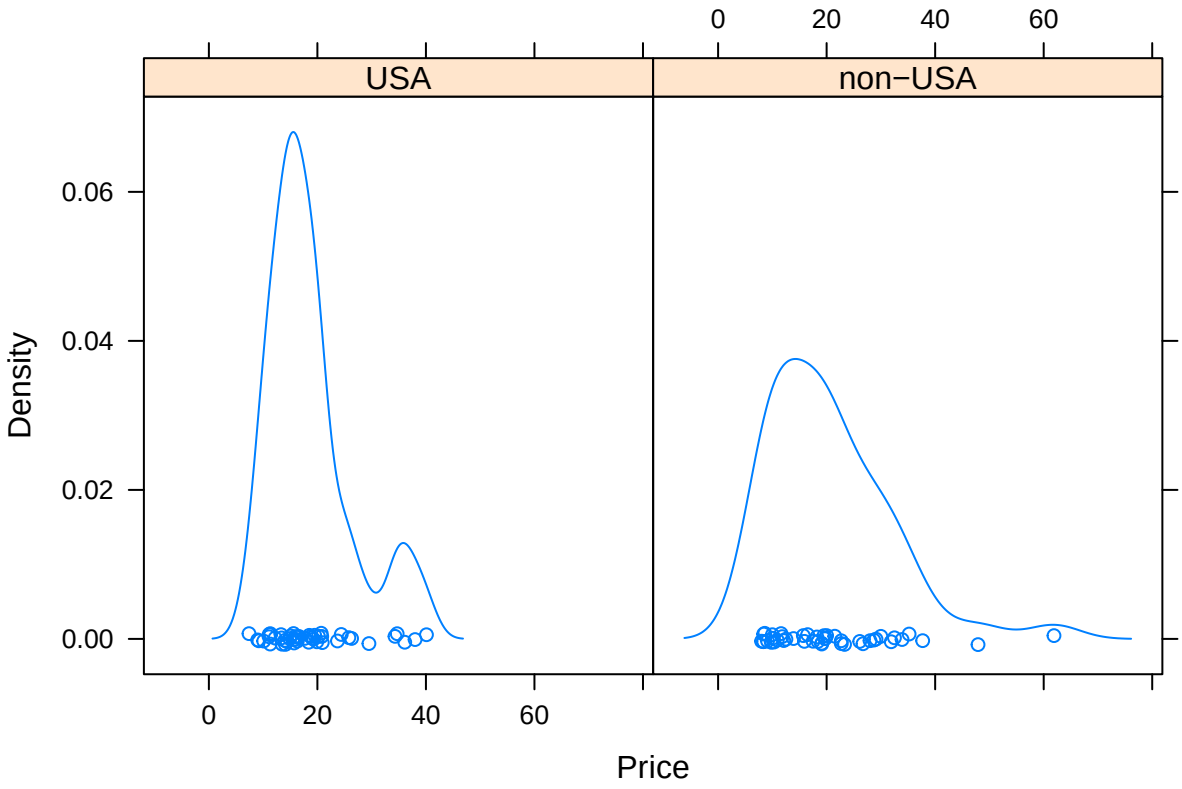

```
bwplot(Price ~ Type | Origin, data = Cars93)
```



```
# We can also make histograms and density plots, though since these do not
# lend well to comparison, we must leave the left side of the formula blank.
histogram(~Price | Origin, data = Cars93)
```




```
densityplot(~Price | Origin, data = Cars93)
```



ggplot2 Plotting

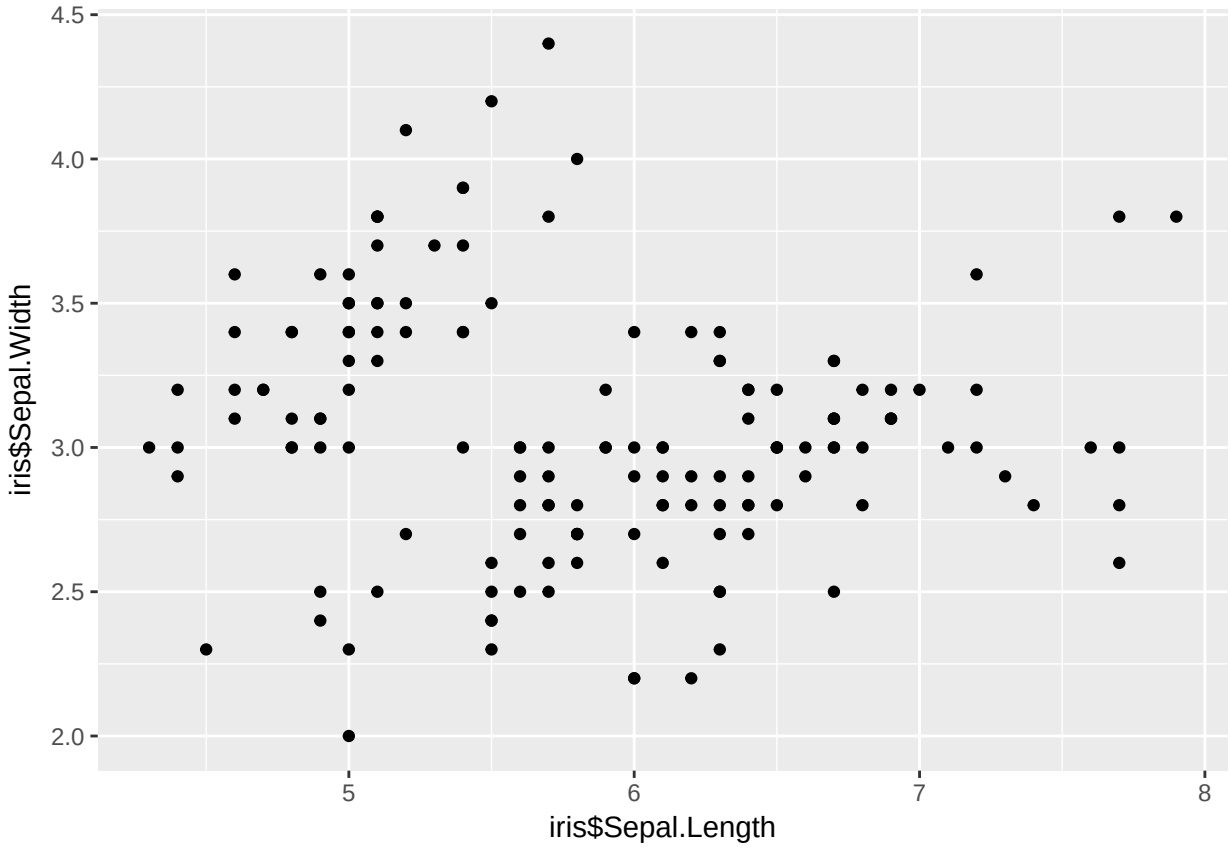
A more recent plotting system than either base R's plotting functions or **lattice** is **ggplot2**, by Hadley Wickham. Again, **ggplot2** builds on existing graphical systems in R (specifically, **grid**, just like **lattice**), but graphics are built using what one may call a mini-language based on Wilkinson's *The Grammar of Graphics*. This means that one does not learn **ggplot2** without additionally learning graphical theory; the two are intertwined here. While it takes some effort to learn **ggplot2**, perhaps more so than **lattice**, once learned, it allows for complex yet visually appealing graphics to be created in a natural way. (**ggplot2** is my preferred system for creating graphics, and the one I used to make the graphics in this report.)

ggplot2 has two primary functions for creating graphics,`qplot()` and `ggplot()`. `qplot()` is intended for making quick plots in a manner similar to `plot()` in base R, though it's not a generic plotting function like `plot()`. `ggplot()` is more involved than `qplot()`, requiring that data be stored in a data frame in order for it to be plotted. That said, `ggplot()` is the go-to plotting function for more involved plots. I will cover both of these functions here.

`qplot()`

The first two arguments passed to `qplot()` are the data to be plotted. For example, I can make a scatterplot with `qplot()` as follows:

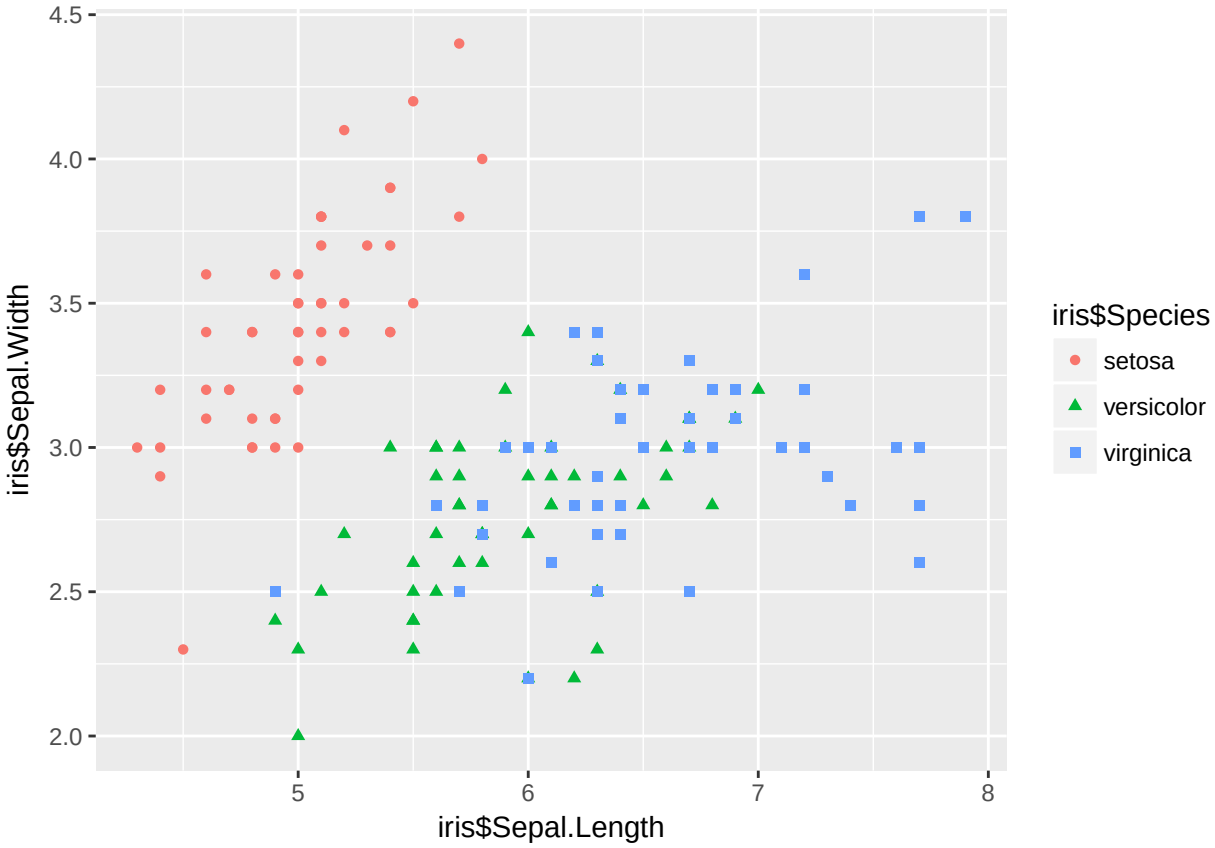
```
library(ggplot2)
qplot(iris$Sepal.Length, iris$Sepal.Width)
```




```
# The $ notation gets annoying after a while. Thankfully, we have a data
# argument to clean things up.
```

It's easier to add complexity to this plot. Additionally, `qplot()` will add a legend automatically (as opposed to the headache of manually adding a legend in base R plotting). Below I color each point based on species, and also change the shape of the point again based on species.

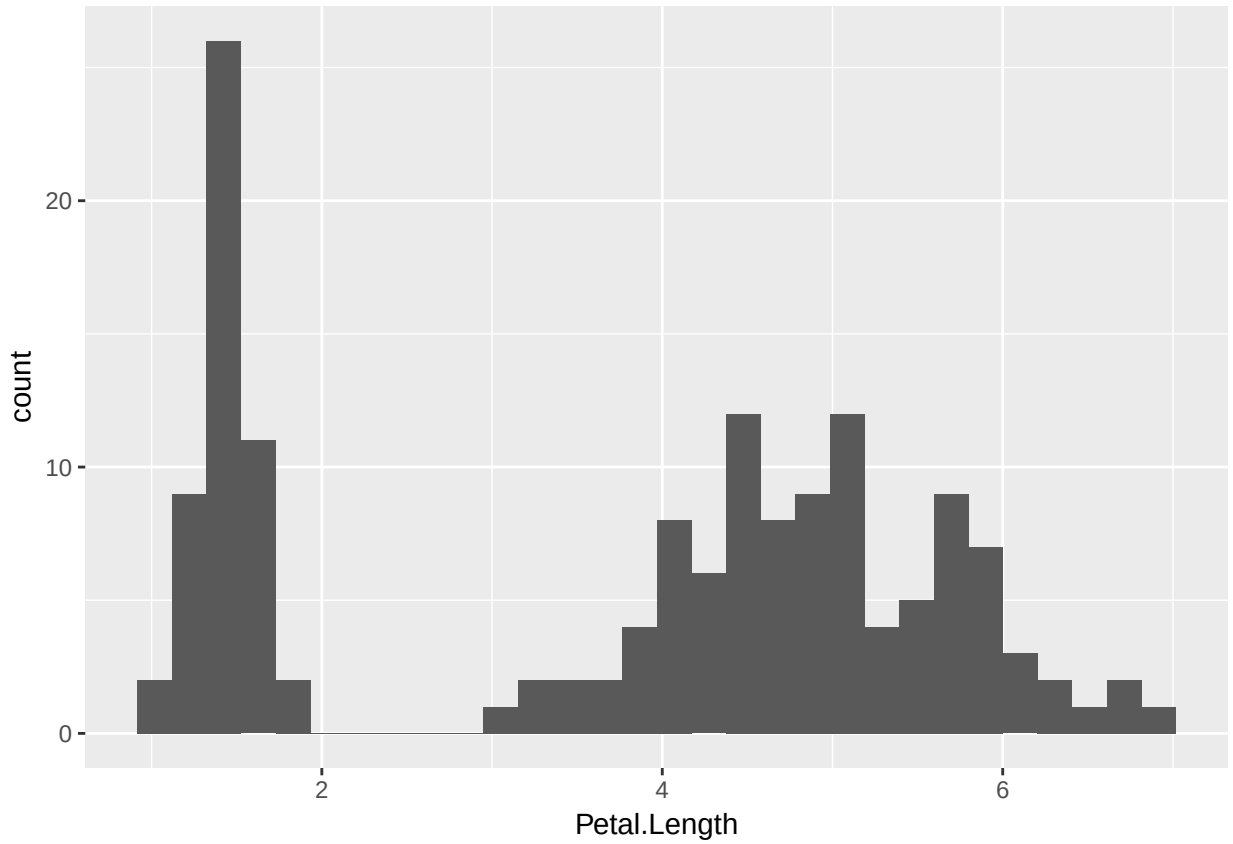
```
qplot(iris$Sepal.Length, iris$Sepal.Width, color = iris$Species, shape = iris$Species)
```



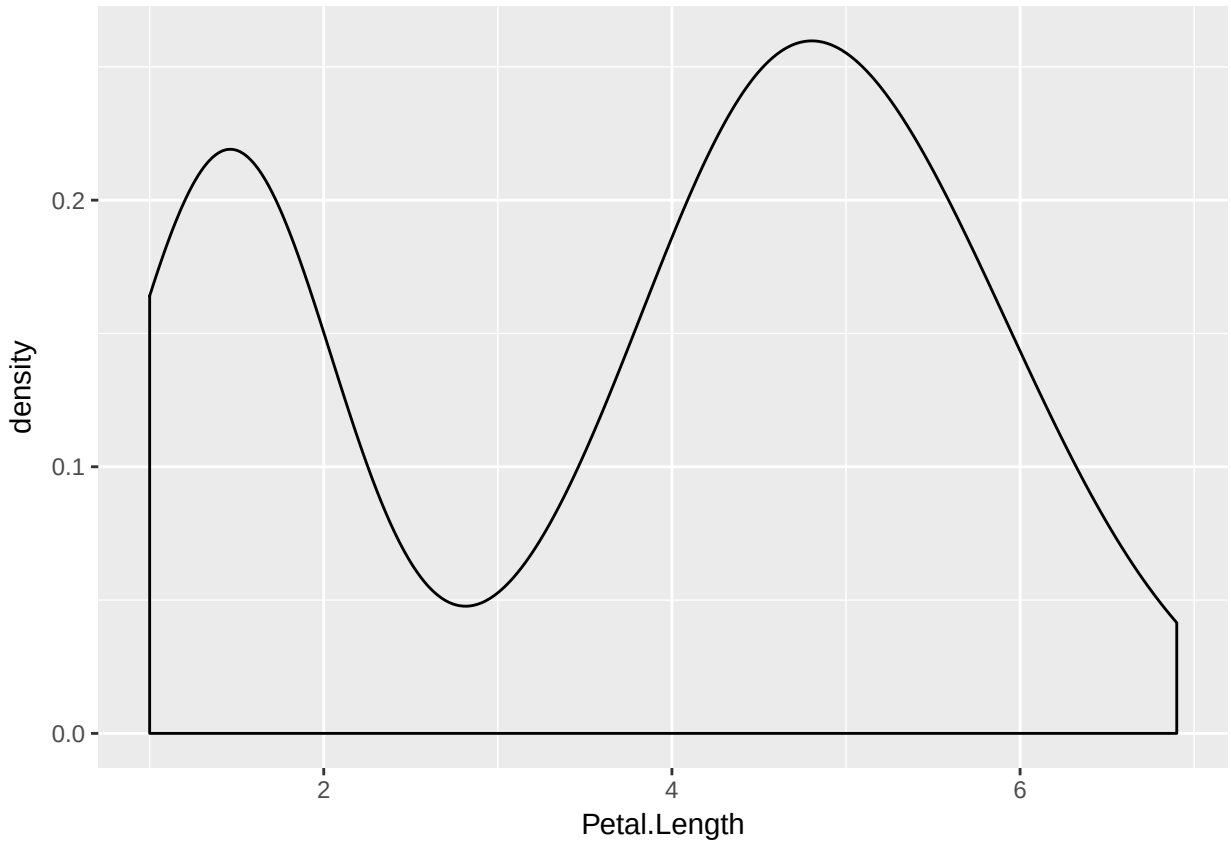
If I want to make a different plot, I need to change the **geom**, the visual channel through which information is communicated, as shown below when making a histogram, density plot, or comparative box plot.

```
# The $ notation gets annoying after a while. Thankfully, we have a data
# argument to clean things up.
qplot(Petal.Length, data = iris, geom = "histogram")
```

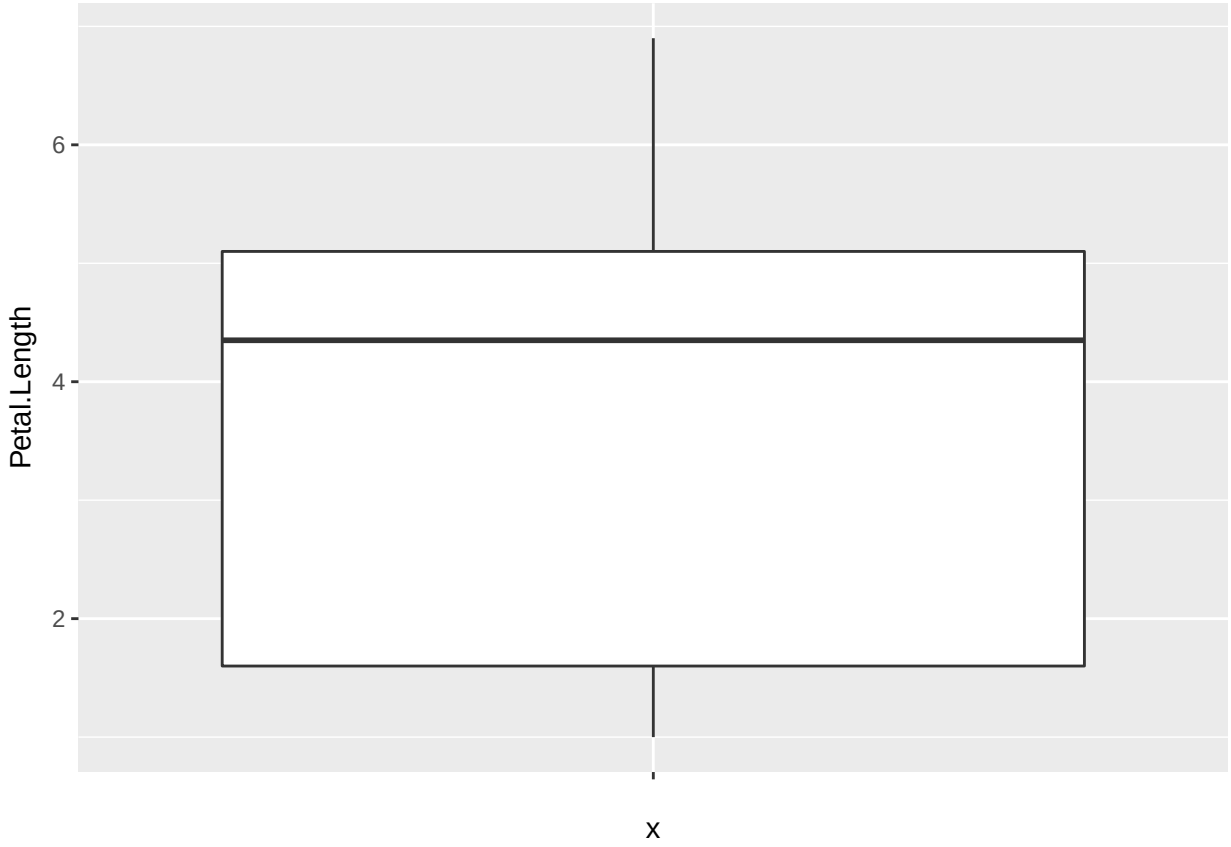
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



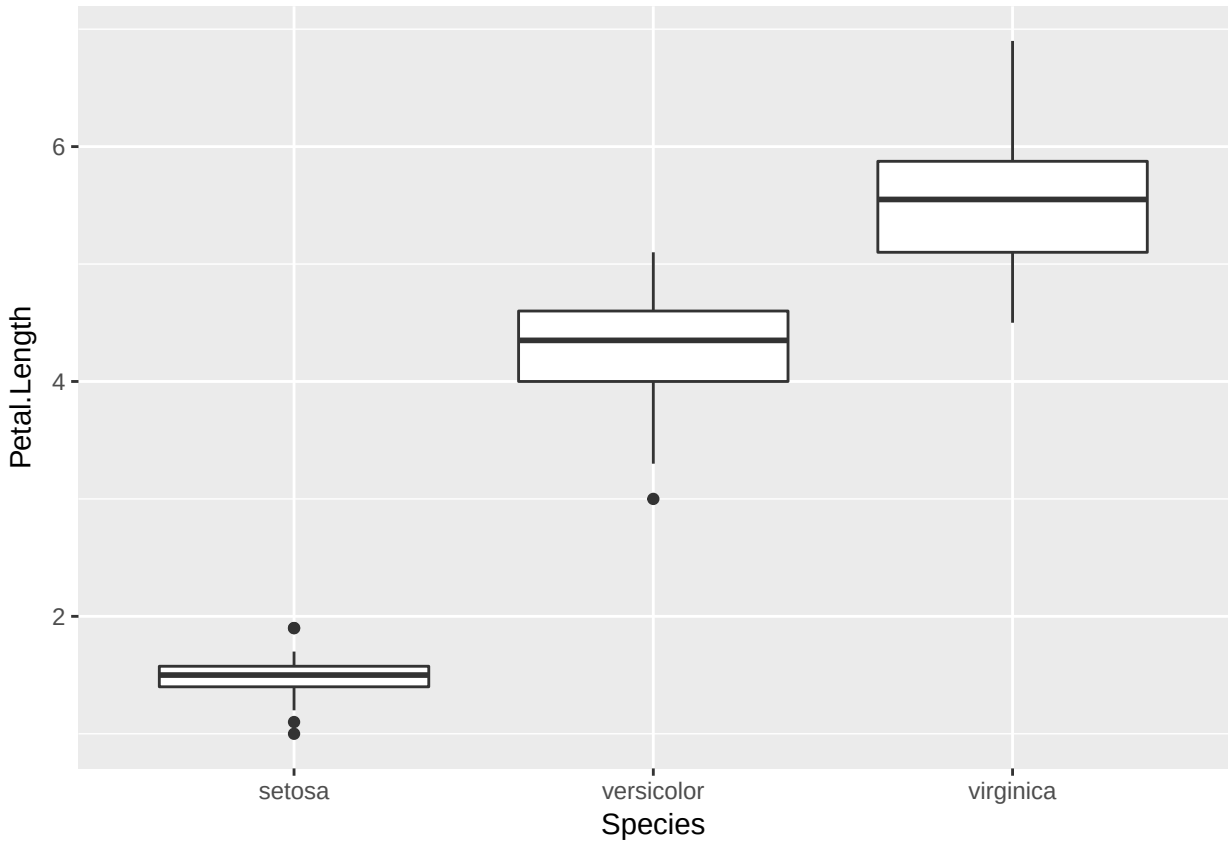

```
qplot(Petal.Length, data = iris, geom = "density")
```




```
# If I only want one box, I could do so with the following;, where group
# goes first (I made a dummy grou, '')
qplot("", Petal.Length, data = iris, geom = "boxplot")
```



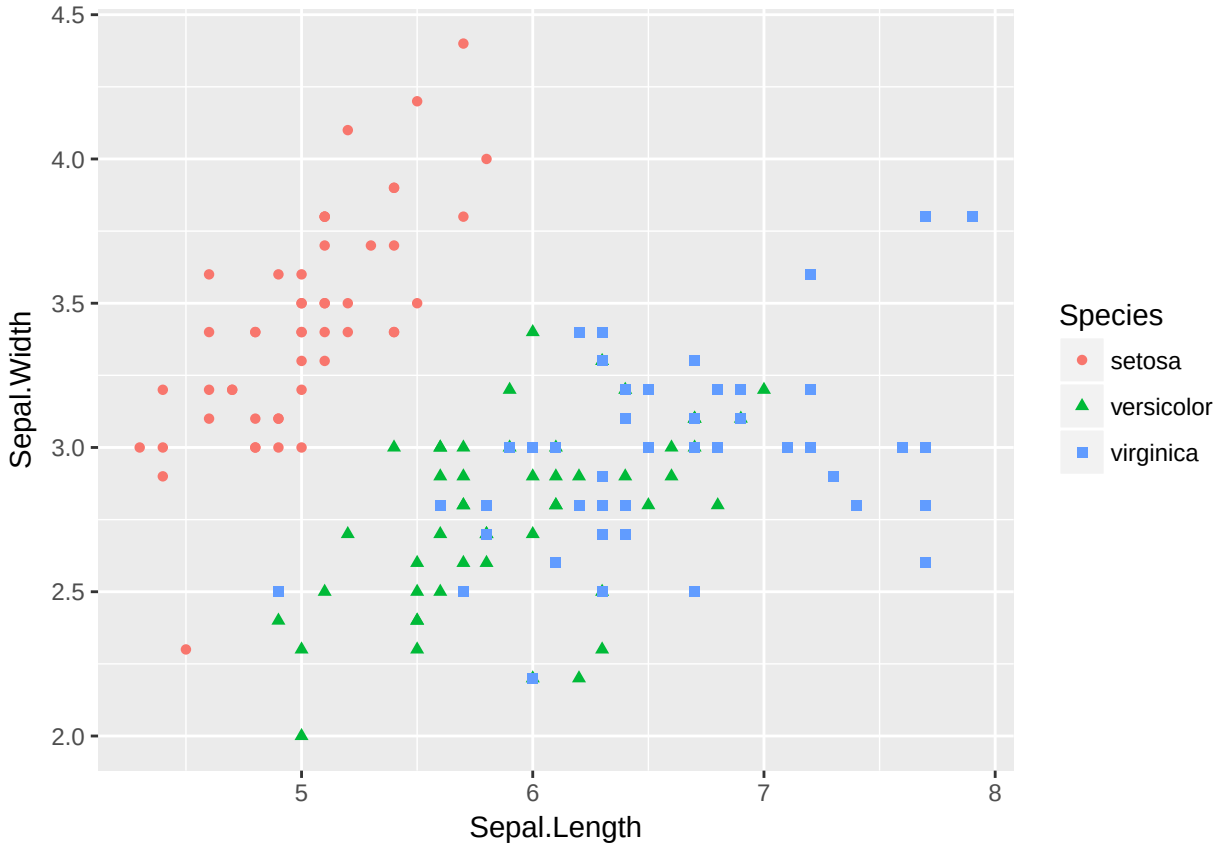

```
# If I want a comparative boxplot, I would do so by specifying population
# first, data second
qplot(Species, Petal.Length, data = iris, geom = "boxplot")
```



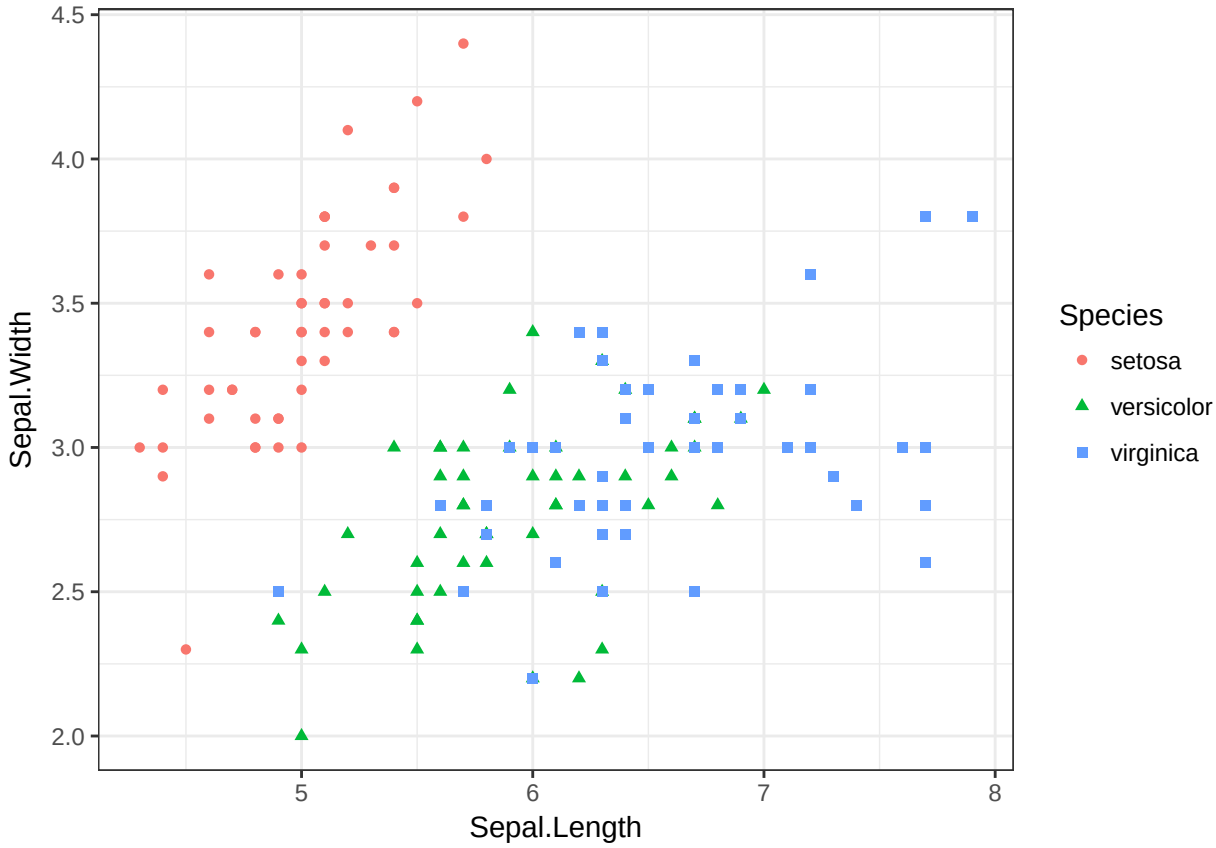
ggplot2 allows graphics to be themed. Color and shape choices are made automatically by default (though they can be changed), and the decision is made based on the current theme being used. Using different themes may lead to different choices in, say, color, being made. This is advantageous since one can worry about how information is being communicated without necessarily worrying about some of the specifics (e.g., we can tell R to communicate groups by color without thinking about what colors are being used).

The default theme used by both `qplot()` and `ggplot()` is `theme_grey()`. Other themes are included in **ggplot2**. The package **ggthemes** includes more themes (often based on graphics created in publications such as *The Economist*, *The Wall Street Journal*, other software packages such as Microsoft Excel, or emulating particular famous authors such as Edward Tufte), and it is possible for you to create your own original theme. You can change a theme by "adding" it to a graphic. (**ggplot2** overloads the `+` operator to have a particular meaning when "adding" things to a `gg`- or `ggplot`-class object).

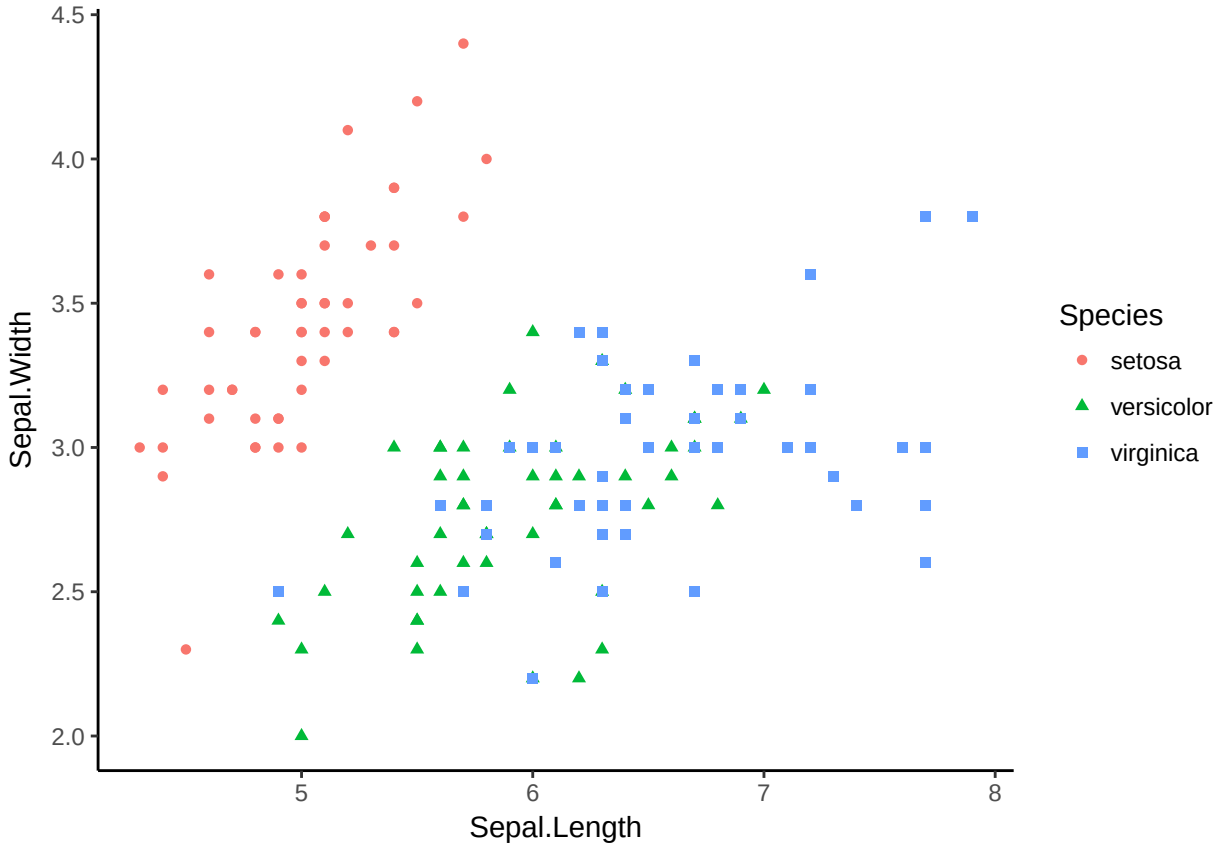
```
# Unlike with base R plots or lattice graphics, we can store plots made with
# ggplot in objects, as I demonstrate below:
p <- qplot(Sepal.Length, Sepal.Width, data = iris, color = Species, shape = Species)
# We can view the plot either with print(p) or by calling the object
# directly (i.e. just p).  The default theme_grey() theme
print(p)
```



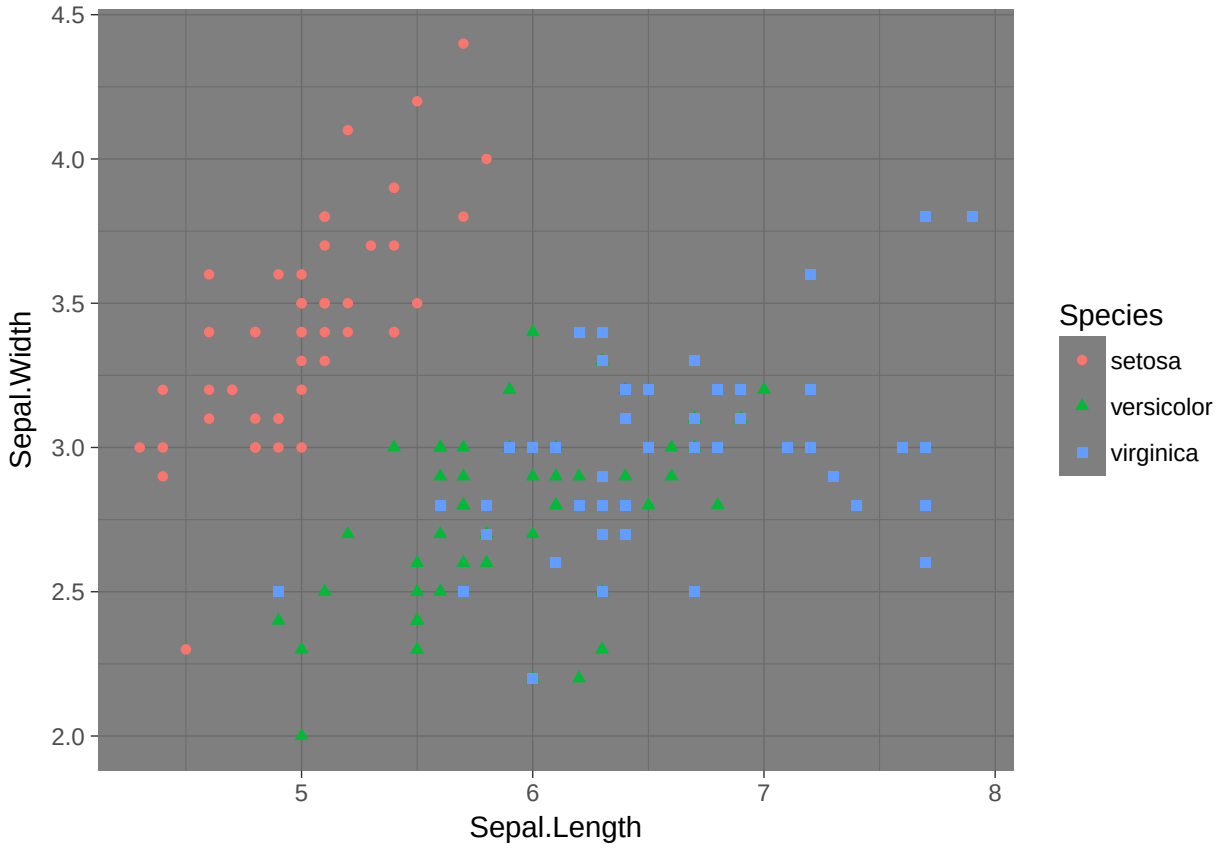

```
# Other themes:
p + theme_bw()
```



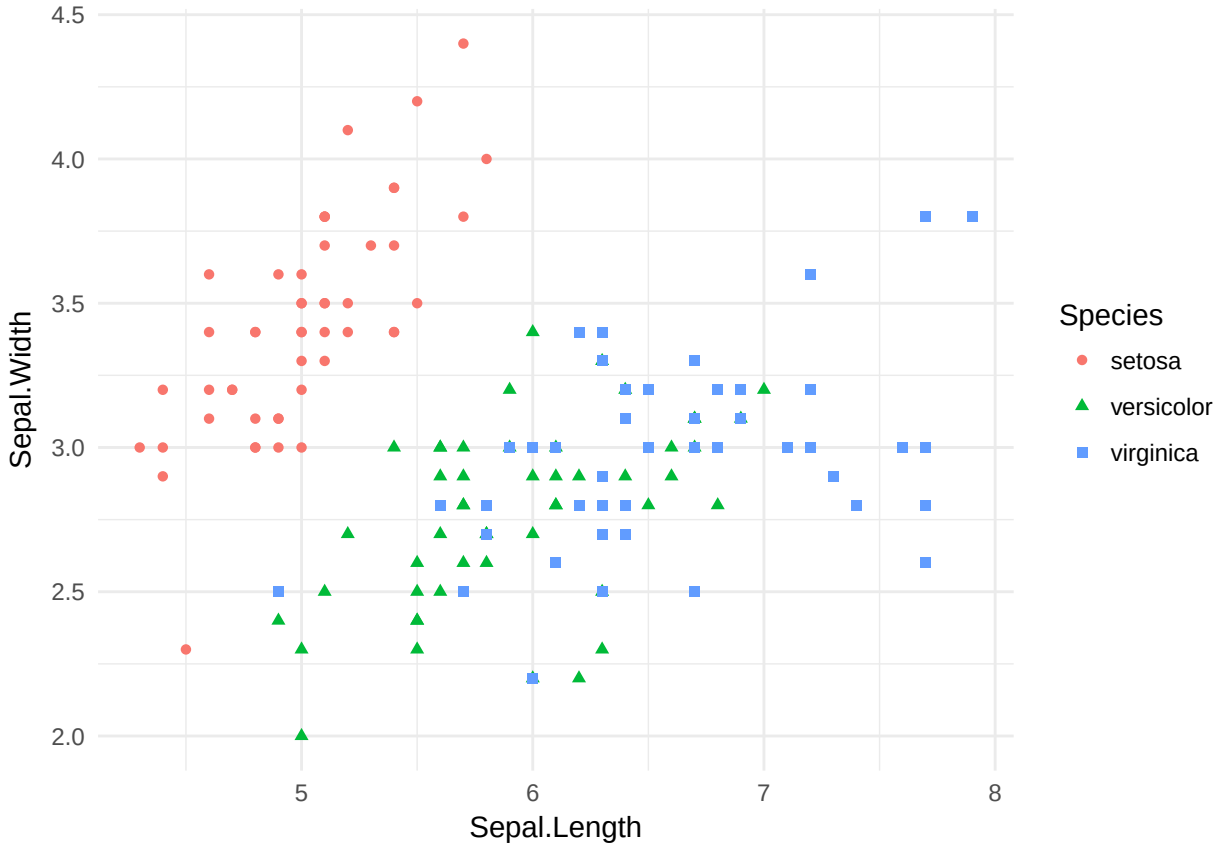

```
p + theme_classic()
```


4.5
4.0
3.5
3.0
2.5
2.0
Sepal.Width
5
6
7
8
Sepal.Length
Species
setosa
versicolor
virginica

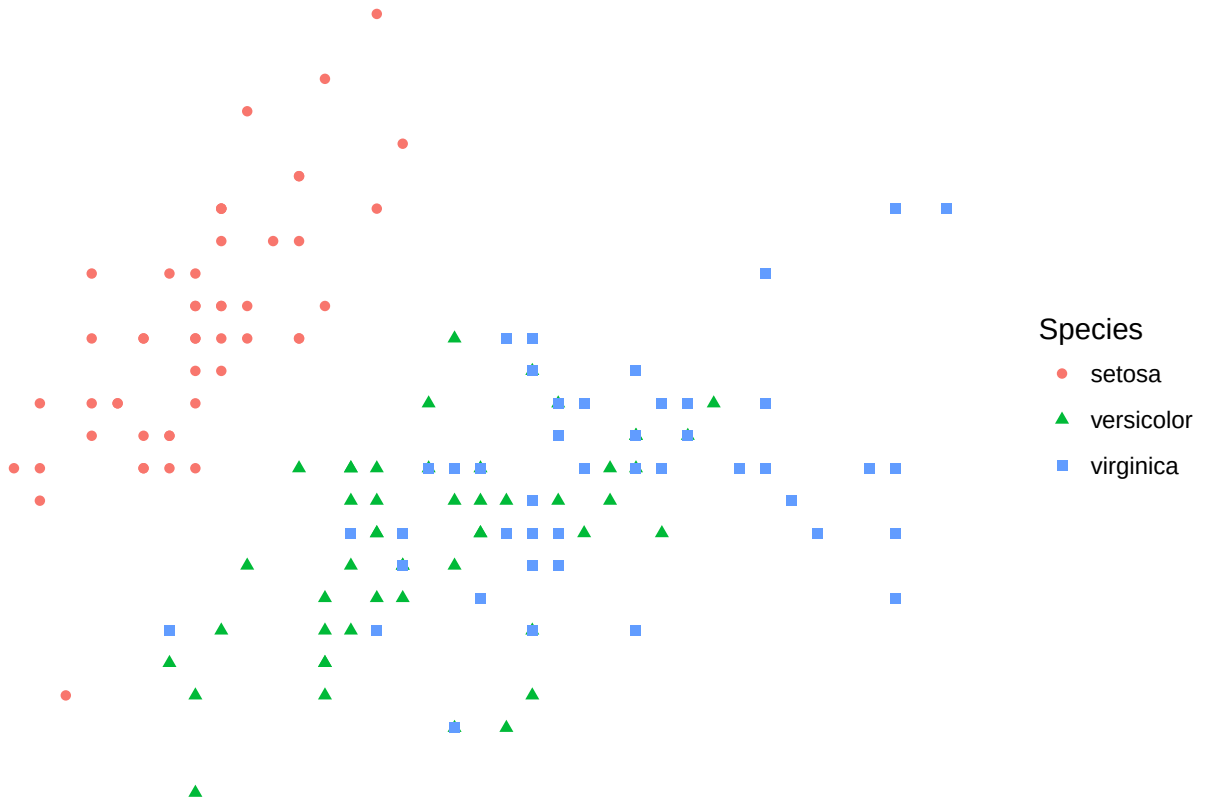
```
p + theme_dark()
```




```
p + theme_minimal()
```


4.5
4.0
3.5
3.0
2.5
2.0
Sepal.Width
5
6
7
8
Sepal.Length
Species
setosa
versicolor
virginica

```
p + theme_void()
```


Species
setosa
versicolor
virginica

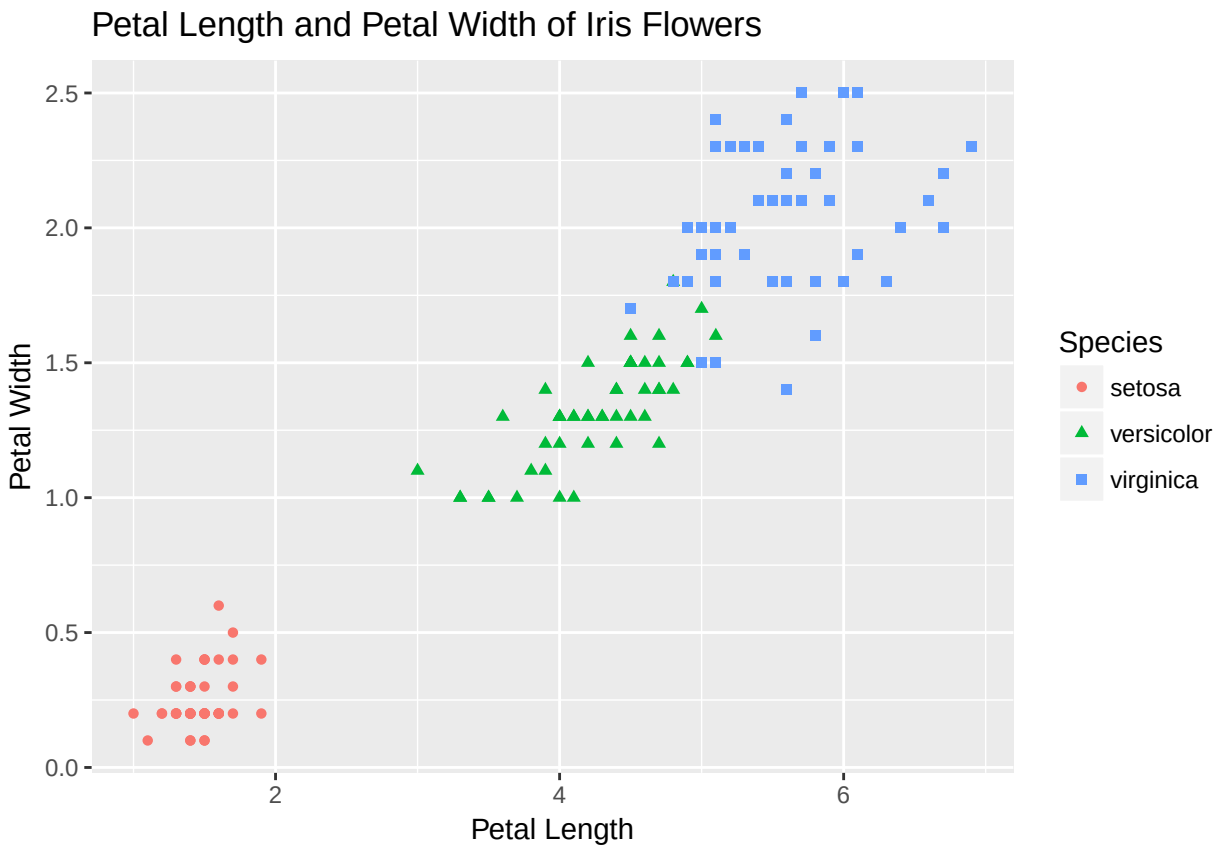
ggplot()

`ggplot()` is the main function of **ggplot2**, with `qplot()` being merely a simplified version of `ggplot()`. Unlike `qplot()`, which can accept data in the form of vectors, data passed to `ggplot()` *must be a data frame!* This restriction allows `ggplot()` to create graphics in a consistent way (you can even design a graphic on one data set, then simply swap that data set out with another and have the graphic still work.)

