Intro to R for Psychologists

Steven W. Nydick

Contents

Part I

The Basics of R

Chapter 1
Introduction to R

Don't call me señor! I'm not a Spanish person. You must call me Mr. Biggles, or Group Captain Biggles, or Mary Biggles if I'm dressed as my wife, but never señor.
—Monty Python's Flying Circus - Episode 33

1.1 Using R

This chapter goes through the basics of using `R`. It is in your best interest to type *all* of the `R` code yourself, on your own. Moreover, it behooves you to make sure that you understand every line of code. Example code will be italicized and placed on its own special line next to a sideways caret, like the following:

```
> x <- 2
```

Some of the code will not be evaluated, so you must type it into your own `R` session to figure out what it does. Never type the caret (`>`) into `R` – it is just to let you know the contents of a code chunk. Furthermore, sometimes I will present "code" only intended to diagram certain functions or procedures without the appropriate syntactical concerns. Any "pseudo-code" or code that I do not indent for you to run as is, I will display as red and boldface. For instance, the following should not be evaluated in your `R` session, as it will return an error.

```
Code(More Code, And More)
```

The first thing that you want to do is download and install `R`, from the following website (for free on any system):

http://cran.r-project.org/

And download one of the *nicer* GUIs in which to run `R`. Either:

1. Tinn-R (Windows only)

http://sourceforge.net/projects/tinn-r/

2. R-Studio (for all platforms)

http://rstudio.org/download/

To configure Tinn-R, go to `R/Configure/Permanent` in the file headings. Once you have configured `R` to work with Tinn-R or R-Studio, opening those programs will automatically start an `R` session. You can also click on the `R` icon itself or (if you are on a Mac or Linux machine) type `R` into the command line. Either way, one you start `R`, you should see a window with the following:

```
R version 2.14.2 (2012-02-29)
Copyright (C) 2012 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[R.app GUI 1.50 (6126) x86_64-apple-darwin9.8.0]

[Workspace restored from /Users/stevennydick/.RData]
[History restored from /Users/stevennydick/.Rhistory]

>
```

That is the window in which you should be typing things (hopefully useful things) for your data analysis. In this case, if I type the simple expression:

```
> 2 + 2
[1] 4
```

you should enter it into that box, and the result will be:

```
R version 2.14.2 (2012-02-29)
Copyright (C) 2012 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[R.app GUI 1.50 (6126) x86_64-apple-darwin9.8.0]

[Workspace restored from /Users/stevennydick/.RData]
[History restored from /Users/stevennydick/.Rhistory]

> 2 + 2
[1] 4
>
```

Magic, isn't it?! I certainly think so.

So, I bet (either by this point in the day or out of sheer frustration) that you want to know what to *do* with the command prompt (i.e., `>`). The first thing that you could do is write a comment - something that helps you understand the code but which is not parsed by the machine. Comments always follow the pound (`#`) sign. Copy the following code, and you will see that nothing happens:

```
> # This is the most pointless code ever!  Seriously!  Evvver!
> # I sometimes do like to use R to talk to myself ... although
> # it usually doesn't really want to talk back.  Hmm...
```

Comments *always* end at the end of the line. Anything *before* `#` (the comment) on a given line will be parsed by the machine, anything *after* `#` will *not* be parsed, but you need a `#` on the next line to continue the comment.

The second thing that you could do is write part of code on one line but screw up and not finish it. For example, I could try to find the square-root of 4 but push enter without closing the parentheses.

```
> sqrt(4
+ )
[1] 2
```

Rather than giving me a new command prompt (`>`), `R` gives me the continuation command prompt (`+`) telling me that whatever I type on that line is going to be evaluated *with* the previous code. It's a ... continuation of the previous line. However, the comment line (`#`) and continuation prompt (`+`) are not really that interesting. There are better uses for `R`.

1.2 R as a Calculator

The most straightforward use of R is as a high-powered calculator. Most of the normal calculator functions work in R:

```
> 2 + 3  # adds two numbers
[1] 5
> 2 - 3  # subtracts one number from the other
[1] -1
> 2 * 3  # multiplies two numbers
[1] 6
> 2 / 3  # divides one number by the other
[1] 0.6666667
> 2 ^ 3  # exponentiates one number by the other
[1] 8
```

In general, white space does not matter in R, so that

```
> 1 + 2 # breathing room
[1] 3
> 1+2   # oh ugh-ness!
[1] 3
```

give the same result.