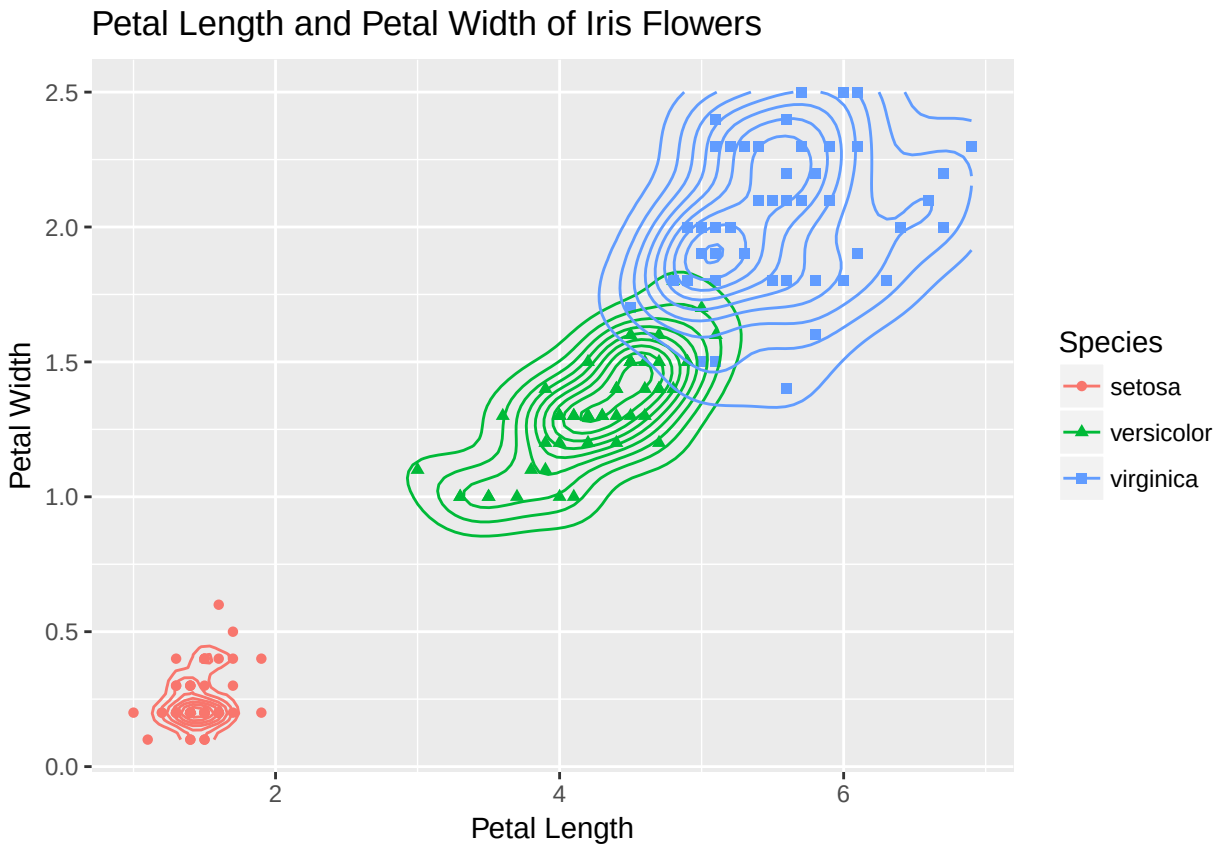
Three important building blocks for building graphics using `ggplot()` (aside from theme elements, like specific colors used, labels, or titles) are **aesthetics**, **geoms**, and **stats**. Aesthetics, controlled by the `aes()` function, determine the visual channels through which information is transmitted (position, color, shape, etc.). Geoms determine how visual information is rendered; in other words, it translates aesthetics into graphics. A few functions, such as `geom_point()`, `geom_line()`, and many others, "add" geoms to a graphic. Finally, stats allow data summaries, such as histograms or density plots, to be created and then drawn, and come in the form of functions such as `stat_summary()`, `stat_quantile()`, and many others.

Let's visualize the `iris` data set, this time using `ggplot()`.

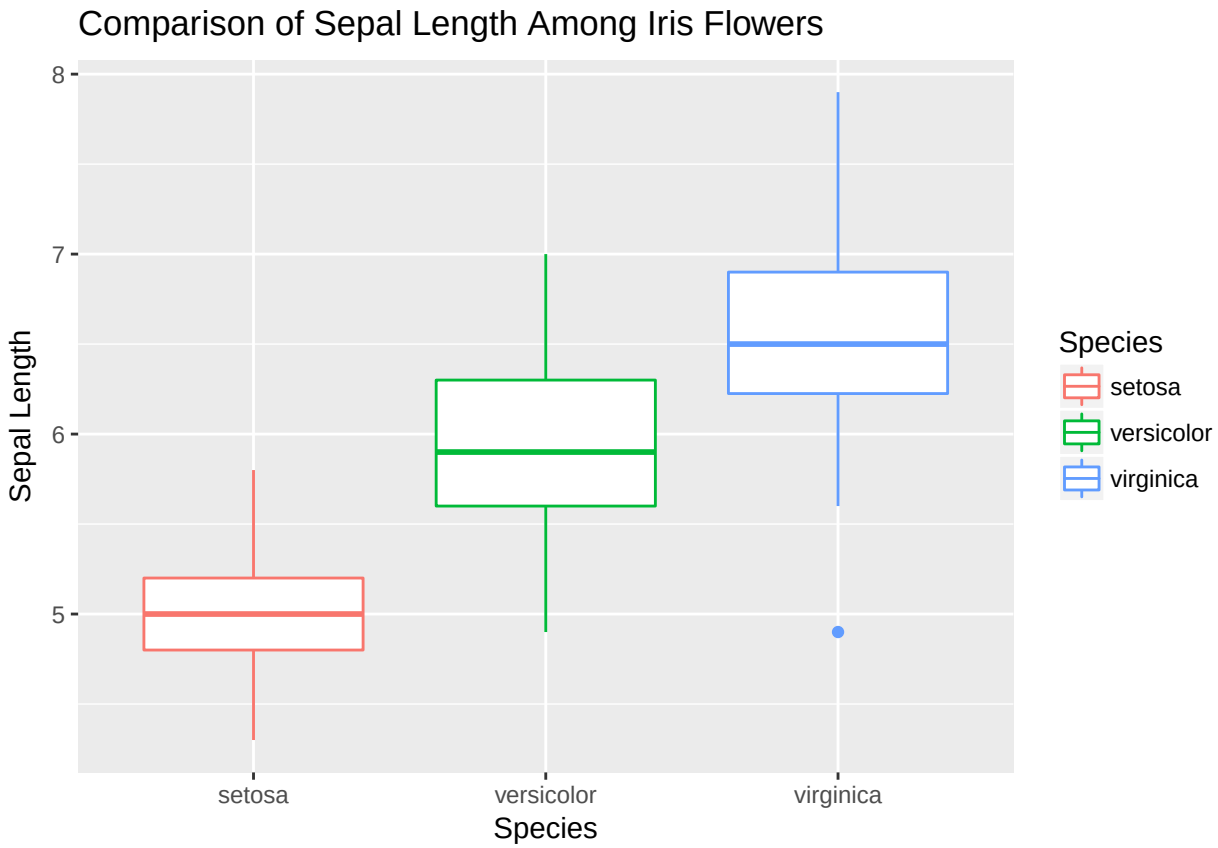
```
# Notice that I layer on geoms with +, which has been overloaded to work appropriately for ggplot2 obje
p <- ggplot(iris) +   # Create the basic plot object
  geom_point(aes(x = Petal.Length, y = Petal.Width, shape = Species, color = Species)) +   # Create a d
  xlab("Petal Length") +  # Add axis labels
  ylab("Petal Width") +
  ggtitle("Petal Length and Petal Width of Iris Flowers")
# Let's see the result!
print(p)
```



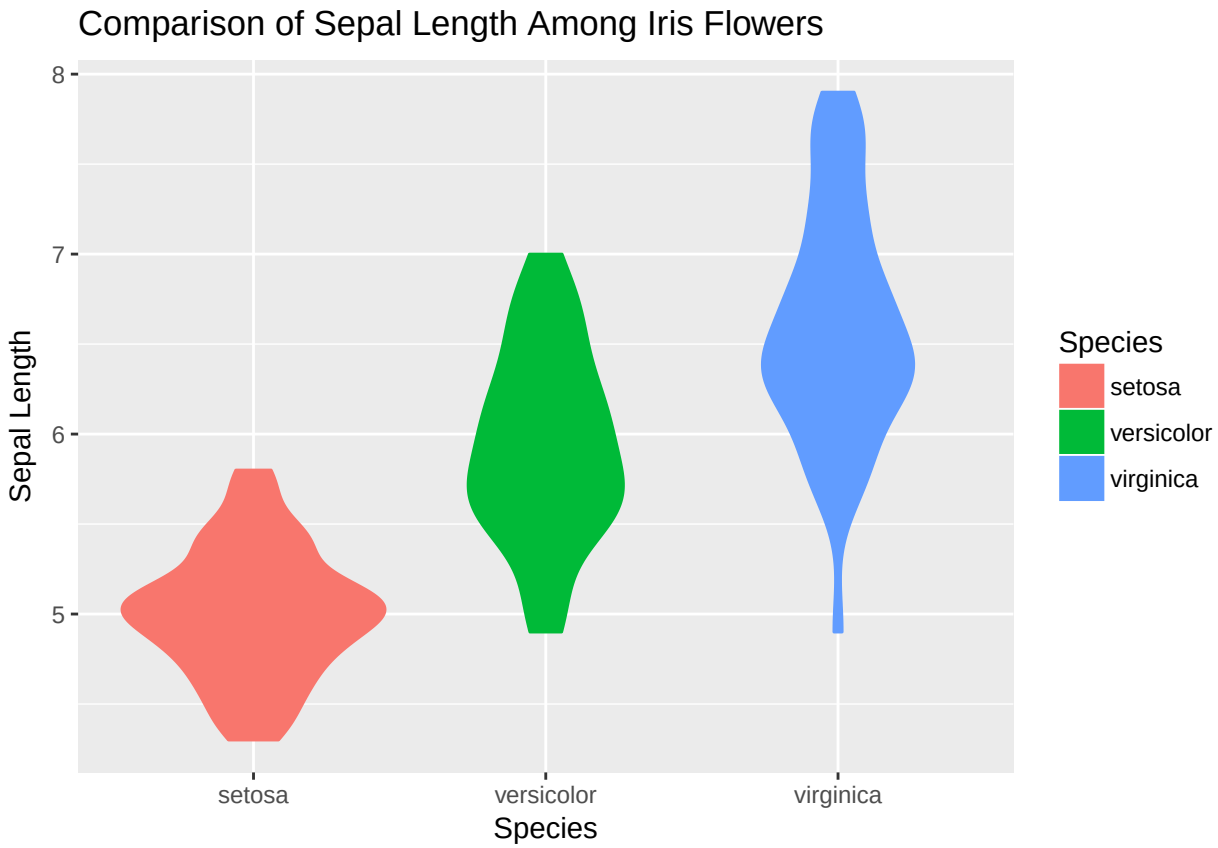

```
# Let's add a stat that creates a 2D density plot
p + stat_density2d(aes(x = Petal.Length, y = Petal.Width, group = Species, color = Species))
```



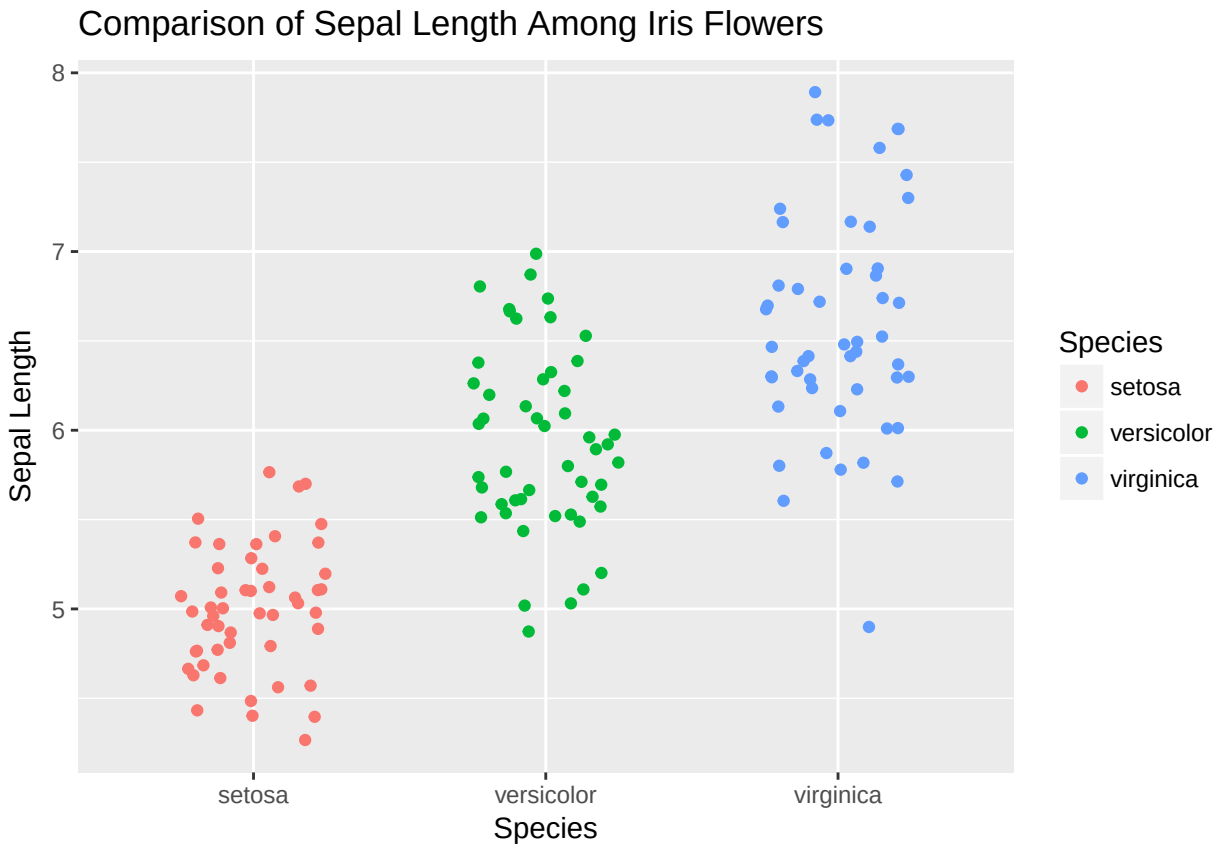

```
# Now let's look just at iris sepal length. We initialize a basic object and swap out geoms to look at
# First, create the basic object:
q <- ggplot(iris, aes(y = Sepal.Length, x = Species, color = Species)) +
  xlab("Species") +
  ylab("Sepal Length") +
  ggtitle("Comparison of Sepal Length Among Iris Flowers")
# I first look at a boxplot
q + geom_boxplot()
```



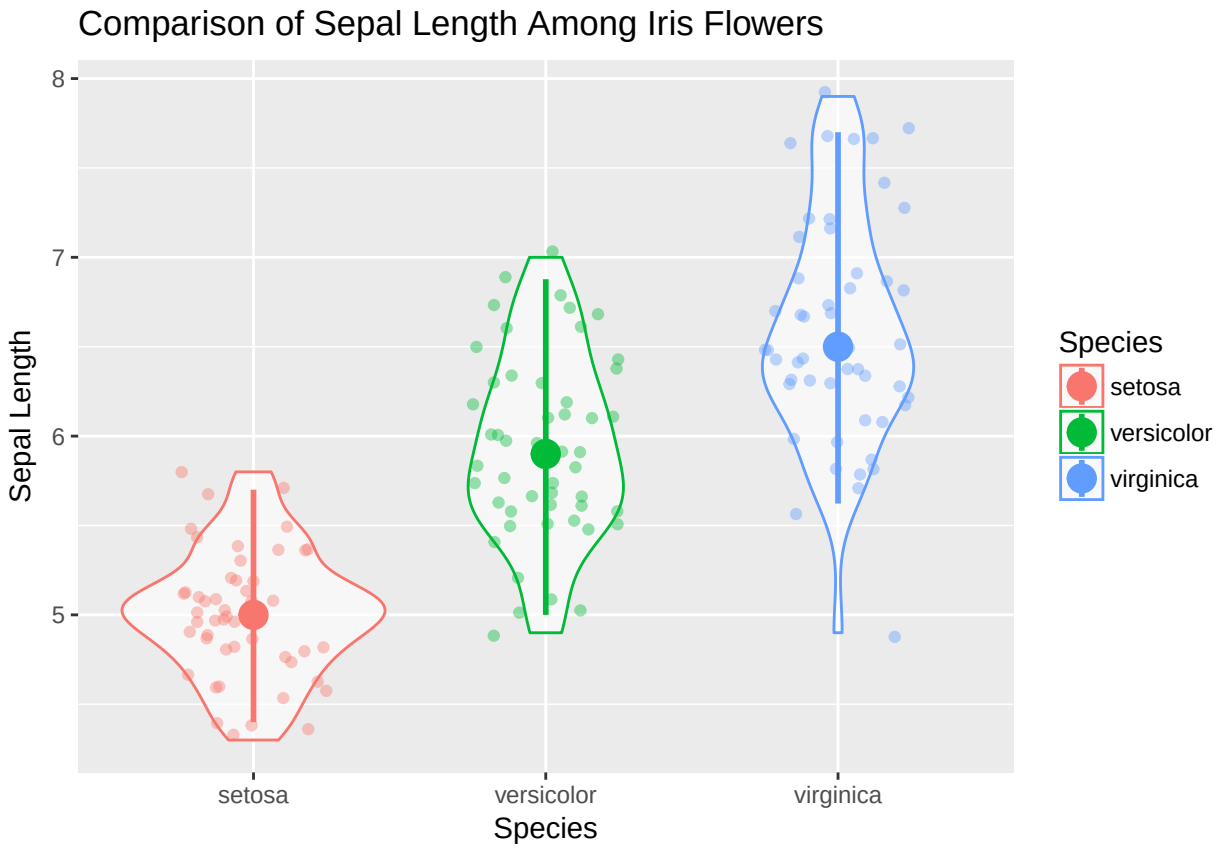

```
# An alternative to the boxplot is the violin plot
q + geom_violin(aes(fill = Species))
```




```
# Or we can plot jittered data
q + geom_jitter(width = .25)
```



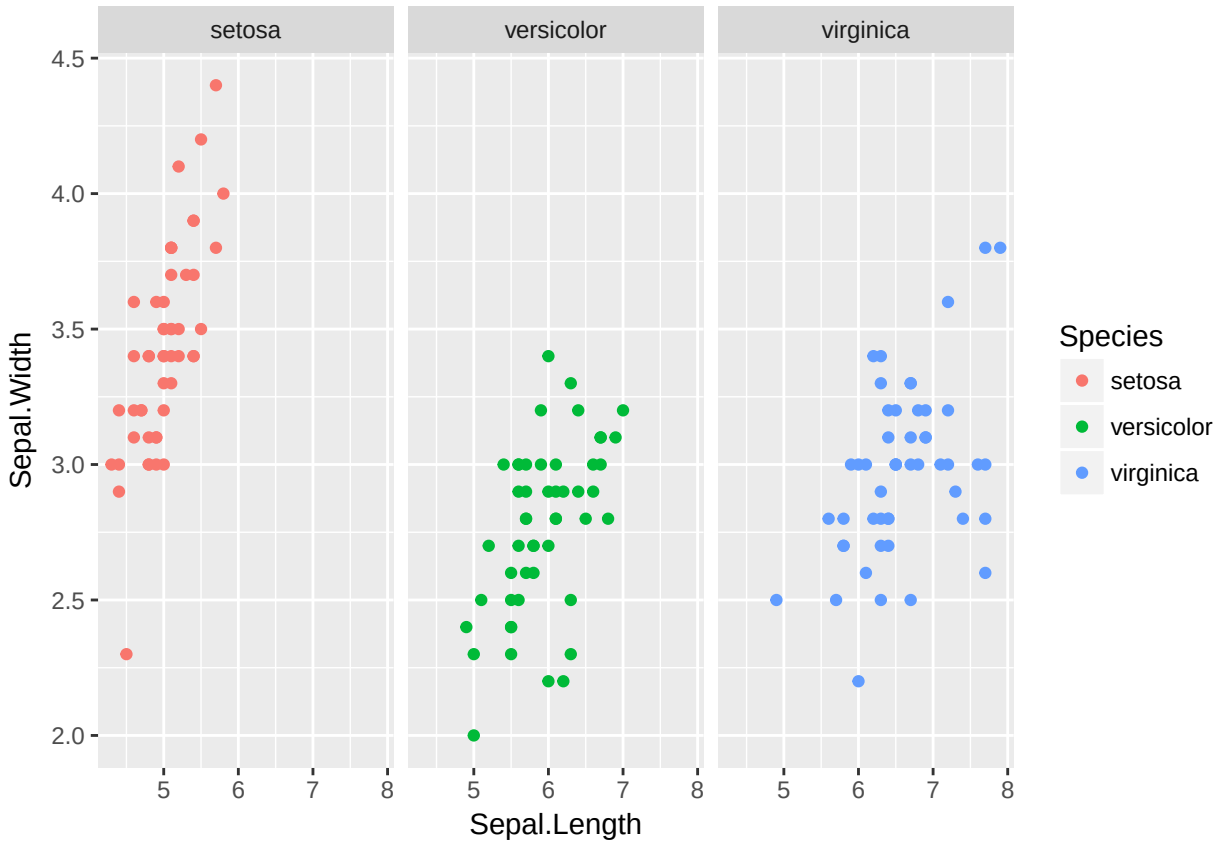

```
# Here is a more complex, layered graphic
q + geom_violin(alpha = .6) +
  geom_jitter(width = .25, alpha = .4) +
  stat_summary(size = 1, fun.data = median_hilow)
```



Graphics that would be very difficult to make in base R or even **lattice** are almost effortless in **ggplot2**!

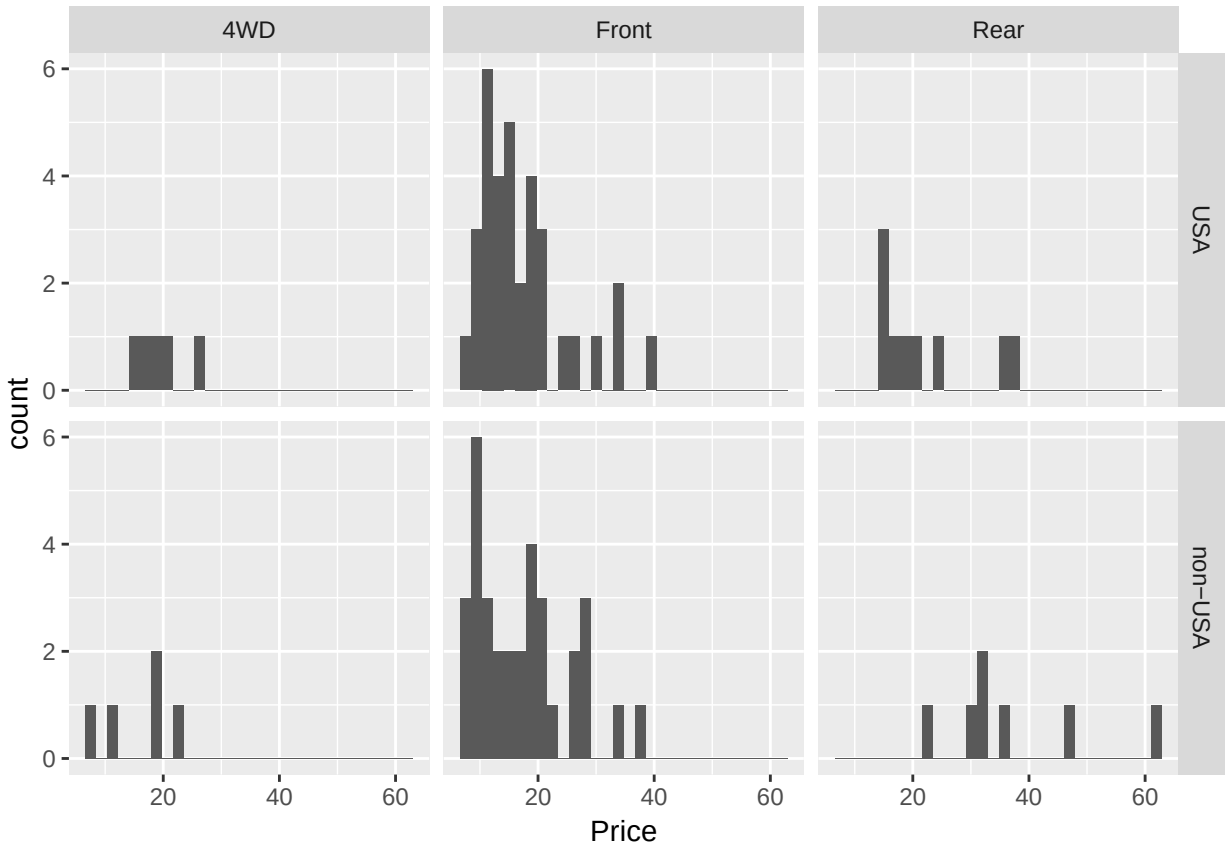
Like **lattice**, **ggplot2** allows for splitting plots based on a variable. This is called **faceting**, and is controlled primarily by the `facet_grid()` function, added to a plot like any other function in **ggplot2**. `face_grid(y ~ x)` will break plots up according to the categorical variables `x` and `y`, where each row represents a different value for `x` and each column a different value for `y`. If you don't wish to facet according to two variables, replace either `x` or `y` with a `.`, like `. ~ y`. I demonstrate faceting first on `iris`, then on `Cars93`.

```
# Create a plot as normal
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) + geom_point() +
    # Create different plots for different species
facet_grid(. ~ Species)
```



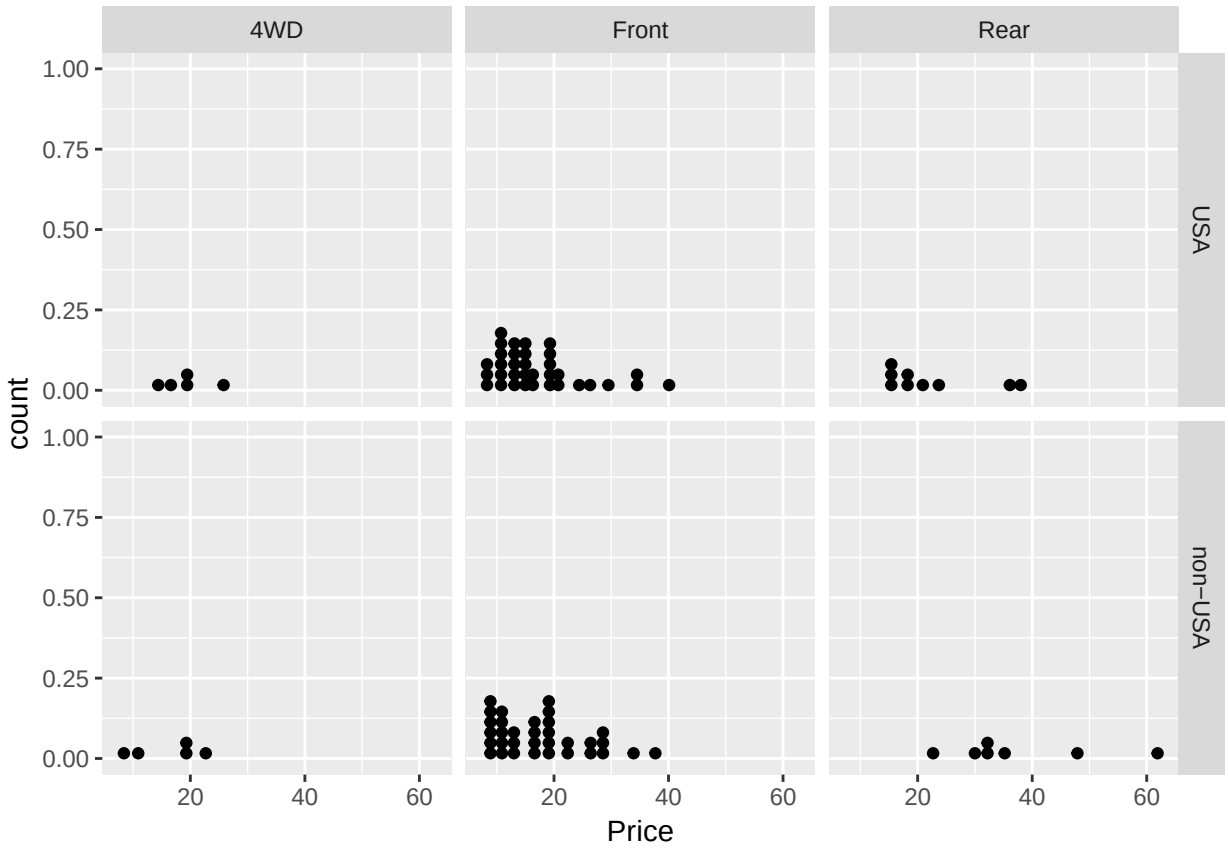

```
# Create a facet grid for price of cars depending on origin and drive train
p1 <- ggplot(Cars93, aes(x = Price))
p1 + geom_histogram() + facet_grid(Origin ~ DriveTrain)
```

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

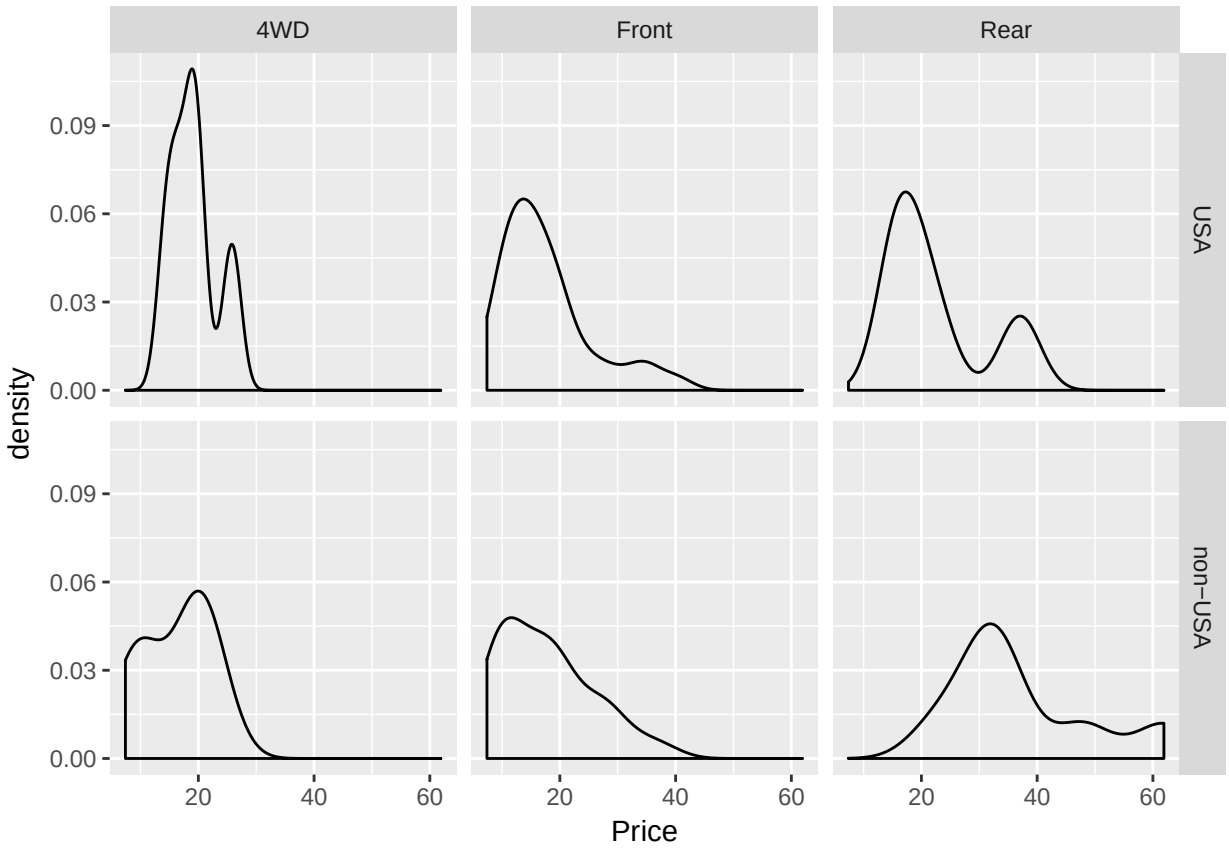



```
# We could try other plots as well
p1 + geom_dotplot() + facet_grid(Origin ~ DriveTrain)
```

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




```
p1 + geom_density() + facet_grid(Origin ~ DriveTrain)
```


4WD
Front
Rear
USA
non−USA
density
0.09
0.06
0.03
0.00
20
40
60
Price

Lecture 7

Distribution Comparison

A common problem is determining whether two distributions of two samples are the same. While statistical tests can help answer this question, visualization techniques are also quite effective. Many of the plots we have seen can be adapted to compare distributions from independent samples.

One way to do so is to create two stem-and-leaf plots back-to-back, sharing common stems but having leaves from different data sets extending out in different directions. Base R will not do this, but the `stem.leaf.backback()` function in the **aplpack** package can create such a chart. `stem.leaf.backback(x, y)` will plot the distributions of the data in vectors `x` and `y` with a back-to-back stem-and-leaf plot. Here we use this function to examine the distribution of tooth lengths of guinea pigs given different supplements, contained in the `ToothGrowth` data set.

```
library(aplpack)
```

```
## Loading required package: tcltk
```

```
str(ToothGrowth)
```

```
## 'data.frame':    60 obs. of  3 variables:
##  $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
##  $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
##  $ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
```

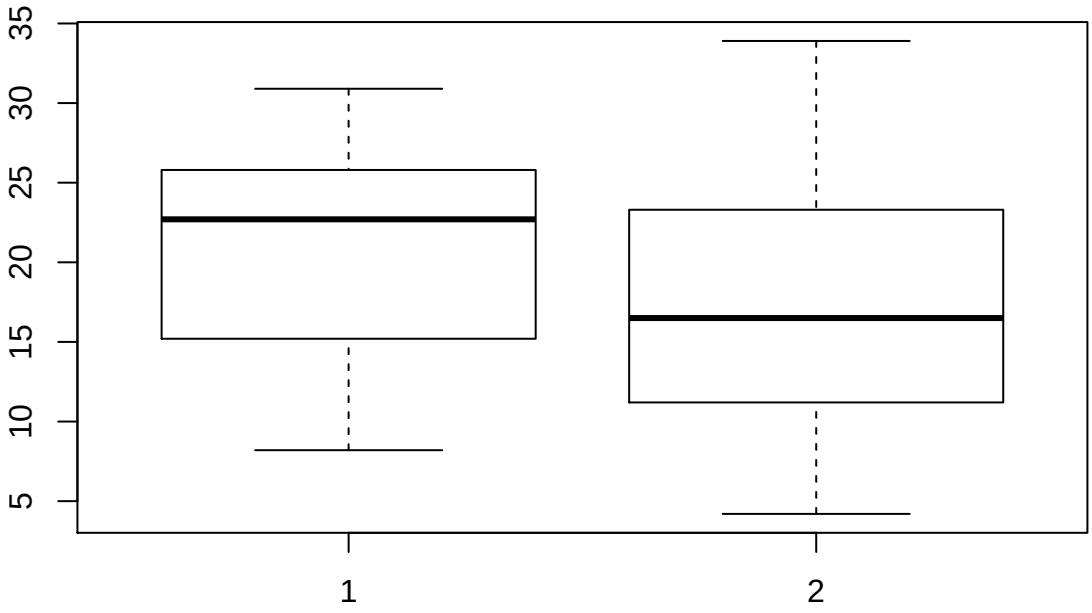
```
# The split function can split data vectors depending on a factor. Here,
# split will split len in ToothGrowth depending on the factor variable supp,
# creating a list with the two variables we want
len_split <- with(ToothGrowth, split(len, supp))
OJ <- len_split$OJ
VC <- len_split$VC
stem.leaf.backback(OJ, VC, rule.line = "Sturges")
```

```
## ----------------------------------
##   1 | 2: represents 12, leaf unit: 1
##               OJ      VC
## ----------------------------------
##                 | 0* |4           1
##    4        9998| 0. |55677       6
##    7         440| 1* |011134     12
##   11        9765| 1. |55667788   (8)
##   (9)  443332110| 2* |1233       10
##   10   977666555| 2. |5669        6
##    1           0| 3* |23          2
##                 | 3. |
```

```
##                 | 4* |
## ____________________________________
## n:            30      30
## ____________________________________
```

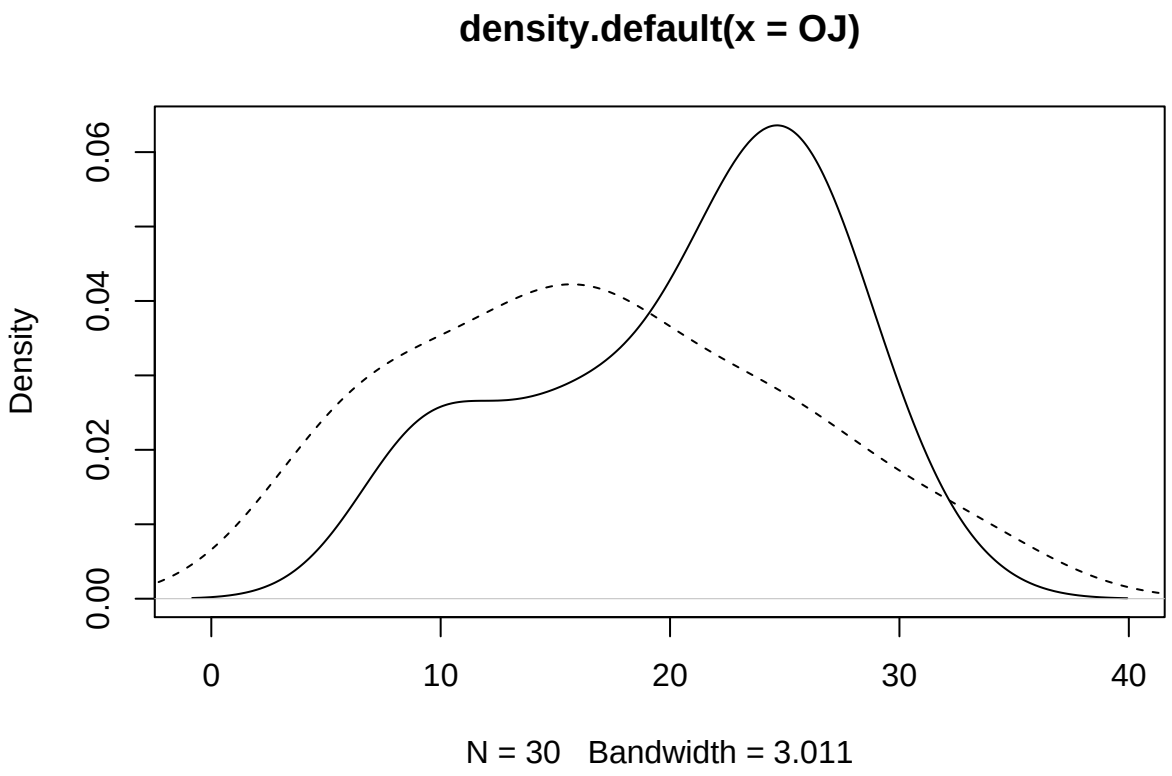
We have seen comparative boxplots before; again, they can be quite useful for comparing distributions. Calling `boxplot(x, y)` with two data vectors `x` and `y` will compare the distributions of the data in the vectors `x` and `y` with a comparative boxplot.

```
boxplot(OJ, VC)
```



We can also use density estimates to compare distributions, like so:

```
plot(density(OJ), lty = 1)
lines(density(VC), lty = 2)
```

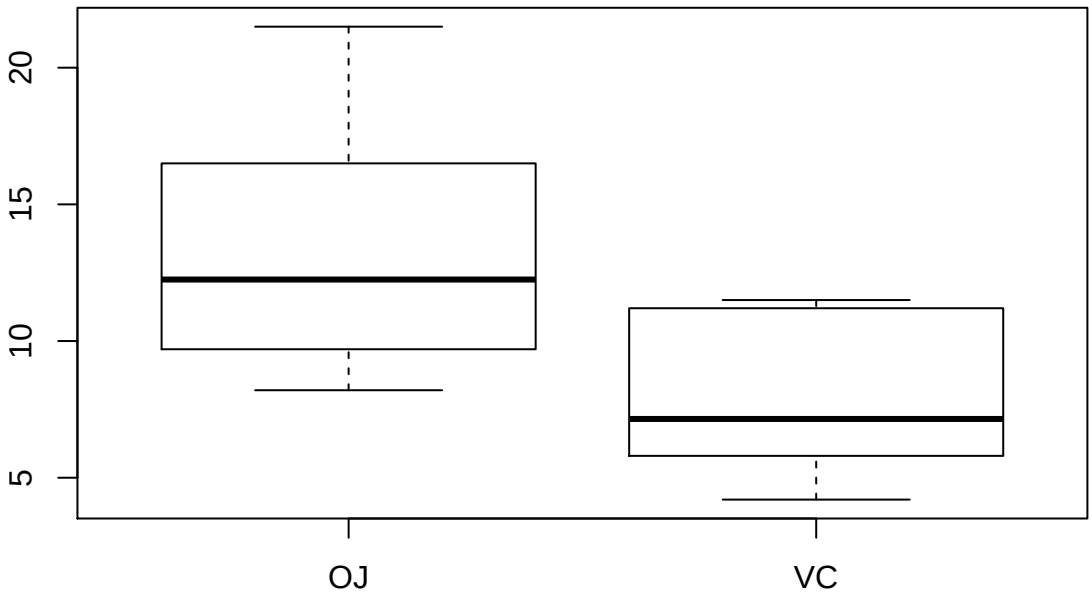


Model Formula Interface

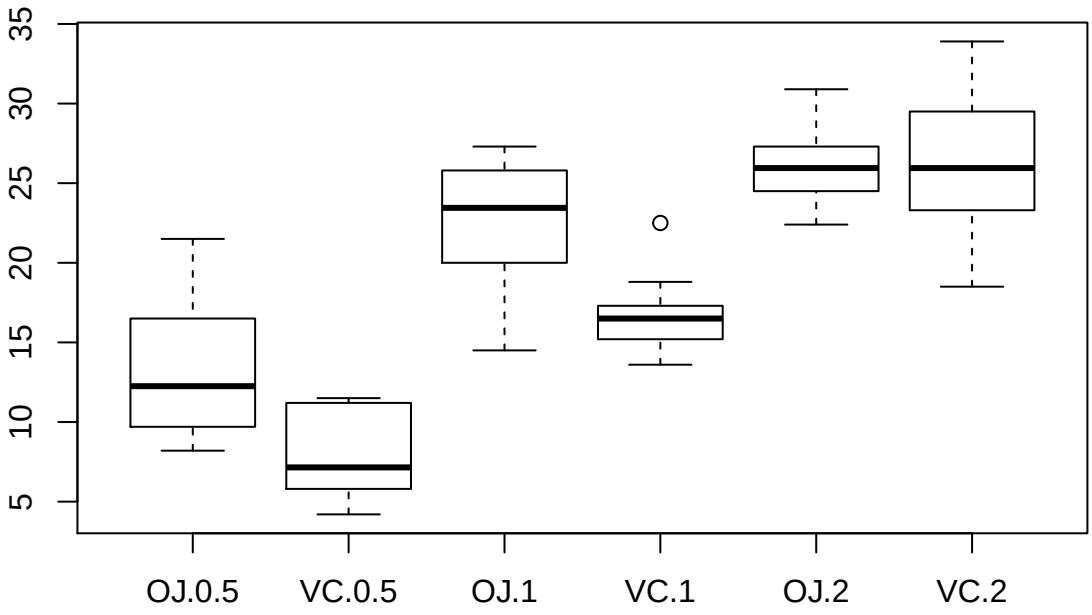
You have seen this interface before, and now we discuss it in more detail. The interface is `y ~ x`, or more descriptively, `response ~ predictor`. Loosely, we see the response variable as being dependent on the predictor, which could by a single variable, as in `y ~ x`, or a combination of variables, as in `y ~ x + z`. (`+` is overloaded to have a special meaning in the model formula interface, and thus does not necessarily mean "addition". If you wish to have `+` mean literal "addition", use the function `I()`, as in `y ~ I(x + z)`.) Many function in R use the formula interface, and often include additional arguments such as `data` (for specifying a data frame containing the data and variables described by the formula) and `subset` (used for selecting a subset of the data, according to some logical rule). Functions that use this formula interface include `boxplot()`, `lm()`, `summary()`, and **lattice** plotting functions.

Below I demonstrate using `boxplot()`'s formula interface for exploring the `ToothGrowth` data more simply.

```
# Here, I plot the tooth growth data depending on supplement when dose ==
# 0.5
boxplot(len ~ supp, data = ToothGrowth, subset = dose == 0.5)
```




```
# I can create a boxplot that depends on both supplement and dosage
boxplot(len ~ supp + dose, data = ToothGrowth)
```



Here I compare means of tooth lengths using the formula interface in `summary()`, provided in the package **Hmisc**.

```
library(Hmisc)
```

```
## Loading required package: lattice
```

```
## Loading required package: survival
```

```
## Loading required package: Formula
```

```
## Loading required package: ggplot2
```

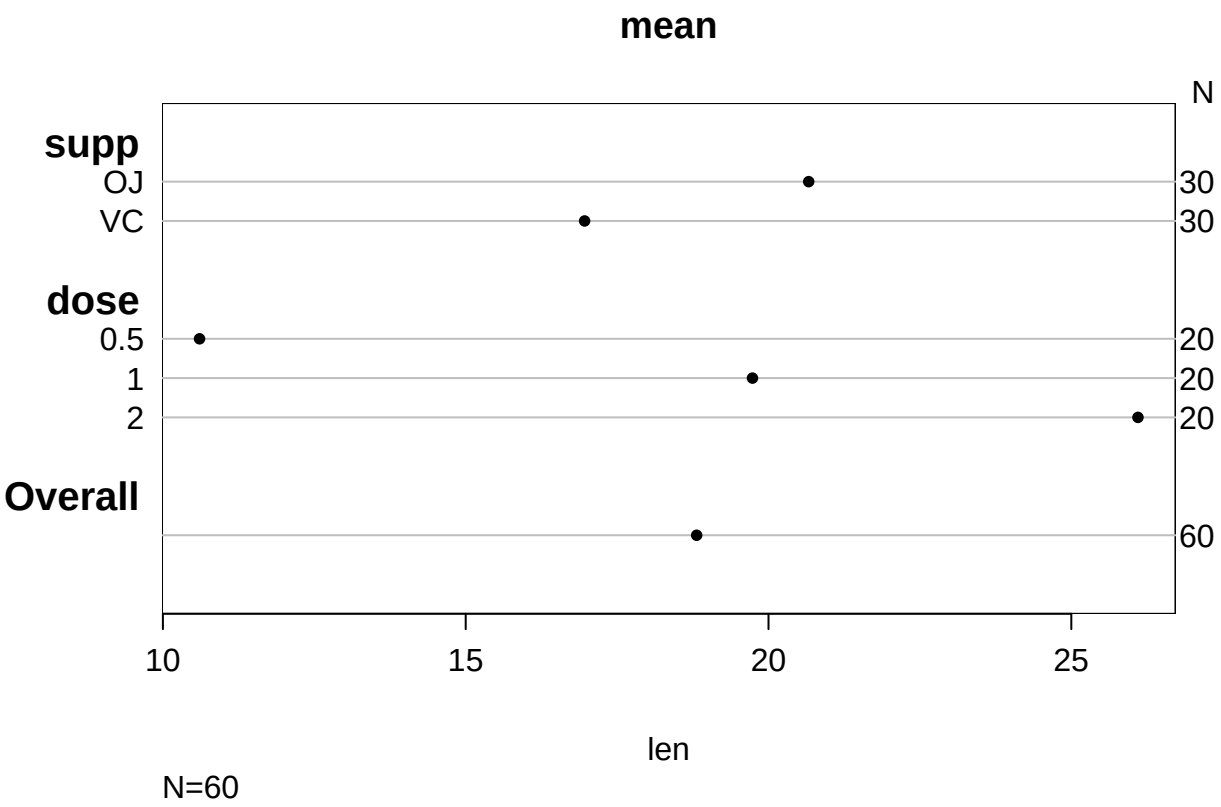
```
##
## Attaching package: 'Hmisc'
```

```
## The following objects are masked from 'package:base':
##
##     format.pval, units
```

```
# First, the result of summary
summary(len ~ supp + dose, data = ToothGrowth)
```

```
## len     N= 60
##
## +-------+---+--+--------+
## |       |   |N |len     |
## +-------+---+--+--------+
## |supp   |OJ |30|20.66333|
## |       |VC |30|16.96333|
## +-------+---+--+--------+
## |dose   |0.5|20|10.60500|
## |       |1  |20|19.73500|
## |       |2  |20|26.10000|
## +-------+---+--+--------+
## |Overall|   |60|18.81333|
## +-------+---+--+--------+
```

```
# A nice plot of this information (though the table is very informative; a
# plot may not be necessary)
plot(summary(len ~ supp + dose, data = ToothGrowth))
```



Splitting and Stacking Data

Consider the function `boxplot()`. We have seen two ways to make a parallel boxplot using `boxplot()`. One way is to have your data stored in separate vectors, say `x` and `y`, then call `boxplot(x, y)`. Another way is to have the data stored in a data frame, say `d`, that has two columns, say `v` and `g`. The variable `v` contains the data stored in the original vectors `x` and `y`, and the variable `g` identifies whether the data belongs to group `x` or group `y`. Then the function call `boxplot(v ~ g, data = d)` will produce the same graphic we described earlier.

Which is the better approach? Here, there may not appear to be much difference. However, in other circumstances, it may be easier to have data stored in separate vectors or in a list containing those vectors (say, if you are using `sapply()` for looping), or it may be easier to have your data stored in "long form", where a data frame contains the actual data points in one column and another column identifies the group to which each data point belongs (for example, if you are plotting using `ggplot()`). Thus it's important to be able to transform data so it fits whichever format we need.

There are two functions that allow for an easy transition between these two formats: `split()` and `stack()`. `split(v, g)` will create a list of vectors where each data point in `v` is included in a vector identified by the vector `g`. (Naturally, `v` and `g` must be the same length.) For example, if `tweet` is a vector of strings representing tweets by Twitter users, and `user` is a vector that identifies the user who wrote each tweet stored in `tweet`, then `l <- split(tweet, user)` will create a list `l` with vectors for each unique user in `user`, each vector in the list containing the respective user's tweets. So if `NTGuardian` is one of the users, `l$NTGuardian` is the vector containing NTGuardian's tweets.

Let's try this out on the `iris` data, where the `Sepal.Length` variable identifies sepal lengths of iris flowers, and `Species` identifies the species of each flower in the sample. Currently the data is in long form, so let's separate it into separate vectors.

```
# A look at the data
head(iris)
```

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

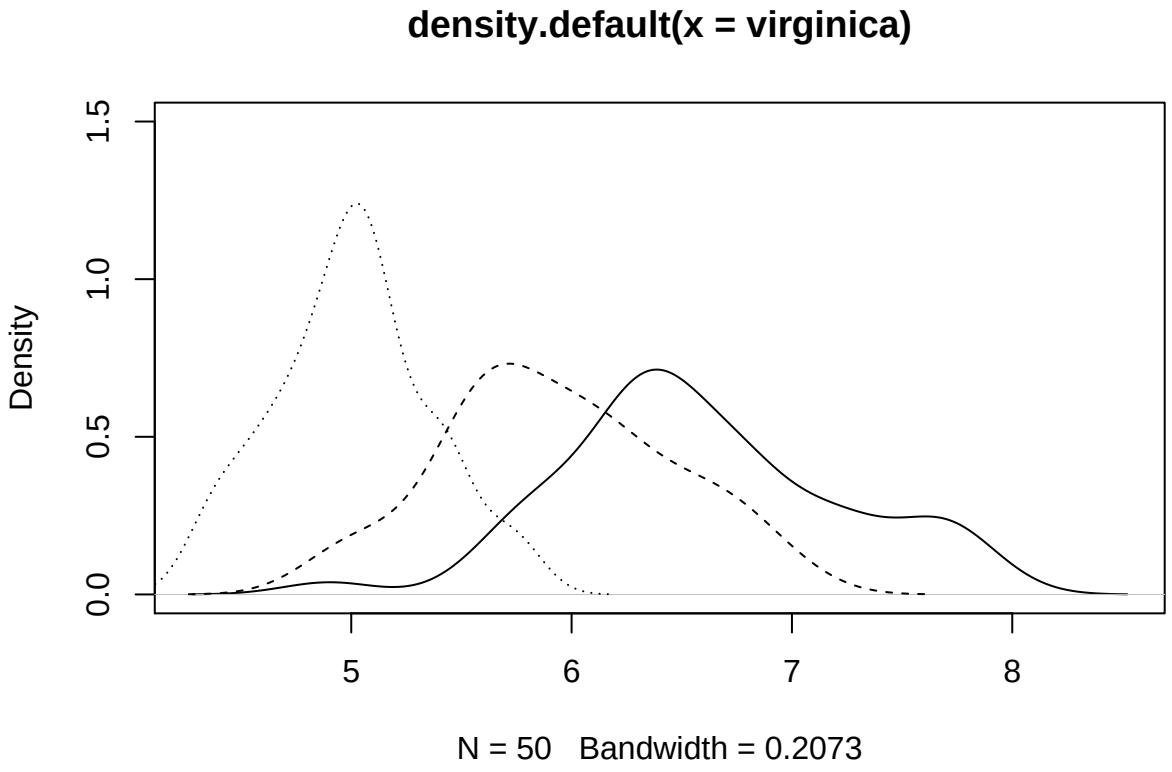
```
# It's in long form; let's change that
l <- split(iris$Sepal.Length, iris$Species)
l
```

```
## $setosa
##  [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
## [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
## [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0
##
## $versicolor
##  [1] 7.0 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6
## [18] 5.8 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0
## [35] 5.4 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7
##
## $virginica
##  [1] 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5
## [18] 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3
## [35] 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

```
# l can be used in sapply; say, we want the median sepal length for each
# flower
sapply(l, median)
```

```
##     setosa versicolor  virginica
##        5.0        5.9        6.5
```

```
# If we want the variables in the working environment, we will need to
# assign them to vectors manually
setosa <- l$setosa
versicolor <- l$versicolor
virginica <- l$virginica
# Make a density plot comparing the distributions
plot(density(virginica), lty = 1, ylim = c(0, 1.5))
lines(density(versicolor), lty = 2)
lines(density(setosa), lty = 3)
```




```
# This task lends naturally to the formula interface, and the DescTools
# package provides this interface
library(DescTools)
```

```
##
## Attaching package: 'DescTools'
```

```
## The following objects are masked from 'package:Hmisc':
##
##     Label, Mean, %nin%, Quantile
```

```
## The following object is masked from 'package:aplpack':
##
##     plot.bagplot
```

```
# Notice this produces the same result as the earlier call
l2 <- split(Sepal.Length ~ Species, data = iris)
l2
```

```
## $setosa
##  [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
## [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
## [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0
##
## $versicolor
##  [1] 7.0 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6
## [18] 5.8 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0
## [35] 5.4 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7
##
## $virginica
##  [1] 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5
## [18] 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3
## [35] 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

`stack()` does the opposite of `split()`. `stack()` takes a list of vectors, named according to the group to which they belong, and returns a data frame with a column for the data and a column identifying the group to which each data point belongs. So if `l` is a list produced by split, `stack(l)` will return the data into the original long form format. Another example: if `ff` is a vector that contains the weight of boxes of Frosted Flakes cereals, `trx` a vector that contains the weights of boxes of Trix cereal, and `ccp` a vector that contains weights of boxes of Cocoa Puffs, then `df <- stack(list("ff" = ff, "trx" = trx, "ccp" = ccp))` will create a data frame with a column containing weights of cereal boxes and a column containing the type of cereal.

Usage of `stack()` is demonstrated below.

```
# Recall setosa, versicolor, and verginica from before; let's reconstruct
# the data frame with which we made them.
setosa
```

```
##  [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
## [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
## [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0
```

```
versicolor
```

```
##  [1] 7.0 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6
## [18] 5.8 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0
## [35] 5.4 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7
```

```
virginica
```

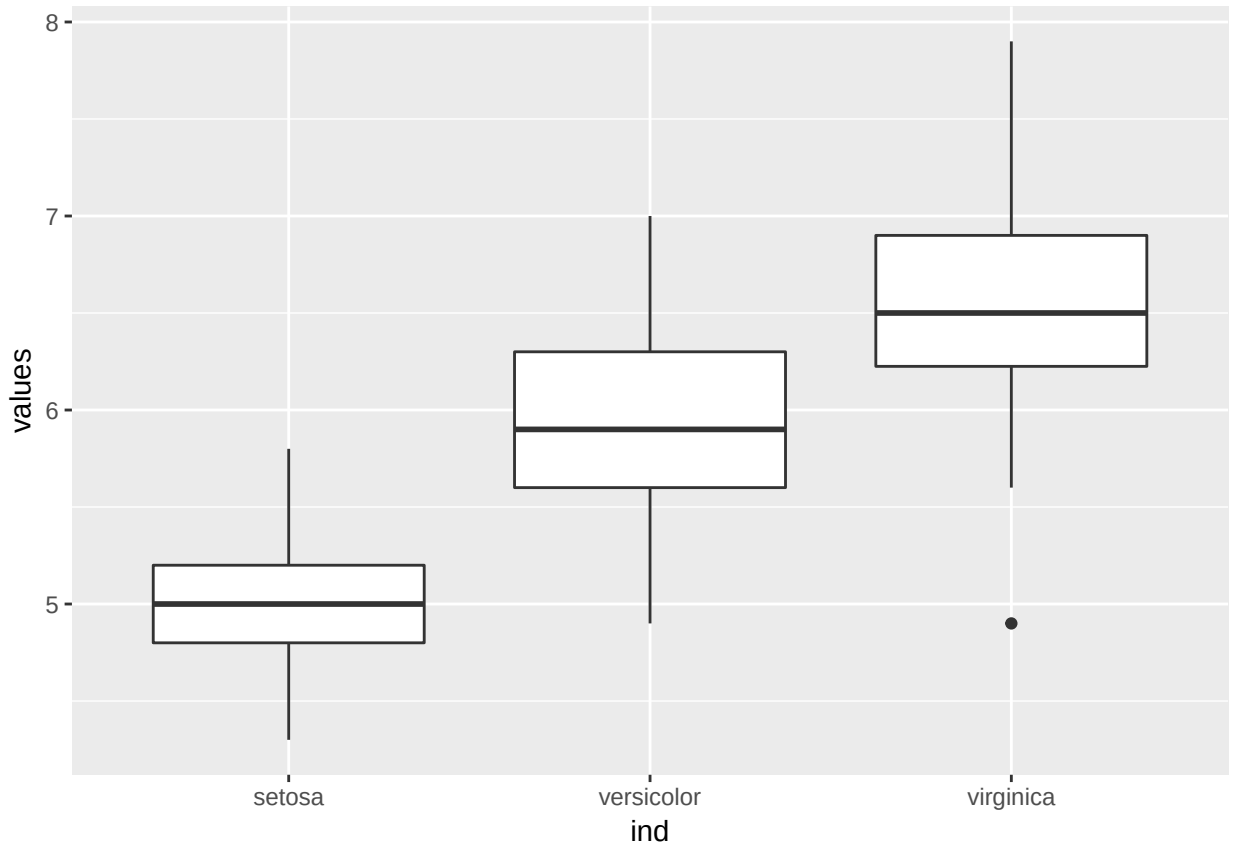
```
##  [1] 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5
## [18] 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3
## [35] 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

```
df <- stack(list(setosa = setosa, versicolor = versicolor, virginica = virginica))
head(df)
```

```
##   values    ind
## 1    5.1 setosa
## 2    4.9 setosa
## 3    4.7 setosa
## 4    4.6 setosa
## 5    5.0 setosa
```

```
## 6    5.4 setosa
```

```
# This is the ideal format for ggplot
library(ggplot2)
ggplot(df, aes(x = ind, y = values)) + geom_boxplot()
```




```
# To see that stack basically undoes what was done by split
head(ToothGrowth)
```

```
##    len supp dose
## 1  4.2   VC  0.5
## 2 11.5   VC  0.5
## 3  7.3   VC  0.5
## 4  5.8   VC  0.5
## 5  6.4   VC  0.5
## 6 10.0   VC  0.5
```

```
l <- with(ToothGrowth, split(len, supp))
l
```

```
## $OJ
##  [1] 15.2 21.5 17.6  9.7 14.5 10.0  8.2  9.4 16.5  9.7 19.7 23.3 23.6 26.4
## [15] 20.0 25.2 25.8 21.2 14.5 27.3 25.5 26.4 22.4 24.5 24.8 30.9 26.4 27.3
## [29] 29.4 23.0
##
## $VC
##  [1]  4.2 11.5  7.3  5.8  6.4 10.0 11.2 11.2  5.2  7.0 16.5 16.5 15.2 17.3
## [15] 22.5 17.3 13.6 14.5 18.8 15.5 23.6 18.5 33.9 25.5 26.4 32.5 26.7 21.5
## [29] 23.3 29.5
```

```
df2 <- stack(l)
head(df2)
```

```
##   values ind
## 1   15.2  OJ
## 2   21.5  OJ
## 3   17.6  OJ
## 4    9.7  OJ
## 5   14.5  OJ
## 6   10.0  OJ
```

Lecture 8

Paired Data

So far we have examined data that came from two independent samples. They may measure the same thing, but they should still be regarded as distinct. Paired data is an entirely separate case. With paired data, two variables were recorded for one observation. Usually we want to know what the relationship between the two variables is, if any.

A good first step to studying paired data (in fact, you should consider it a *necessary* step!) is to visualize the relationship between the variables with a scatterplot. You could do so with `plot(y ~ x, data = d)`, where `x` is the variable plotted along the horizontal axis of a scatterplot, `y` the variable plotted along the vertical axis, and `d` the data set containing `x` and `y` (if applicable).

The data set `fat`contains various measurements from people's bodies. Let's examine the relationship between height and weight with a scatterplot.

```
library(UsingR)
```

```
## Loading required package: MASS
```

```
## Loading required package: HistData
```

```
## Loading required package: Hmisc
```

```
## Loading required package: lattice
```

```
## Loading required package: survival
```

```
## Loading required package: Formula
```

```
## Loading required package: ggplot2
```

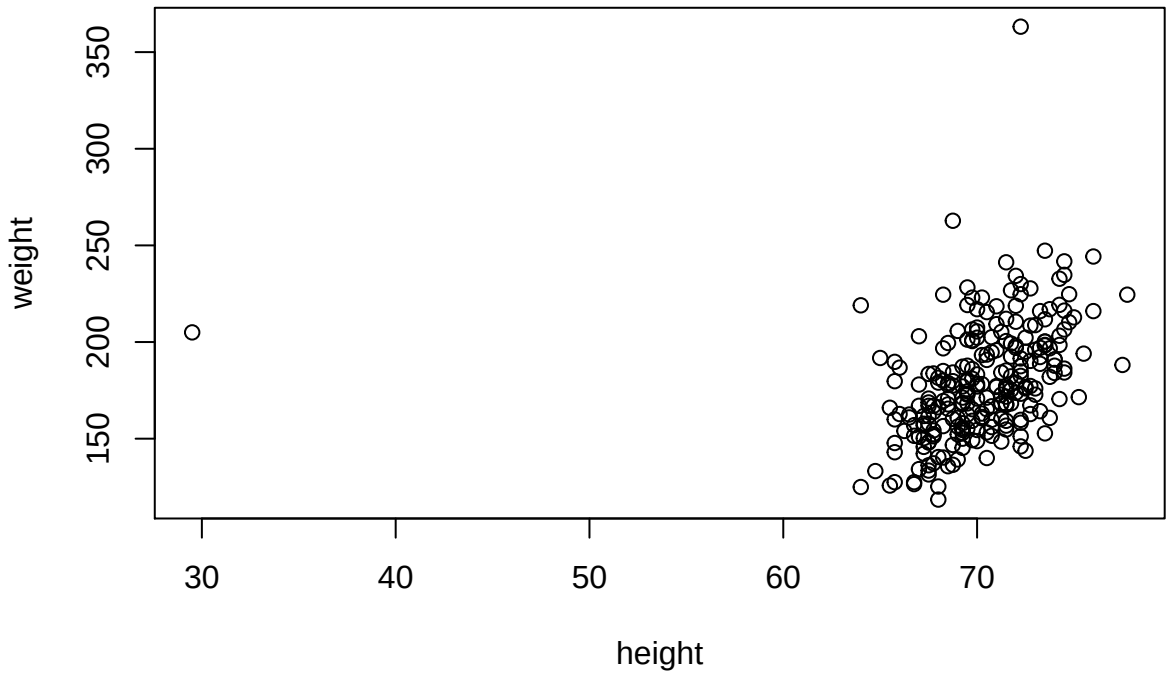
```
##
## Attaching package: 'Hmisc'
```

```
## The following objects are masked from 'package:base':
##
##     format.pval, units
```

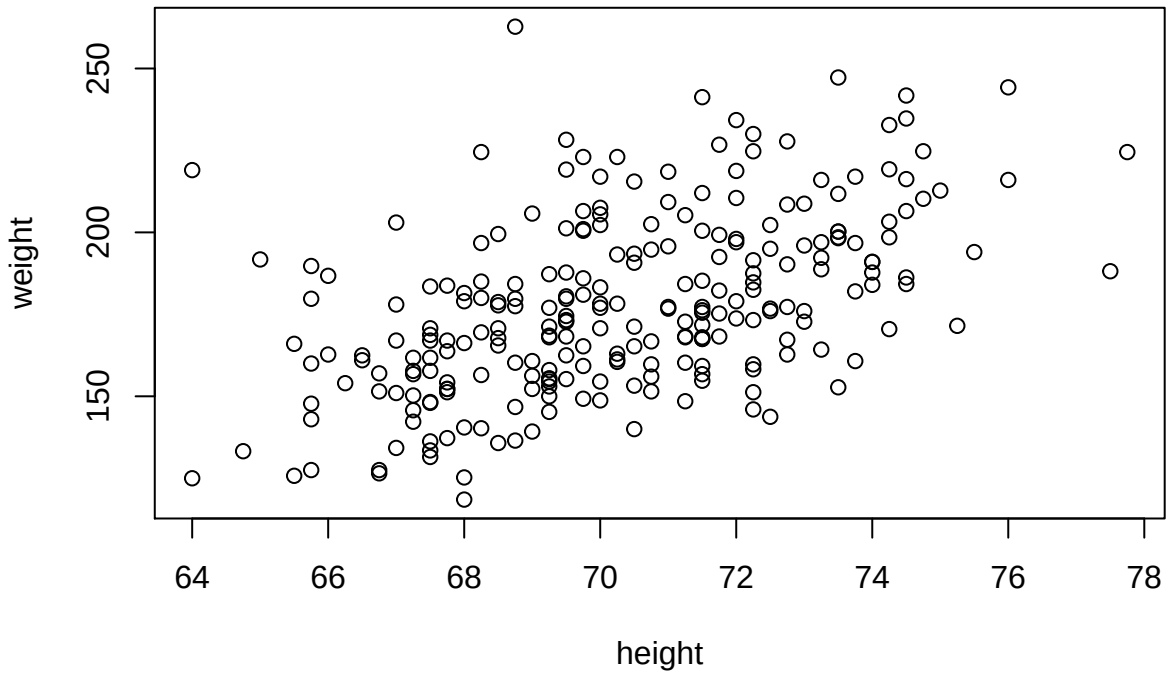
```
##
## Attaching package: 'UsingR'
```

```
## The following object is masked from 'package:survival':
##
##     cancer
```

```
plot(weight ~ height, data = fat)
```




```
# Using a scatterplot, we immediately identified an outlier, an individual
# that, while not necessarily having an unusual weight, is much shorter than
# usual (probably an individual with dwarfism). There is also an individual
# who is unusually heavy, though having a typical height. We filter out
# those poiints in another plot
plot(weight ~ height, data = fat, subset = height > 60 & weight < 300)
```



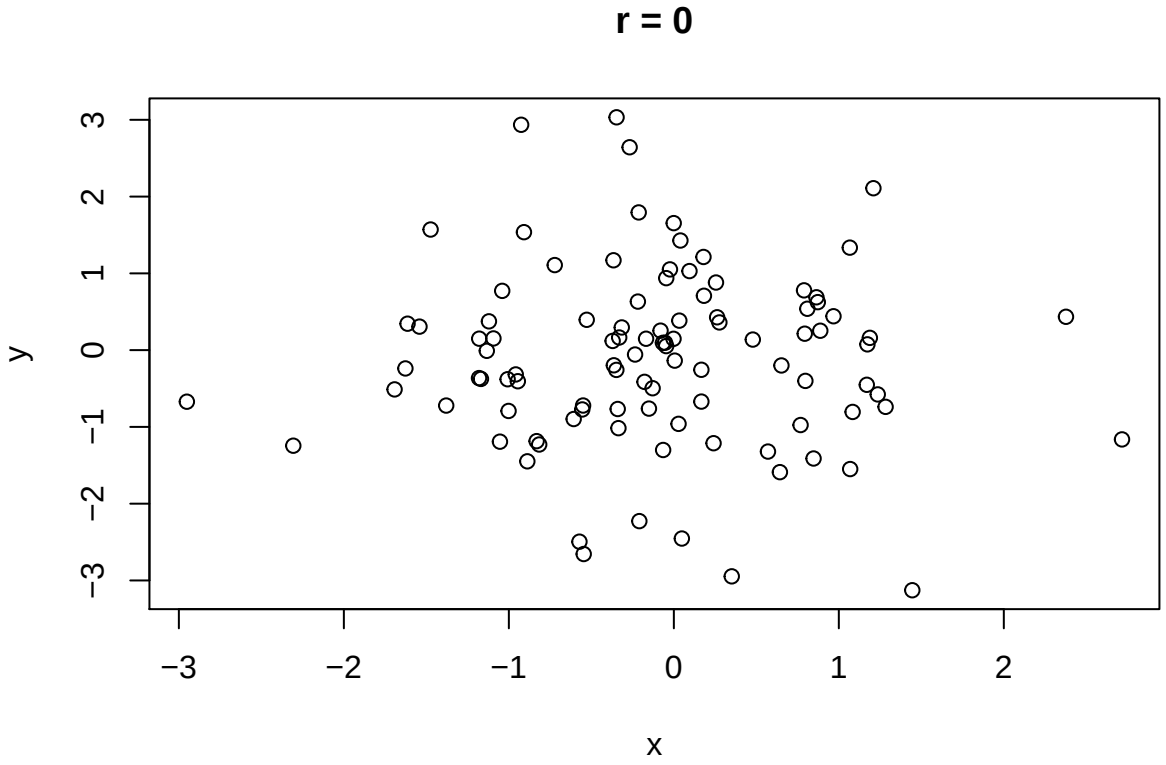
It's a good idea to plot data before numerically analyzing it. Outliers and non-linear relationships between variables are often easy to identify graphically, but can hide in numeric summaries.

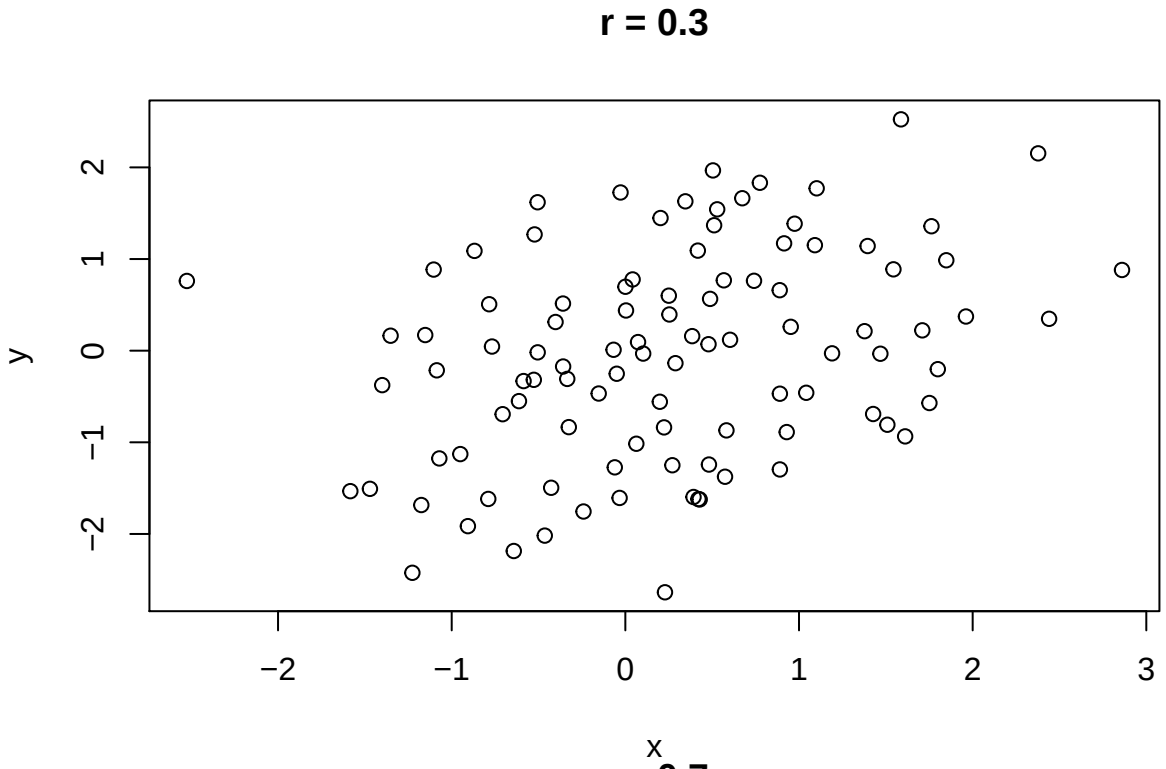
Correlation

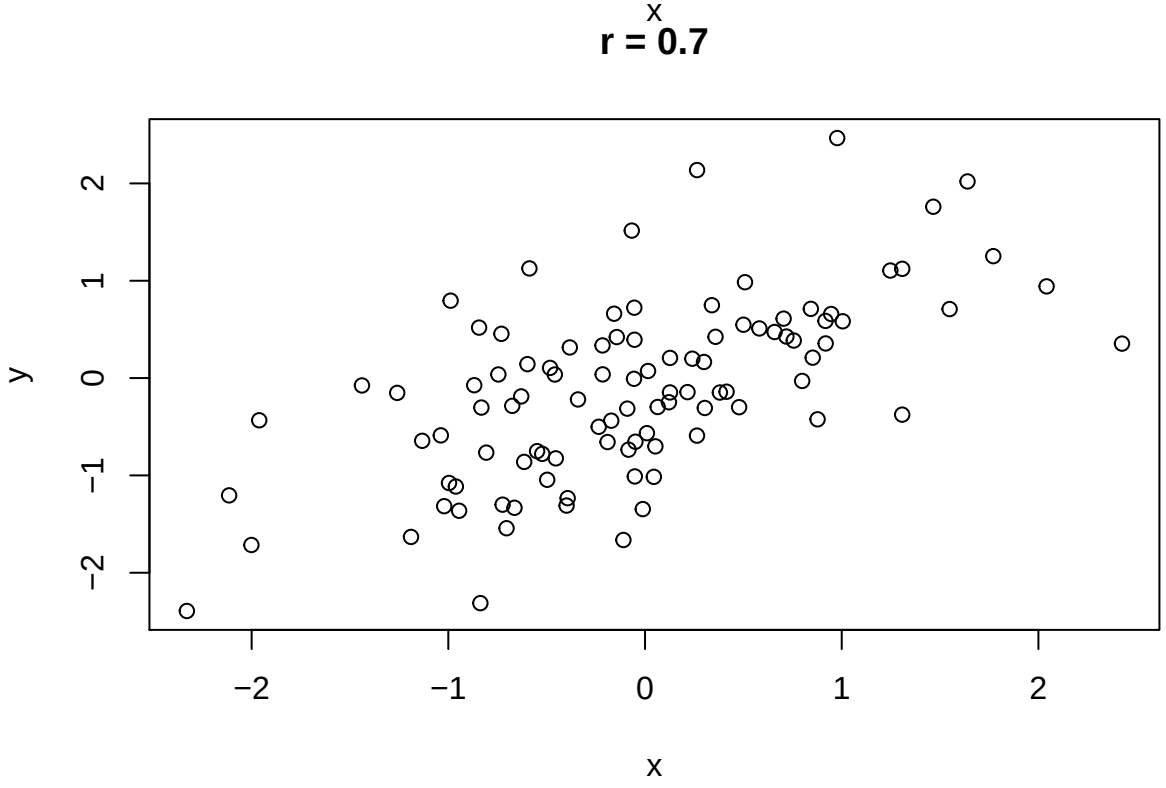
The **correlation** between two variables measures the strength of their relationship. Without discussing how correlation is computed, let r represent the correlation between two variables. It is always true that $-1 \leq r \leq 1$. If $r = 0$, there is no *linear* relationship between the variables (there may still be a relationship, just not a linear one). If $|r| = 1$, there is a perfect linear relationship between the variables; if plotted in a scatterplot, the variables would fall along a perfectly straight line. sign(r) indicates the direction of the relationship. If $r > 0$, there is a positive relationship between the variables; as one variable increases, the other also tends to increase. If $r < 0$, there is a negative relationship between the variables; as one variable increases, the other tends to decrease.

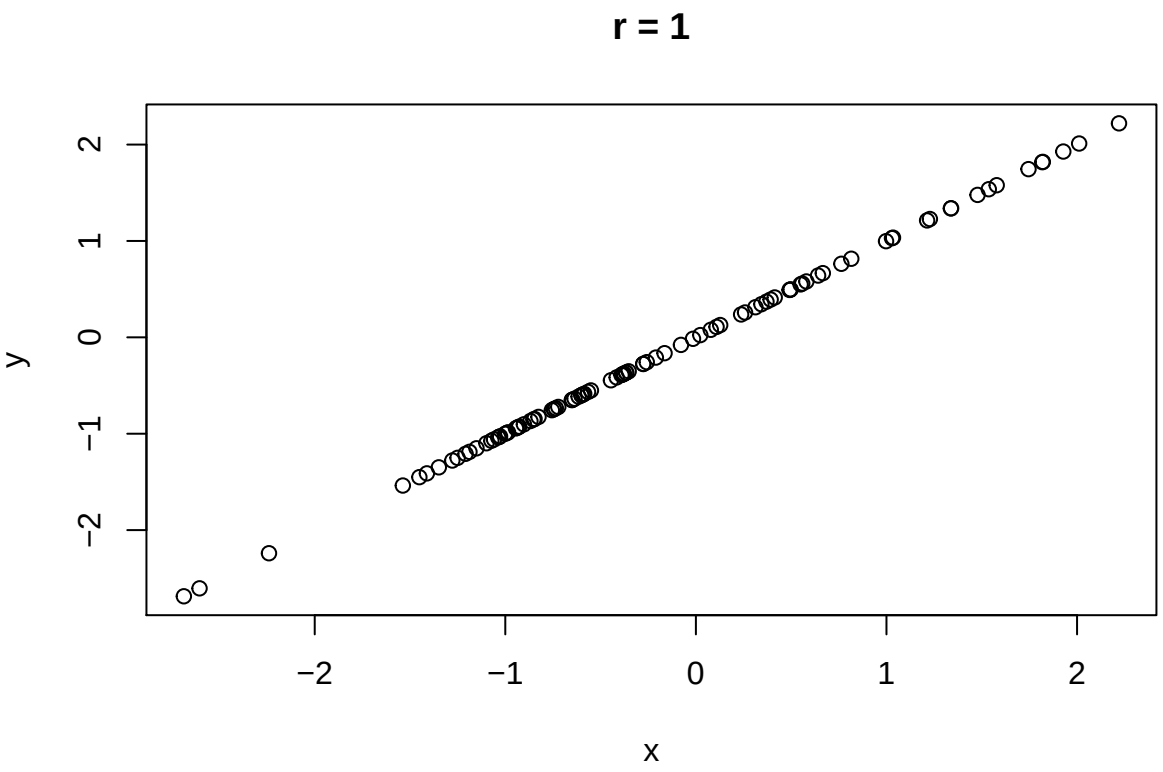
No real-world data set has a correlation that is perfectly zero, one, or negative one (unless engineered). A rule of thumb is that if $0 \leq |r| < .3$, there is no correlation. If $.3 \leq |r| < .7$, the correlation is weak. If $|r| \geq .7$, the correlation is strong.

Some illustrations of different correlations follow.




r = 0.3
y
x
−2
−1
0
1
2
3


r = 0.7
y
x
−2
−1
0
1
2



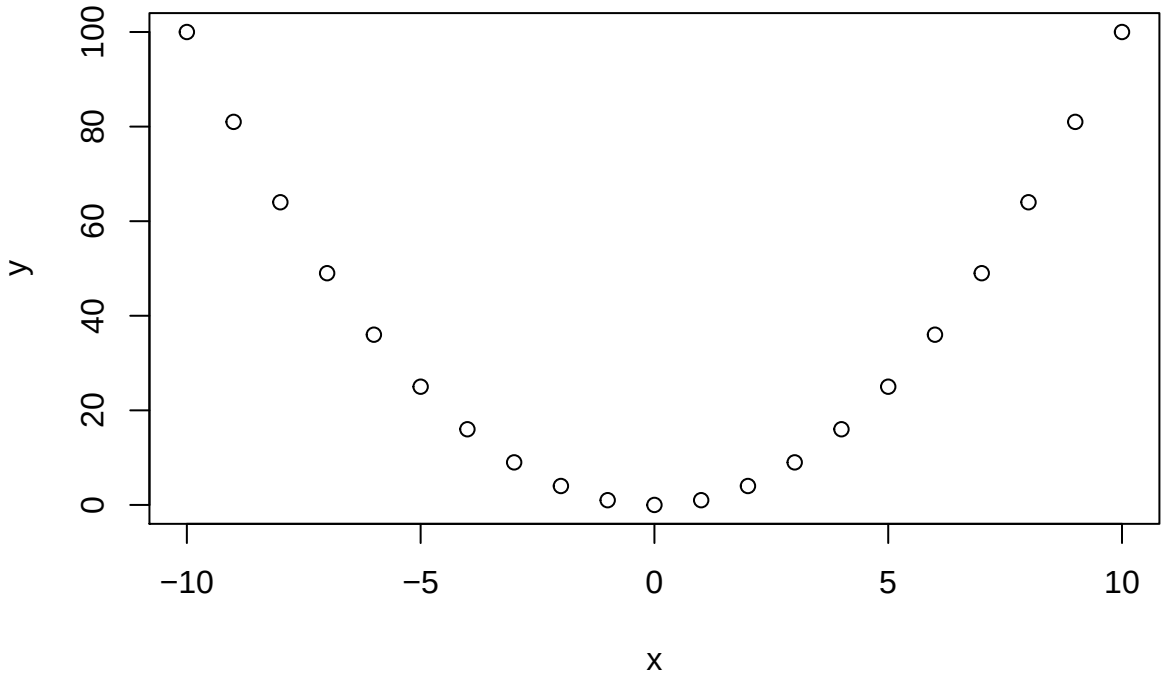
Let's compute a correlation. The `cor()` function will compute correlation for a data set, taking two data vectors `cor(x, y)` and computing the correlation between `x` and `y`.

```
# The correlation of height and weight
cor(fat$height, fat$weight)
```

```
## [1] 0.3082785
```

Remember, we are computing *linear* correlation! There could be a linear correlation of 0, but a perfect non-linear correlation!

```
x <- (-10):10
y <- x^2
plot(x, y)
```



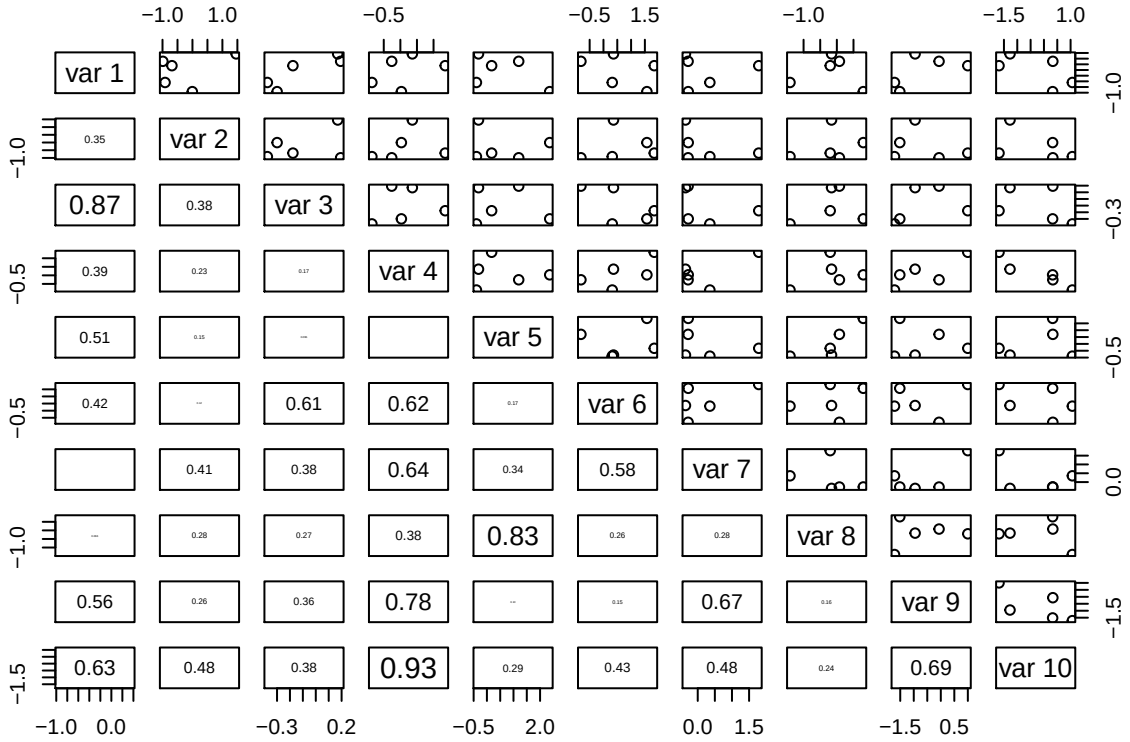

```
cor(x, y)
```

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

One thing a researcher must always bear in mind when studying the correlation between variables is that **correlation is not causation!** A causal relationship between the two variables in study may be responsible for the correlation, but so could a causal relationship between each individual variable and a third, unobserved variable (what we term a **latent variable**), a causal relationship running in the opposite direction of the one observed (y causes x rather than x causing y), or a complex confounding relationship involving feedback loops between the variables observed and unobserved latent variables (x causes y, which in turn causes x, while also being caused by unobserved z while also causing z). Additionally, if there is a time component, a correlation between two variables may be strong for no reason other than both variables trend (this may translate simply into time being a latent, unaccounted-for variable, and both variables need to be detrended).

The website Spurious correlations contains many strong correlations that clearly are not causal (though they are quite amusing). How do people discover these? The best explanation may be that when a very large number of variables are observed, correlations *will* appear. Think of it this way: if every pair of variables in a data set, generated independently, have a probability $\epsilon(n)$ of producing data sets that are not correlated for data sets of size n, the probability that no correlations appear is roughly $(\epsilon(n))^{\binom{k}{2}}$ (keep in mind that $\binom{k}{2}$ grows very quickly in k). This number is approaching zero very fast, so it's highly unlikely that no correlations at all will appear in the data set. The situation is not hopeless, of course; increasing the sample size n will increase $\epsilon(n)$ and make observing misleading sample correlations less likely. But the point remains; it's easier to find *some* correlation in data sets that track lots of variables than in data sets that track only a handful, given the same sample size.

As an example, I generated a small data set using random variables I know are not correlated, yet the plot demonstrates that sample correlations still manage to appear, even large ones.



In general, identifying causality between variables is not easy. There are methods for doing so (they are outside the scope of this course), but bear in mind that they usually must inject additional knowledge, be it about the experiment and data collection method or subject matter knowledge, in order to identify causality. Mathematics alone is not sufficient, and no single statistical analysis may have the final say on a subject. It may take numerous replications and permutations of methodology along with clever argument and domain area knowledge to establish a causal link.

Trend Lines

A trend line is a "best fit" line passing through data in a data set that shows what linear relationship between data tends to prevail throughout the data set. Without going into detail about how this line is computed, the trend line typically computed minimizes the squared (vertical) distance of each data point to the line (this distance being called the **residual** or **error**).

We investigate here lines of the form $\hat{y}_i = \beta_0 + \beta_1 x_i$, where $\hat{y}_i$ is the value of the y variable predicted by the trend line for the value x_i. β_0 is the y-intercept of the trend line, and β_1 the slope of the trend line. We can compute the coefficients of the trend line using the R function `lm(y ~ x, data = d)`, where `y` is the variable being predicted (you may think "dependent variable"), `x` is the predicting variable (think "independent variable"), and `d` is the data set containing both `x` and `y` (if applicable). We can save the output of `lm()` by using a call akin to `fit <- lm(y ~ x, data = d)`, and we can extract the coefficients of the trend line from `fit` using `coefficients(fit)` or simply accessing them directly with `fit$coefficients`.

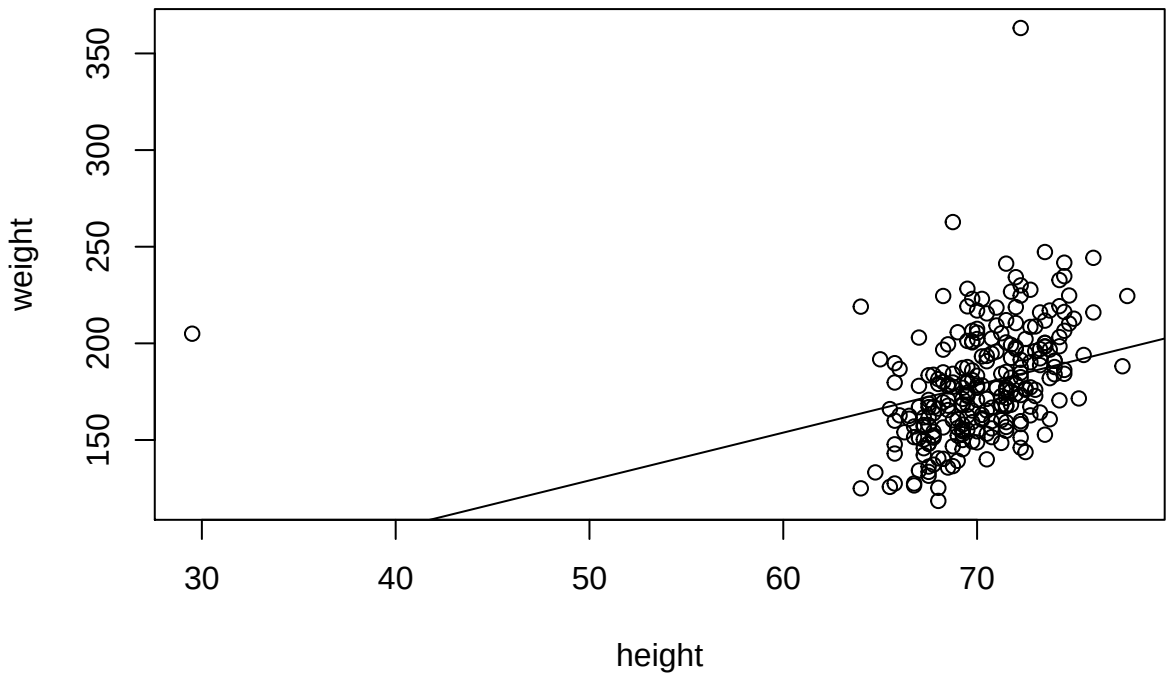
Below I demonstrate fitting a trend line to the height and weight data.

```
hw_fit <- lm(weight ~ height, data = fat)
coefficients(hw_fit)
```

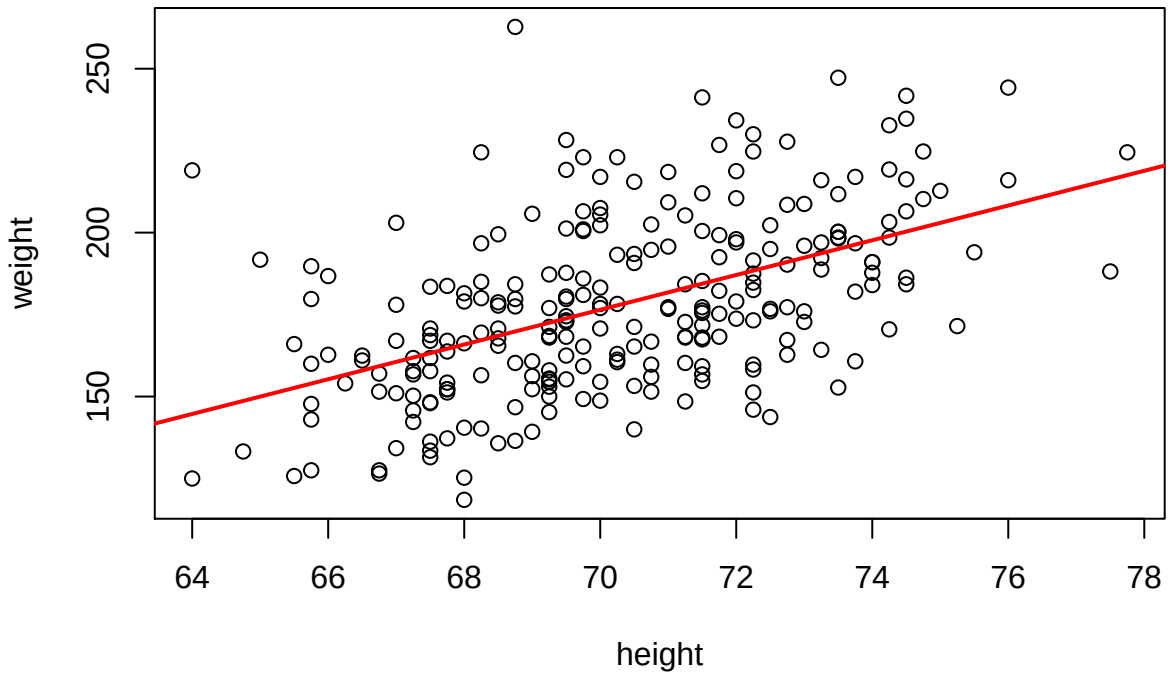
```
## (Intercept)      height
##    5.411832    2.473493
```

We can see the model, but we would like to plot the relationship on a line. After making a plot with `plot()`, we can add a trendline with `abline(fit)`, which adds the trend line stored in `fit` to the plot.

```
plot(weight ~ height, data = fat)
abline(hw_fit)
```




```
# Least-squares regression is sensitive to outliers, so let's compute the
# trend line when the outliers are left out
hw_fit2 <- lm(weight ~ height, data = fat, subset = height > 50 & weight < 300)
plot(weight ~ height, data = fat, subset = height > 50 & weight < 300)
abline(hw_fit2, col = "red", lwd = 2)
```



Computation and analysis of trends is a topic discussed much more extensively in MATH 3080, and so I end the discussion here.

Categorical Bivariate Data

So far we have examined only relationships in quantitative data. We can also examine relationships between categorical data.

Numerically we analyze categorical data with tables of counts, with each cell of the table containing a count of the observations having a particular combination of the categorical variables in question. We usually want to consider the joint distribution of the variables in question as well as the margins of the tables. The `xtabs()` function can allow us to quickly construct join distribution tables using formula notation. `xtabs(~ x + y, data = d)` will construct a table depending on variables `x` and `y`, with data stored in `d`. One could extend this table to as many variables as desired; for example, `xtabs(~ x + y + z, data = d)` constructs a three-dimensional array examining the relationship between variables `x`, `y`, and `z`. When creating such an array, you may want to use the `ftable()` function for viewing the information in the table in a more legible format. For example, if we saved the results of the earlier `xtabs()` output in a variable `tab`, `ftable(tab, row.vars = 2, col.vars = c(1, 3))` will create a table where the variable associated with dimension 2 (`y`) will be shown in rows, and the variables associated with dimensions 1 and 3 (`x` and `z`) are shown in the columns.

I demonstrate constructing tables this way below, exploring the `Cars93` (**MASS**) data set.

```
# Two-way table exploring origin and type
tab1 <- xtabs(~Origin + Type, data = Cars93)
tab1
```

```
##          Type
## Origin    Compact Large Midsize Small Sporty Van
##   USA           7    11      10     7      8   5
##   non-USA       9     0      12    14      6   4
```

```
# A three-way table
tab2 <- xtabs(~Origin + Type + Cylinders, data = Cars93)
# The following output is hard to parse
tab2
```

```
## , , Cylinders = 3
##
##          Type
## Origin    Compact Large Midsize Small Sporty Van
##   USA           0     0       0     0      0   0
##   non-USA       0     0       0     3      0   0
##
## , , Cylinders = 4
##
##          Type
## Origin    Compact Large Midsize Small Sporty Van
##   USA           7     0       4     7      4   0
##   non-USA       8     0       3    11      4   1
##
## , , Cylinders = 5
##
##          Type
## Origin    Compact Large Midsize Small Sporty Van
##   USA           0     0       0     0      0   0
##   non-USA       0     0       1     0      0   1
##
## , , Cylinders = 6
##
```

```
##          Type
## Origin    Compact Large Midsize Small Sporty Van
##   USA           0     7       5     0      3   5
##   non-USA       1     0       7     0      1   2
##
## , , Cylinders = 8
##
##          Type
## Origin    Compact Large Midsize Small Sporty Van
##   USA           0     4       1     0      1   0
##   non-USA       0     0       1     0      0   0
##
## , , Cylinders = rotary
##
##          Type
## Origin    Compact Large Midsize Small Sporty Van
##   USA           0     0       0     0      0   0
##   non-USA       0     0       0     0      1   0
```

```
# This is easier to read
ftable(tab2, row.vars = 2, col.vars = c(1, 3))
```

```
##         Origin    USA                    non-USA
##         Cylinders   3  4  5  6  8 rotary       3  4  5  6  8 rotary
## Type
## Compact             0  7  0  0  0      0       0  8  0  1  0      0
## Large               0  0  0  7  4      0       0  0  0  0  0      0
## Midsize             0  4  0  5  1      0       0  3  1  7  1      0
## Small               0  7  0  0  0      0       3 11  0  0  0      0
## Sporty              0  4  0  3  1      0       0  4  0  1  0      1
## Van                 0  0  0  5  0      0       0  1  1  2  0      0
```

```
# A four-way table
tab3 <- xtabs(~Origin + Type + Cylinders + Man.trans.avail, data = Cars93)
ftable(tab3, row.vars = c(2, 4), col.vars = c(1, 3))
```

```
##                         Origin    USA                    non-USA
##                         Cylinders   3  4  5  6  8 rotary       3  4  5  6  8 rotary
## Type    Man.trans.avail
## Compact No                          0  2  0  0  0      0       0  0  0  0  0      0
##         Yes                         0  5  0  0  0      0       0  8  0  1  0      0
## Large   No                          0  0  0  7  4      0       0  0  0  0  0      0
##         Yes                         0  0  0  0  0      0       0  0  0  0  0      0
## Midsize No                          0  4  0  4  1      0       0  0  0  3  1      0
##         Yes                         0  0  0  1  0      0       0  3  1  4  0      0
## Small   No                          0  0  0  0  0      0       0  0  0  0  0      0
##         Yes                         0  7  0  0  0      0       3 11  0  0  0      0
## Sporty  No                          0  0  0  0  0      0       0  0  0  0  0      0
##         Yes                         0  4  0  3  1      0       0  4  0  1  0      1
## Van     No                          0  0  0  4  0      0       0  0  0  2  0      0
##         Yes                         0  0  0  1  0      0       0  1  1  0  0      0
```

When faced with a table, one often wishesw to know the marginal distributions, which is the distribution of just one of the variables without any knowledge of any other variables. We can obtain the margins of the tables produced by `xtabs()` with `margin.table()`. The call `margin.table(tbl, margin = i)` will find the marginal distribution of `tbl` for margin `i`, which may be 1 or 2 for a two-way table (corresponding to rows

and columns, respectively), but could be higher for more complex tables. I demonstrate `margin.table()` below:

```
margin.table(tab3, margin = 1)
```

```
## Origin
##     USA non-USA
##      48      45
```

```
margin.table(tab3, margin = 2)
```

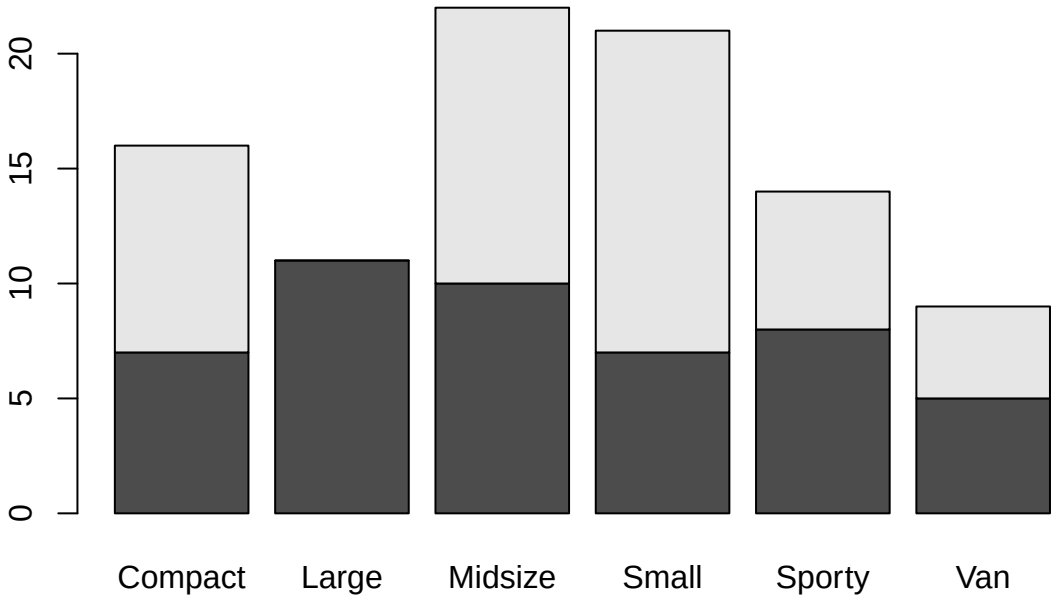
```
## Type
## Compact   Large Midsize   Small  Sporty     Van
##      16      11      22      21      14       9
```

```
margin.table(tab3, margin = 3)
```

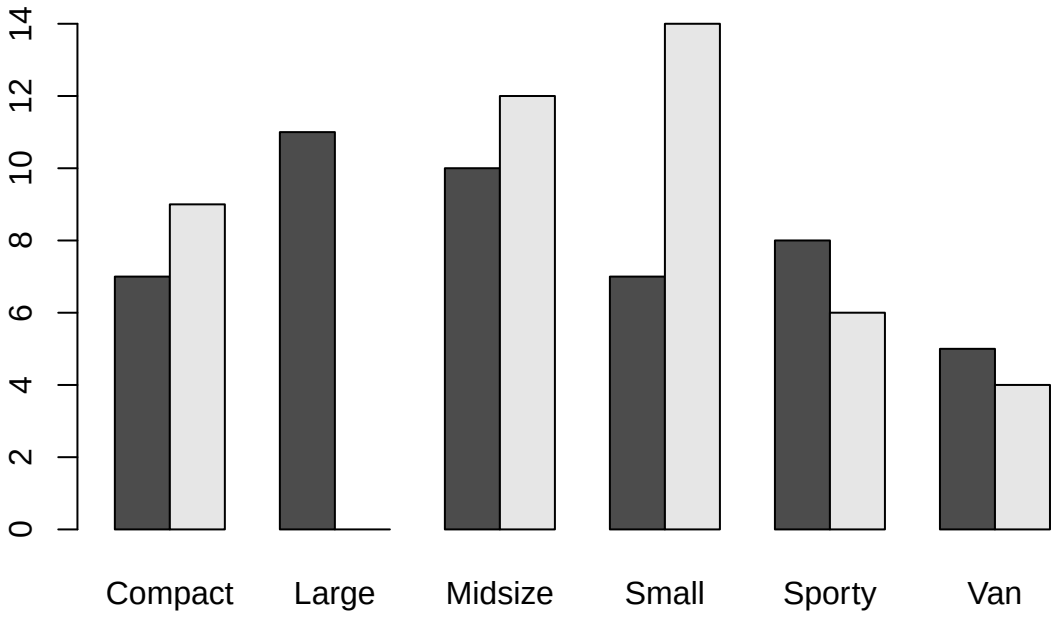
```
## Cylinders
##      3      4      5      6      8 rotary
##      3     49      2     31      7      1
```

We have a few options for visualizing data in a two-way table (tables with more dimensions would be more complex and more demanding from visualization techniques). One way would be with a stacked bar plot, where the height of the bar corresponds to the marginal distribution of one variable and sub-bars denote the breakdown for the second category. Better though would be side-by-side bar plots, which don't stack the breakdown categories one atop the other but instead place them side by side yet in close proxomity. `barplot()` can create such plots.

```
barplot(tab1)
```

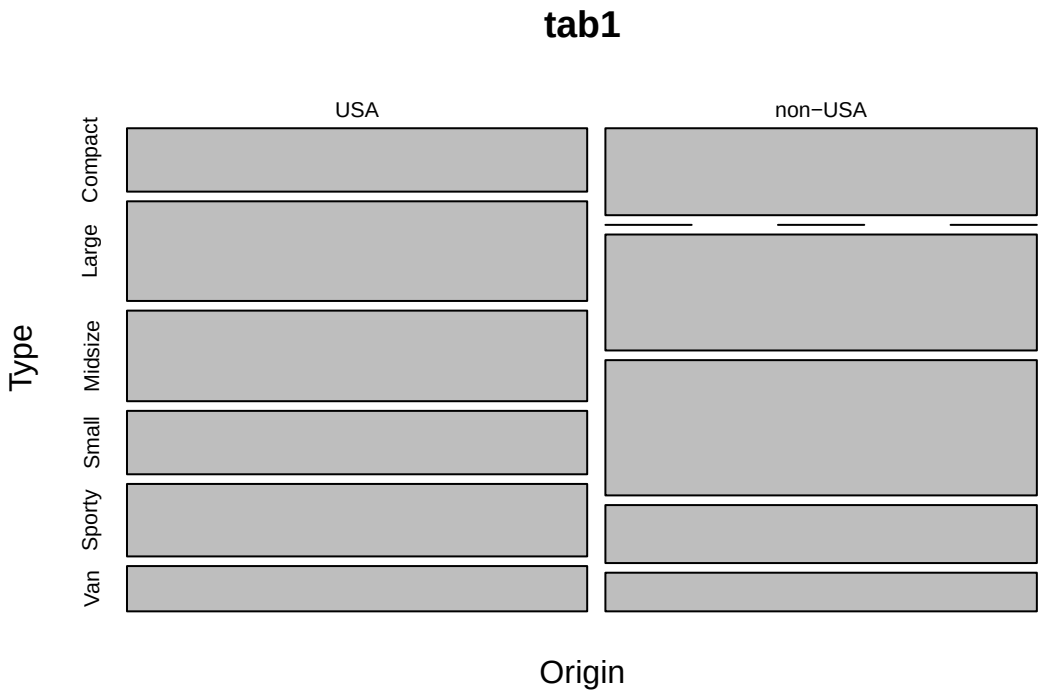



```
barplot(tab1, beside = TRUE)
```

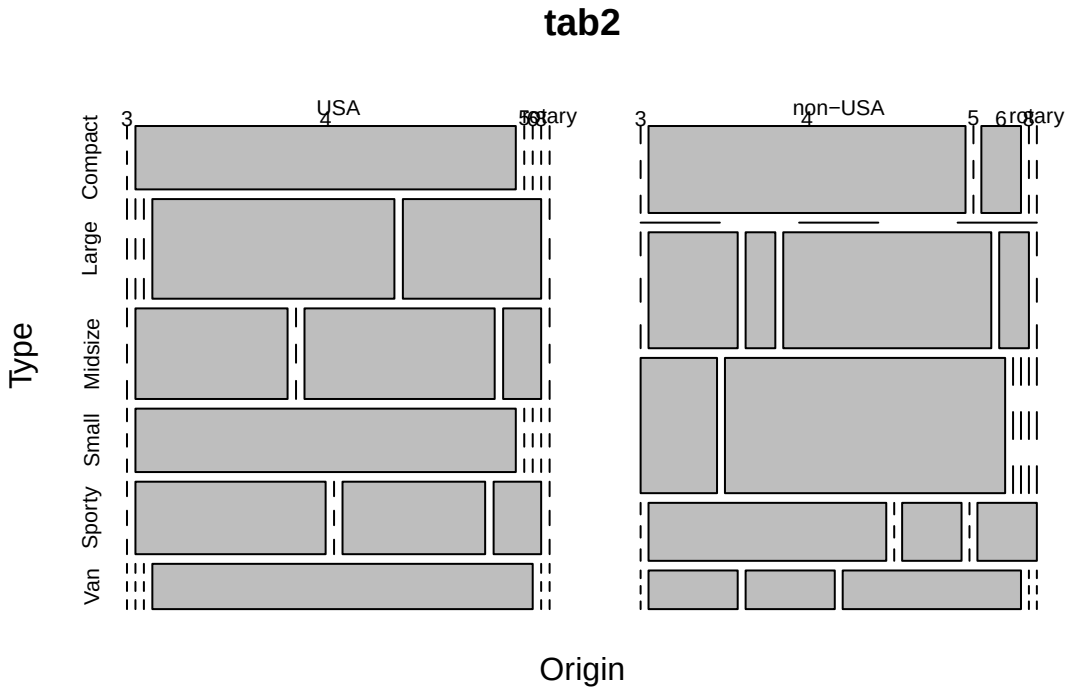


Another option is a **mosaic plot**, which shows the frequency of each combination of variables as the size of rectangles. This can be created in R using the function `mosaicplot()`, as below.

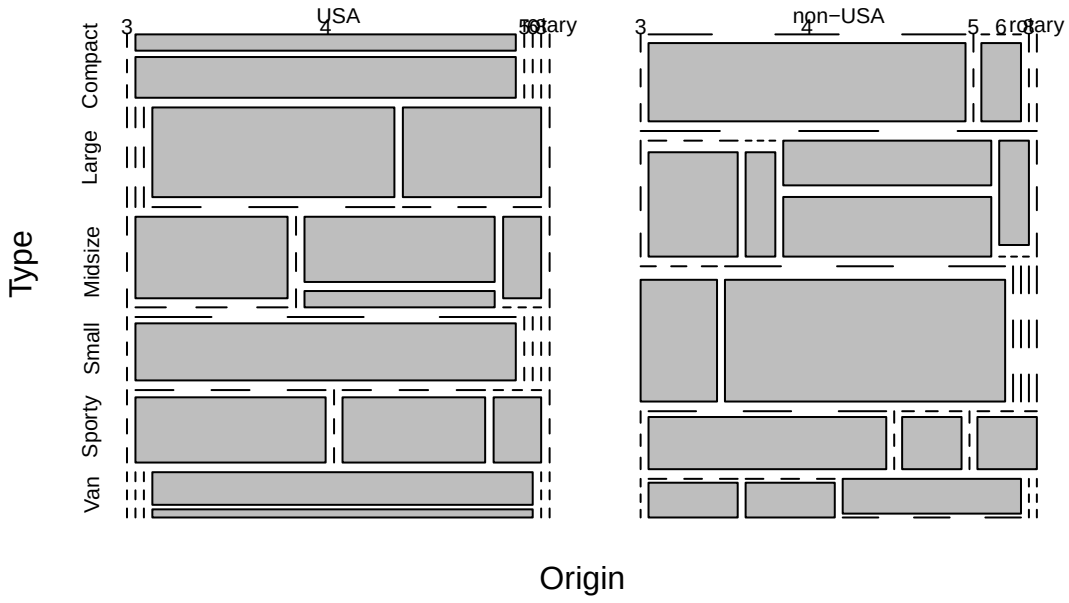
```
mosaicplot(tab1)
```




```
mosaicplot(tab2)
```




```
mosaicplot(tab3)
```



Lecture 9

Random Number Generation

Computers cannot create random numbers. They are Turing-complete, and a Turing machine is deterministic, not random. Instead, computers generate pseudo-random numbers. They start with a seed, an initial number in a sequence, and from there generate further numbers in the sequence that, without knowing the initial seed, appear to be random. That said, if the seed were known along with the generating process, the random sequence could be recreated exactly. Usually the system time is used to initialize a random number generator, since this may look random to a user. In R, you can use the `set.seed()` function to set a random seed. With no parameters, R will reset the random seed. If you wish to set the seed to a specific (numeric) value, you can do so with `set.seed(seed)`. (You do not have to set a seed; R will automatically pick one at the start of a session, though if you want reproducible results, you should set the seed and communicate what it is.) For this document, I set the seed below.

```
set.seed(52716)
```

Random number generators generate uniformly distributed random numbers. Once they can generate random numbers from the uniform distribution, random numbers following other distributions can be generated as well, so only one random number generator is needed.

The `sample()` function allows for some basic random value generation. Two arguments are required. The first is a vector containing the values that could be generated. This is your **sample space**, the possible values that could be generated by the random process. The second is the number of random values to generate. The call `sample(x, size)` will generate a random vector consisting of values from `x` of length `size`.

```
# Here I simulate the classic ball-in-bag probability model where random
# balls are pulled from a bag, without replacement. There are 32 red balls
# and 16 blue balls. I pull five balls from the bag.
sample(rep(c("red", "blue"), times = c(32, 16)), 5)
```

```
## [1] "blue" "red"  "red"  "red"  "red"
```

By default, `sample()` will use a model where random values are generated **without replacement**. This means that once a value is observed, it cannot be observed again (in the ball-and-bag model, this equates to pulling a ball from the bag and not putting the ball back in; once drawn, it cannot be drawn again). This naturally means that you must have `size < length(x)`, or an error will be thrown; you can't draw random values when you run out! To switch to a model **with replacement** (i.e. balls are put back in the bag after being drawn), set the parameter `replace = TRUE`.

```
# A model without replacement
ball_pull1 <- sample(rep(c("red", "blue"), times = c(32, 16)), 32 + 16)
ball_pull1
```

```
##  [1] "red"  "red"  "red"  "red"  "blue" "blue" "blue" "blue" "red"  "red"
## [11] "blue" "red"  "red"  "red"  "red"  "red"  "blue" "blue" "red"  "red"
## [21] "red"  "red"  "red"  "blue" "red"  "red"  "red"  "blue" "red"  "red"
```

```
## [31] "blue" "red"  "blue" "blue" "red"  "red"  "red"  "blue" "red"  "red"
## [41] "blue" "red"  "red"  "red"  "red"  "blue" "red"  "blue"
```

```
# Since I pulled all balls out of the bag, I get exactly 32 reds and 16
# blues
table(ball_pull1)
```

```
## ball_pull1
## blue  red
##   16   32
```

```
# Now with replacement
ball_pull2 <- sample(rep(c("red", "blue"), times = c(32, 16)), 32 + 16, replace = TRUE)
ball_pull2
```

```
##  [1] "red"  "red"  "red"  "blue" "red"  "red"  "red"  "red"  "blue" "red"
## [11] "red"  "red"  "red"  "red"  "red"  "blue" "red"  "red"  "red"  "red"
## [21] "red"  "red"  "red"  "red"  "red"  "blue" "red"  "red"  "blue" "red"
## [31] "red"  "red"  "red"  "red"  "red"  "red"  "blue" "red"  "red"  "blue"
## [41] "red"  "red"  "blue" "red"  "blue" "red"  "red"  "blue"
```

```
# While close to the true frequencies, I don't see red balls exactly 32
# times, or blue balls exactly 16 times.
table(ball_pull2)
```

```
## ball_pull2
## blue  red
##   10   38
```

By default, `sample()` will assume that each element in `x` is equally likely. This can be changed by setting `prob` to a vector assigning probabilities to each outcome.

```
# We can change the sampling with replacement model by making the sample
# space only c('red', 'blue'), and setting the probability of seeing these
# respective values to those consistent with our model.
sample(c("red", "blue"), 5, replace = TRUE, prob = c(32/48, 16/48))
```

```
## [1] "red"  "red"  "red"  "red"  "blue"
```

Families of Distributions

R facilitates both description and simulation of random variables from various families of probability distributions, using similar formats. Each family will have four R functions associated with it, preceded by a character and followed by the identifier of the family.

- `d` functions are associated with the probability density function (pdf) or probability mass function (pmf) of a distribution. An input vector of quantiles will get the value of the pdf/pmf at those quantiles.
- `p` functions are associated with the cumulative distribution function (cdf) of a probability distribution. Thus they describe the probability $P(X \leq x)$, or the probability of being less than the input. This is often referred to as the lower tail of the distribution. If instead you want the upper tail, or $P(X > x)$, you can set `lower.tail = FALSE` in a `p` function to obtain the upper tail.
- `q` functions are associated with the quantiles of the probability distribution, and thus are the inverse of the cdf. There is likewise a `lower.tail` parameter for the `q` function.
- `r` functions generate random values drawn from the distribution.

I give examples below for some distributions, starting with some discrete distributions.

Bernoulli

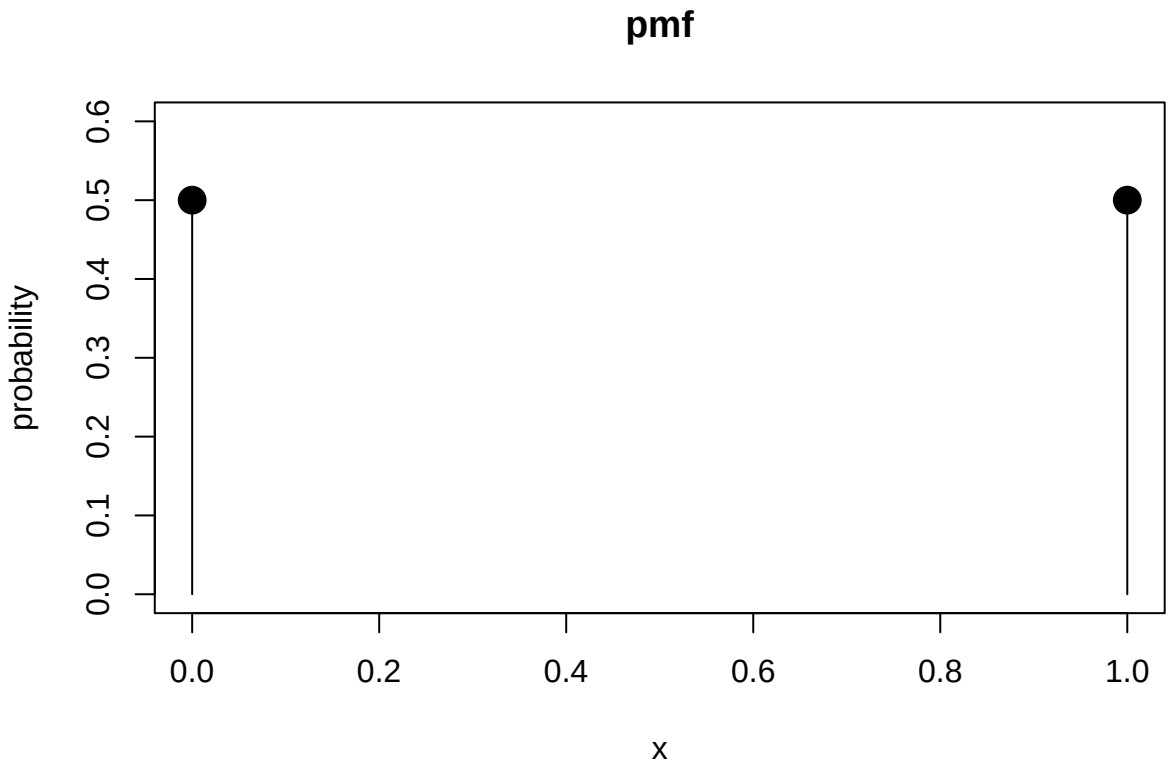
A Bernoulli random variable is a random variable that is either 1 or 0, and is described by the probability of obtaining 1, often denoted p. Thus we denote a Bernoulli random variable with:

$$X \sim \text{Ber}(p)$$

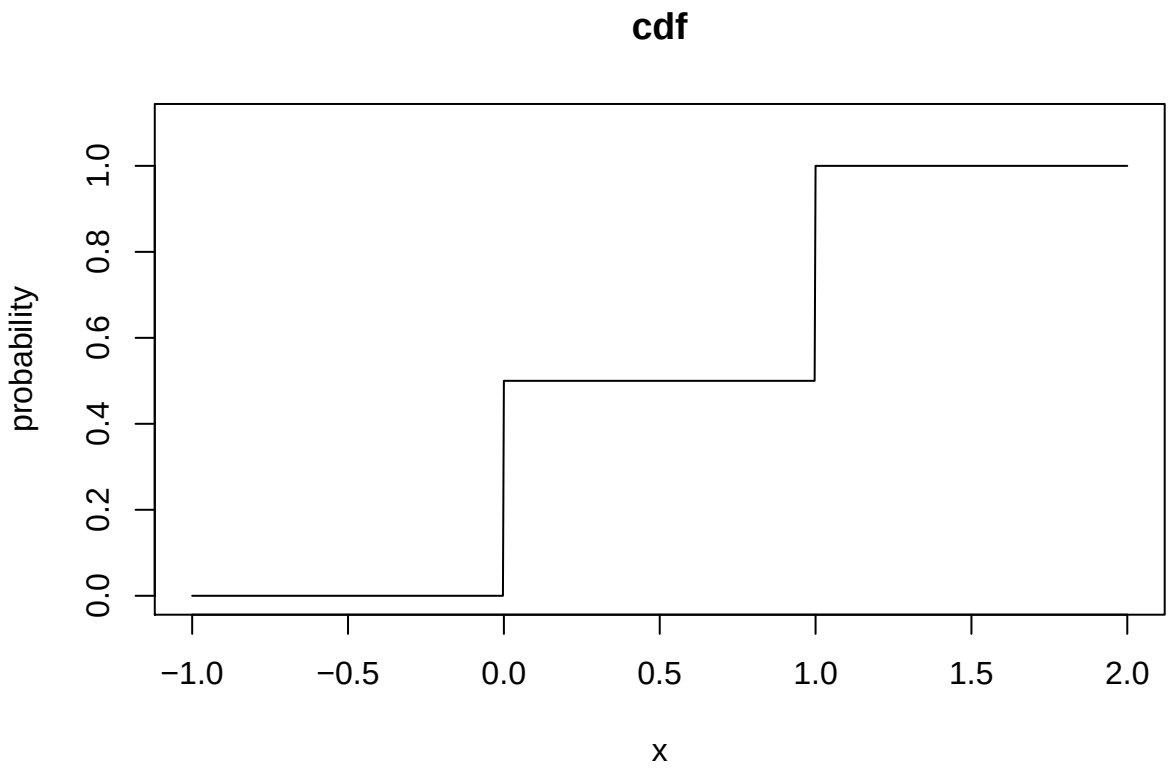
Note that if $p = \frac{1}{2}$, this is the model of a single fair coin flip.

Bellow, I plot the pdf's and cdf's of a random variable $X \sim \text{Ber}(\frac{1}{2})$, and also simulate some values.

```
# I will be plotting a lot of pmf's in this document, so I create a function
# to help save effort.  The first argument, q, represents the quantiles of
# the random variable (the values that are possible). The second argument
# represents the value of the pmf at each q (and thus should be of the same
# length); in other words, for each q, p is the probability of seeing q
plot_pmf <- function(q, p) {
    # This will plot a series of horizontal lines at q with height p, setting
    # the y limits to a reasonable heights
    plot(q, p, type = "h", xlab = "x", ylab = "probability", main = "pmf", ylim = c(0,
        max(p) + 0.1))
    # Usually these plots have a dot at the end of the line; the point function
    # will add these dots to the plot created above
    points(q, p, pch = 16, cex = 2)
}

# Plot the pmf of a Bernoulli rv
ber_q <- c(0, 1)
ber_pmf <- dbinom(ber_q, 1, 0.5)
plot_pmf(ber_q, ber_pmf)
```



```r
# Plot the cdf of a Bernoulli rv
ber_q2 <- seq(-1, 2, length = 1000)
ber_cdf <- pbinom(ber_q2, 1, 0.5)
# Plot the cdf with a line plot
plot(ber_q2, ber_cdf, type = "l", xlab = "x", ylab = "probability", main = "cdf",
    ylim = c(0, 1.1))
```




```r
# The quartiles of a Bernoulli distribution
qbinom(seq(0, 1, by = 0.25), 1, 0.5)
```

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

```r
# Generate 10 random values from a Bernoulli distribution
rbinom(10, 1, 0.5)
```

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

Binomial

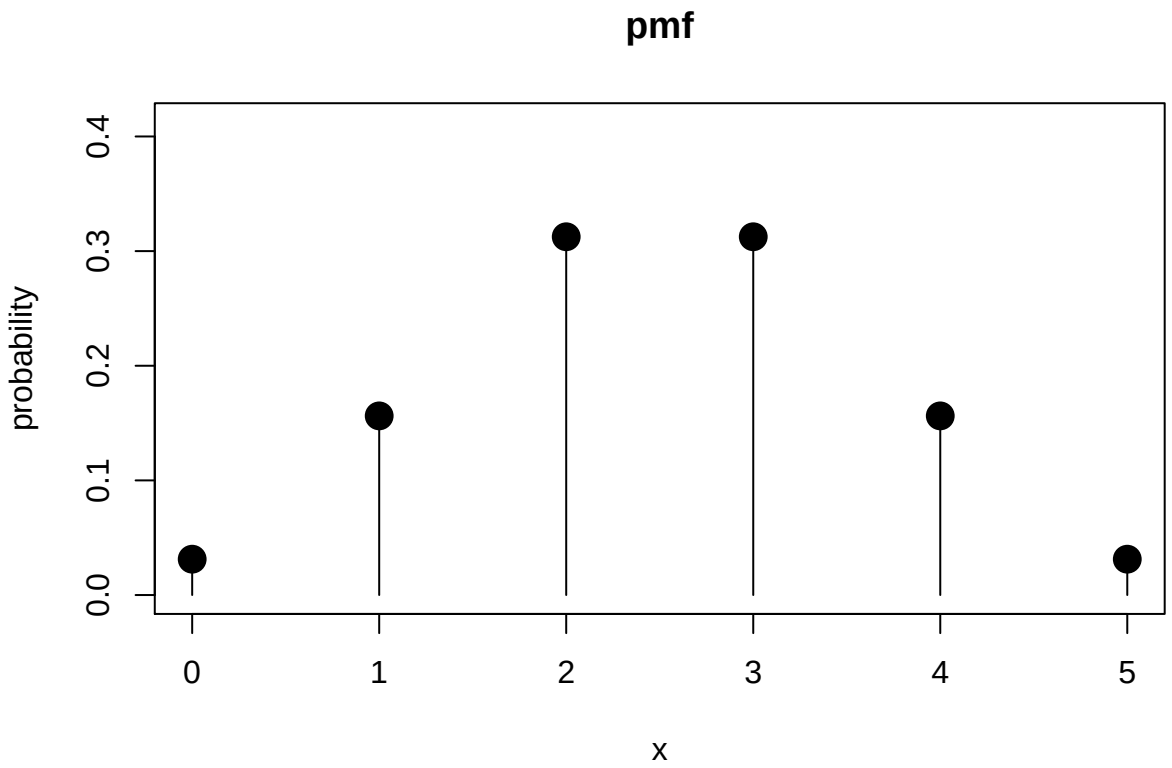
A binomial random variable is specified by two parameters, n and p, and is denoted by:

$$X \sim \text{BINOM}(n, p)$$

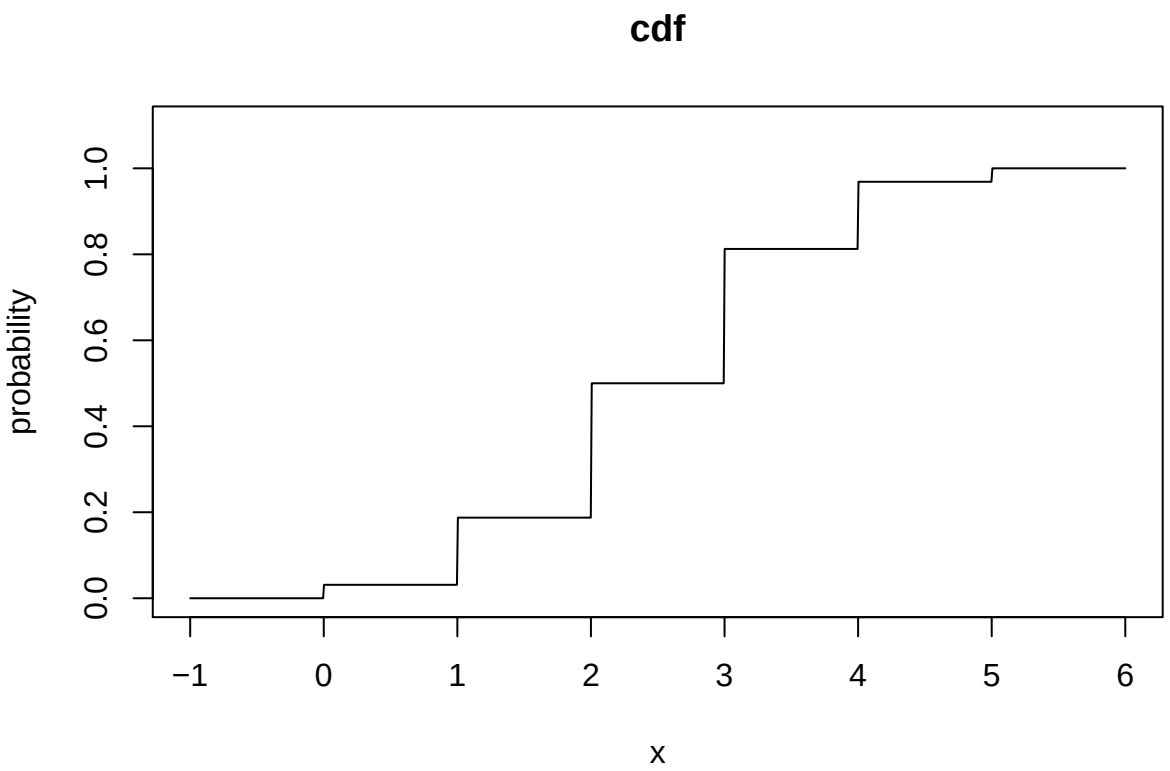
Binomial random variables represent the number of times n Bernoulli random variables are 1 when the probability of seeing 1 is p (another way to think of this is that if a coin flip is a Bernoulli random variable, then the number of times the coin lands heads-up when flipped n times is a binomial random variable).

The functions for the binomial distribution are `dbinom()`, `pbinom()`, `qbinom()`, and `rbinom()`. All of these have arguments `size` and `prob` for specifying the parameters `n` and `p` of the binomial distribution respectively.

```
# Let's consider a model where a fair coin is flipped five times.  The pmf
# of the binomial rv:
binom_q <- 0:5
binom_pmf <- dbinom(binom_q, size = 5, prob = 0.5)
plot_pmf(binom_q, binom_pmf)
```




```
# The cdf of a binomial rv
binom_q2 <- seq(-1, 6, length = 1000)
binom_cdf <- pbinom(binom_q2, size = 5, prob = 0.5)
plot(binom_q2, binom_cdf, type = "l", xlab = "x", ylab = "probability", main = "cdf",
    ylim = c(0, 1.1))
```




```
# The quartiles of this binomial rv
qbinom(seq(0, 1, by = 0.25), size = 5, prob = 0.5)
```

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

```
# 10 simulated binomial rv
rbinom(10, size = 5, prob = 0.5)
```

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

Geometric

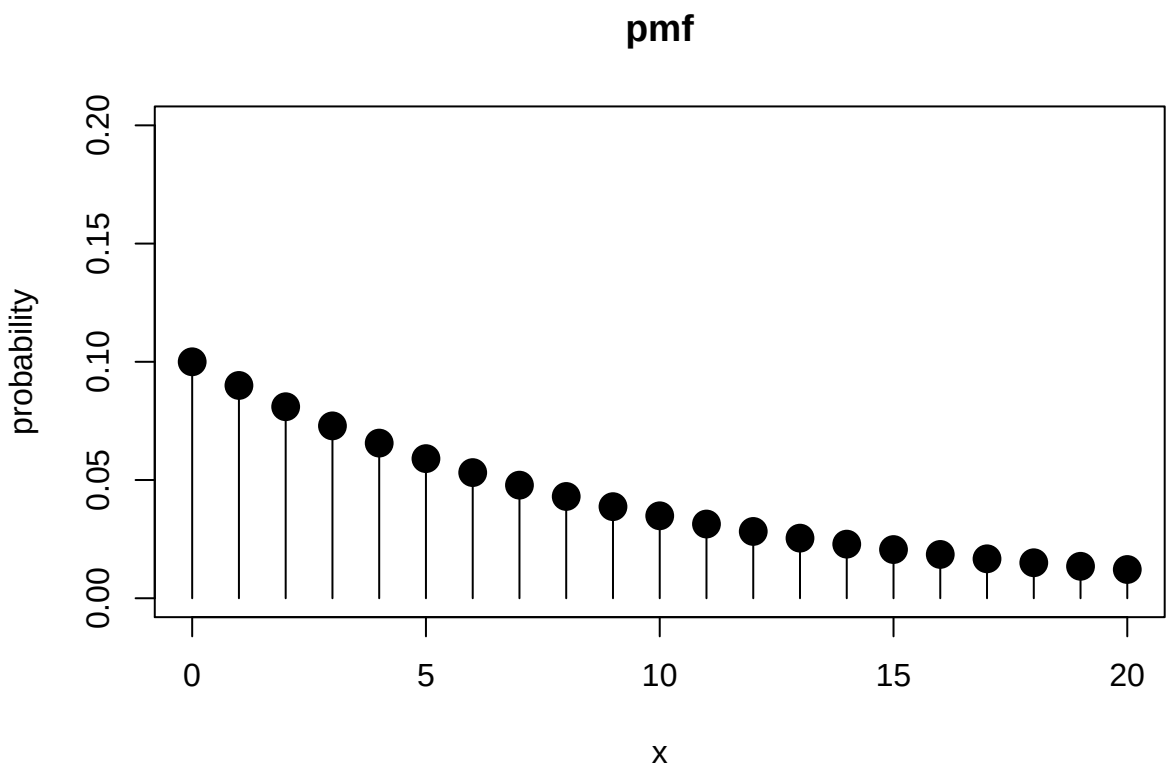
A geometric random variable is specified by one parameter, p, and is denoted by:

$$X \sim \text{GEOM}(p)$$

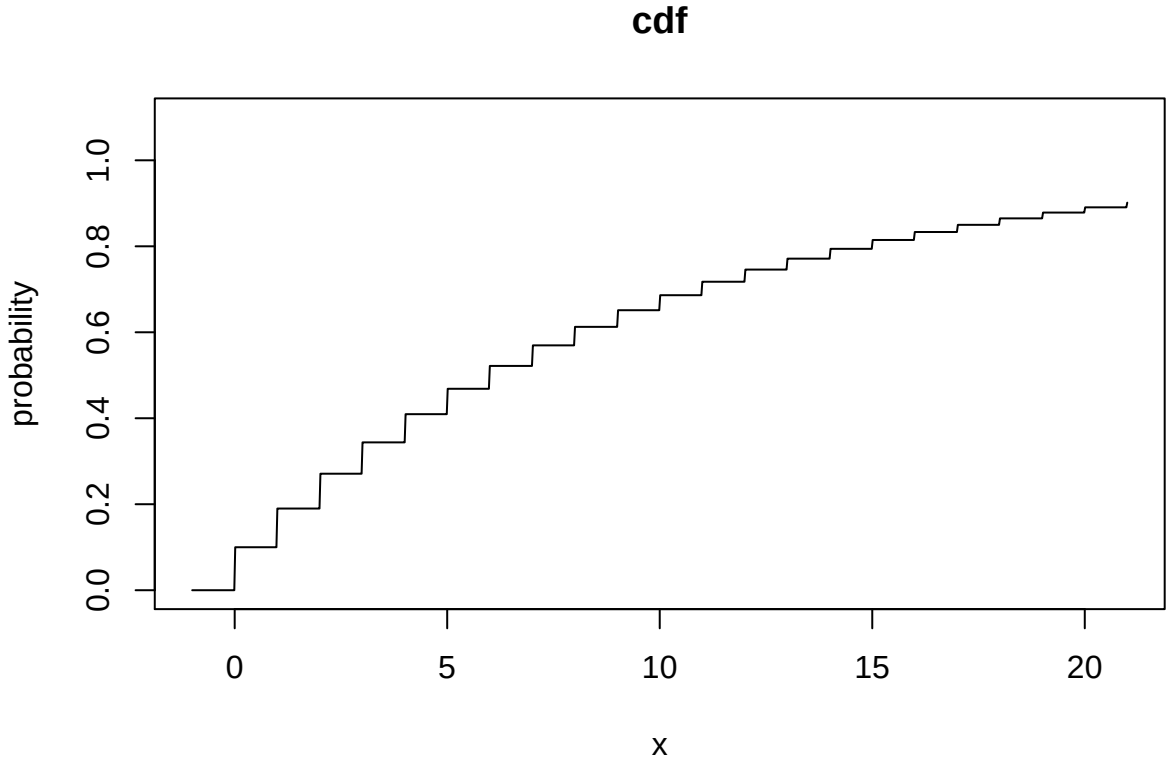
This random variable represents how many times a Bernoulli random variable with parameter p is created until a 1 is seen. (In other words, the number of times a coin is flipped until it lands heads-up is a geometric random variable). The name comes from the fact that the pmf of a geometric random variable is a geometric series, or $p(x) = p(1-p)^{x-1}$.

The functions for working with geometric random variables are `dgeom()`, `pgeom()`, `qgeom()`, and `rgeom()`. All of these have an argument `prob` used to represent the parameter `p` of the geometric distribution.

```
# In these examples we will work with a geometric rv where p = 1. I now show
# part of its pmf
geom_q <- 0:20
geom_pmf <- dgeom(geom_q, prob = 0.1)
plot_pmf(geom_q, geom_pmf)
```




```
# The cdf of this variable
geom_q2 <- seq(-1, 21, length = 1000)
geom_cdf <- pgeom(geom_q2, prob = 0.1)
plot(geom_q2, geom_cdf, type = "l", xlab = "x", ylab = "probability", main = "cdf",
    ylim = c(0, 1.1))
```



```r
# Quartiles of a geometric rv
qgeom(seq(0, 1, by = 0.25), prob = 0.1)
```

```
## [1]   0   2   6  13 Inf
```

```r
# 10 simulated geometric rv's
rgeom(10, prob = 0.1)
```

```
##  [1] 19  3 11  8 10  4 14 13 11  0
```

Poisson

A Poisson random variable is specified by one parameter, λ, and is denoted by:

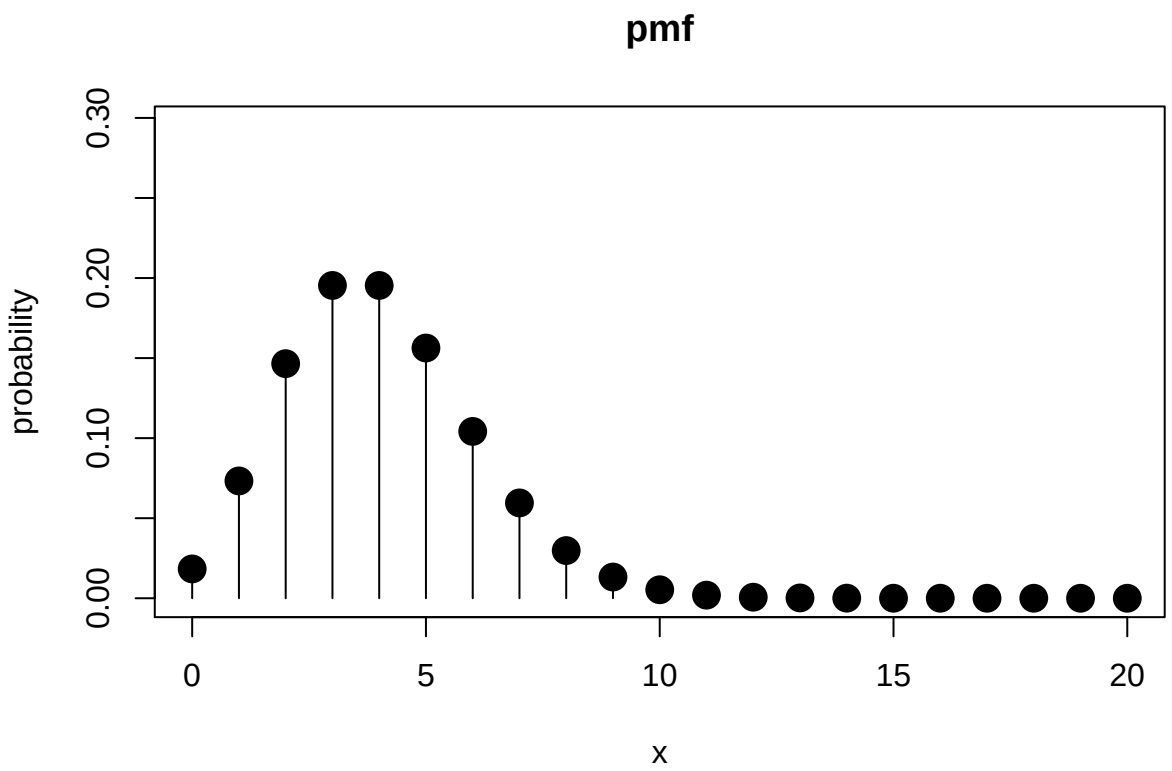
$$X \sim \text{POI}(\lambda)$$

Poisson random variables are used to model counts of an event that occur in a finite frame of time. They are, in some sense, the inverse of a geometric random variable; while a geometric random variable represents the length of time to see one "success", a Poisson random variable represents how many "successes" given a certain length of time. The parameter λ represents the average "success" rate. Some examples of Poisson random variables include:

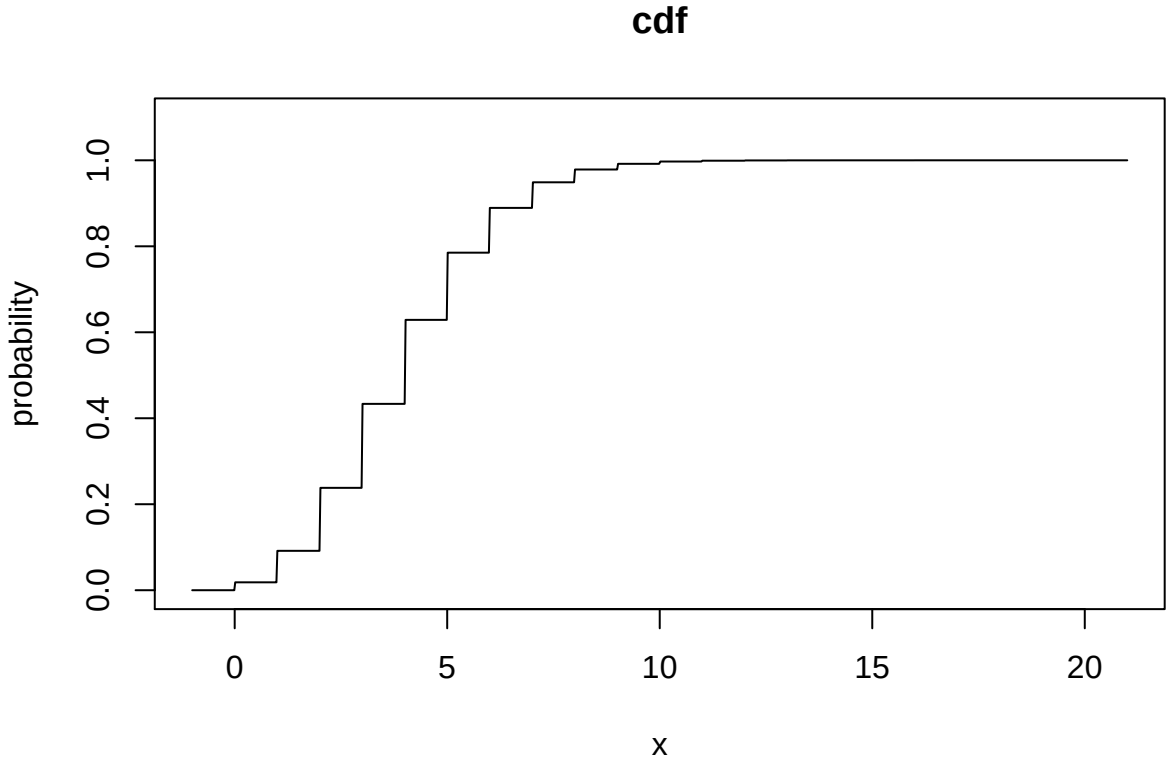
- The number of calls a call center receives in a work day.
- How many goals a soccer team scores in a game.
- The number of deaths from horse kicks in the Prussian army annually.

The R functions for working with Poisson random variables are `dpois()`, `ppois()`, `qpois()`, and `rpois()`. The argument `lambda` corresponds to the parameter λ.

```r
# For these examples, the average success rate is 4.  Creating a pmf:
pois_q <- 0:20
pois_pmf <- dpois(pois_q, lambda = 4)
plot_pmf(pois_q, pois_pmf)
```




```
# Creating a cdf
pois_q2 <- seq(-1, 21, length = 1000)
pois_cdf <- ppois(pois_q2, lambda = 4)
plot(pois_q2, pois_cdf, type = "l", xlab = "x", ylab = "probability", main = "cdf",
    ylim = c(0, 1.1))
```



```
# Quartiles of the Poisson rv
qpois(seq(0, 1, by = 0.25), lambda = 4)
```

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

```
# Simulating Poisson rv's
rpois(10, lambda = 4)
```

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

Uniform

The first continuous random variable I consider here is a uniformly distributed random variable, which is described by two parameters, a and b, and represented with:

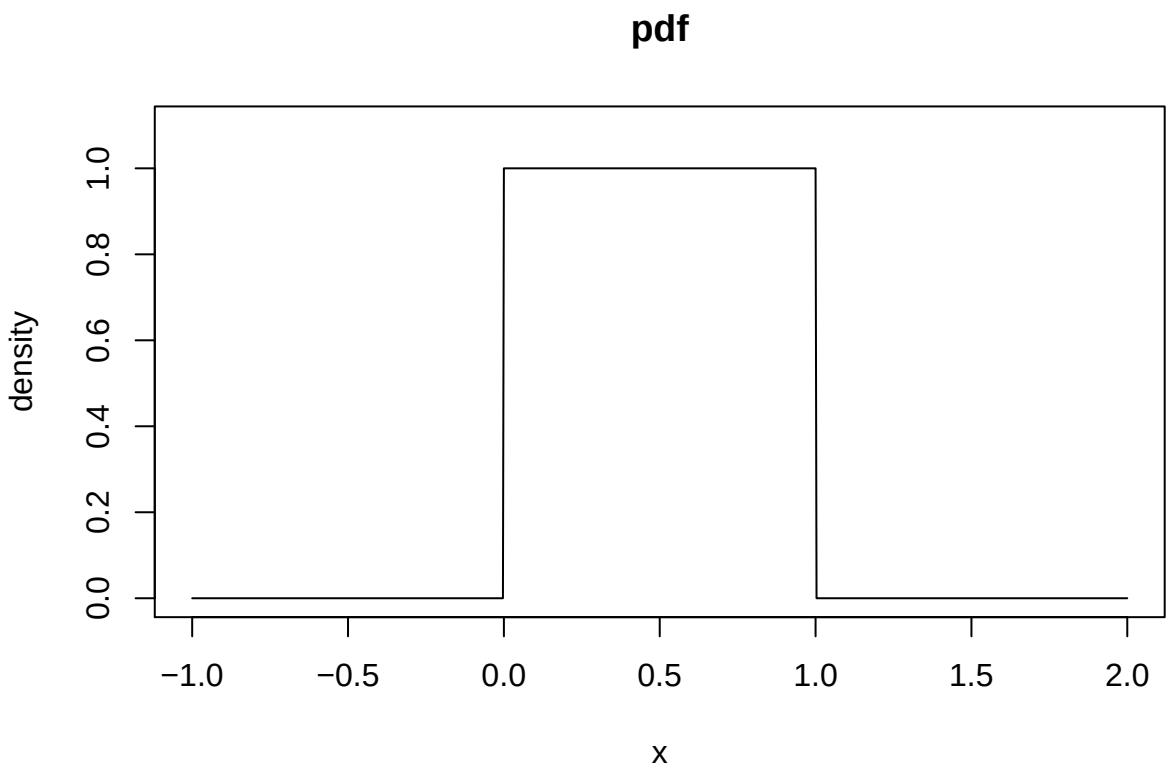
$$X \sim \text{UNIF}(a, b)$$

a is the smallest value possible a uniformly distributed rv could take, and b the largest. Every number between a and b is equally weighted. It is a very important random variable, since it describes the probability for any random variable the probability that the random variable is a certain quantile of its distribution. Additionally, computers simulate uniform random variables, and using them you can simulate every other random variable.

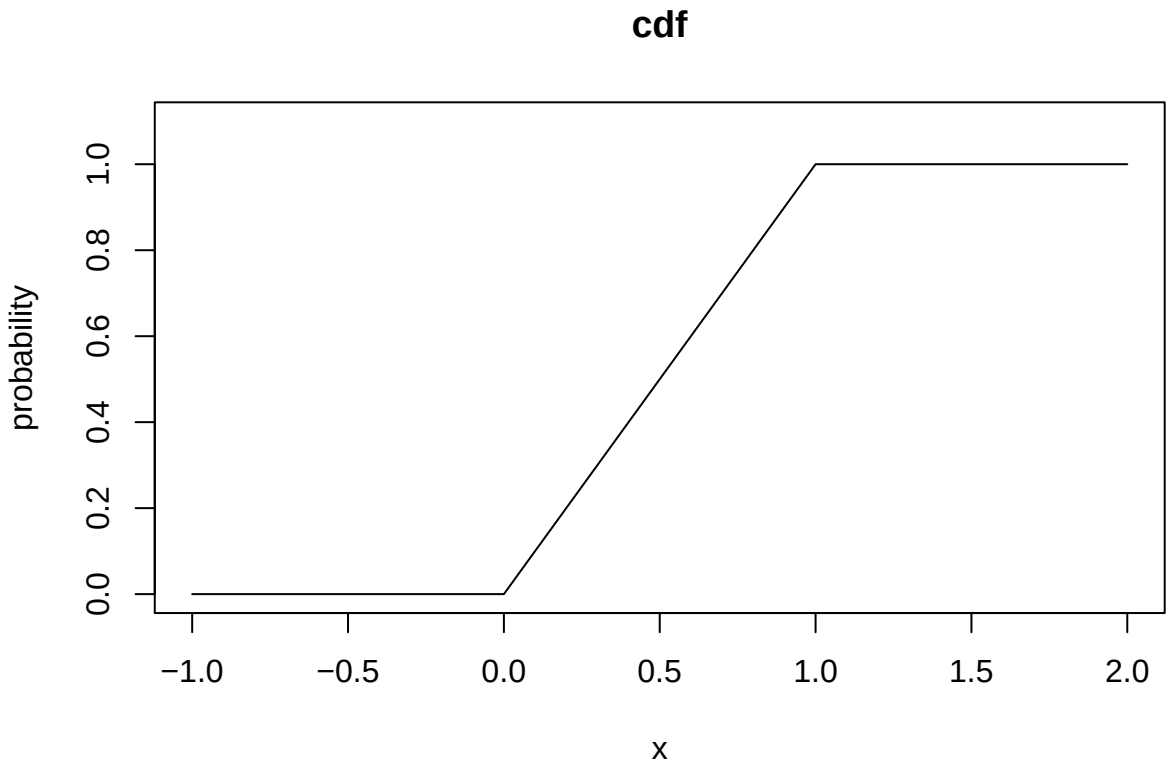
The `dunif()`, `punif()`, `qunif()`, and `runif()` functions are used for working with uniform rv's in R. They all have arguments `min` and `max` that correspond to parameters a and b mentioned above.

Being a continuous random variable, rather than having a pmf, a probability density function (pdf) is used for describing the random variable.

```
# Throughout I will set a = 0 and b = 1. This could be considered the
# standard Uniform random variable.  The pdf of a uniform rv:
unif_q <- seq(-1, 2, length = 1000)
unif_pdf <- dunif(unif_q, min = 0, max = 1)
plot(unif_q, unif_pdf, type = "l", xlab = "x", ylab = "density", main = "pdf",
    ylim = c(0, max(unif_pdf) + 0.1))
```




```
# The cdf of a uniform rv
unif_cdf <- punif(unif_q, min = 0, max = 1)
plot(unif_q, unif_cdf, type = "l", xlab = "x", ylab = "probability", main = "cdf",
    ylim = c(0, 1.1))
```



```r
# Quartiles of a uniform rv
qunif(seq(0, 1, by = 0.25), min = 0, max = 1)
```

```
## [1] 0.00 0.25 0.50 0.75 1.00
```

```r
# Simulated uniform rv's
runif(10, min = 0, max = 1)
```

```
##  [1] 0.7562467 0.9696724 0.8479128 0.2943741 0.9111995 0.2399134 0.7144876
##  [8] 0.3392606 0.8428848 0.8038884
```

Exponential

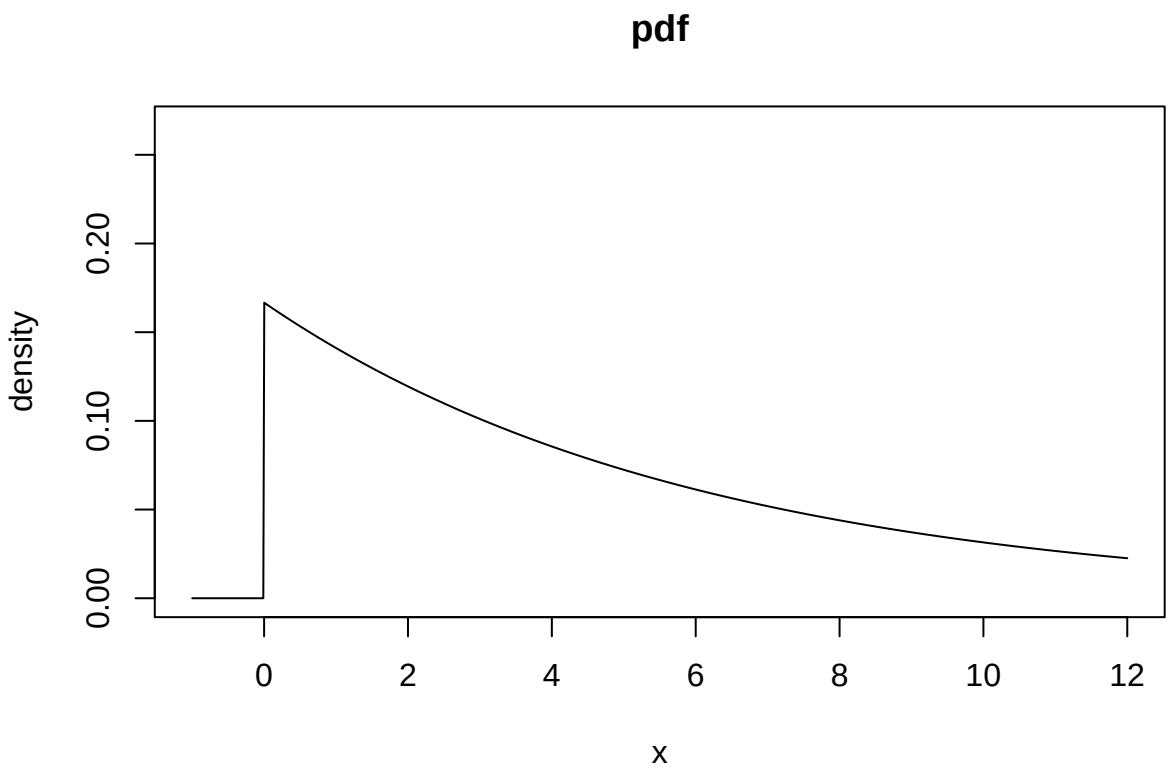
The exponential distribution is specified by a parameter μ and is described by:

$$X \sim \text{EXP}(\mu)$$

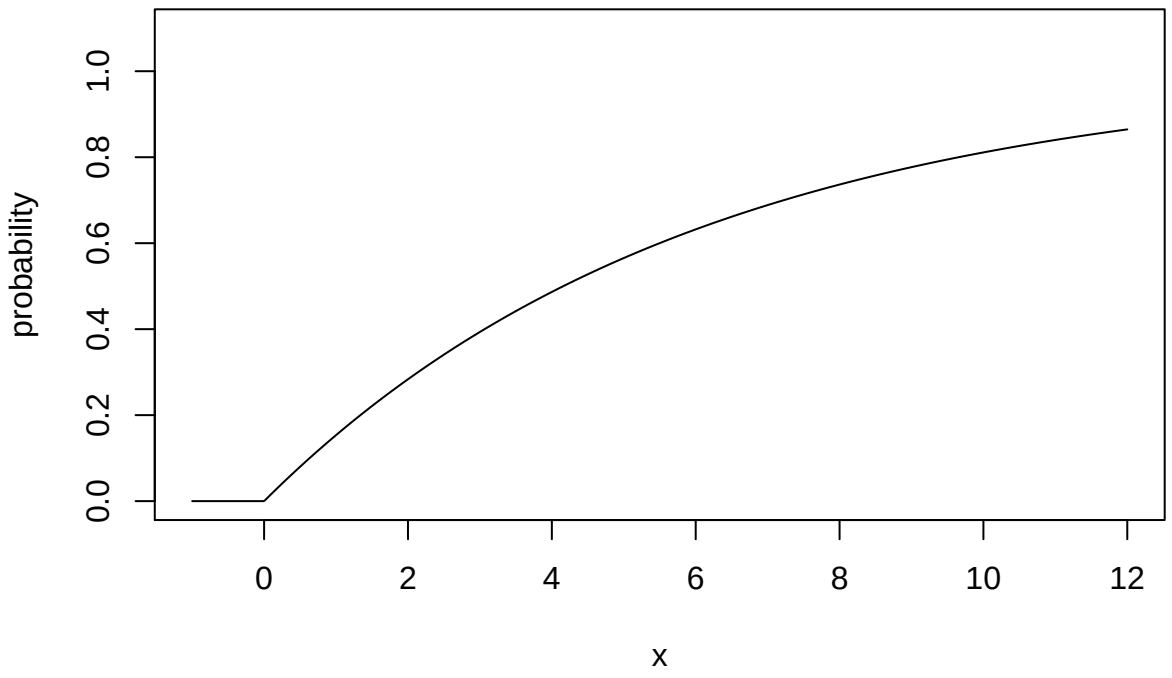
μ describes both the mean and the standard deviation of an exponential random variable. These are used to model waiting times, such as how long it takes a server to service a a customer and move on to the next one in the queue. Thus, μ is the average length of time to service one customer and move on to the next. One could also consider instead $\frac{1}{\mu}$, which is the rate at which customers are serviced. So if a call center takes on average half a minute to service a customer, the average rate at which it services its customers is two customers per minute. (You may notice some similarity between an exponential random variable and a geometric random variabe; the exponential rv is the continuous version of the geometric rv.)

The function `dexp()`, `pexp()`, `qexp()`, and `rexp()` are used for working with exponential random variables in R. You do not directly specify the mean wait time, μ, in R; instead you specify the rate, $\frac{1}{\mu}$, with the `rate` argument. Thus if you wanted a mean of $\mu = 6$, you would set `rate = 1/6`.

```r
# Here we will model an exponential rv where the mean is 6. This means that
# the rate is 1/6.  The pdf of an exponential rv:
exp_q <- seq(-1, 12, length = 1000)
exp_pdf <- dexp(exp_q, rate = 1/6)
plot(exp_q, exp_pdf, type = "l", xlab = "x", ylab = "density", main = "pdf",
    ylim = c(0, max(exp_pdf) + 0.1))
```




```
# The cdf of an exponential rv
exp_cdf <- pexp(exp_q, rate = 1/6)
plot(exp_q, exp_cdf, type = "l", xlab = "x", ylab = "probability", main = "cdf",
    ylim = c(0, 1.1))
```



```r
# Quartiles of an exponential rv
qexp(seq(0, 1, by = 0.25), rate = 1/6)
```

```
## [1] 0.000000 1.726092 4.158883 8.317766      Inf
```

```r
# Simulated exponential rv's
rexp(10, rate = 1/6)
```

```
##  [1]  4.181027  1.381246  2.254482  2.814817  1.777893 15.998047 13.546735
##  [8] 13.562386  4.017361  4.070043
```

Normal

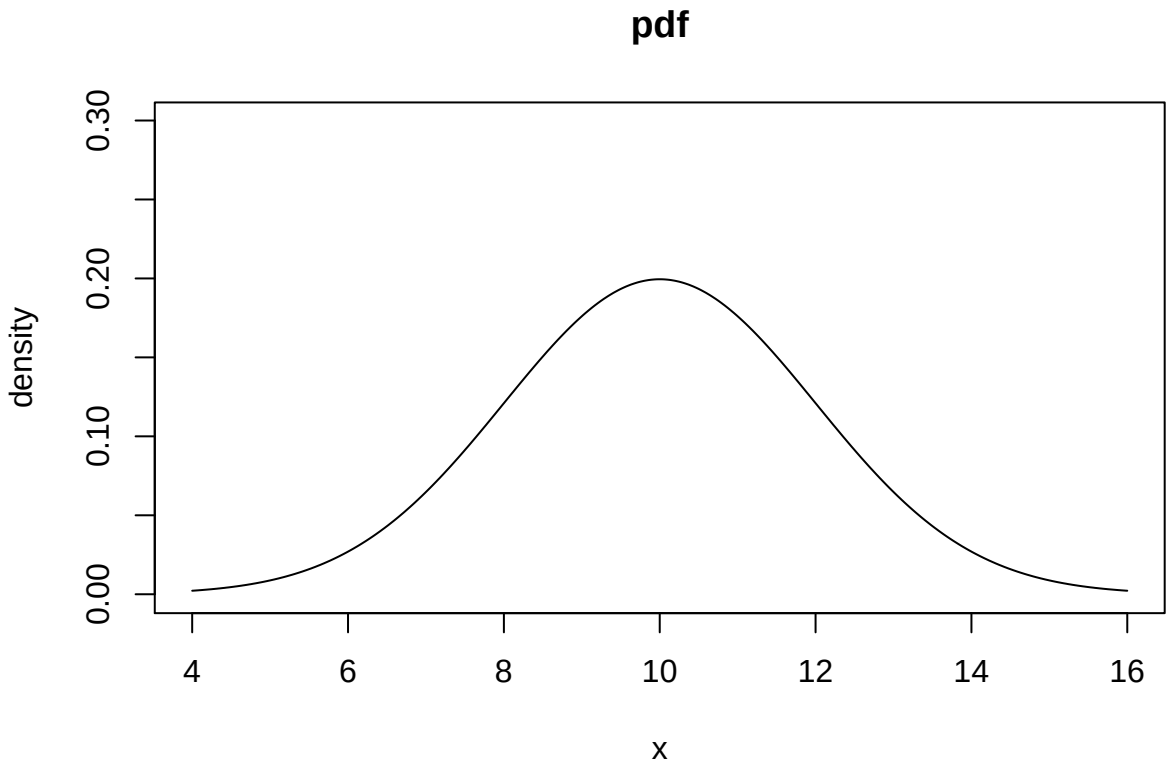
The Normal distribution is specified by two parameter, μ and σ, and is described by:

$$X \sim N(\mu, \sigma)$$

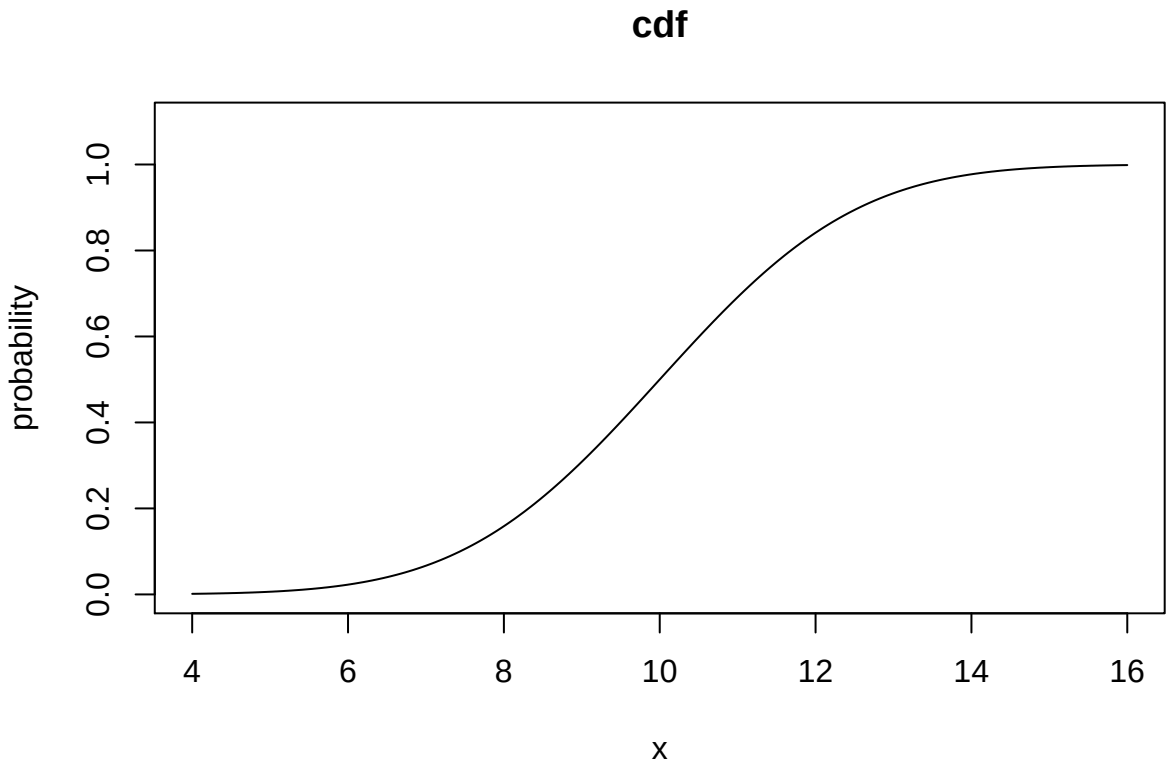
μ is the mean of a Normal rv, and σ is the rv's standard deviation. The Normal distribution describes the distribution of "errors", or now far away some random variable is from its mean. It is perhaps the most important distribution in probability and statistics.

The functions `dnorm()`, `pnorm()`, `qnorm()`, and `rnorm()` are used for working with Normal rv's. They have two arguments, `mean` and `sd`, used for specifying the parameters μ and σ, respectively.

```r
# We will set the mean to 10 and the standard deviation to 2 when working
# with Normal rv's in this example.  The pdf of a Normal rv:
norm_q <- seq(4, 16, length = 1000)
norm_pdf <- dnorm(norm_q, mean = 10, sd = 2)
plot(norm_q, norm_pdf, type = "l", xlab = "x", ylab = "density", main = "pdf",
    ylim = c(0, max(norm_pdf) + 0.1))
```



```r
# The cdf of a Normal rv:
norm_cdf <- pnorm(norm_q, mean = 10, sd = 2)
plot(norm_q, norm_cdf, type = "l", xlab = "x", ylab = "probability", main = "cdf",
    ylim = c(0, 1.1))
```




```r
# The quartiles of a Normal rv
qnorm(seq(0, 1, by = 0.25), mean = 10, sd = 2)
```

```
## [1]     -Inf  8.65102 10.00000 11.34898      Inf
```

```r
# Simulated Normal rv's
rnorm(10, mean = 10, sd = 2)
```

```
##  [1] 12.541297 11.221620  7.411588 13.483226 12.046666  8.065685 10.076769
##  [8] 10.338442  7.755875  6.666931
```

Central Limit Theorem

One of the most important theorems in all of statistics is the central limit theorem. This theorem describes the distribution of the sample mean as the sample size grows large. Let X_i be a random variable with mean μ and standard deviation σ. For every i such that $1 \leq i \leq n$ (n representing the sample size), X_i is independent and identically distributed, or iid (meaning that every data point is drawn from the exact same distribution and do not depend on the value of other X_i in the data set). Let $\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ be the sample mean of this data set. Then:

$$\overline{X} \sim N\left(\mu, \frac{\sigma}{\sqrt{n}}\right) \text{ as } n \to \infty$$

This is saying that if the above conditions hold, the distribution of the sample mean $\overline{X}$ begins to resemble

the Normal distribution with larger sample sizes, *regardless of the original distribution of* X_i. This is the basis of much of statistical inference.

We can show the central limit theorem at work in R. Let $X_i \sim \text{EXP}(5)$. Notice that $\mu = 5$ and $\sigma = \mu = 5$. If the sample size is n, the central limit theorem says that the distribution of $\overline{X}$ should begin to resemble the Normal distribution with mean $\mu = 5$ and standard deviation:

$$\frac{\sigma}{\sqrt{n}} = \frac{5}{\sqrt{n}}$$

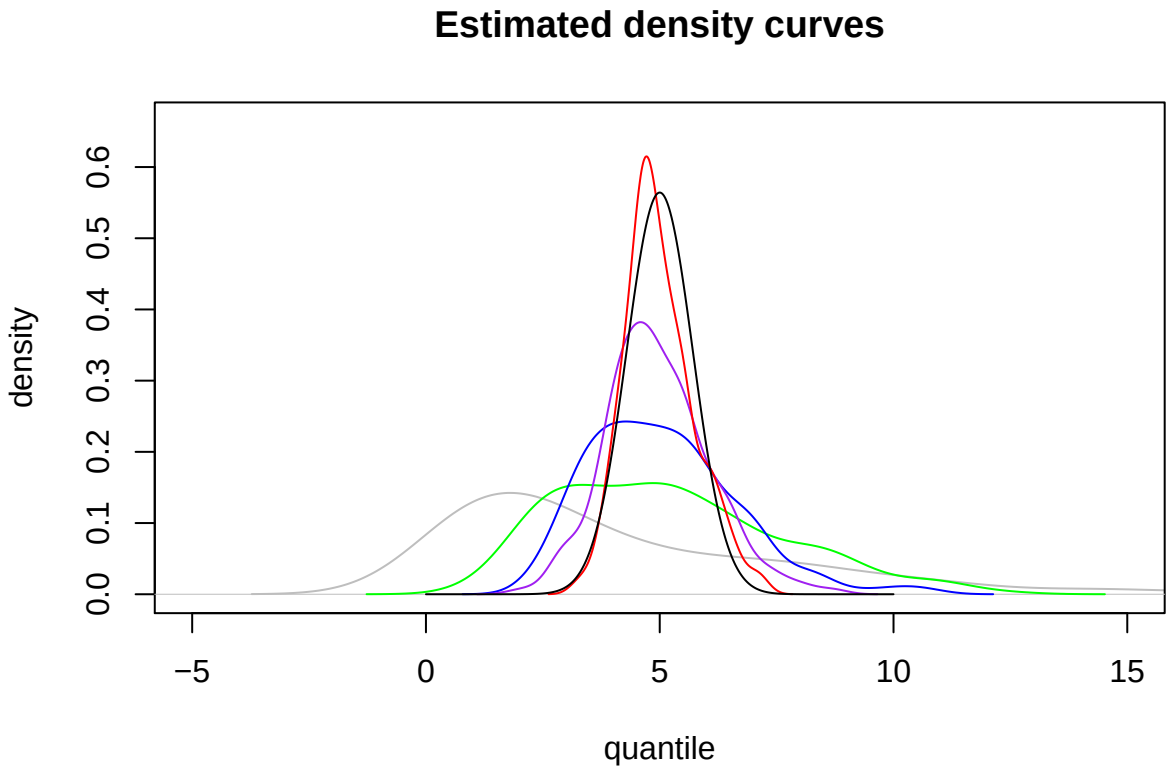
as n grows large.

Let's consider samples of size 10, 50, 200, and 1000. I first generate 200 simulated means from samples for each of these sizes.

```
# We will loop over the vector c(10, 50, 200, 500), each of these
# representing a sample size. lapply() applies a function (the second
# argument) to each element of this vector, and stores the result as a list,
# which we will call sim_means. The (anonymous) function will take only one
# argument, n, representing the sample size. It will return a vector of
# simulated means when the sample size is n.
sim_means <- lapply(c(5, 10, 20, 50), function(n) {
    # I now simulate 200 data sets of size n from the exponential distribution,
    # setting rate = 1/5 for exponentially distributed random variables with
    # mean 5, and store the resulting data sets in a list called sim_data_sets.
    # (I add a throwaway argument because something will be passed to the
    # function, but I don't want to use it)
    sim_data_sets <- lapply(1:200, function(throwaway) rexp(n, rate = 1/5))
    # Return a vector containing means for each of these data sets.
    sapply(sim_data_sets, mean)
})
# Let's look at what we just made
str(sim_means)
```

```
## List of 4
##  $ : num [1:200] 3.51 4.13 2.59 2.91 4.67 ...
##  $ : num [1:200] 5.38 3.29 5.43 2.68 4.21 ...
##  $ : num [1:200] 4.01 2.83 6.64 6.62 7.09 ...
##  $ : num [1:200] 7.06 5.27 5.61 4.31 4.78 ...
```

```
# First, I plot an estimated density function for simulated exponential
# random variables with mean 5 (rate 1/5)
plot(density(rexp(200, rate = 1/5)), xlab = "quantile", ylab = "density", main = "Estimated density cur
    col = "gray", type = "l", xlim = c(-5, 15), ylim = c(0, dnorm(5, mean = 5,
        sd = 5/sqrt(50)) + 0.1))
# I add density curves for each of the simulated means
lines(density(sim_means[[1]]), col = "green")
lines(density(sim_means[[2]]), col = "blue")
lines(density(sim_means[[3]]), col = "purple")
lines(density(sim_means[[4]]), col = "red")
# Finally, add a Normal curve showing what the distribution of the final
# sample mean should be near according to the central limit theorem
clt_norm_q <- seq(0, 10, length = 1000)
lines(clt_norm_q, dnorm(clt_norm_q, mean = 5, sd = 5/sqrt(50)), col = "black")
```


Estimated density curves
density
0.0
0.1
0.2
0.3
0.4
0.5
0.6
−5
0
5
10
15
quantile

Lecture 10

In this lecture, we will discuss increasingly popular packages and methods in R that are intended to make code more readable or make otherwise complicated tasks simpler. No package listed here is included in the base R installation, and must be installed from CRAN. That said, they can make working in R much easier.

magrittr and the Pipe Operator

Let's say that you have a long list of tasks that you want to perform on a data set. For example, you may want to take a data set, subset it, fit a linear model to it, and then get the coefficients. There are a few ways to do these four steps:

1. Perform them all in one line, like so (for the `iris` data set):

```
coefficients(lm(Sepal.Length ~ Sepal.Width,
                data = subset(iris, select = -c(Species),
                              subset = Species == "virginica")))
```

```
## (Intercept) Sepal.Width
##   3.9068365   0.9015345
```

2. Perform each step on its own line, saving the results of the previous step and using the object you created in the next step:

```
iris_subset <- subset(iris, select = -c(Species),
                      subset = Species == "virginica")
mod <- lm(Sepal.Length ~ Sepal.Width, data = iris_subset)
coefficients(mod)
```

```
## (Intercept) Sepal.Width
##   3.9068365   0.9015345
```

Not everyone is satisfied with this approach. The first can become quite long, and furthermore, you need to effectively read from right to left in order to understand what was done. The second approach requires keeping track of what each object in each step is, and thus can also be confusing.

The **magrittr** package, though, provides a third option: the pipe operator, `%>%`. The logic of the operator is simple: `x %>% f == f(x)`. In other words, `x` becomes the first argument passed to the function `f()`. If `f()` takes multiple arguments, then the following holds: `x %>% f(y) == f(x, y)`. If you wish for the pipe operator to pass the object on the left-hand side to something other than the first argument of `f()`, you can substitute `.` for the argument you wish to pass the object to: in other words, `y %>% f(x, .) == f(x, y)`. You can also use `.` to pass the object on the left-hand side to a named argument, so `y %>% f(x, foo = .) == f(x, foo = y)`.

While there's no clear advantage to using the `%>%` operator when you're dealing with just one object, when you wish to *chain* operations, it suddenly becomes very clear why many like using it. Here's the above

example repeated using %>%:

```
library(magrittr)
iris %>%
  subset(select = -c(Species), subset = Species == "versicolor") %>%
  lm(Sepal.Length ~ Sepal.Width, data = .) %>%
  coefficients
```

```
## (Intercept) Sepal.Width
##   3.5397347   0.8650777
```

Furthermore, you can read the operation from left to right to understand what was done, and easily insert additional steps in the chain. Maybe you with to **standardize** the variables, wich is subtracting each observation by the mean and dividing by the standard deviation. It's easy to insert that step into the above pipeline:

```
iris %>%
  subset(select = -c(Species), subset = Species == "versicolor") %>%
  scale %>%
  as.data.frame %>% # scale returns a matrix, but we want a data frame
  lm(Sepal.Length ~ Sepal.Width, data = .) %>%
  coefficients
```

```
##  (Intercept)  Sepal.Width
## 8.949085e-17 5.259107e-01
```

If we were to "unwrap" the above code, it would be equivalent to:

```
# Have fun reading this and understanding what it does
coefficients(lm(Sepal.Length ~ Sepal.Width, data = as.data.frame(
  scale(subset(iris, select = -c(Species),
               subset = Species == "versicolor")))))
```

```
##  (Intercept)  Sepal.Width
## 8.949085e-17 5.259107e-01
```

magrittr includes operators other than the %>% operator, such as:

- The %<>% operator, where the results of the function on the right are saved to the object on the left, or `x %<>% f == {x <- x %>% f} == {x <- f(x)}`:

```
# For demonstration purposes, make three copies of rivers
rivers1 <- rivers
rivers2 <- rivers
rivers3 <- rivers

# We will save the mean of rivers in these variables in three different ways
# 1: Without any pipe operators (By the way, did you know that if you wrap a
# command in parentheses, the object that is saved will also be printed in
# the console? This is a convenient way to both save and see the result of
# an operation.)
(rivers1 <- mean(rivers1))
```

```
## [1] 591.1844
```

```
# 2: With %>%
(rivers2 <- rivers2 %>% mean)
```

```
## [1] 591.1844
```

```
# 3: With %<>%
(rivers3 %<>% mean)
```

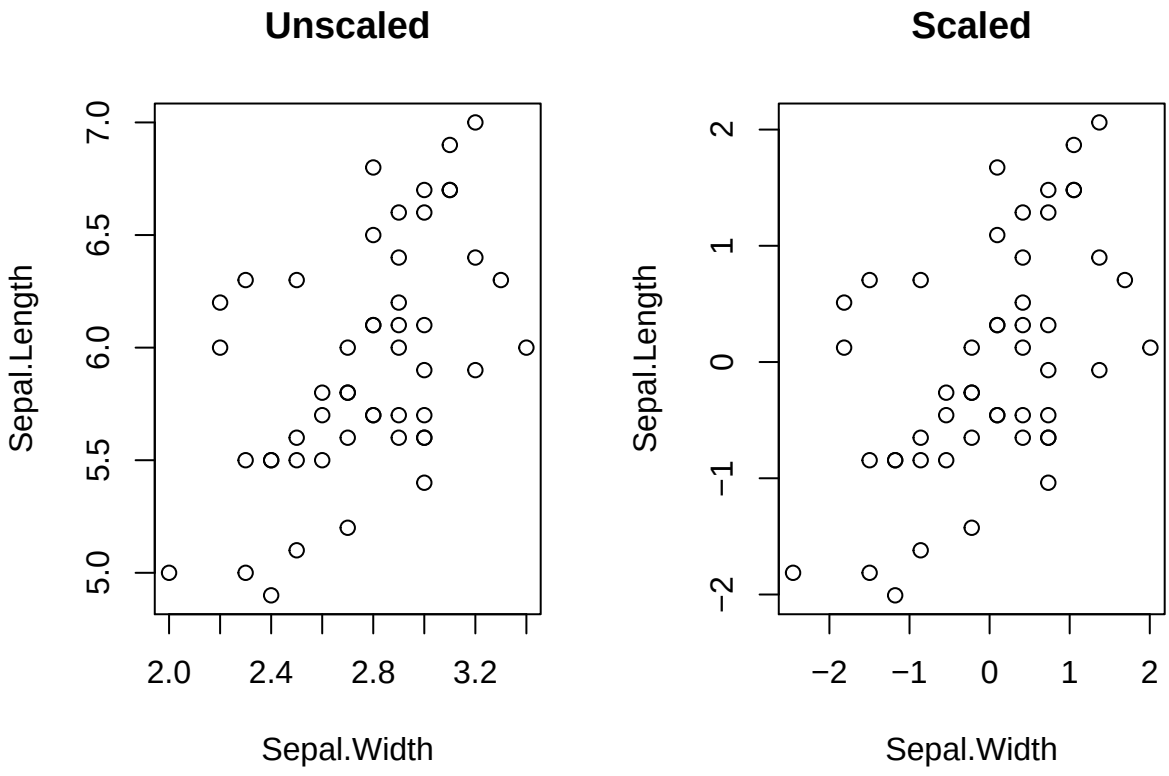
```
## [1] 591.1844
```

```
c(rivers1, rivers2, rivers3) %>% print
```

```
## [1] 591.1844 591.1844 591.1844
```

- The %T>% operator, where the value on the left is returned instead of the operation on the right (useful if you want to see something about an intermediate step):

```
# Save current par settings
old_par <- par()
par(mfrow = c(1,2))

iris %>%
  subset(select = -c(Species), subset = Species == "versicolor") %T>%
  # The next step in the chain will not be passed to the following step, since
  # above is %T>% rather than %>%; this is good because we don't want to use a
  # plot object, just see it
  plot(Sepal.Length ~ Sepal.Width, data = ., main = "Unscaled") %>%
  scale %>%
  as.data.frame %T>% # scale returns a matrix, but we want a data frame
  # The next step in the chain will not be passed to the following step, since
  # above is %T>% rather than %>%; this is good because we don't want to use a
  # plot object, just see it
  plot(Sepal.Length ~ Sepal.Width, data = ., main = "Scaled") %>%
  lm(Sepal.Length ~ Sepal.Width, data = .) %>%
  coefficients
```




```
##  (Intercept)  Sepal.Width
```

```
## 8.949085e-17 5.259107e-01
```

```
# Return to original settings
par(old_par)
```

```
## Warning in par(old_par): graphical parameter "cin" cannot be set

## Warning in par(old_par): graphical parameter "cra" cannot be set

## Warning in par(old_par): graphical parameter "csi" cannot be set

## Warning in par(old_par): graphical parameter "cxy" cannot be set

## Warning in par(old_par): graphical parameter "din" cannot be set

## Warning in par(old_par): graphical parameter "page" cannot be set
```

- The `%$%` operator works like `%>%` but also exposes the names of the object on the left-hand side to the operation on the right-hand side; in effect, it's a combination of `%>%` and `with()`. It other words, if `z` is a variable in `x` (perhaps `x` is a data frame or list), then `x %$% f(z) == with(x, z %>% f) == with(x, f(z))`:

```
# Intelligent use of the %$% operator allows us to avoid using the data argument of lm, making the pipe
par(mfrow = c(1,2))

iris %>%
  subset(select = -c(Species), subset = Species == "versicolor") %>%
  scale %>%
  as.data.frame %$% # Now we don't need data = . ; there is an implicit with()
  lm(Sepal.Length ~ Sepal.Width) %>%
  coefficients
```

```
##  (Intercept)  Sepal.Width
## 8.949085e-17 5.259107e-01
```

```
# Here is an equivalent, ugly one-line command
coefficients(with(
  as.data.frame(
    scale(
      subset(iris, select = -c(Species), subset = Species == "versicolor"))),
  lm(Sepal.Length ~ Sepal.Width)))
```

```
##  (Intercept)  Sepal.Width
## 8.949085e-17 5.259107e-01
```

You can learn more about **magrittr** and the pipe operator here.

dplyr: Subsetting With Power

Throughout this course, we have seen many methods for subsetting a data frame, including bracket notation (that is, `df[x,y]`) and, in particular, the `subset()` function. The bracket notation can easily become unruly (especially if you are using `which()`). `subset()` is much better, but the notation is not necessarily convenient and the function itself is slow and limited in use (it is basically a convenience method for the bracket notation). Furthermore, these functions will not do everything you want (say, order the rows of a data frame, or conveniently rename columns).

Hadley Wickham, the main creator of **ggplot2**, created a package called **dplyr** that provides alternative subsetting methods. The package is designed to support use of the `%>%` operator (**magrittr** is loaded in automatically when you load **dplyr**) with data manipulation functions. Also, while `subset()` is intended to

work just with data frames and doesn't provide any speed advantages, **dplyr** functions are not only often faster, they can work with objects other than data frames, such as a connection to a SQL data base (the logic of **dplyr** functions is similar in many ways to SQL logic, and with both can be modeled abstractly with relational algebra).

Basic subsetting using `select()` and `filter()`

The **dplyr** functions `select()` and `filter()` together give you all the capabilities of `subset()` in base R.

`select(df, ...)` chooses the columns from the data frame `df` to include in the output. You can make this selection exactly like you would using the `select` argument in `subset()` (with negative indices, `:`, or just listing out the columns you want).

`filter(df, logical_statement)` chooses the rows from the data frame `df` to include in the output, using syntax identical to that of the `subset` argument of the `subset` function. You write out a logical statement describing which rows you want, depending on column values.

Here are two ways to get the same data frame, using `subset()` and the **dplyr** functions.

```
# The subset way
subset(iris, select = Petal.Length:Species, subset = Species == "virginica" &
    Petal.Length > mean(Petal.Length))
```

```
##     Petal.Length Petal.Width   Species
## 101          6.0         2.5 virginica
## 102          5.1         1.9 virginica
## 103          5.9         2.1 virginica
## 104          5.6         1.8 virginica
## 105          5.8         2.2 virginica
## 106          6.6         2.1 virginica
## 107          4.5         1.7 virginica
## 108          6.3         1.8 virginica
## 109          5.8         1.8 virginica
## 110          6.1         2.5 virginica
## 111          5.1         2.0 virginica
## 112          5.3         1.9 virginica
## 113          5.5         2.1 virginica
## 114          5.0         2.0 virginica
## 115          5.1         2.4 virginica
## 116          5.3         2.3 virginica
## 117          5.5         1.8 virginica
## 118          6.7         2.2 virginica
## 119          6.9         2.3 virginica
## 120          5.0         1.5 virginica
## 121          5.7         2.3 virginica
## 122          4.9         2.0 virginica
## 123          6.7         2.0 virginica
## 124          4.9         1.8 virginica
## 125          5.7         2.1 virginica
## 126          6.0         1.8 virginica
## 127          4.8         1.8 virginica
## 128          4.9         1.8 virginica
## 129          5.6         2.1 virginica
## 130          5.8         1.6 virginica
## 131          6.1         1.9 virginica
```

```
## 132          6.4         2.0 virginica
## 133          5.6         2.2 virginica
## 134          5.1         1.5 virginica
## 135          5.6         1.4 virginica
## 136          6.1         2.3 virginica
## 137          5.6         2.4 virginica
## 138          5.5         1.8 virginica
## 139          4.8         1.8 virginica
## 140          5.4         2.1 virginica
## 141          5.6         2.4 virginica
## 142          5.1         2.3 virginica
## 143          5.1         1.9 virginica
## 144          5.9         2.3 virginica
## 145          5.7         2.5 virginica
## 146          5.2         2.3 virginica
## 147          5.0         1.9 virginica
## 148          5.2         2.0 virginica
## 149          5.4         2.3 virginica
## 150          5.1         1.8 virginica
```

```
# The dplyr way
library(dplyr)
```

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

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

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

```
iris %>% filter(Species == "virginica" & Petal.Length > mean(Petal.Length)) %>%
    select(Petal.Length:Species)
```

```
##    Petal.Length Petal.Width   Species
## 1           6.0         2.5 virginica
## 2           5.1         1.9 virginica
## 3           5.9         2.1 virginica
## 4           5.6         1.8 virginica
## 5           5.8         2.2 virginica
## 6           6.6         2.1 virginica
## 7           4.5         1.7 virginica
## 8           6.3         1.8 virginica
## 9           5.8         1.8 virginica
## 10          6.1         2.5 virginica
## 11          5.1         2.0 virginica
## 12          5.3         1.9 virginica
## 13          5.5         2.1 virginica
## 14          5.0         2.0 virginica
## 15          5.1         2.4 virginica
## 16          5.3         2.3 virginica
## 17          5.5         1.8 virginica
## 18          6.7         2.2 virginica
```

```
## 19          6.9         2.3 virginica
## 20          5.0         1.5 virginica
## 21          5.7         2.3 virginica
## 22          4.9         2.0 virginica
## 23          6.7         2.0 virginica
## 24          4.9         1.8 virginica
## 25          5.7         2.1 virginica
## 26          6.0         1.8 virginica
## 27          4.8         1.8 virginica
## 28          4.9         1.8 virginica
## 29          5.6         2.1 virginica
## 30          5.8         1.6 virginica
## 31          6.1         1.9 virginica
## 32          6.4         2.0 virginica
## 33          5.6         2.2 virginica
## 34          5.1         1.5 virginica
## 35          5.6         1.4 virginica
## 36          6.1         2.3 virginica
## 37          5.6         2.4 virginica
## 38          5.5         1.8 virginica
## 39          4.8         1.8 virginica
## 40          5.4         2.1 virginica
## 41          5.6         2.4 virginica
## 42          5.1         2.3 virginica
## 43          5.1         1.9 virginica
## 44          5.9         2.3 virginica
## 45          5.7         2.5 virginica
## 46          5.2         2.3 virginica
## 47          5.0         1.9 virginica
## 48          5.2         2.0 virginica
## 49          5.4         2.3 virginica
## 50          5.1         1.8 virginica
```

```
# The slice() function in dplyr allows selecting rows by index
iris %>% filter(Species == "virginica" & Petal.Length > mean(Petal.Length)) %>%
    select(Petal.Length:Species) %>% slice(1:10)
```

```
## # A tibble: 10 x 3
##    Petal.Length Petal.Width Species
##           <dbl>       <dbl> <fct>
##  1          6           2.5 virginica
##  2          5.1         1.9 virginica
##  3          5.9         2.1 virginica
##  4          5.6         1.8 virginica
##  5          5.8         2.2 virginica
##  6          6.6         2.1 virginica
##  7          4.5         1.7 virginica
##  8          6.3         1.8 virginica
##  9          5.8         1.8 virginica
## 10          6.1         2.5 virginica
```

Unlike `subset()`, though, you can easily rename columns if necessary when using `select()`, like so:

```
iris %>% filter(Species == "virginica" & Petal.Length > mean(Petal.Length)) %>%
    select(Length = Petal.Length, Width = Petal.Width, Species) %>% slice(1:10)
```

```
## # A tibble: 10 x 3
##    Length Width Species
##     <dbl> <dbl> <fct>
##  1    6     2.5 virginica
##  2    5.1   1.9 virginica
##  3    5.9   2.1 virginica
##  4    5.6   1.8 virginica
##  5    5.8   2.2 virginica
##  6    6.6   2.1 virginica
##  7    4.5   1.7 virginica
##  8    6.3   1.8 virginica
##  9    5.8   1.8 virginica
## 10    6.1   2.5 virginica
```

If you wish to rename the columns of an existing data frame without removing those not mentioned, use `rename()` instead:

```
(new_iris <- iris %>% select(Petal.Length:Species) %>% slice(1:10))
```

```
## # A tibble: 10 x 3
##    Petal.Length Petal.Width Species
##           <dbl>       <dbl> <fct>
##  1          1.4         0.2 setosa
##  2          1.4         0.2 setosa
##  3          1.3         0.2 setosa
##  4          1.5         0.2 setosa
##  5          1.4         0.2 setosa
##  6          1.7         0.4 setosa
##  7          1.4         0.3 setosa
##  8          1.5         0.2 setosa
##  9          1.4         0.2 setosa
## 10          1.5         0.1 setosa
```

```
new_iris %>% rename(Length = Petal.Length, Width = Petal.Width)
```

```
## # A tibble: 10 x 3
##    Length Width Species
##     <dbl> <dbl> <fct>
##  1    1.4   0.2 setosa
##  2    1.4   0.2 setosa
##  3    1.3   0.2 setosa
##  4    1.5   0.2 setosa
##  5    1.4   0.2 setosa
##  6    1.7   0.4 setosa
##  7    1.4   0.3 setosa
##  8    1.5   0.2 setosa
##  9    1.4   0.2 setosa
## 10    1.5   0.1 setosa
```

Adding new variables with `mutate()`

Suppose you have a data frame and you want to add a new variable; perhaps you want to track the difference between `Sepal.Length` and `Sepal.Width` in the `iris` data set. `mutate()` (which is similar to the `transform()` function in base R) will allow for easily adding new variables in a data frame using existing variables. Suppose the data frame `df` contains the variables `x` and `y`. The call `mutate(df, z = f(x,y))` will

return a data frame identical to `df` but including a new variable `z` which is the result of applying `f` to the entry of the variable `x` and `y` to each row (note that `f` could be an expressions; say, subtraction). (Bear in mind that this works row-wise). If you want to return a data frame that *only* includes `z` (it does not include the original columns), use the `transmute()` function instead (not demonstrated).

```
iris %>% select(Petal.Length:Species) %>% filter(Species == "versicolor") %>%
    mutate(Difference = Petal.Length - Petal.Width, Ratio = Petal.Length/Petal.Width) %>%
    head
```

```
##   Petal.Length Petal.Width    Species Difference    Ratio
## 1          4.7         1.4 versicolor        3.3 3.357143
## 2          4.5         1.5 versicolor        3.0 3.000000
## 3          4.9         1.5 versicolor        3.4 3.266667
## 4          4.0         1.3 versicolor        2.7 3.076923
## 5          4.6         1.5 versicolor        3.1 3.066667
## 6          4.5         1.3 versicolor        3.2 3.461538
```

Other useful dplyr functions

I will mention three other functions you may find handy. The `summarize()` function will create a data frame with one row that contains requested numerical summaries, with the user specifying how to compute those summaries in the function call. An example is shown below:

```
iris %>% summarize(sepal_mean = mean(Sepal.Length), sepal_sd = sd(Sepal.Length),
    sepal_Q1 = quantile(Sepal.Width, 0.25), versicolor_count = sum(Species ==
        "versicolor"))
```

```
##   sepal_mean  sepal_sd sepal_Q1 versicolor_count
## 1   5.843333 0.8280661      2.8               50
```

Finally, the functions `sample_n()` and `sample_frac()` allow for choosing random rows from a data frame, with `sample_n()` allowing you to set the number of rows you desire, and `sample_frac()` the fraction of rows from the original data frame you want. They are demonstrated below:

```
iris %>% sample_n(10)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 147          6.3         2.5          5.0         1.9  virginica
## 66           6.7         3.1          4.4         1.4 versicolor
## 33           5.2         4.1          1.5         0.1     setosa
## 42           4.5         2.3          1.3         0.3     setosa
## 85           5.4         3.0          4.5         1.5 versicolor
## 123          7.7         2.8          6.7         2.0  virginica
## 36           5.0         3.2          1.2         0.2     setosa
## 68           5.8         2.7          4.1         1.0 versicolor
## 81           5.5         2.4          3.8         1.1 versicolor
## 1            5.1         3.5          1.4         0.2     setosa
```

```
iris %>% sample_frac(.15)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 47           5.1         3.8          1.6         0.2     setosa
## 131          7.4         2.8          6.1         1.9  virginica
## 25           4.8         3.4          1.9         0.2     setosa
## 18           5.1         3.5          1.4         0.3     setosa
## 62           5.9         3.0          4.2         1.5 versicolor
## 60           5.2         2.7          3.9         1.4 versicolor
```

```
## 143          5.8         2.7          5.1         1.9  virginica
## 82           5.5         2.4          3.7         1.0 versicolor
## 132          7.9         3.8          6.4         2.0  virginica
## 51           7.0         3.2          4.7         1.4 versicolor
## 114          5.7         2.5          5.0         2.0  virginica
## 28           5.2         3.5          1.5         0.2     setosa
## 89           5.6         3.0          4.1         1.3 versicolor
## 124          6.3         2.7          4.9         1.8  virginica
## 27           5.0         3.4          1.6         0.4     setosa
## 4            4.6         3.1          1.5         0.2     setosa
## 125          6.7         3.3          5.7         2.1  virginica
## 76           6.6         3.0          4.4         1.4 versicolor
## 139          6.0         3.0          4.8         1.8  virginica
## 122          5.6         2.8          4.9         2.0  virginica
## 73           6.3         2.5          4.9         1.5 versicolor
## 126          7.2         3.2          6.0         1.8  virginica
```

This may be handy for some procedures, such as bootstrapping.

dplyr is capable of much more than I have described. You can find a short introduction here, and a cheat sheet here, the latter showing that **dplyr** is capable of more complex data manipulation tasks such as joins, grouping, and aggregation.

reshape2: Melting and casting data

You may recall that in an earlier lecture I discussed the notion of data being in **long-form format**, where there are two variables in a data frame, one representing a value and another representing the variable to which the value corresponds. **Wide-form format**, in contrast, gives each variable its own column and the values of the variable are recorded as rows. We may need to reshape data so that it more resembles long-form or wide-form formats; this may be important for, say, plotting with **ggplot2**.

Hadley Wickham wrote the **reshape2** package to help with transforming the format of data. The package is based around two functions: `melt()` and `cast()`. `melt()` takes a wide-form data set and brings it into long-form format, while `cast()` takes a long-form data set and brings it into wide-form format.

Here's an example of what `melt()` will do to the `mtcars` data set:

```
library(reshape2)
melt(mtcars) %>% head
```

```
## No id variables; using all as measure variables
```

```
##   variable value
## 1      mpg  21.0
## 2      mpg  21.0
## 3      mpg  22.8
## 4      mpg  21.4
## 5      mpg  18.7
## 6      mpg  18.1
```

`melt()`'s default behavior is to assume that all variables with numeric data are variables with data you want to flatten, but you can explicitly idenfity the variables you want to be designated "id" variables (and thus not "melted"), like so:

```
melt(mtcars, id = c("gear", "carb")) %>% head
```

```
##   gear carb variable value
```

```
## 1    4    4      mpg  21.0
## 2    4    4      mpg  21.0
## 3    4    1      mpg  22.8
## 4    3    1      mpg  21.4
## 5    3    2      mpg  18.7
## 6    3    1      mpg  18.1
```

If we want to specify the names of the variables that contain names of variables and values (as opposed to the default names of `variable` and `value`) we can do so by setting the parameters `variable.name` and `value.name` in `melt()`:

```
melt(mtcars, id = c("gear", "carb"),
     variable.name = "foo",
     value.name = "bar") %>% head
```

```
##   gear carb foo  bar
## 1    4    4 mpg 21.0
## 2    4    4 mpg 21.0
## 3    4    1 mpg 22.8
## 4    3    1 mpg 21.4
## 5    3    2 mpg 18.7
## 6    3    1 mpg 18.1
```

While it's easy to "melt" a data set, "casting" it back from long-form format to wide-form format is more difficult. There are two functions, `dcast()` and `acast()`, which casts the data back to wide-form format, with `dcast()` returning a data frame and `acast()` array-like objects (vectors, matrices, arrays). We will look at `dcast()` here (they are similar).

A general `dcast()` call has the form `dcast(df, f, value.var = "value")`. `df`, the first parameter, is the data frame being cast. `f` is a formula that determines how the data is to be cast. Finally, `value.var` tells `dcast()` which variable in `df` is the variable containing values.

The parameter `f` is where most of the action happens. `f` is of the form `id ~ variable`, where `id` is the combination of variables that compose the id variables, and `variable` is the data frame variable that contains the variables each entry in `value` describes. `id` could be a combination of variables that identify an observation: for example, if in some imaginary data frame the variables `day` and `time` uniquely identify an observation, then our formula `f` may resemble `day + time ~ variable`. If we do not have id variables, then `dcast()` should not be used: use `unstack(df, value ~ variable)` instead.

It is possible that in casting you encounter rows that will be mapped in the casting process back to the same cell, and thus their observations must somehow be combined. The `fun.aggregate` parameter in `dcast()` will tell the `dcast()` function what function to use to aggregate those values. By default, `fun.aggregate = length`; the resulting matrix will simply count how many rows were mapped back to the same cell. But you may prefer some other aggregation means, such as `mean()` or `median()` or `sum()`.

Below, I demonstrate casting melted data into wide form.

```
# Fictitious long-form data
(df <- data.frame(idvar = rep(1:20, times = 3), variable = rep(c("height", "weight",
    "girth"), each = 20), value = c(rnorm(20, mean = 5.5, sd = 1), rnorm(20,
    mean = 180, sd = 20), rnorm(20, mean = 30, sd = 4))))
```

```
##    idvar variable      value
## 1      1   height   5.493245
## 2      2   height   6.374324
## 3      3   height   5.425054
## 4      4   height   4.935022
## 5      5   height   6.881408
## 6      6   height   4.244969
```

```
## 7      7   height   4.568275
## 8      8   height   7.656028
## 9      9   height   6.065840
## 10    10   height   5.657912
## 11    11   height   6.261456
## 12    12   height   7.296769
## 13    13   height   3.980199
## 14    14   height   4.589487
## 15    15   height   4.832898
## 16    16   height   6.638288
## 17    17   height   6.123158
## 18    18   height   4.174483
## 19    19   height   6.211407
## 20    20   height   2.449331
## 21     1   weight 179.569512
## 22     2   weight 212.917198
## 23     3   weight 159.510770
## 24     4   weight 189.677433
## 25     5   weight 167.449159
## 26     6   weight 199.500540
## 27     7   weight 186.641073
## 28     8   weight 164.593473
## 29     9   weight 217.218592
## 30    10   weight 157.206968
## 31    11   weight 149.422822
## 32    12   weight 185.327842
## 33    13   weight 233.427509
## 34    14   weight 164.449186
## 35    15   weight 186.748269
## 36    16   weight 188.519095
## 37    17   weight 186.161091
## 38    18   weight 214.162013
## 39    19   weight 201.450273
## 40    20   weight 171.129761
## 41     1    girth  27.672681
## 42     2    girth  28.851358
## 43     3    girth  33.859699
## 44     4    girth  29.380395
## 45     5    girth  30.217894
## 46     6    girth  28.217576
## 47     7    girth  34.391800
## 48     8    girth  28.457569
## 49     9    girth  34.913874
## 50    10    girth  29.417434
## 51    11    girth  28.208254
## 52    12    girth  35.534026
## 53    13    girth  32.266757
## 54    14    girth  29.167722
## 55    15    girth  23.502902
## 56    16    girth  25.229127
## 57    17    girth  26.029380
## 58    18    girth  30.973059
## 59    19    girth  31.162895
## 60    20    girth  33.581216
```

```
dcast(df, idvar ~ variable)
```

```
##    idvar    girth   height   weight
## 1      1 27.67268 5.493245 179.5695
## 2      2 28.85136 6.374324 212.9172
## 3      3 33.85970 5.425054 159.5108
## 4      4 29.38040 4.935022 189.6774
## 5      5 30.21789 6.881408 167.4492
## 6      6 28.21758 4.244969 199.5005
## 7      7 34.39180 4.568275 186.6411
## 8      8 28.45757 7.656028 164.5935
## 9      9 34.91387 6.065840 217.2186
## 10    10 29.41743 5.657912 157.2070
## 11    11 28.20825 6.261456 149.4228
## 12    12 35.53403 7.296769 185.3278
## 13    13 32.26676 3.980199 233.4275
## 14    14 29.16772 4.589487 164.4492
## 15    15 23.50290 4.832898 186.7483
## 16    16 25.22913 6.638288 188.5191
## 17    17 26.02938 6.123158 186.1611
## 18    18 30.97306 4.174483 214.1620
## 19    19 31.16289 6.211407 201.4503
## 20    20 33.58122 2.449331 171.1298
```

```
# Modifying this example so there are two id variables: recall %% is the mod
# operator
(df2 <- df %>% mutate(idvar2 = idvar%%2, idvar = idvar%%10))
```

```
##    idvar variable      value idvar2
## 1      1   height   5.493245      1
## 2      2   height   6.374324      0
## 3      3   height   5.425054      1
## 4      4   height   4.935022      0
## 5      5   height   6.881408      1
## 6      6   height   4.244969      0
## 7      7   height   4.568275      1
## 8      8   height   7.656028      0
## 9      9   height   6.065840      1
## 10     0   height   5.657912      0
## 11     1   height   6.261456      1
## 12     2   height   7.296769      0
## 13     3   height   3.980199      1
## 14     4   height   4.589487      0
## 15     5   height   4.832898      1
## 16     6   height   6.638288      0
## 17     7   height   6.123158      1
## 18     8   height   4.174483      0
## 19     9   height   6.211407      1
## 20     0   height   2.449331      0
## 21     1   weight 179.569512      1
## 22     2   weight 212.917198      0
## 23     3   weight 159.510770      1
## 24     4   weight 189.677433      0
## 25     5   weight 167.449159      1
```

```
## 26     6   weight 199.500540      0
## 27     7   weight 186.641073      1
## 28     8   weight 164.593473      0
## 29     9   weight 217.218592      1
## 30     0   weight 157.206968      0
## 31     1   weight 149.422822      1
## 32     2   weight 185.327842      0
## 33     3   weight 233.427509      1
## 34     4   weight 164.449186      0
## 35     5   weight 186.748269      1
## 36     6   weight 188.519095      0
## 37     7   weight 186.161091      1
## 38     8   weight 214.162013      0
## 39     9   weight 201.450273      1
## 40     0   weight 171.129761      0
## 41     1    girth  27.672681      1
## 42     2    girth  28.851358      0
## 43     3    girth  33.859699      1
## 44     4    girth  29.380395      0
## 45     5    girth  30.217894      1
## 46     6    girth  28.217576      0
## 47     7    girth  34.391800      1
## 48     8    girth  28.457569      0
## 49     9    girth  34.913874      1
## 50     0    girth  29.417434      0
## 51     1    girth  28.208254      1
## 52     2    girth  35.534026      0
## 53     3    girth  32.266757      1
## 54     4    girth  29.167722      0
## 55     5    girth  23.502902      1
## 56     6    girth  25.229127      0
## 57     7    girth  26.029380      1
## 58     8    girth  30.973059      0
## 59     9    girth  31.162895      1
## 60     0    girth  33.581216      0
```

```
# Use both id variables, and for collisions, take the mean of the values
dcast(df2, idvar + idvar2 ~ variable, fun.aggregate = mean)
```

```
##    idvar idvar2    girth   height   weight
## 1      0      0 31.49933 4.053622 164.1684
## 2      1      1 27.94047 5.877351 164.4962
## 3      2      0 32.19269 6.835547 199.1225
## 4      3      1 33.06323 4.702626 196.4691
## 5      4      0 29.27406 4.762255 177.0633
## 6      5      1 26.86040 5.857153 177.0987
## 7      6      0 26.72335 5.441628 194.0098
## 8      7      1 30.21059 5.345716 186.4011
## 9      8      0 29.71531 5.915255 189.3777
## 10     9      1 33.03838 6.138623 209.3344
```

You can learn more about the **reshape2** package here and here.

Lecture 11

Inferential Statistics via Computational Methods

The objective of **inferential statistics** is to describe population parameters using samples. In the lecture, we will study extensively inferential statistics when constructed from a probabilistic model, and in the lab we will see how to use R to perform analysis based on these methods. Prior to that, though, we explore techniques based on computational methods such as simulation and bootstrapping. Not only do computational methods provide a good first step for thinking about later methods, they are also very useful when one does not have a probability model available for inference, perhaps because such a model would be very difficult to construct.

Simulation

Let X_i be a Normal random variable with mean 100 and standard deviation 16, so $X_i \sim N(100, 16^2)$. We know that if X_i are random variables, the sample mean $\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ is also a random variable. What distribution does $\overline{X}$ follow? While we can compute this distribution exactly, here we show how it could be simulated.

We could simulate a single sample mean for a sample of size 10 with the following code:

```
set.seed(6222016)
rand_dat <- rnorm(10, mean = 100, sd = 16)
mean(rand_dat)
```

```
## [1] 101.0052
```

Obviously 101.00519 is not 100, which we know to be the true mean of the data, nor should we expect that to ever happen. But is it "close" to the true mean? It's hard to say; we would need to know how much variability is possible.

We would like to generate a number of sample means for data coming from the distirbution in question. An initial approach may involve a `for` loop, which is not the best approach when using R. Another approach may involve `sapply()`, like so:

```
sim_means <- sapply(1:100, function(throwaway) mean(rnorm(10, mean = 100, sd = 16)))
```

While better than a `for` loop, this solution uses `sapply()` in a way it was not designed for. We have to create a variable with a parameter that is unused by the function, and also pass a vector with a length corresponding to the number of simulated values we want, when in truth it's not the vector we want but the length of the vector. A better solution would be to use the `replicate()` function. The call `replicate(n, expr)` will repeat the expression `expr` `n` times, and return a vector containing the results. I show an example for simulated means below:

```
sim_means <- replicate(100, mean(rnorm(10, mean = 100, sd = 16)))
sim_means
```

```
##   [1] 108.35565 100.66750 108.19234  95.89265 101.20817 105.76444 104.89860
##   [8] 104.22216 102.06229  94.31038  99.55700  94.44461 102.06794  89.95609
##  [15] 101.17615 100.63431  95.18799 104.70427 107.69503 104.04101 107.86346
##  [22] 103.06723  97.43702 102.05654  96.08053 101.36263 103.91291  96.41028
##  [29]  96.81695 102.50987 104.38500  99.17687  97.74994  92.30596  99.26339
##  [36] 101.67710 102.39232 105.79338 106.72293  99.22185  95.99618  98.91788
##  [43] 101.01230  96.25258  91.20321 104.81500  96.05610 106.60535  93.97403
##  [50] 103.89452 107.67794 105.31947  99.81982  99.84518  98.65894  93.04366
##  [57]  99.36829  97.92127 105.95773  93.25400  97.91482 100.05437  93.78229
##  [64] 105.21112  97.93044 100.19164  95.70649  97.18679  93.44727 101.54060
##  [71]  98.67888 103.98927  95.20378 101.56096 108.71335 105.40760 105.08654
##  [78] 102.58549  95.86958  97.53956  92.42496 103.78224 102.66197  93.44900
##  [85]  98.92777  97.65882  96.09172 108.16021 100.97451  97.53069  98.50505
##  [92] 102.83923  91.39683  98.91011 102.07212  99.36859 100.66825 102.34920
##  [99]  99.55977 105.28871
```

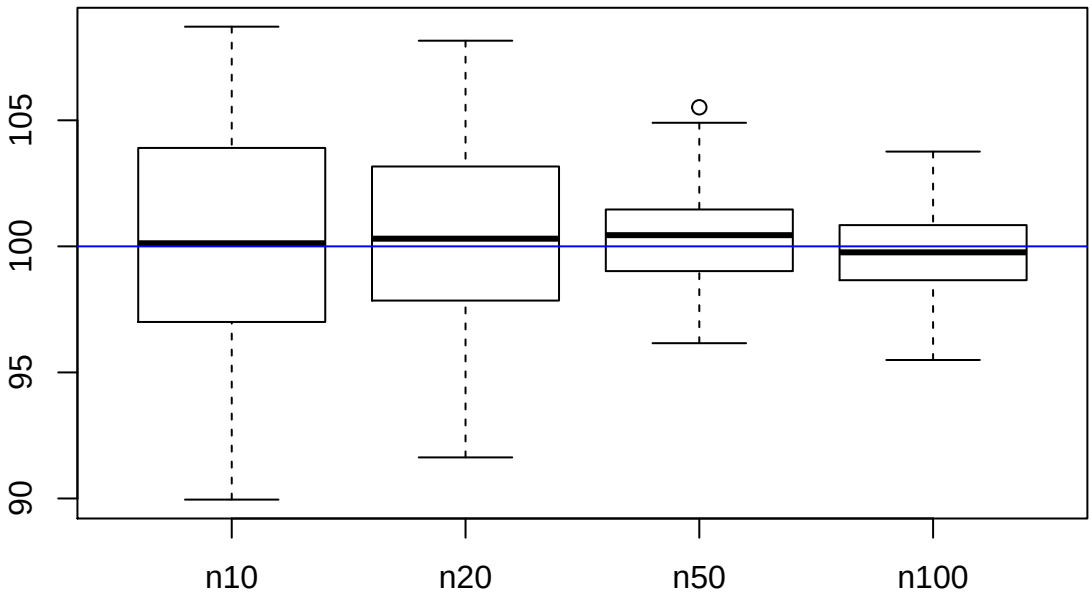
Looking at the simulated means, we see a lot of variability, with 53% of the means being more than 3 away from the true mean. The only way to reduce this variability would be to increase the sample size.

```
sim_means2 <- replicate(100, mean(rnorm(20, mean = 100, sd = 16)))
sim_means2
```

```
##   [1]  99.69189  95.21360 105.52372 104.45792  93.52940  91.62906 105.26088
##   [8] 100.36841 105.73402 102.46795  99.09634  98.09586 103.41291 102.24939
##  [15]  95.19029  97.46231 101.84167 106.39167  99.76708  96.41046  94.57369
##  [22] 100.30044 107.70845 105.03082 103.03096  96.89259 101.63428 106.77924
##  [29]  93.78310  99.20373  99.21529  96.04547 100.02290 100.23826  98.78401
##  [36]  96.60201  97.90106  95.12226  98.53885 101.81787 107.67086 102.87544
##  [43] 102.54591 100.30953 104.98270  99.21484  96.38804 100.71951 100.54088
##  [50]  96.36608 102.28295 103.48824 104.53096 100.48312  97.77604 100.40682
##  [57]  99.59780  95.34798  97.79904  94.79671 103.44435 103.94647 101.33339
##  [64]  98.72290  98.94278 102.88865  97.95571  98.96574 101.67445  98.99390
##  [71] 101.10229 100.38276 100.99239 102.65524 106.64794  95.42338 103.30498
##  [78] 104.93425 108.15721  99.01630  96.63749  96.91546 101.56492 104.53852
##  [85]  99.35392  97.48784  98.20285  92.61517 106.96066 100.54894 100.45520
##  [92]  99.29380  99.20205 104.19171 103.59921  97.42151  95.17621 106.17301
##  [99]  98.95303 106.53663
```

Now, only 46% of the simulated means are more than 5 away from the true mean. If we repeat this for ever increasing sample sizes, we can see the distribution of the sample means concentrating around the true mean.

```
sim_means3 <- replicate(100, mean(rnorm(50, mean = 100, sd = 16)))
sim_means4 <- replicate(100, mean(rnorm(100, mean = 100, sd = 16)))
boxplot(list("n10" = sim_means, "n20" = sim_means2, "n50" = sim_means3, "n100" = sim_means4))
abline(h = 100, col = "blue")
```



As can be seen in the chart, larger sample sizes have smaller variability around the true mean. This is what we want; we want estimators for the mean to be both accurate (they center around the correct result, not elsewhere) and precise (they are consistently near the correct answer). The term for the first property is **unbiasedness**, where $\mathbb{E}[X] = \mu$, and the term for the second property is **minimum variance**.

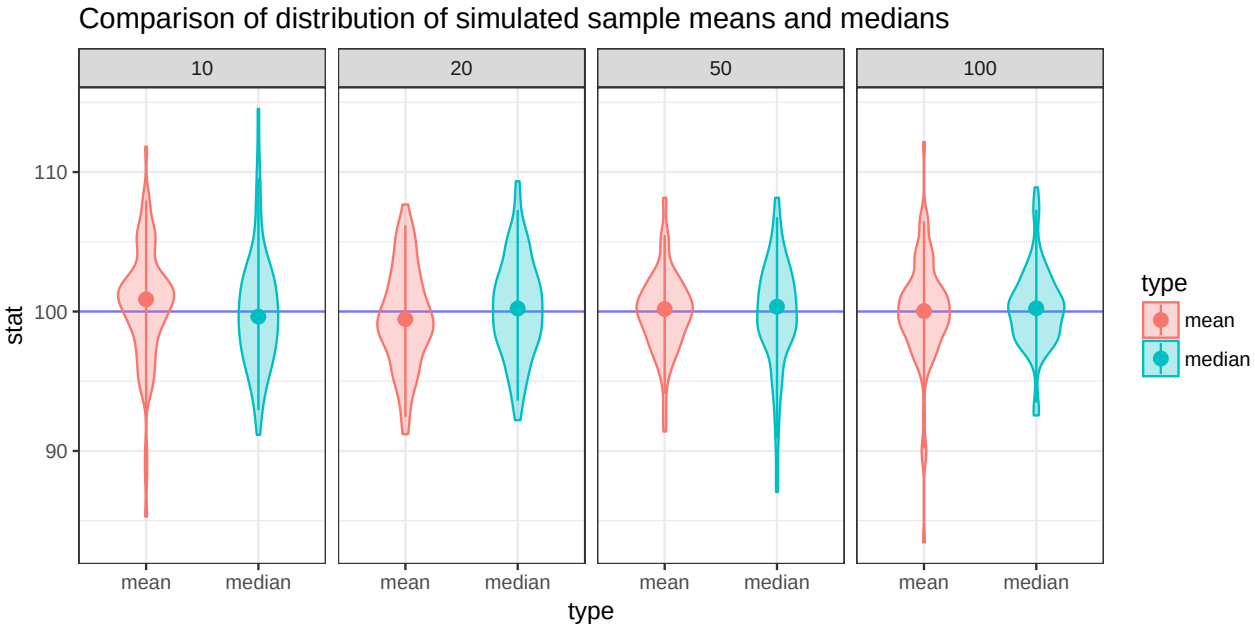
Recall that for the Normal distribution, the mean is also the median. This suggests the sample median as an alternative to the sample mean for estimating the same parameter. How do the two compare? Let's simulate them and find out!

```
library(ggplot2)

# Simulate sample medians
sim_medians <- replicate(100, median(rnorm(10, mean = 100, sd = 16)))
sim_medians2 <- replicate(100, median(rnorm(20, mean = 100, sd = 16)))
sim_medians3 <- replicate(100, median(rnorm(50, mean = 100, sd = 16)))
sim_medians4 <- replicate(100, median(rnorm(100, mean = 100, sd = 16)))

# Make a data frame to contain the data for the sake of easier plotting
dat <- data.frame(stat = c(sim_means, sim_medians, sim_means2, sim_medians2, sim_means3, sim_medians3, s
                  type = rep(c("mean", "median"), times = 4, each = 20),
                  n = as.factor(rep(c(10, 20, 50, 100), each = 2 * 20)))
head(dat)
```

```
##        stat type  n
## 1 108.35565 mean 10
## 2 100.66750 mean 10
## 3 108.19234 mean 10
## 4  95.89265 mean 10
## 5 101.20817 mean 10
## 6 105.76444 mean 10
```

```
# Using ggplot2 to make the graphic comparing distributions
ggplot(dat, aes(y = stat, x = type, color = type, fill = type)) +
  geom_hline(yintercept = 100, color = "blue", alpha = .5) + # Horizontal line for true center
  geom_violin(width = .5, alpha = .3) + # Violin plot
  stat_summary(fun.data = "median_hilow") + # A statistical summary, showing median and 5th/95th percen
  facet_grid(. ~ n) + # Split based on sample size
```

```
  theme_bw() +  # Sometimes I don't like the grey theme
  ggtitle("Comparison of distribution of simulated sample means and medians")
```



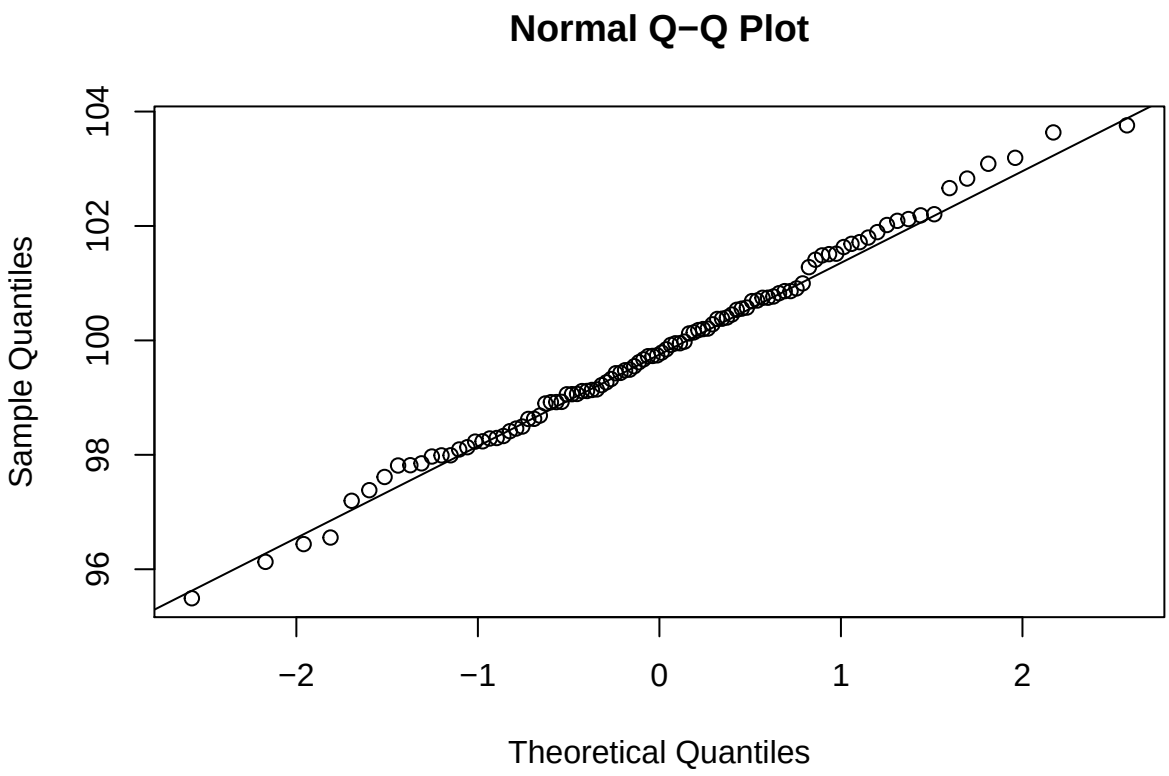
From the above graphic, one can see that the sample median has more variance than the sample median, and is thus not minimum variance. The sample mean is a more reliable way to estimate unknown μ than the sample median.

These analyses suggest that when trying to estimate the value of a parameter, we should follow these principals:

1. We should use unbiased estimators. If this is not possible, we should use estimators that are at least **consistent** (that is, the bias approaches 0 as the sample size grows large).
2. We should use estimators that vary as little as possible. This in turn implies that we should use as large a sample size as possible when estimating unknown parameters.

Clearly, $\overline{X}$ is a random variable. With that said, what is its distribution? We could explore the possibility that $\overline{X}$ is Normally distributed by looking at a **Q-Q plot**, which compares sample quantiles to theoretical quantiles if a random variable were to follow some particular distribution. If we wanted to see if $\overline{X}$ were Normally distributed, we could use the `qnorm()` function to create a Q-Q plot comparing the distribution of $\overline{X}$ to the Normal distribution. The call `qqnorm(x)` create a Q-Q plot comparing the distribution of `x` to the Normal distribution. I next create such a plot.

```
qqnorm(sim_means4)
# A line, for comparison
qqline(sim_means4)
```



If the points in the plot fit closely to a straight line, then the distributions are similar. In this case, it seems that the Normal distribution fits the data well (as it should).

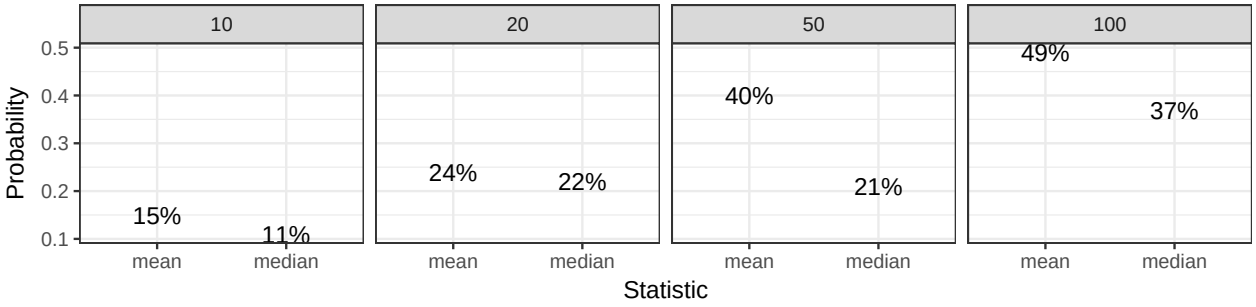
Simulation is frequently used for estimating probabilities that are otherwise difficult to compute by hand. The idea is to simulate the phenomena under question and how often the event in question happened in the simulated data. If done correctly, the sample proportion should approach the true probability as more simulations are done. Here, I demonstrate by estimating for each of the sample sizes investigated, the probability of being "close" to the true mean, both for the sample mean and the sample median (these probabilities are no necessarily difficult to compute by hand, and you can investigate for yourself using the results of the Central Limit Theorem whether the estimated probabilities are close to the true probabilities).

```
library(reshape)
dat_list <- list("mean" = list(
  # Compute probability of being "close" for sample means
  "10" = mean(abs(sim_means - 100) < 1), "20" = mean(abs(sim_means2 - 100) < 1), "50" = mean(abs(sim_mea
  # Probabilities of being "close" for sample medians
  "median" = list("10" = mean(abs(sim_medians - 100) < 1), "20" = mean(abs(sim_medians2 - 100) < 1), "5
# Convert this list into a workable data frame. Here I use the melt function in reshape, which will cre
nice_df <- melt(dat_list)
# Data cleanup
names(nice_df) <- c("Probability", "Size", "Statistic")
nice_df$Size <- factor(nice_df$Size, levels = c("10", "20", "50", "100"))
# The actual probabilities
nice_df
```

```
##   Probability Size Statistic
## 1        0.15   10      mean
## 2        0.24   20      mean
## 3        0.40   50      mean
## 4        0.49  100      mean
## 5        0.11   10    median
```

```
## 6        0.22   20    median
## 7        0.21   50    median
## 8        0.37  100    median
```

```
# A plot of the probabilities
ggplot(nice_df, aes(y = Probability, x = Statistic)) +
  # Instead of plotting points, I simply plot what the probability is, as a point.
  geom_text(aes(label = paste0(Probability * 100, "%"))) +
  facet_grid(. ~ Size) +
  theme_bw()
```



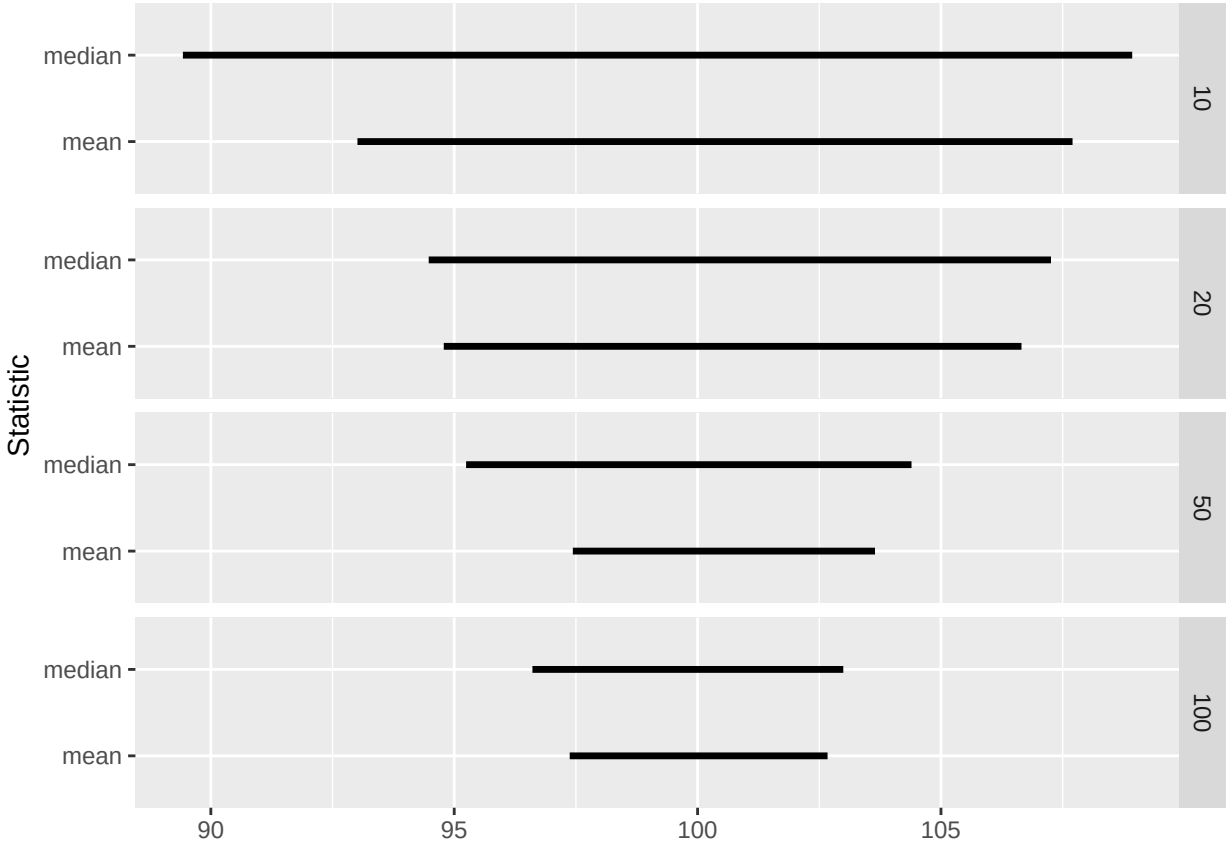
The above information suggests that the sample mean tends to be closer than the sample median to the true mean. This can be proven mathematically, but the simulation approach is much easier, although not nearly as precise.

We could also use sample quantiles to estimate true quantiles for a statistic, and thus get some sense as to where the true parameter may lie. I demonstrate in the code block below.

```
quant_list = list("mean" = list("10" = as.list(quantile(sim_means, c(.05, .95))), "20" = as.list(quanti
quant_df <- cast(melt(quant_list), L1 + L2 ~ L3)
names(quant_df) <- c("Statistic", "Size", "Lower", "Upper")
quant_df$Size <- factor(quant_df$Size, levels = c("10", "20", "50", "100"))

quant_df
```

```
##   Statistic Size    Lower    Upper
## 1      mean   10 93.01272 107.7034
## 2      mean  100 97.37297 102.6705
## 3      mean   20 94.78556 106.6545
## 4      mean   50 97.43711 103.6452
## 5    median   10 89.42588 108.9288
## 6    median  100 96.60708 102.9936
## 7    median   20 94.47622 107.2607
## 8    median   50 95.24452 104.3953
```

```
ggplot(quant_df) +
  geom_segment(aes(x = Lower, xend = Upper, y = Statistic, yend = Statistic), size = 1.2) +
  facet_grid(Size ~ .) +
  xlab(NULL)
```



We can see that in all cases the true mean lies between the 5^{th} and 95^{th} quantiles, and that the sample mean has a narrower range than the sample median, thus giving a more precise description as to where the true mean lies.

Significance Testing

Statistical inference's *raison d'etre* is detecting signal from noise. More precisely, **statistical inference** is used to determine if there is enough evidence to detect an effect or determine the location of a parameter in the presence of "noise", or random effects.

Suppose a researcher is trying to determine if a new drug under study lowers blood pressure. The researcher randomly assigns study participants to control and treatment groups. He stores his results in R vectors like so:

```
control <- c(124, 155, 120, 116)
treatment <- c(120, 108, 132, 112)
```

Is there a difference beteen control and treatment? Let's check.

```
mean(treatment) - mean(control)
```

```
## [1] -10.75
```

A difference of -10.75 looks promising, but there is lots of variation in the data. Is a difference of -10.75 large enough to conclude there is a difference?

Suppose not. Let's assume that the treatment had no effect, and the observed difference is due only to the random assignment to control and treatment groups. If that were the case, other random assignment schemes would have differences just as or even more extreme than the one observed.

Let's investigate this possibility. We will use the `combn()` function to find all possible combinations of assigning individuals to the treatment group, with the rest going to the control group. `combn(vec, n)` will find all ways to choose `n` elements from `vec`, storing the result in a matrix with each column representing one possible combination. We will assume that we will be pooling the control and treatment groups into a single vector with 8 elements.

```
# Find all ways to choose the four indices that will represent a new random
# assignment to the treatment group. This is all we need to know to
# determine who is assigned to both control and treatment.
assn <- combn(1:(length(control) + length(treatment)), 4)
# Look at the result
assn[, 1:4]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
## [2,]    2    2    2    2
## [3,]    3    3    3    3
## [4,]    4    5    6    7
```

```
# How many combinations are possible?
ncol(assn)
```

```
## [1] 70
```

Now, let's actually investigate the difference and determine if it is "large".

```
# Lump all data into a single vector
blood_pressure <- c(control, treatment)
blood_pressure
```

```
## [1] 124 155 120 116 120 108 132 112
```

```
# To demonstrate the idea of this analysis, let's consider just one new
# assignment of control and treatment, using the 4th column of assn. The new
# assignment is in ind.
ind <- assn[, 4]
# If blood_pressure[ind] is the treatment group, blood_pressure[-ind] is the
# control group.
blood_pressure[ind]
```

```
## [1] 124 155 120 132
```

```
blood_pressure[-ind]
```

```
## [1] 116 120 108 112
```

```
# What is the new difference?
mean(blood_pressure[ind]) - mean(blood_pressure[-ind])
```

```
## [1] 18.75
```

```
# Now, let's do this for all possible combinations, using apply
diffs <- apply(assn, 2, function(ind) {
    mean(blood_pressure[ind]) - mean(blood_pressure[-ind])
})
```

Now we can decide how "unusual" our initial difference between treatment and control of -10.75 is. Since we are trying to determine if the treatment reduces blood pressure, we decide that if this particular assignment is unusually negative, there would be evidence that the treatment works. So we see how many assignments have means more negative than the one seen.

```
sum(diffs < mean(treatment) - mean(control))
```

```
## [1] 11
```

```
mean(diffs < mean(treatment) - mean(control))
```

```
## [1] 0.1571429
```

It seems that 0.16% of assigments to control and treatment result in differences more "extreme" than the one observed. This is not convincing evidence that the difference we saw reflects any significant effect from the new drug; there is too much noise in the data to reach such a conclusion.

Let's consider another problem. In the `ToothGrowth` data set, we can see the results of an experiment where different supplements were used to examine the tooth growth of guinea pigs. Is there a difference?

Let's first see what the difference is.

```
# Find the means
supp_means <- aggregate(len ~ supp, data = ToothGrowth, mean)
supp_means
```

```
##   supp      len
## 1   OJ 20.66333
## 2   VC 16.96333
```

```
# The difference in means (VC - OJ)
diff(supp_means$len)
```

```
## [1] -3.7
```

There is a difference; it appears that the supplement `VC` (vitamin C in absorbic acid) has smaller tooth growth than the supplement `OJ` (orange juice). But is this evidence *significant*?

In this case, our earlier trick will not work. There are 60 observations in `ToothGrowth`, 30 of which are for the group `OJ`. This means that there are $\binom{60}{30}$ = `choose(60, 30)` $= 118,264,581,564,861,152$ possible random assignments to the two groups. R will surely choke on that much computation, so we must use a new approach.

Rather than examine *all* possible permutations, let's randomly select permutations and see how often in our random sample we get results more extreme than what was observed. The principle is the same as what was done before, but it's probabilistic rather than exhaustive.

```
# Let's randomly assign to VC supplement just once for proof of concept;
# each sample has 30 observations, so we will randomly select 30 to be the
# new VC group
ind <- sample(1:60, size = 30)
with(ToothGrowth, mean(len[ind]) - mean(len[-ind]))
```

```
## [1] 0.7066667
```

```
# We will now do this many times
sim_diffs <- replicate(2000, {
    ind <- sample(1:60, size = 30)
    with(ToothGrowth, mean(len[ind]) - mean(len[-ind]))
})
# Proportion with bigger difference
mean(sim_diffs < diff(supp_means$len))
```

```
## [1] 0.0325
```

3.25% of means are "more extreme" than what was observed, which seems convincing evidence that `VC` is, on average, less than `OJ`.

Lecture 12

Bootstrapping

When looking at polls in the news, you may notice a margin of error attached to the numbers in the poll. The **margin of error** quantifies the uncertainty we attach to a statistic estimated from data, and the **confidence interval**, found by adding and subtracting the margin of error from the statistic, represents the range of values in which the true value of the parameter being estimated could plausibly lie. These are computed for many statistics we estimate.

We cover in the lecture how confidence intervals are computed using probabilistic methods. Here, we will use a computational technique called **bootstrapping** for computing these intervals. Even though the statistics we discuss in this course could have confidence intervals computed exactly, this is not always the case. We may not always have a formula for the margin of error in a closed form, either due to it simply not being derived yet or because it's intractable. Additionally, bootstrapping may be preferred even if a formula for a margin error exists because bootstrapping may be a more robust means of computing this margin of error when compared to a procedure very sensitive to the assumptions under which it was derived.

Last lecture, we examined techniques for obtaining, computationally, a confidence interval for the location of the true mean of a Normal distribution. Unfortunately, doing so required knowing what the true mean was, which clearly is never the case (otherwise there would be no reason for the investigation). Bootstrapping does what we did earlier, but instead of using the Normal distribution to estimate the standard error, it estimates the standard error by drawing from the distribution of the data set under investigation (the **empirical distribution**). We re-estimate the statistic in the simulated data sets to obtain a distribution of the statistic under question, and use this distribution to estimate the margin of error we should attach to the statistic.

Suppose `x` is a vector containing the data set under investigation. We can sample from the empirical distribution of `x` via `sample(x, size = length(x), replace = TRUE)` (I specify `size = length(x)` to draw simulated data sets that are the same size as `x`, which is what should be done when bootstrapping to obtain confidence intervals, but in principle one can sample from the empirical distribution simulated data sets of any size). We can then compute the statistic of interest from the simulated data sets, and use `quantile()` to obtain a confidence interval.

Let's demonstrate by estimating the mean determinations of copper in wholemeal flour, in parts per million, contained in the `chem` data set (**MASS**).

```
library(MASS)
chem
```

```
##  [1]  2.90  3.10  3.40  3.40  3.70  3.70  2.80  2.50  2.40  2.40  2.70
## [12]  2.20  5.28  3.37  3.03  3.03 28.95  3.77  3.40  2.20  3.50  3.60
## [23]  3.70  3.70
```

```
# First, the sample mean of chem
mean(chem)
```

```
## [1] 4.280417
```

```
# To demonstrate how we sample from the empirical distribution of chem, I
# simulate once, and also compute the mean of the simulated data set
chem_sim <- sample(chem, size = length(chem), replace = TRUE)
chem_sim
```

```
##  [1]  3.37  3.70 28.95  2.40  2.50  3.70  3.03  5.28  3.40  3.40  3.77
## [12]  3.40  3.40 28.95  2.90  2.80  3.60  2.50 28.95  3.70  3.40  3.77
## [23]  2.90  2.50
```

```
mean(chem_sim)
```

```
## [1] 6.51125
```

```
# Now let's obtain a standard error by simulating 2000 means from the
# empirical distribution
mean_boot <- replicate(2000, {
    mean(sample(chem, size = length(chem), replace = TRUE))
})
# The 95% confidence interval
quantile(mean_boot, c(0.025, 0.975))
```

```
##     2.5%    97.5%
## 3.028719 6.518438
```

Bayesian Statistics

The statistical techniques you have seen so far, and the approach taken to statistics in the lecture portion of the course and after this lecture in the lab as well, all are considered to be **frequentist statistics**. In frequentist statistics, we consider nature as having a fixed yet unknown state. If we are trying to learn about a parameter θ, we consider θ to be unknown but fixed, and we try to make inference about the value of the fixed number θ using data. In a hypothesis test, we assume that either the null hypothesis or the alternative hypothesis are true, and we use data to determine which hypothesis is more likely to be true given the data we observe. If the data we observe is highly unlikely under the assumption that the null hypothesis is true, then we reject the null hypothesis in favor of the alternative.

This school of thought in statistics arose in the late 19th and early 20th centuries, with mathematicians such as Fisher and Student (William Gosset) developing frequentist methods. They were very popular throughout the 20th century and remain the dominant statistical school of thought and the methods taught in introductory statistics courses.

Frequentist statistics, though, is not the only school of thought in statistics. **Bayesian statistics** is arguably older, with Bayes and Laplace being seen as the earliest individuals to develop the methods. Unlike Frequentist statistics, where we try to learn about a state of nature we presume to be fixed, the true state of nature under Bayesian statistics is seen as being random. The parameter θ we try to learn about is not a fixed number but a random number with a probability distribution of its own. Before collecting data, we assign to θ a *prior* distribution, which reflects our belief about the possible values of θ prior to analysis. We collect data so we may compute a *posterior* distribution, which reflects where we believe θ to be after having observed data. Likewise, with hypothesis testing, rather than assume one of two hypotheses to be true, or rejecting one hypothesis in favor of another, we assign probabilities to the hypotheses desribing how likely we believe one or the other to be true, then after collecting data, we update our beliefs in which hypothesis is true.

As you can imagine, Bayesian methods are highly reliant on **Bayes' Theorem**:

$$P(H|D) = \frac{P(D|H)P(H)}{P(D|H)P(H) + P(D|H^C)P(H^C)} = \frac{P(D|H)P(H)}{P(D)} \propto P(D|H)P(H)$$

(The symbol $\propto$ means "proportional to", meaning that the terms on the right is equal to the term on the left up to a multiplicative constant, or $A \propto B \iff A = cB$ for some constant $c \neq 0$.) If H represents the event the hypothesis is true, and D represents the event of observing a particular data set, then the line $P(H|D) \propto P(D|H)P(H)$ effectively sums up the Bayesian mindset: the probability the hypothesis H is true given the observed data set D is proportional to the likelihood of observing our data D if H were actually true, multiplied with the probaility that H is actually true (we then divide by a constant that ensures that $P(H|D) + P(H^C|D) = 1$, but the normalizing constant is uninteresting). Thus, using the *prior* belief (distribution) that H is true (that is, $P(H)$), after collecting data, we compute a *posterior* belief (distribution) that H is true, $P(H|D)$.

Bayesian methods used to not be as common as frequentist methods, and the frequentist approach still dominates. But they are becoming more common, especially in fields such as computer science. (And if you follow politics and Nate Silver at FiveThirtyEight and use his forecasts for elections, you are using the results of Bayesian statistics; his probabilistic forecasts are Bayesian in nature.) There are many advantages and challenges either approach has, and reasons to use one approach or the other range from philosophical to computational. We will not go into the debate.

Let's consider a simple example. Suppose we are considering a Bernoulli experiment; perhaps we are flipping a coin and we wish to know what the proportion of flips land heads-up. Thus, $X_i \sim Ber(p)$, where p is the probability the coin lands heads-up.

In the Bayesian framework, p is a random variable. We assign a prior distribution to p; perhaps we will say that $P(p = .05) = P(p = .1) = P(p = .15) = ... = P(p = .95) = \frac{1}{19}$. We collect data from our coin by performing n flips, and counting the number of times the coin lands heads-up. Then the likelihood for seeing s flips, given the population proportion of heads p is p_0, is $P(s \text{ heads out of } n \text{ flips}|p = p_0) \propto {p_0}^s(1 - p_0)^{n-s}$. Combining these together and Bayes' formula, we now have a formula for the posterior distribution:

$$\begin{aligned} P(p = p_0|s \text{ heads out of } n \text{ flips}) &\propto P(s \text{ heads out of } n \text{ flips}|p = p_0)P(p = p_0) \\ &\propto {p_0}^s(1 - p_0)^{n-s}P(p = p_0) \\ &= \frac{1}{19}{p_0}^s(1 - p_0)^{n-s} \end{aligned}$$

Bear in mind, these are not proper probabilities; as it stands, $c = \sum_{p_0 \in \{.05,...,.95\}} \frac{1}{19}{p_0}^s(1 - p_0)^{n-s} \neq 1$. That said, after computing the posterior distribution, it's not too difficult to find this c (in this problem; in general, it's very difficult), and then $P(p = p_0|s \text{ heads out of } n \text{ flips}) = \frac{1}{c}\frac{1}{19}{p_0}^s(1 - p_0)^{n-s}$.

Let's suppose we perform $n = 20$ flips and obtain $s = 12$ heads. Let's see what the posterior distribution is with this data, using R.

```
# I will be plotting a lot of pmf's in this document, so I create a function
# to help save effort.  The first argument, q, represents the quantiles of
# the random variable (the values that are possible). The second argument
# represents the value of the pmf at each q (and thus should be of the same
# length); in other words, for each q, p is the probability of seeing q
plot_pmf <- function(q, p, main = "pmf") {
    # This will plot a series of horizontal lines at q with height p, setting
    # the y limits to a reasonable heights
    plot(q, p, type = "h", xlab = "x", ylab = "probability", main = main, ylim = c(0,
        max(p) + 0.1))
    # Usually these plots have a dot at the end of the line; the point function
    # will add these dots to the plot created above
    points(q, p, pch = 16, cex = 1.5)
}

# Change plot settings
old_par <- par()
```

```r
par(mfrow = c(2, 1))

# Possible probabilities
(p <- seq(0.05, 0.95, by = 0.05))
```

```
##  [1] 0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40 0.45 0.50 0.55 0.60 0.65 0.70
## [15] 0.75 0.80 0.85 0.90 0.95
```

```r
# The prior distribution of p
(prior <- rep(1/length(p), times = length(p)))
```

```
##  [1] 0.05263158 0.05263158 0.05263158 0.05263158 0.05263158 0.05263158
##  [7] 0.05263158 0.05263158 0.05263158 0.05263158 0.05263158 0.05263158
## [13] 0.05263158 0.05263158 0.05263158 0.05263158 0.05263158 0.05263158
## [19] 0.05263158
```

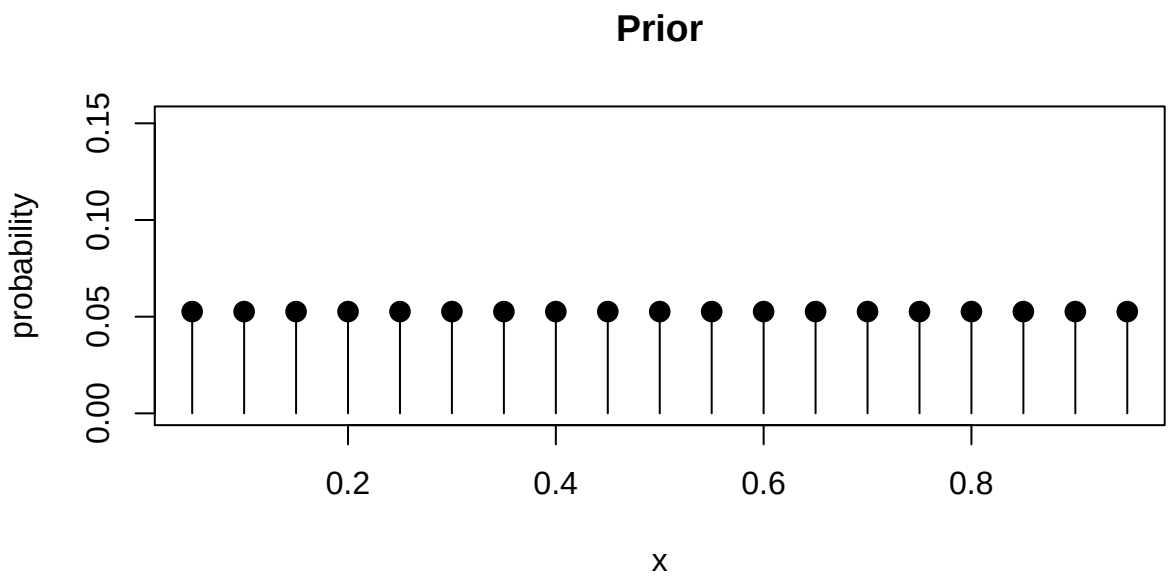
```r
# Plot the prior
plot_pmf(p, prior, main = "Prior")

# Compute the likelihood function
n <- 20  # Number of flips
s <- 12  # Number of heads
(like <- p^s * (1 - p)^(n - s))
```

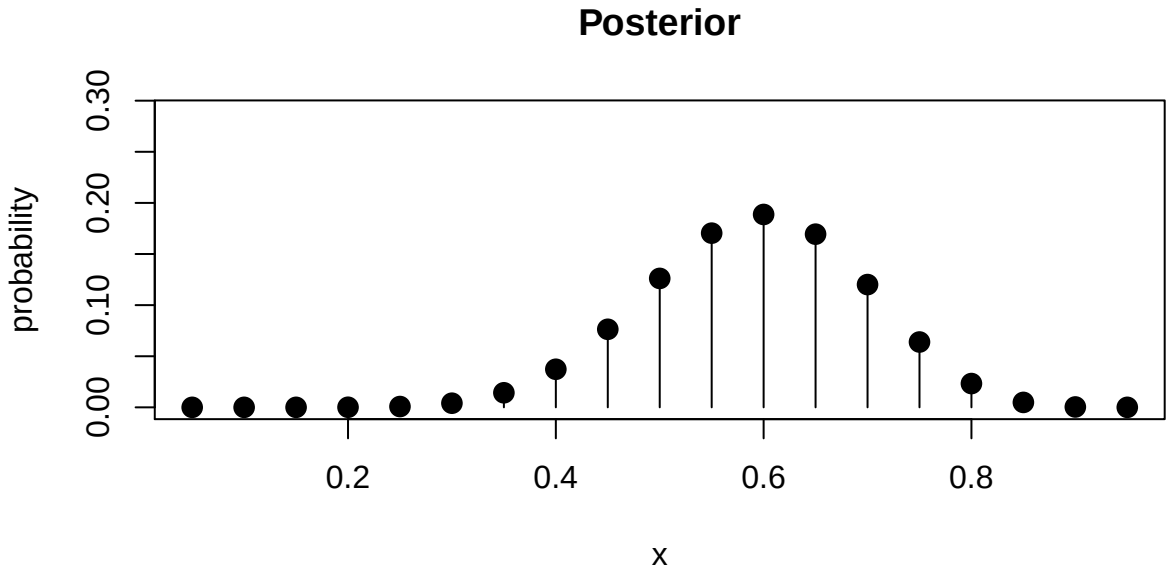
```
##  [1] 1.619679e-16 4.304672e-13 3.535465e-11 6.871948e-10 5.967195e-09
##  [6] 3.063652e-08 1.076771e-07 2.817928e-07 5.773666e-07 9.536743e-07
## [11] 1.288405e-06 1.426576e-06 1.280869e-06 9.081269e-07 4.833428e-07
## [16] 1.759219e-07 3.645501e-08 2.824295e-09 2.110782e-11
```

```r
# With the likelihood and prior computed, we now obtain the posterior
# distribution.
post <- like * prior
(post <- post/sum(post))  # Normalize; make proper probabilities
```

```
##  [1] 2.142325e-11 5.693725e-08 4.676306e-06 9.089422e-05 7.892719e-04
##  [6] 4.052246e-03 1.424229e-02 3.727231e-02 7.636742e-02 1.261411e-01
## [11] 1.704154e-01 1.886911e-01 1.694186e-01 1.201166e-01 6.393102e-02
## [16] 2.326892e-02 4.821849e-03 3.735653e-04 2.791899e-06
```

```r
plot_pmf(p, post, main = "Posterior")
```





Notice the shape of the posterior. It centers around $p = .6$, which was the sample proportion of successes. Nevertheless, other values of p could occur with high probability. We would call $p = .6$ our *maximum a-posteriori (MAP) estimate* of p; it is the most likely value for p.

```
# Getting the MAP estimator for p
p[which.max(post)]
```

```
## [1] 0.6
```

Using the posterior distribution, we can make a number of statements about the value of p. For example, if we wanted to determine whether the coin is biased in favor of heads, we compute $P(p \geq .5)$ and compare it to $P(p < .5)$, like so:

```
# Is the coin biased?
sum(post[which(p >= .5)])   # Probability p >= .5
```

```
## [1] 0.8671808
```

```
sum(post[which(p < .5)])   # Probability p < .5
```

```
## [1] 0.1328192
```

With $P(p \geq .5) = 0.8671808$, the coin is likely to be biased in favor of heads, though there is still a strong chance this may be false.

We can compute a **credible interval**, which is the Bayesian equivalent of a confidence interval. We will say that a $C \times 100\%$ **credible interval** $[a; b]$ **for** p is an interval such that $P(a \leq p \leq b) = C$. I compute a (approximately) 95% credible interval for p as follows:

```
# Get the cdf of the posterior distribution using cumsum()
(post_cdf <- cumsum(post))
```

```
##   [1] 2.142325e-11 5.695867e-08 4.733265e-06 9.562748e-05 8.848994e-04
##   [6] 4.937145e-03 1.917943e-02 5.645175e-02 1.328192e-01 2.589602e-01
## [11] 4.293756e-01 6.180667e-01 7.874853e-01 9.076019e-01 9.715329e-01
## [16] 9.948018e-01 9.996236e-01 9.999972e-01 1.000000e+00
```

```
# If we want the credible interval to be centered around p, we want <2.5% of
# the distribution in the lower tail, and <2.5% in the upper tail.
(a <- p[max(which(post_cdf < 0.025))])
```

```
## [1] 0.35
```

```
(b <- p[min(which(post_cdf > 0.975))])
```

```
## [1] 0.8
```

```
# What is the actual probability that a <= p <= b
sum(post[which(p >= a & p <= b)])
```

```
## [1] 0.9898646
```

The 98.99% credible interval (the closest we can get to a 95% credible interval) is [0.35, 0.8]. You *can* interpret this interval this interval as being the interval where there is a 98.99% chance that p lies between 0.35 and 0.8. (If this were a confidence interval, this interpretation would be *wrong!*)

Lecture 13

Confidence Interval Basics

The sample mean, $\bar{x}$, is usually not considered enough to know where the true mean of a population is. It is seen as one instantiation of the random variable $\overline{X}$, and since we interpret it as being the result of a random process, we would like to describe the uncertainty we associate with its position relative to the population mean. This is true not only of the sample mean but any statistic we estimate from a random sample.

The last lecture was concerned with computational methods for performing statistical inference. In this lecture, we consider procedures for inferential statistics derived from probability models. In particular, I focus on **parametric methods**, which aim to uncover information regarding the location of a parameter of a probability model assumed to describe the data, such as the mean μ in the Normal distribution, or the population proportion p. (The methods explored last lecture, such as bootstrapping, are **non-parametric**; they do not assume a probability model describing the underlying distribution of the data. MATH 3080 covers other non-parametric methods.)

Why use probabilistic methods when we could use computational methods for many procedures? First, computational power and complexity may limit the effectiveness of computational methods; large or complex data sets may be time consuming to process using computational methods alone. Second, methods relying on simulation, such as bootstrapping, are imprecise. Many times this is fine, but the price of imprecision is large when handling small-probability events and other contexts where precision is called for. Third, while computational methods are **robust** (deviation from underlying assumptions has little effect on the end result), this robustness comes at the price of **power** (the ability to detect effects, even very small ones), and this may not be acceptable to a practitioner (much of statistical methodology can be seen as a battle between power and robustness, which are often mutually exclusive properties). Furthermore, many computational methods can be seen as a means to approximate some probabilistic method that is for some reason inaccessible or not quite optimal.

A $100C\%$ **confidence interval** is an interval or region constructed in such a way that the probability the *procedure* used to construct the interval results in an interval containing the true population parameter of interest is C. (Note that this is *not* the same as the probability that the parameter of interest is in the confidence interval, which does not even make sense in this context since the confidence interval computed from the sample is fixed and the parameter of interest is also considered fixed, albeit unknown; this is a common misinterpretation of the meaning of a confidence interval and I do not intend to spread it.) We call C the **confidence level** of the confidence interval. Smaller C result in intervals that capture the true parameter less frequently, while larger C result in intervals capturing the desired parameter more frequently. There is (always) a price, though; smaller C yield intervals that are more precise (and more likely to be wrong), while larger C yield intervals that are less precise (and less likely to be wrong). There is usually only one way to get more precise confidence intervals while maintaining the same confidence level C: collect more data.

In this lecture we consider only confidence intervals for univariate data (or data that could be univariate). When discussing confidence intervals, the **estimate** is the estimated value of the parameter of interest, and the **margin of error** (which I abbreviate with "moe") is the quantity used to describe the uncertainty

associated with the estimate. The most common and familiar type of confidence interval is a two-sided confidence interval of the form:

$$\text{estimate} \pm \text{moe}$$

For estimates of parameters from symmetric distributions, such a confidence interval is called an **equal-tail confidence interval** (so called because the confidence interval is equally likely to err on either side of the estimate). For estimates for parameters from non-symmetric distributions, an equal-tail confidence interval may not be symmetric around the estimate. Also, confidence intervals need not be two-sided; one-sided confidence intervals, which effectively are a lower-bound or upper-bound for the location of the parameter of interest, may also be constructed.

Confidence Intervals "By Hand"

One approach for constructing confidence intervals in R is "by hand", where the user computes the estimate and the moe. This approach is the only one available if no function for constructing the desired confidence interval exists (that will not be the case for the confidence intervals we will see, but may be the case for others).

Let's consider a confidence interval for the population mean when the population standard deviation is known. The estimate is the sample mean, $\bar{x}$, and the moe is $z_{\frac{1-C}{2}} \frac{\sigma}{\sqrt{n}}$, where C is the confidence level, z_α is the quantile of the standard Normal distribution such that $P(Z > z_\alpha) = 1 - \Phi(z_\alpha) = \alpha$, σ is the population standard deviation (presumed known, which is unrealistic), and n is the sample size. So the two-sided confidence interval is:

$$\bar{x} \pm z_{\frac{1-C}{2}} \frac{\sigma}{\sqrt{n}}$$

Such an interval relies on the Central Limit Theorem to ensure the quality of the interval. For large n, it should be safe (with "large" meaning over 30, according to DeVore), and otherwise one may want to look at the shape of the distribution of the data to decide whether it is safe to use these procedures (the more "Normal", the better).

The "by hand" approach finds all involved quantities individually and uses them to construct the confidence intervals.

Let's construct a confidence interval for sepal length of versicolor iris flowers in the `iris` data set. There are 50 observations, so it should be safe to use the formula for the CI described above. We will construct a 95% confidence interval assume that $\sigma = 0.5$.

```
# Get the data
vers <- split(iris$Sepal.Length, iris$Species)$versicolor
xbar <- mean(vers)
xbar
```

```
## [1] 5.936
```

```
# Get critical value for confidence level
zstar <- qnorm(0.025, lower.tail = FALSE)
sigma <- 0.5
# Compute margin of error
moe <- zstar * sigma/sqrt(length(vers))
moe
```

```
## [1] 0.1385904
```

```
# The confidence interval
ci <- c(Lower = xbar - moe, Upper = xbar + moe)
ci
```

```
##   Lower   Upper
## 5.79741 6.07459
```

Of course, this is the long way to compute confidence intervals, and R has built-in functions for computing most of the confidence intervals we want. The "by hand" method relies simply on knowing how to compute the values used in the definition of the desired confidence interval, and combining them to get the desired interval. Since this uses R as little more than a glorified calculator and alternative to a table, I will say no more about the "by hand" approach (of course, if you are computing a confidence interval for which there is no R function, the "by hand" approach is the only available route, though you may save some time and make a contribution to the R community by writing your own R function for computing this novel confidence interval in general).

R Functions for Confidence Intervals

Confidence intervals and hypothesis testing are closely related concepts, so the R functions that find confidence intervals are also the functions that perform hypothesis testing (all of these functions being in the **stats** package, which is loaded by default). If desired, it is possible to extract just the desired confidence interval from the output of these functions, though when used interactively, the confidence interval and the result of some hypothesis test are often shown together.

When constructing confidence intervals, there are two parameters to be aware of: `conf.level` and `alternative`. `conf.level` specifies the desired confidence level associated with the interval, C. Thus, `conf.level = .99` tells the function computing the confidence interval that a 99% confidence interval is desired. (The default behavior is to construct a 95% confidence interval.) `alternative` determines whether a two-sided interval, an upper-bound, or a lower-bound are computed. `alternative = "two.sided"` creates a two-sided confidence interval (this is usually the default behavior, though, so specifying this may not be necessary). `alternative = "greater"` computes a one-sided '00C% confidence lower bound, and `alternative = "less"` returns a one-sided 100C% confidence upper bound. (The parameter `alternative` gets its name and behavior from the alternative hypothesis when performing hypothesis testing, which we will discuss next lecture.)

I now discuss different confidence intervals intended to capture different population parameters.

Population Mean, Single Sample

We saw a procedure for constructing a confidence interval for the population mean earlier, and that procedure assumed the population standard deviation σ is known. That assumption is unrealistic, and instead we base our confidence intervals on the random variable $T = \frac{\overline{X}-\mu}{\frac{S}{\sqrt{n}}}$. If the data X_i is drawn from some Normal distribution, then T will follow Student's t distribution with $\nu = n-1$ degrees of freedom: $T \sim t(n-1)$. Thus the confidence interval is constructed using critical values from the t distribution, and the resulting confidence interval is:

$$\bar{x} \pm t_{\frac{\alpha}{2},n-1}\frac{s}{\sqrt{n}} \equiv \left(\bar{x} - t_{\frac{\alpha}{2},n-1}\frac{s}{\sqrt{n}}; \bar{x} + t_{\frac{\alpha}{2},n-1}\frac{s}{\sqrt{n}}\right)$$

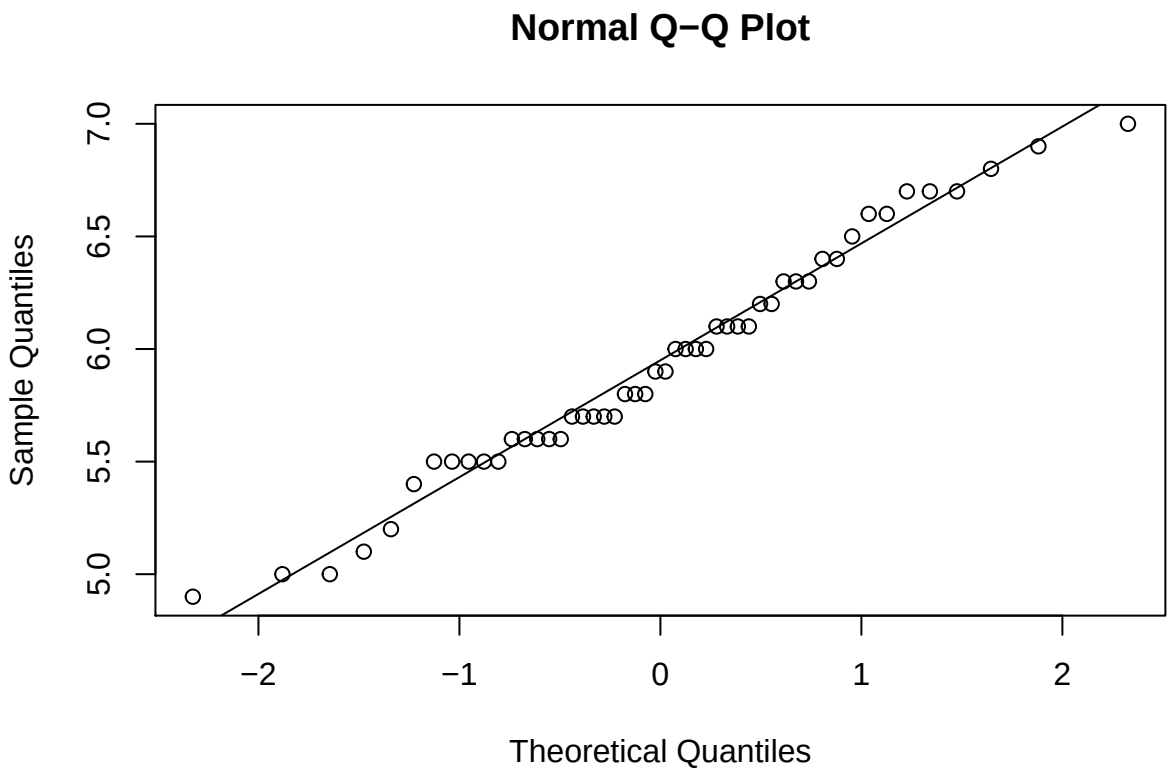
where if $T \sim t(\nu)$, $P\left(T \geq t_{\alpha,\nu}\right) = \alpha$.

One should check the Normality assumption made by this confidence interval before using it. A **Q-Q plot** is a good way to do so, and a Q-Q plot checking for Normality can be created using the `qnorm()` function.

`qnorm(x)` creates a Q-Q plot checking normality for a data vector `x`, and `qqline(x)` will add a line by which to judge how well the distribution fits the data (being clost to the line suggests a good fit, while strange patterns like an S-shape curve suggest a poor fit).

I check how well the Normal distribution describes the versicolor iris flower sepal length data below. It is clear that the assumption of Normality holds quite well for this data.

```
qqnorm(vers)
qqline(vers)
```



That said, the Normality assumption turns out not to be crucial for these confidence intervals, and the methods turn out to be robust enough to work well even when that assumption does not hold. If a distribution is roughly symmetric with no outliers, it's probably safe to build a confidence interval using methods based on the t distribution, and even if these two do not quite hold, there are circumstances when even then the t distribution will still work well.

The function $t.test()$ will construct a confidence interval for the true mean μ, and I demonstrate its use below.

```
# Construct a 95% two-sided confidence interval
t.test(vers)
```

```
##
##  One Sample t-test
##
## data:  vers
## t = 81.318, df = 49, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  5.789306 6.082694
## sample estimates:
## mean of x
##     5.936
```

```r
# Construct a 99% upper confidence bound
t.test(vers, conf.level = 0.99, alternative = "less")
```

```
##
##  One Sample t-test
##
## data:  vers
## t = 81.318, df = 49, p-value = 1
## alternative hypothesis: true mean is less than 0
## 99 percent confidence interval:
##      -Inf 6.111551
## sample estimates:
## mean of x
##     5.936
```

```r
# t.test creates a list containing all computed stats, including the
# confidence interval. I can extract the CI by accessing the conf.int
# variable in the list
tt1 <- t.test(vers, conf.level = 0.9)
tt2 <- t.test(vers, conf.level = 0.95)
tt3 <- t.test(vers, conf.level = 0.99)
str(tt1)
```

```
## List of 9
##  $ statistic  : Named num 81.3
##   ..- attr(*, "names")= chr "t"
##  $ parameter  : Named num 49
##   ..- attr(*, "names")= chr "df"
##  $ p.value    : num 6.14e-54
##  $ conf.int   : atomic [1:2] 5.81 6.06
##   ..- attr(*, "conf.level")= num 0.9
##  $ estimate   : Named num 5.94
##   ..- attr(*, "names")= chr "mean of x"
##  $ null.value : Named num 0
##   ..- attr(*, "names")= chr "mean"
##  $ alternative: chr "two.sided"
##  $ method     : chr "One Sample t-test"
##  $ data.name  : chr "vers"
##  - attr(*, "class")= chr "htest"
```

```r
# Create a graphic comparing these CIs
library(ggplot2)
# The end goal is to create an object easily handled by ggplot. I start by
# making a list with my data
conf_int_dat <- list(`90% CI` = as.list(tt1$conf.int), `95% CI` = as.list(tt2$conf.int),
    `99% CI` = as.list(tt3$conf.int))
# Use melt and cast from reshape package to make a data frame I can work
# with
library(reshape)
melted_cid <- melt(conf_int_dat)
melted_cid
```

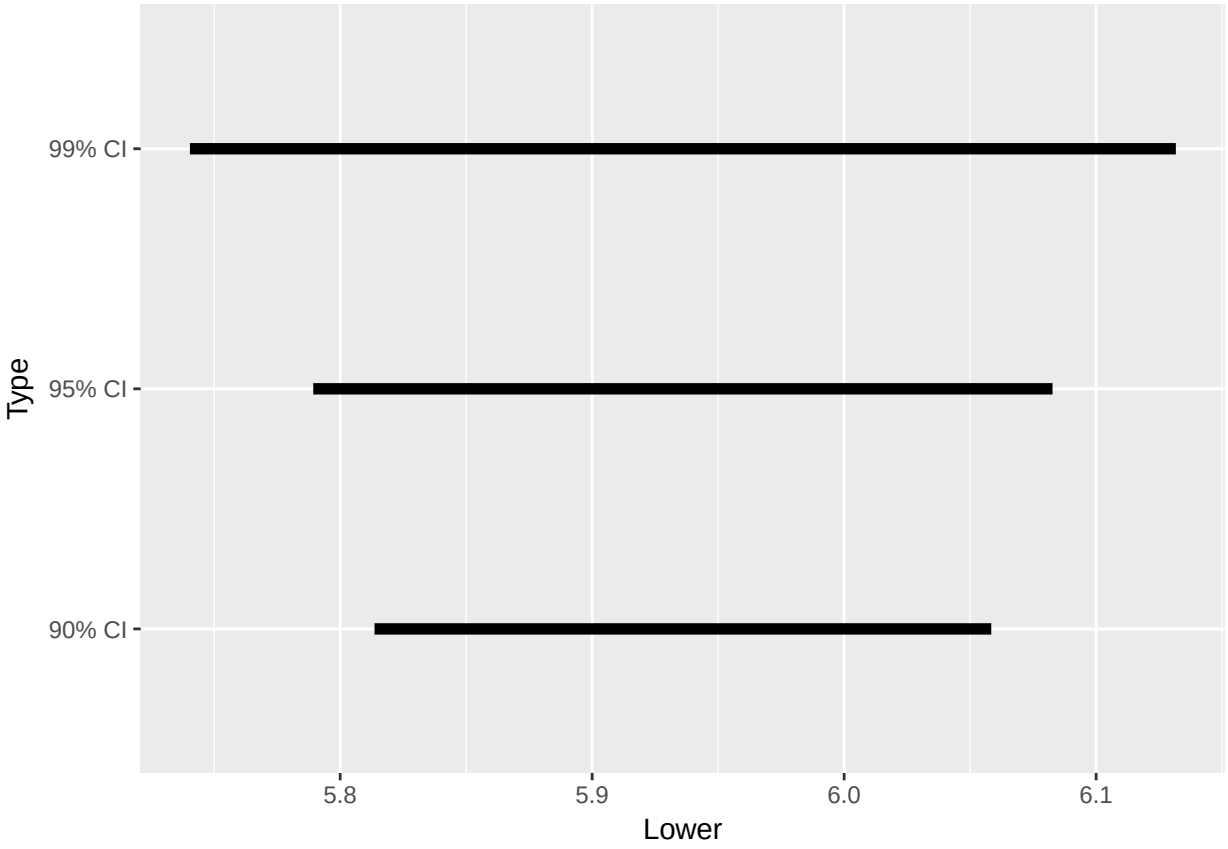
```
##      value L2     L1
## 1 5.813616  1 90% CI
## 2 6.058384  2 90% CI
```

```
## 3 5.789306  1 95% CI
## 4 6.082694  2 95% CI
## 5 5.740370  1 99% CI
## 6 6.131630  2 99% CI
```

```
plot_conf_int_dat <- cast(melted_cid, L1 ~ L2)
names(plot_conf_int_dat) <- c("Type", "Lower", "Upper")
plot_conf_int_dat
```

```
##     Type    Lower    Upper
## 1 90% CI 5.813616 6.058384
## 2 95% CI 5.789306 6.082694
## 3 99% CI 5.740370 6.131630
```

```
# Now, create desired graphic
ggplot(plot_conf_int_dat) + geom_segment(aes(x = Lower, xend = Upper, y = Type,
    yend = Type), size = 2)
```




```
# It's clear that larger confidence levels lead to larger intervals, as can
# be seen in this graphic
```

Population Proportion, Single Sample

Suppose we collect categorical data, and encode observations that have some trait being tracked ("successes") with 1, and the rest 0. This is a Bernoulli random variable, and our objective is to estimate the population proportion of successes, p. The number of successes in the sample, $\sum_{i=1}^{n} X_i$, is a binomial random variable, and the natural estimator for the population proportion is the sample proportion, $\hat{p} = \frac{1}{n}\sum_{i=1}^{n} X_i$.

Computing a confidence interval for p has been revisited many times, each method trying to rectify faults of others (there is even an R package, **binom**, devoted to providing all the different methods for computing confidence intervals for the population proportion). A method that used to be popular, based on the Normal approximation for the distribution of the sample proportion $\hat{p}$, is the confidence interval:

$$\hat{p} \pm z_{\frac{\alpha}{2}} \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \equiv \left(\hat{p} - z_{\frac{\alpha}{2}} \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}; \hat{p} + z_{\frac{\alpha}{2}} \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \right)$$

The function `prop.test()` will compute confidence intervals using this method, having the format `prop.test(x, n)`, where `x` is the number of successes and `n` the sample size.

A June CNN-ORC poll found 420 individuals surveyed out of 1001 would vote for Donald Trump if the election were held that day. Let's use this data to get a confidence interval for the proportion of voters who would vote for Trump.

```
prop.test(420, 1001)
```

```
##
##  1-sample proportions test with continuity correction
##
## data:  420 out of 1001, null probability 0.5
## X-squared = 25.574, df = 1, p-value = 4.256e-07
## alternative hypothesis: true p is not equal to 0.5
## 95 percent confidence interval:
##  0.3888812 0.4509045
## sample estimates:
##         p
## 0.4195804
```

`prop.test()` found a confidence interval with a margin of error of roughly 3% (the target margin of error of the survey).

This confidence interval was popular in textbooks for years, but it is flawed to the point few advocate its use. The primary reason is that this interval is not guaranteed to capture the true p with the probability we specify, behaving erratically for different combinations of sample sizes and with more-or-less extreme p. It's thus considered unreliable, and alternatives are preferred.

The **Hmisc** package contains a function for computing confidence intervals for the population proportion, `binconf()`, that allows for a few other confidence intervals for the population proportion to be computed. The score-based method is advocated by the textbook author used in the lecture course, and is supported by `binconf()`. The interface of the function is different from the interface of similar functions in the **stats** (base R) package (it was not written by the same authors); instead of specifying `conf.level`, one specifies `alpha` in `binconf()`, with $\alpha = 1 - C$ (the default value for `alpha` is .05, corresponding to a 95% confidence interval). Additionally, `binconf()` will not directly compute one-sided confidence bounds like `prop.test()` would. But otherwise usage is similar, with `binconf(x, n, method = "wilson")` computing a 95% confidence interval for a data set in which, out of `n` trials, there were `x` successes, using the score confidence interval.

Let's use `binconf()` to compute the score confidence interval for the proportion of voters supporting Donald Trump, obtaining both a 95% and 99% confidence interval.

```
library(Hmisc)
# 95% CI
binconf(420, 1001, method = "wilson")
```

```
##   PointEst     Lower     Upper
##  0.4195804 0.3893738 0.4504019
```

```
# 99% CI
binconf(420, 1001, alpha = 0.01, method = "wilson")
```

```
##   PointEst     Lower     Upper
##  0.4195804 0.3800618 0.4601581
```

The 95% confidence interval using the score method is not the same as the result from `prop.test()`, but not drastically different either.

`binom.test()` supports the exact confidence interval, which uses the CDF of the binomial distribution rather than a Normal approximation. The exact confidence interval can be computed via `binom.test(x, n)` (with all other common parameters such as `conf.level` and `alternative` being supported). `binconf()` also supports the exact confidence interval, where one merely sets `method = "exact"` to compute it.

```
binom.test(420, 1001)
```

```
##
##  Exact binomial test
##
## data:  420 and 1001
## number of successes = 420, number of trials = 1001, p-value =
## 4.029e-07
## alternative hypothesis: true probability of success is not equal to 0.5
## 95 percent confidence interval:
##  0.3887849 0.4508502
## sample estimates:
## probability of success
##              0.4195804
```

```
# 99% confidence lower bound for Trump support, using binom.test
binom.test(420, 1001, alternative = "greater", conf.level = 0.99)
```

```
##
##  Exact binomial test
##
## data:  420 and 1001
## number of successes = 420, number of trials = 1001, p-value = 1
## alternative hypothesis: true probability of success is greater than 0.5
## 99 percent confidence interval:
##  0.3831763 1.0000000
## sample estimates:
## probability of success
##              0.4195804
```

Paired Sample Confidence Interval for Difference in Means

Suppose you have paired data, X_i and a corresponding Y_i. Examples of paired data include weight before (X_i) and after (Y_i) a weight-loss program, or the political views of a husband (X_i) and wife (Y_i). There are two different means for X_i and Y_i, denoted μ_X and μ_Y, and we want to know how these means compare.

Since the data is paired, we create a confidence interval using the differences of each X_i and Y_i, $D_i = X_i - Y_i$, and the confidence interval will describe the location of the mean difference μ_D. Thus, for paired data, after computing the differences, the procedure for computing a confidence interval is the same as the univariate case.

One approach for computing the confidence interval for the mean difference of paired data is with `t.test(x`

`- y)`, where `x` and `y` are vectors containing the paired data. Another approach would be to use `t.test(x, y, paired = TRUE)`.

I illustrate by examining whether stricter enforcement of speed limit laws helps reduce the number of accidents. The `Traffic` data set (**MASS**) shows the effects of Swedish speed limits on accidents after an experiment where specific days in 1961 and 1962 saw different speed limits applied, in a paired study (days each year were matched). Thus a matched pairs design could be applied to see if the speed limit laws helped reduce accidents.

```
library(MASS)
library(reshape)
head(Traffic)
```

```
##   year day limit  y
## 1 1961   1    no  9
## 2 1961   2    no 11
## 3 1961   3    no  9
## 4 1961   4    no 20
## 5 1961   5    no 31
## 6 1961   6    no 26
```

```
# The Traffic data set needs to be formed into a format that is more easily
# worked with. The following code reshapes the data into a form that allows
# for easier comparison of paired days
new_Traffic <- with(Traffic, data.frame(year = year, day = day, limit = as.character(limit),
    y = as.character(y)))
melt_Traffic <- melt(new_Traffic, id.vars = c("year", "day"), measure.vars = c("limit",
    "y"))
form_Traffic <- cast(melt_Traffic, day ~ year + variable)
form_Traffic <- with(form_Traffic, data.frame(day = day, limit_61 = `1961_limit`,
    accidents_61 = as.numeric(as.character(`1961_y`)), limit_62 = `1962_limit`,
    accidents_62 = as.numeric(as.character(`1962_y`))))
head(form_Traffic)
```

```
##   day limit_61 accidents_61 limit_62 accidents_62
## 1   1       no            9       no            9
## 2   2       no           11       no           20
## 3   3       no            9       no           15
## 4   4       no           20       no           14
## 5   5       no           31       no           30
## 6   6       no           26       no           23
```

```
# Now we subset this data so that only days where speed limits were enforced
# differently between the two years
diff_Traffic <- subset(form_Traffic, select = c("accidents_61", "accidents_62",
    "limit_61", "limit_62"), subset = limit_61 != limit_62)
head(diff_Traffic)
```

```
##    accidents_61 accidents_62 limit_61 limit_62
## 11           29           17       no      yes
## 12           40           23       no      yes
## 13           28           16       no      yes
## 14           17           20       no      yes
## 15           15           13       no      yes
## 16           21           13       no      yes
```

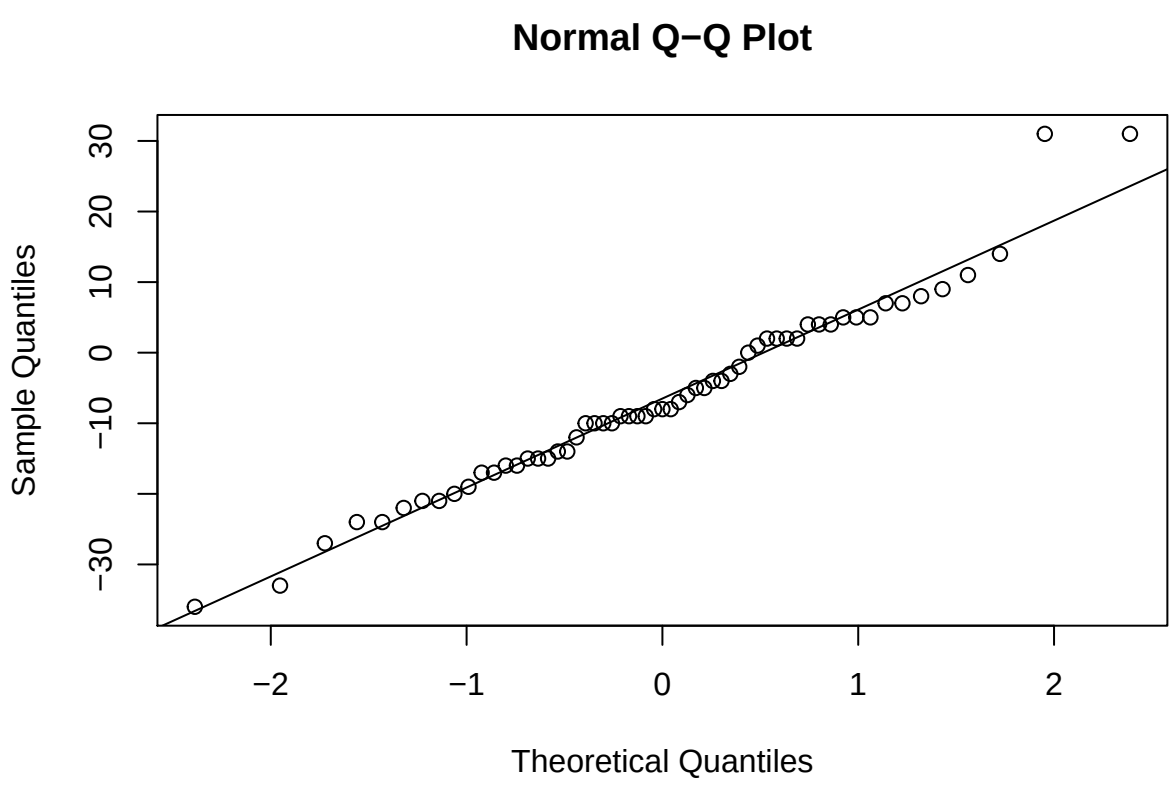
```
# Get a vector for accidents when the speed limit was not enforced, and a
# vector for accidents for when it was
accident_no_limit <- with(diff_Traffic, c(accidents_61[limit_61 == "no"], accidents_62[limit_62 ==
    "no"]))
accident_limit <- with(diff_Traffic, c(accidents_61[limit_61 == "yes"], accidents_62[limit_62 ==
    "yes"]))
accident_limit
```

```
##  [1] 19 19  9 21 22 23 14 19 15 13 22 42 29 21 12 16 17 23 16 20 13 13  9
## [24] 10 27 12  7 11 15 29 17 17 15 25  9 16 25 25 16 22 21 17 26 41 25 12
## [47] 17 24 26 16 15 12 22 24 16 25 14 15  9
```

```
accident_no_limit
```

```
##  [1] 29 40 28 17 15 21 24 15 32 22 24 11 27 37 32 25 20 40 21 18 35 21 25
## [24] 34 42 27 34 47 36 15 26 21 39 39 21 15 17 20 24 30 25  8 21 10 14 18
## [47] 26 38 31 12  8 22 17 31 49 23 14 25 24
```

```
# It took a lot of work, but finally we have the data in a format we want.
# Now, the first thing I check is whether the differences are Normally
# distributed
qqnorm(accident_limit - accident_no_limit)
qqline(accident_limit - accident_no_limit)
```




```
# Aside from a couple observations, the Normal distribution seems to fit
# very well. t procedures are safe to use.
t.test(accident_limit, accident_no_limit, paired = TRUE)
```

```
##
##  Paired t-test
##
```

```
## data:  accident_limit and accident_no_limit
## t = -3.7744, df = 58, p-value = 0.0003794
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -9.856470 -3.024886
## sample estimates:
## mean of the differences
##               -6.440678
```

The above analysis suggests that enforcing speed limits reduces the number of accidents, since zero is not in the interval.

Test for Difference in Population Means Between Two Samples

Analyzing paired data is simple since the analysis reduces to the univariate case. Two independent samples, though, are not as simple.

Suppose we have two *independent* data sets, with $X_1, ..., X_n$ the first and $Y_1, ..., Y_m$ the second, each drawn from two distinct populations. We typically wish to know whether the means of the two populations, μ_X and μ_Y, differ. Thus, we wish to obtain a confidence interval for the difference in means of the populations, $\mu_X - \mu_Y$.

The natural estimator for $\mu_X - \mu_Y$ is $\overline{X} - \overline{Y}$, so the point estimate of the difference is $\bar{x} - \bar{y}$. The margin of error, though, depends on whether we believe the populations are homoskedastic or heteroskedastic. **Homoskedastic** populations have equal variances, so $\sigma_X^2 = \sigma_Y^2$. **Heteroskedastic** populations, though, have different variances, so $\sigma_X^2 \neq \sigma_Y^2$. Between the two, homoskedasticity is a much stronger assumption, so unless you have a reason for believing otherwise, you should assume heteroskedasticity. (It is the correct assumption the populations are, in fact, heteroskedastic, and if they are actually homoskedastic, the confidence interval should still be reasonably precise.)

For homoskedastic populations, the confidence interval is:

$$\bar{x} - \bar{y} \pm t_{\frac{\alpha}{2}, n+m-2} s \sqrt{\frac{1}{n} + \frac{1}{m}} \equiv \left(\bar{x} - \bar{y} - t_{\frac{\alpha}{2}, n+m-2} s \sqrt{\frac{1}{n} + \frac{1}{m}}; \bar{x} - \bar{y} + t_{\frac{\alpha}{2}, n+m-2} s \sqrt{\frac{1}{n} + \frac{1}{m}} \right)$$

where s is the sample standard deviation of the pooled sample (that is, when $X_1, ..., X_n$ and $Y_1, ..., Y_m$ are treated as one sample).

Confidence intervals for heteroskedastic data are more complicated, since the t distribution is no longer the correct distribution to use to describe the behavior of the random variable from which the confidence interval is derived. That said, the t distribution can approximate the true distribution when the degrees of freedom used is:

$$\nu = \frac{\left(\frac{s_X^2}{n} + \frac{s_Y^2}{m} \right)^2}{\frac{\left(s_X^2/n \right)^2}{n-1} + \frac{\left(s_Y^2/m \right)^2}{m-1}}$$

with s_X^2 and s_Y^2 being the variances of the data sets $x_1, ..., x_n$ and $y_1, ..., y_m$, respectively. Then the confidence interval is approximately:

$$\bar{x} - \bar{y} \pm t_{\frac{\alpha}{2}, \nu} \sqrt{\frac{s_X^2}{n} + \frac{s_Y^2}{m}}$$

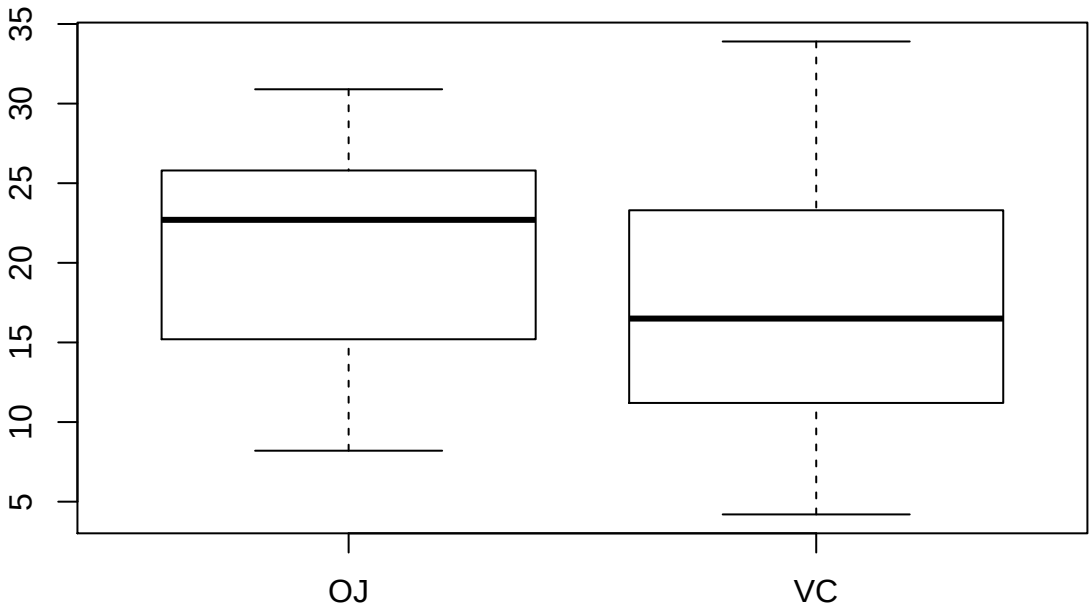
`t.test()` can compute confidence intervals in both these cases. The function call is similar to the case of paired data, though without setting `paired = TRUE`. Instead, set `var.equal = TRUE` for homoskedastic populations, and `var.equal = FALSE` (the default) for heteroskedastic populations.

Let's illustrate this by determining whether orange juice or a vitamin C supplement increase tooth growth in guinea pigs (in the `ToothGrowth` data set).

```
# First, let's get the data into separate vectors
split_len <- split(ToothGrowth$len, ToothGrowth$supp)
OJ <- split_len$OJ
VC <- split_len$VC
# Gets CI for difference in means
t.test(OJ, VC)
```

```
##
##  Welch Two Sample t-test
##
## data:  OJ and VC
## t = 1.9153, df = 55.309, p-value = 0.06063
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.1710156  7.5710156
## sample estimates:
## mean of x mean of y
##  20.66333  16.96333
```

```
# Is there homoskedasticity? Let's check a boxplot
boxplot(len ~ supp, data = ToothGrowth)
```




```
# These populations look homoskedastic; spread does not differ drastically
# between the two. A CI that assumes homoskedasticity is thus computed.
t.test(OJ, VC, var.equal = TRUE)
```

```
##
##  Two Sample t-test
##
## data:  OJ and VC
```

```
## t = 1.9153, df = 58, p-value = 0.06039
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.1670064  7.5670064
## sample estimates:
## mean of x mean of y
##  20.66333  16.96333
```

```
# Not much changed (only slightly narrower).
```

Difference in Population Proportions

If we have two populations, each with a respective population proportion representing the proportion of that population possessing some trait, we can construct a confidence interval for the difference in the proportions, $p_X - p_Y$. The form of the confidence interval depends on which type is being used, and I will not discuss the details of how these confidence intervals are computed, but simply say that some of the functions we have seen for single sample confidence intervals, such as `prop.test()`, support comparing proportions (`binconf()` and `binom.test()` do not). The only difference is that rather than passing a single count and a single sample size to these functions, you pass a vector containing the count of successes in the two population, and a vector containing the two sample sizes (naturally they should be in the same order).

In the next example, I compare the proportions of men and women with melanoma who die from the disease.

```
mel_split <- split(Melanoma$status, Melanoma$sex)
# Get logical vectors for whether patient died from melanoma. Group 0 is
# women and group 1 is men, and the code 1 in the data indicates death from
# melanoma
fem_death <- mel_split[["0"]] == 1
man_death <- mel_split[["1"]] == 1
# Vector containing the number of deaths for both men and women
deaths <- c(Female = sum(fem_death), Male = sum(man_death))
# Vector containing sample sizes
size <- c(Female = length(fem_death), Male = length(man_death))
deaths/size
```

```
##    Female      Male
## 0.2222222 0.3670886
```

```
# prop.test, with the 'classic' CLT-based CI
prop.test(deaths, size)
```

```
##
##  2-sample test for equality of proportions with continuity
##  correction
##
## data:  deaths out of size
## X-squared = 4.3803, df = 1, p-value = 0.03636
## alternative hypothesis: two.sided
## 95 percent confidence interval:
##  -0.283876633 -0.005856138
## sample estimates:
##    prop 1    prop 2
## 0.2222222 0.3670886
```

Lecture 14

Hypothesis Testing Basics

Hypothesis testing is another common form of statistical inference. In hypothesis testing, our objective is to decide whether we have enough evidence to reject the **null hypothesis**, a statement about the population distribution that *a priori* we believe to be true, in favor of the **alternative hypothesis**, a statement about the population disagreeing with the null hypothesis. Usually the null hypothesis is denoted by H_0 and the alternative by H_A.

The first statistical tests introduced to students are tests about the value of a population parameter (other tests are possible, though). These tests generally take the form:

$$H_0 : \theta = \theta_o$$
$$H_A : \begin{cases} \theta < \theta_0 \\ \theta \neq \theta_0 \\ \theta > \theta_0 \end{cases}$$

These are the tests that I discuss.

In statistical testing, one computes a **test statistic** used to compute a **p-value**, which represents the probability of observing a test statistic at least as "extreme" as the one actually observed if H_0 were in fact true. Very small p-values indicate a false null hypothesis. Usually what constitutes a "small" p-value is decided beforehand by choosing a **level of significance**, typically denoted α. If the p-value is less than α, H_0 is rejected in favor of H_A; otherwise, we fail to reject H_0. Common α include 0.05, 0.01, 0.001, and 0.1. The smaller α, the more difficult it is to reject H_0.

In hypothesis testing, we need to be aware of two types of errors, called Type I and Type II errors. A **Type I** error is rejecting H_0 when H_0 is true. A **Type II** error is failing to reject H_0 when H_A is true. For any test, we want to know the probability of making either type of error. The probability of a Type I error is α, the level of significance; this means we specify beforehand what Type I error we want. Type II errors are much more complicated, since they depend not only on what the true value of θ is (which we will call θ_A, the value of θ under the alternative assumed true for Type II error analysis) but additionally on the specified α and sample size n (other parameters may be involved as well, but they are assumed constant and unable to be changed). For any testing scheme we represent the Type II error with $\beta(\theta_A)$, the probability of failing to reject H_0 when the true value of θ is θ_A. Generally $\beta(\theta_A)$ is large when θ_A is close to θ_0 (in fact, $\beta(\theta_0) = 1 - \alpha$), and small when θ_A is distant from θ_0. This should make intuitive sense; a big difference between θ_0 and θ_A should be easy to detect, but a small difference may be more difficult. In practice, researchers pick a θ_A for which they want to be able to detect a difference with some specified probability β. They then use this to pick a sample size that gives the test the desired property.

Some prefer to discuss the **power** of a test rather than the probability of a Type II error. The power of a test is the probability of rejecting H_0 when $\theta = \theta_A$. Power connects both Type I and Type II errors, since the power of a test is defined to be $\pi(\theta_A) = 1 - \beta(\theta_A)$ and $\pi(\theta_0) = \alpha$. The same principles discussed with regard to the probability of Type I and Type II errors hold with power.

As with confidence intervals, R has many functions for handling hypothesis testing (in fact, you have already seen most of them). We can perform hypothesis testing "by hand" (not using any functions designed for performing an entire test), or using functions designed for hypothesis testing. We start with methods "by hand".

Hypothesis Testing "By Hand"

Hypothesis testing "by hand" involves computing the test statistic and finding the appropriate p-value for a test directly. This is the only means of hypothesis testing if a function for performing some desired test does not exist (which is probably not the case unless you're working with a very obscure test).

Let's start by considering a simple z-test, where the hypotheses are:

$$H_0 : \mu = \mu_0$$
$$H_A : \begin{cases} \mu < \mu_0 \\ \mu \neq \mu_0 \\ \mu > \mu_0 \end{cases}$$

The test statistic is $z = \frac{\bar{x} - \mu_0}{\frac{\sigma}{\sqrt{n}}}$ and the p-value (denoted p_{val}) is:

$$p_{\text{val}} = \begin{cases} \Phi(z) & \text{if } H_A \text{ is } \mu < \mu_0 \\ 2\left(1 - \Phi(|z|)\right) & \text{if } H_A \text{ is } \mu \neq \mu_0 \\ 1 - \Phi(z) & \text{if } H_A \text{ is } \mu > \mu_0 \end{cases}$$

This test is unrealistic since it assumes σ is known, but it is simple to analyze.

I demonstrate these procedures by testing whether the diameter of black cherry trees is 12 in. I test the hypotheses:

$$H_0 : \mu = 12$$
$$H_A : \mu \neq 12$$

I use the data set `trees` and assume that the population standard deviation is 3. I will base my test on the significance level of $\alpha = .05$.

```
# Some basic numbers
xbar <- mean(trees$Girth)
xbar
```

```
## [1] 13.24839
```

```
n <- nrow(trees)
mu_0 <- 12
sigma <- 3
# Test statistic
z <- (xbar - mu_0)/(sigma/sqrt(n))
z
```

```
## [1] 2.316908
```

```
# Get p-value, using pnorm
pval <- 2 * (1 - pnorm(abs(z)))
pval
```

```
## [1] 0.02050872
```

Since 0.02 is less than my significance level of .05, I reject the null hypothesis; the mean diameter of black cherry trees is not 12. That said, if I were to use a significance level of $\alpha = .01$, I would not reject the null hypothesis. Remember that the p-value represents the largest level of significance at which I would fail to reject the null hypothesis. Thus the p-value measures how unlikely my data is if the null hypothesis were true, with smaller p-values indicating more evidence against the null hypothesis.

Testing hypotheses "by hand" follows a similar format to the one demonstrated here, so I do not demonstrate this approach any more in this lecture.

R Functions for Statistical Testing

R has functions for performing many common statistical tests, including many that come with any R installation in the **stats** package. We already saw all the functions for performing statistical tests I consider in this lecture when we discussed confidence intervals. There may be additional parameters to specify depending on the statistical test, such as what the population mean under the null hypothesis is, but otherwise little has changed. Many of these functions have a common parameter `alternative` that determines what the alternative hypothesis is. For a two-sided test (that is, when the alternative hypothesis is that $\theta \neq \theta_0$), you may set `alternative = "two.sided"` (this is the default, though, so setting this may not be necessary). If the alternative hypothesis says $\theta > \theta_0$, set `alternative = "greater"`, and if the alternative hypothesis says $\theta < \theta_0$, set `alternative = "less"`.

The statistical testing functions in **stats** will report the results of a statistical test and many other important quantities, such as the estimate of the parameters investigated, the sample size, degrees of freedom, the value of a test statistic, the p-value, and even the corresponding confidence interval. They do not state whether to reject or not reject the null hypothesis; you are expected to tell from the p-value reported whether to reject or not reject based on the significance level you have chosen.

stats includes some functions for Type II error analysis (often in the form of power analysis, which amounts to the same sort of analysis), but this is a difficult analysis in general. Functions for study planning may be included in other packages.

Test for Location of Population Mean

Suppose you wish to test the hypotheses:

$$H_0 : \mu = \mu_0$$

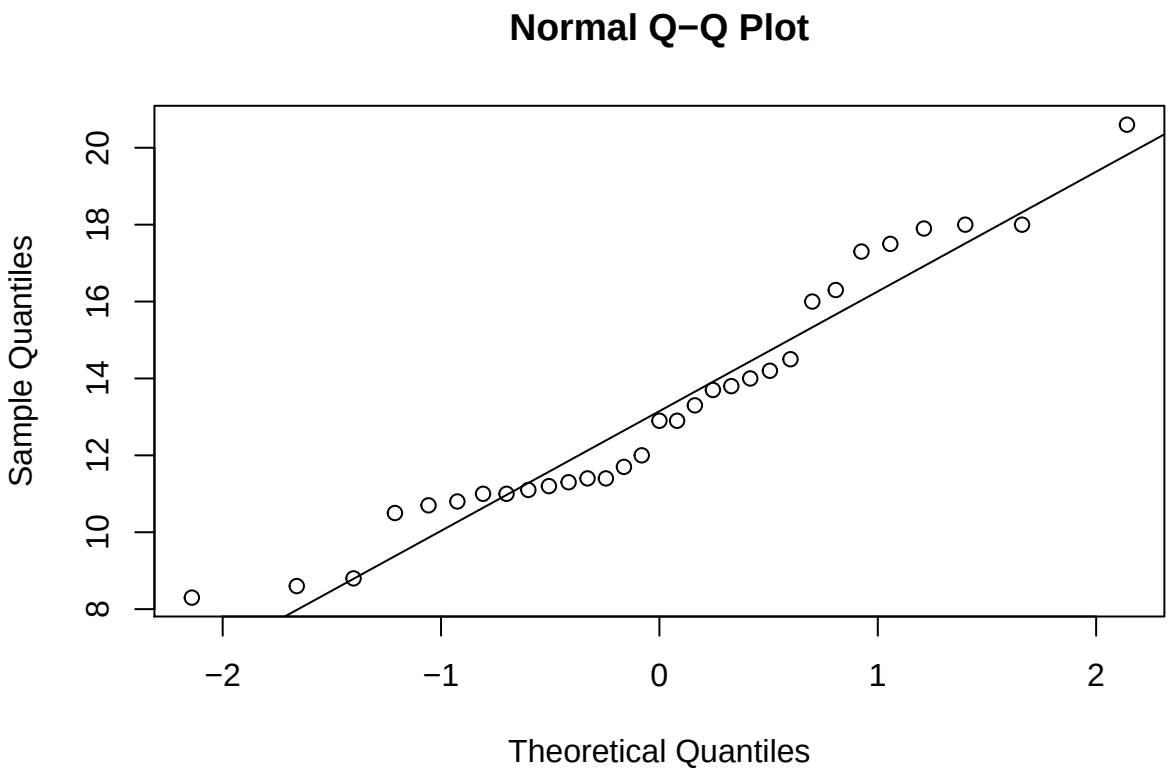
$$H_A : \begin{cases} \mu < \mu_0 \\ \mu \neq \mu_0 \\ \mu > \mu_0 \end{cases}$$

Furthermore, you assume (after perhaps using `qqnorm()` or some other procedure to check) that your data is Normally distributed, and thus it is safe to use the t-test (or your sample size is large enough for this to be safe to use anyway). The function `t.test()` allows you to test these hypotheses. The call `t.test(x, mu = mu0)` will test whether the data stored in vector `x` has a mean of `mu0` or whether the true mean differs from `mu0` (remember: change the form of the alternative hypothesis with the parameter `alternative`), using the test statistic:

$$t = \frac{\bar{x} - \mu_0}{\frac{s}{\sqrt{n}}}$$

I test the hypotheses regarding tree girth in the `trees` data set mentioned earlier in this lecture, but this time using `t.test()`, using a level of significance of $\alpha = .05$.

```
# Check the normality assumption
qqnorm(trees$Girth)
qqline(trees$Girth)
```




```
# Data appears Normally distributed, so it's safe to use t.test
t.test(trees$Girth, mu = 12)
```

```
##
##  One Sample t-test
##
## data:  trees$Girth
## t = 2.2149, df = 30, p-value = 0.0345
## alternative hypothesis: true mean is not equal to 12
## 95 percent confidence interval:
##  12.09731 14.39947
## sample estimates:
## mean of x
##  13.24839
```

With a p-value of 0.0345, I reject the null hypothesis in favor of the alternative.

Test for Value of a Proportion

Suppose you wish to test the hypotheses:

$$H_0 : p = p_0$$

$$H_A : \begin{cases} p < p_0 \\ p \neq p_0 \\ p > p_0 \end{cases}$$

Suppose your sample size is large enough to invoke the Central Limit Theorem to describe the sampling distribution of the statistic $\hat{p} = \frac{1}{n}\sum_{i=1}^{n} X_i$, and thus use the test statistic:

$$z = \frac{\hat{p} - p_0}{\sqrt{\frac{p_0(1-p_0)}{n}}}$$

You can use `prop.test()` to test the hypotheses if these conditions hold. The call `prop.test(x, n, p = p0)` will test whether the true proportion is `p0` when there were `x` successes out of `n` trials (both `x` and `n` are single non-negative integers).

Suppose that out of 1118 survey participants, 562 favored Hillary Clinton over Donald Trump (this data is fictitious). I test whether the candidates are tied or whether Hillary Clinton is winning below:

```
prop.test(562, 1118, alternative = "greater", p = 0.5)
```

```
##
##  1-sample proportions test with continuity correction
##
## data:  562 out of 1118, null probability 0.5
## X-squared = 0.022361, df = 1, p-value = 0.4406
## alternative hypothesis: true p is greater than 0.5
## 95 percent confidence interval:
##  0.477664 1.000000
## sample estimates:
##         p
## 0.5026834
```

With a p-value of 0.4406, I soundly fail to reject the null hypothesis.

Suppose that your sample size is not large enough to use the above procedure (or you would simply rather not risk it), and you would rather use the binomial distribution to perform an exact test of the null and alternative hypotheses. `binom.test()` will allow you to perform such a test with a function call similar to `prop.test()`.

Suppose you wish to know what the political alignment of your Facebook friends are. You conduct a survey, randomly selecting 15 friends from your friends list and determining their political affiliation, with a "success" being a friend being the same political party as yours. You find that 10 of your friends have the same political views as you, and use this to test whether most of your friends agree with you. You use a significance level of $\alpha = .1$ to decide whether to reject the null hypothesis.

```
binom.test(10, 15, p = .5, alternative = "greater")
```

```
##
##  Exact binomial test
##
## data:  10 and 15
## number of successes = 10, number of trials = 15, p-value = 0.1509
## alternative hypothesis: true probability of success is greater than 0.5
## 95 percent confidence interval:
##  0.4225563 1.0000000
## sample estimates:
## probability of success
##              0.6666667
```

Since the p-value of 0.1509 is greater than the significance level, you fail to reject the null hypothesis.

Testing for Difference Between Population Means Using Paired Data

Suppose you have data sets from two populations X_i and Y_i, each possibly having their own mean, and you wish to test:

$$H_0 : \mu_X = \mu_Y \equiv \mu_X - \mu_Y = 0$$
$$H_A : \begin{cases} \mu_X < \mu_Y \equiv \mu_X - \mu_Y < 0 \\ \mu_X \neq \mu_Y \equiv \mu_X - \mu_Y \neq 0 \\ \mu_X > \mu_Y \equiv \mu_X - \mu_Y > 0 \end{cases}$$

Since the data is paired, testing these hypotheses is similar to inference with univariate data after you find $D_i = X_i - Y_i$ and replace $\mu_X - \mu_Y$ with μ_D.

If the differences D_i are Normally distributed, you can use the t-test to decide whether to reject H_0 (you would also use the t-test if your data was not Normally distributed but the sample size was large enough that little would change if a more exact test were used). In R, use `t.test()` setting the parameter `paired = TRUE`. By default the parameter `mu` is `0`, so you do not need to change it unless you want to test for a difference between the two populations other than zero. (In other words, you would be testing whether the samples differ by a certain amount as opposed to whether they differ at all.)

Below I test whether stronger speed limits reduce accidents, using the Swedish study we saw in the lecture on confidence intervals (in fact, you may see that most of the following code is identical to that lecture's code). I will reject H_0 if the p-value is less than 0.1.

```
library(MASS)
library(reshape)
head(Traffic)
```

```
##   year day limit  y
## 1 1961   1    no  9
## 2 1961   2    no 11
## 3 1961   3    no  9
## 4 1961   4    no 20
## 5 1961   5    no 31
## 6 1961   6    no 26
```

```
# The Traffic data set needs to be formed into a format that is more easily worked with. The following
new_Traffic <- with(Traffic, data.frame("year" = year, "day" = day, "limit" = as.character(limit), "y" =
melt_Traffic <- melt(new_Traffic, id.vars = c("year", "day"), measure.vars = c("limit", "y"))
form_Traffic <- cast(melt_Traffic, day ~ year + variable)
form_Traffic <- with(form_Traffic, data.frame("day" = day, "limit_61" = `1961_limit`, "accidents_61" = a
head(form_Traffic)
```

```
##   day limit_61 accidents_61 limit_62 accidents_62
## 1   1       no            9       no            9
## 2   2       no           11       no           20
## 3   3       no            9       no           15
## 4   4       no           20       no           14
## 5   5       no           31       no           30
## 6   6       no           26       no           23
```

```
# Now we subset this data so that only days where speed limits were enforced differently between the tw
diff_Traffic <- subset(form_Traffic, select = c("accidents_61", "accidents_62", "limit_61", "limit_62")
head(diff_Traffic)
```

```
##    accidents_61 accidents_62 limit_61 limit_62
## 11           29           17       no      yes
## 12           40           23       no      yes
## 13           28           16       no      yes
## 14           17           20       no      yes
## 15           15           13       no      yes
## 16           21           13       no      yes
```

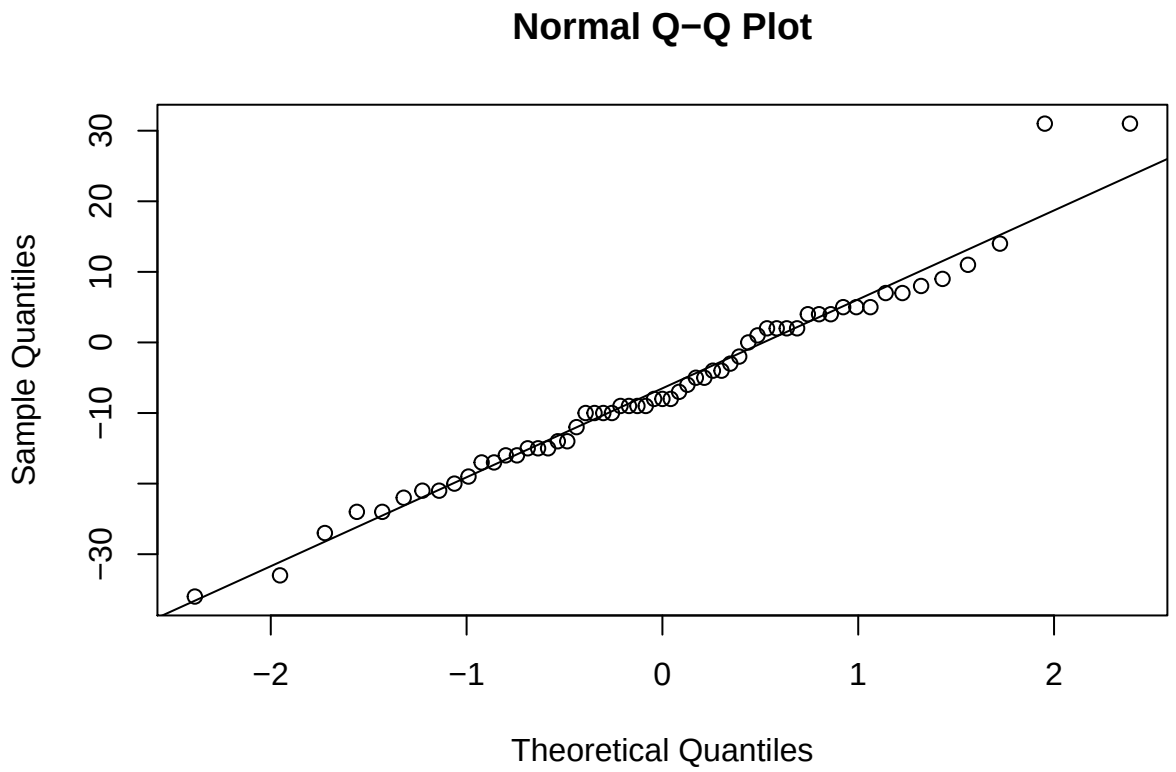
```
# Get a vector for accidents when the speed limit was not enforced, and a vector for accidents for when
accident_no_limit <- with(diff_Traffic, c(accidents_61[limit_61 == "no"], accidents_62[limit_62 == "no"]
accident_limit <- with(diff_Traffic, c(accidents_61[limit_61 == "yes"], accidents_62[limit_62 == "yes"])
accident_limit
```

```
##  [1] 19 19  9 21 22 23 14 19 15 13 22 42 29 21 12 16 17 23 16 20 13 13  9
## [24] 10 27 12  7 11 15 29 17 17 15 25  9 16 25 25 16 22 21 17 26 41 25 12
## [47] 17 24 26 16 15 12 22 24 16 25 14 15  9
```

```
accident_no_limit
```

```
##  [1] 29 40 28 17 15 21 24 15 32 22 24 11 27 37 32 25 20 40 21 18 35 21 25
## [24] 34 42 27 34 47 36 15 26 21 39 39 21 15 17 20 24 30 25  8 21 10 14 18
## [47] 26 38 31 12  8 22 17 31 49 23 14 25 24
```

```
# It took a lot of work, but finally we have the data in a format we want. Now, the first thing I check
qqnorm(accident_limit - accident_no_limit)
qqline(accident_limit - accident_no_limit)
```




```
# Aside from a couple observations, the Normal distribution seems to fit very well. t procedures are sa
t.test(accident_limit, accident_no_limit, paired = TRUE, alternative = "less")
```

```
##
##  Paired t-test
```

```
##
## data:  accident_limit and accident_no_limit
## t = -3.7744, df = 58, p-value = 0.0001897
## alternative hypothesis: true difference in means is less than 0
## 95 percent confidence interval:
##       -Inf -3.588289
## sample estimates:
## mean of the differences
##               -6.440678
```

With a p-value of 0.0002, we soundly reject H_0.

Test for Difference in Mean Between Populations With Independent Samples

Again, consider the set of hypotheses considered in the above section, only the data is not paired; we have two samples, $X_1, ..., X_n$ and $Y_1, ..., Y_m$ (I assume both are drawn from Normal distributions, so we may use t procedures). Our test statistic depends on whether we believe our populations are homoskedastic (common standard deviation σ) or heteroskedastic (possibly differing standard deviations σ_X and σ_Y) under the null hypothesis (whether they are homoskedastic or heteroskedastic if the alternative hypothesis is true doesn't matter). If we believe that, under H_0, the populations are homoskedastic, our test statistic will be:

$$t = \frac{\bar{x} - \bar{y}}{s\sqrt{\frac{1}{n} + \frac{1}{m}}}$$

where s is the pooled sample standard deviation. The degrees of freedom of the t distribution used to compute the p-value are $\nu = n + m - 2$.

If we do not assume that the populations are homoskedastic under H_0 (so we either believe they are heteroskedastic or that we have no reason to believe they are homoskedastic and thus assume heteroskedasticity by default), our test statistic will be:

$$t = \frac{\bar{x} - \bar{y}}{\sqrt{\frac{s_x^2}{n} + \frac{s_y^2}{m}}}$$

The t distribution is used as an approximation of the true distribution this test statistic follows under H_0, with degrees of freedom:

$$\nu = \frac{\left(\frac{s_X^2}{n} + \frac{s_Y^2}{m}\right)^2}{\frac{\left(s_X^2/n\right)^2}{n-1} + \frac{\left(s_Y^2/m\right)^2}{m-1}}$$

`t.test()` allows for testing these hypotheses. There is no need to set `paired = FALSE`, and you only set `var.equal = TRUE` if you want a test assuming that the populations are homoskedastic under H_0. (If this assumption holds, the homoskedastic test is more powerful than the heteroskedastic test, but the difference is only minor.)

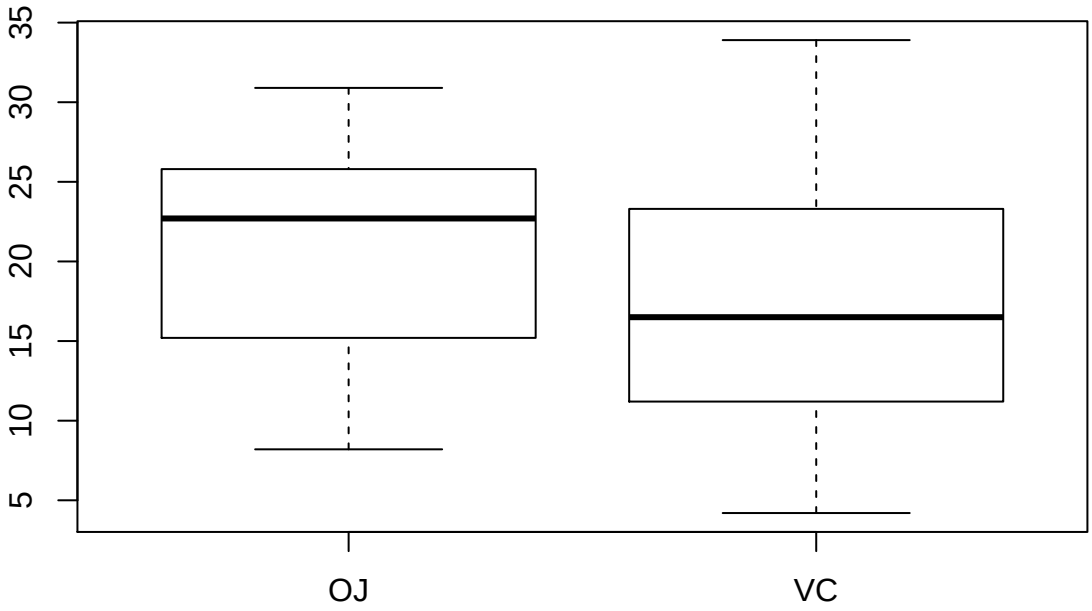
Let's test whether orange juice increases the tooth growth of guinea pigs, using the `ToothGrowth` data set. I use a significance level of $\alpha = .05$ to decide whether to reject H_0.

```
# First, let's get the data into separate vectors
split_len <- split(ToothGrowth$len, ToothGrowth$supp)
OJ <- split_len$OJ
VC <- split_len$VC
```

```r
# Perform statistical test
t.test(OJ, VC, alternative = "greater")
```

```
##
##  Welch Two Sample t-test
##
## data:  OJ and VC
## t = 1.9153, df = 55.309, p-value = 0.03032
## alternative hypothesis: true difference in means is greater than 0
## 95 percent confidence interval:
##  0.4682687       Inf
## sample estimates:
## mean of x mean of y
##  20.66333  16.96333
```

```r
# Is there homoskedasticity? Let's check a boxplot
boxplot(len ~ supp, data = ToothGrowth)
```




```r
# These populations look homoskedastic; spread does not differ drastically
# between the two. A test that assumes homoskedasticity is thus performed.
t.test(OJ, VC, var.equal = TRUE, alternative = "greater")
```

```
##
##  Two Sample t-test
##
## data:  OJ and VC
## t = 1.9153, df = 58, p-value = 0.0302
## alternative hypothesis: true difference in means is greater than 0
## 95 percent confidence interval:
##  0.4708204       Inf
## sample estimates:
## mean of x mean of y
##  20.66333  16.96333
```

```r
# Not much changed.
```

In both versions of the test, I reject the null hypothesis.

Testing for Difference in Population Proportion

Suppose you have two data sets of Bernoulli random variables, $X_1, ..., X_n$ and $Y_1, ..., Y_m$, and you wish to know if the population proportion of "successes" for the first population, p_X, differs from the corresponding proportion in the second population, p_Y. Your set of hypotheses are:

$$H_0 : p_X = p_Y \equiv p_X - p_Y = 0$$

$$H_A : \begin{cases} p_X < p_Y \equiv p_X - p_Y < 0 \\ p_X \neq p_Y \equiv p_X - p_Y \neq 0 \\ p_X > p_Y \equiv p_X - p_Y > 0 \end{cases}$$

If m and n are reasonably large, the Central Limit Theorem can be used to obtain a test statistic:

$$z = \frac{\hat{p}_X - \hat{p}_Y}{\sqrt{\hat{p}(1-\hat{p})\left(\frac{1}{n} + \frac{1}{m}\right)}}$$

where $\hat{p}$ is the pooled sample proportion (that is, the proportion of "successes" from the *combined* sample). Under H_0, this test statistic follows a standard Normal distribution (when applying the Central Limit Theorem).

In R, `prop.test()` can be used to conduct this test, using a call similar to `prop.test(x, n)` where `x` is a vector containing the number of successes in the samples, and `n` is the size of both samples.

Below, I use `prop.test()` and the `Melanoma` data set to test whether males and females are equally likely to die from melanoma after being diagnosed, or whether they differ. I use a significance level of $\alpha = .05$.

```
# First, I obtain sample sizes, using the fact that sex == 1 for males and 0
# for females
n <- c(Female = nrow(Melanoma) - sum(Melanoma$sex), Male = sum(Melanoma$sex))
# Now, find how many in each group died from melanoma
x <- aggregate(Melanoma$status == 1, list(Melanoma$sex), sum)
x <- c(Female = x[1, "x"], Male = x[2, "x"])
prop.test(x, n)
```

```
##
##  2-sample test for equality of proportions with continuity
##  correction
##
## data:  x out of n
## X-squared = 4.3803, df = 1, p-value = 0.03636
## alternative hypothesis: two.sided
## 95 percent confidence interval:
##  -0.283876633 -0.005856138
## sample estimates:
##    prop 1    prop 2
## 0.2222222 0.3670886
```

With a p-value of 0.0364, I reject the null hypothesis; men and women are not equally likely to die from melanoma.

Power Analysis

Power analysis is an important part of planning a statistical study. Researchers usually try to make a test as powerful as reasonable (there is always a more powerful study than the one currently employed; simply use a sample size of $n + 1$ rather than n). A useful technique in planning a study is to decide what effect size the test should be able to detect with some specified probability, then choose a sample size that will give this property to the test.

Whole R packages are devoted to providing tools for study planning, but the **stats** package included with any R installation has some function for power analysis, including those for the two classes of tests we have studied: tests for population mean, and tests for population proportion.

Power Analysis for the t-Test

`power.t.test()` allows you to perform power analysis for the t-test. It can be used in different ways depending on which parameters are passed to it and which are set to `NULL`, and you are encouraged to read the documentation (with, say, `help("power.t.test")`) to see how this function behaves, but I will focus on two applications: computing power for a test, and computing a sample size for a given power.

`power.t.test(n, delta, sd, sig.level, type = "some.type.of.test", alternative = "some.alternative")` will compute the power of a test with sample size `n`, where the difference between the mean under the null hypothesis and the mean under the alternative hypothesis, $\mu_0 - \mu_A$, is `delta`, the population standard deviation is `sd`, and the significance level is `sig.level` (by default, `sig.level = .05`). The type of the test administered is specified by `type`, and can be either `"one.sample"`, `"two.sample"`, or `"paired"` (the meaning should be self-evident). `alternative` specifies whether the alternative hypothesis is one-sided or two-sided. Notice that if `alternative = "two.sided"`, `delta` will be perceived as also communicating which direction the difference between μ_0 and μ_A occurs, so if you want the power to include the probability of rejecting in the opposite direction as well, you should set the parameter `strict` to `TRUE` (by default, `strict = FALSE`).

Suppose you plan to conduct a study to determine whether a new drug would induce weight loss. You plan to give all study participants both the drug and a placebo (in random order, with neither the study participants or experiment staff knowing which treatment is the drug or placebo, thus helping combat bias), and measure the difference in weight loss when the two treatments are administered. Thus your hypotheses are:

$$H_0 : \mu_{\text{drug}} = \mu_{\text{placebo}}$$

$$H_A : \mu_{\text{drug}} > \mu_{\text{placebo}}$$

Your test will use a significance level of $\alpha = .01$. You believe that $\sigma = 20$ (you estimate high to be on the safe side). A researcher on staff suggests a sample size of 20. You are skeptical that a study with that sample size will be able to detect a five-pound difference in weight loss between the drug and the placebo, and thus compute $\pi(5)$, the power of the test when the true difference is 5 lbs.

```
power.t.test(n = 20, delta = 5, sd = 20, sig.level = 0.01, type = "paired",
    alternative = "one.sided")
```

```
##
##      Paired t test power calculation
##
##               n = 20
##           delta = 5
##              sd = 20
##       sig.level = 0.01
##           power = 0.09924502
```

```
##     alternative = one.sided
##
## NOTE: n is number of *pairs*, sd is std.dev. of *differences* within pairs
```

This study will only detect a five-pound difference about 10% of the time, which is too low for your liking. You want to find a sample size to guarantee detecting this difference with some higher probability. This may involve a much larger sample size.

`power.t.test(delta = d, sd = s, sig.level = alpha, power = p, type = "some.type.of.test", alternative = "some.alternative")` is similar in usage to the earlier command but instead of finding power, the sample size will be found such that the power of the test when the difference between μ_0 and μ_A is `d` is `power`. Thus this call is useful when planning a test and choosing an appropriate sample size for detecting a specified effect with some desired probability.

You have decided that you want the study to detect a five-pound difference in weight loss 90% of the time, and want to find a sample size that will give your test this property. You use R to find this sample size:

```
power.t.test(power = 0.9, delta = 5, sd = 20, sig.level = 0.01, type = "paired",
    alternative = "one.sided")
```

```
##
##      Paired t test power calculation
##
##               n = 210.9878
##           delta = 5
##              sd = 20
##       sig.level = 0.01
##           power = 0.9
##     alternative = one.sided
##
## NOTE: n is number of *pairs*, sd is std.dev. of *differences* within pairs
```

The results suggest that you need 211 study participants for your study to have the desired property.

Power Analysis for Tests of Proportion

`power.prop.test()` does for tests for population proportion what `power.t.test()` does for tests for population mean. The syntax is similar, except there is no parameter `type`, and `delta` is replaced with `p1` and `p2`, which specify the population proportions under the two hypotheses. (There is no need to specify `sd`, so clearly that is not a parameter either.)

Gallup polls often survey samples of 1500 adults. Suppose a Gallup poll asks individuals whether they support Hillary Clinton or Donald Trump for President, and the poll uses a significance level of $\alpha = .05$ (the default for `power.prop.test()`, thus allowing us to ignore the parameter `sig.level`). Suppose we wish to use the results of the poll to test:

$$H_0 : p = .5$$

$$H_A : p > .5$$

where p is the proportion of the population supporting Hillary Clinton. We would like to know if the Gallup poll can reasonably detect a 1% advantage for Clinton, and use `power.prop.test()` to detect this:

```
power.prop.test(n = 1500, p1 = .5, p2 = .51, alternative = "one.sided")
```

```
##
##      Two-sample comparison of proportions power calculation
```

```
##
##               n = 1500
##              p1 = 0.5
##              p2 = 0.51
##       sig.level = 0.05
##           power = 0.136286
##     alternative = one.sided
##
## NOTE: n is number in *each* group
```

The Gallup poll will detect this difference only 14% of the time. If we wanted to detect this advantage for Clinton 95% of the time, what sample size do we need? By specifying `power = .95` and omitting `n`, we can find the desired sample size.

```
power.prop.test(power = .95, p1 = .5, p2 = .51, alternative = "one.sided")
```

```
##
##      Two-sample comparison of proportions power calculation
##
##               n = 54102.75
##              p1 = 0.5
##              p2 = 0.51
##       sig.level = 0.05
##           power = 0.95
##     alternative = one.sided
##
## NOTE: n is number in *each* group
```

We would need a sample size of 54,103 people to have a test with these properties.

p-Hacking

p-hacking (also known as data dredging or other names) is the practice of reformulating hypotheses and research questions and computing p-values for the same data set until a statistically significant result is obtained. This is considered to be very bad practice, and results found by p-hacking are untrustworthy. The probability of a Type I error is inflated when p-hacking is used, yet nowhere is this communicated or adjusted for. You do not want to be accused of p-hacking.

When conducting a study, you should have a well-defined problem prior to any testing. Exploratory analysis with visualization, or deeper analysis on *small subsets* of the data (that you do not use later when testing) You should also report what you did. If you perform an analysis and do not get the results you expected and you are certain that the analysis was done correctly, or you simply are unsatisfied with the conclusion, you need to perform analysis on *new* data, not the same data. Furthermore, you should report any prior analyses. Keep in mind that if you perform lots of tests and end when you find a significant result, you may be p-hacking, especially if you do not report that this was done.

There are many do's and don't's with regards to how to properly conduct hypothesis testing and avoid p-hacking. Some rules of thumb would be to have a well-defined plan and research question prior to analysis and keep statistical testing critical to the research question at a minimum. The website FiveThirtyEight has made a web app where you can see what p-hacking is and why it cannot be trusted.

Conclusion

This concludes the MATH 3070 R lab. The MATH 3080 R lab will resume where this lab left off, focusing on using R for the statistical problems encountered in MATH 3080 (ANOVA, linear regression, nonparametric tests, and other topics).

Curtis Miller has written a blog post with MATH 3070 students learning R in mind for ways to improve your R skills in particular (though his post applies to learning programming in general). We require this course because we believe that programming skills are essential to any career involving statistics. The learning process *never* stops. Consider reading the article to know where to go from here.