R abides by the order-of-operations (... hmm ... "please excuse my dear aunt sally"), so that

```
> 1 + 2 * 3 ^ 2
[1] 19
```

squares the 3 first, then multiplies that result by 2, and then adds 1, so that the result is 19. If you want to do the arithmetic sequentially, you can include parentheses:

```
> ( (1 + 2) * 3 ) ^ 2
[1] 81
```

which adds 1 to 2, then multiplies that by 3, then squares the whole thing, so that the result is 81. Notice that after you type one of these commands into R, the first thing on the next line is `[1]`. We will get to *why* the magic `[1]` appears shortly.

1.3 Function Basics

R is a function-oriented, interpretive programming language. It's interpretive because the R machine (or big, bad, hungry R monster) evaluates every line of code immediately after you press "return." It is function-oriented because you do everything through *functions*. In fact, those simple commands that you typed earlier (`+` or `-` or any other arithmetic thingy) are really functions that are put into a more convenient form. The form of most functions is as follows:

```
name(arg1, arg2, ... )
```

They start out with a *name* (the thing you call) followed by a bunch of arguments. Many familiar things are functions. For instance, the `sqrt` from before is a function (the square-root function). There is also:

```
> sqrt(7)  # the square-root of 2
[1] 2.645751
> sin(pi)  # the sin = o/h of pi
[1] 1.224647e-16
> exp(1)   # the exponent of 1 (i.e. 2.72^1)
[1] 2.718282
> log(10)  # the natural log of 10
[1] 2.302585
```

Notice a few things. First, `pi` is predefined to be 3.14... - one does not have to define it him/herself. Second, `sin(pi)` *should be* 0 but actually shows up as `1.224647e-16` (read 1.22 times 10 to the negative 16 – ugh scientific notation). The reason for this is because of computer calculation error.

Most functions have *extra arguments* so that we can change the default behavior. For instance:

```
> log(10)
[1] 2.302585
```

calculates the *natural log* of 10, which is to the base e (if you remember your algebra). `R` allows us to change the base to 2, just by altering the second argument:

```
> log(10, base = 2)
[1] 3.321928
```

"Base"-ically, what we are saying is "take the log of 10, but we want the base argument to be 2." There are two additional methods of altering the base argument. First, all of the functions have an *order* to their arguments, e.g.:

```
log(x, base)
```

so the first argument is the thing we want to take the logarithm of, and the second argument is the base. If we do not use the *name* of the argument followed by an equal sign (e.g., `base=10`), then `R` will try to assign function values in order. So, for instance, if we type:

```
> log(10, 2)
[1] 3.321928
```

`R` will think that 10 is the thing we want to evaluate and 2 is the base.

Second, we could use the name of the argument but only the first couple of letters until the argument is uniquely identified. Because the two arguments are `x` and `base`, if

we type `b` rather than `base`, `R` will *know* that we mean to modify the `base` argument. With this in mind, all of the following should give the same result:

```
> log(10, 2)
[1] 3.321928
> log(10, base = 2)
[1] 3.321928
> log(10, b = 2)
[1] 3.321928
> log(base = 2, x = 10)
[1] 3.321928
> log(b = 2, x = 10)
[1] 3.321928
```

Strangely, even things that don't look like functions in the standard sense (e.g., `+`) are really functions. Knowing that a function is defined by a name followed by parentheses and arguments, the `+` function is *really* the following:

```
> "+"(2, 3)
[1] 5
```

which (as you would suppose) adds 2 to 3. But nobody uses the `"+"` function this way – they added in the standard, calculator sense and let `R` figure out which function to call.

1.4 Warnings and Errors

Unlike SPSS and other data analysis programs, the user can type something into `R` that it does not recognize or cannot work. If you do this `R` will give you an error. Most of your time (unfortunately) will be spent trying to *avoid* getting errors. Yet if your code works, you probably did the correct thing. Errors could happen for the following reasons:

```
> # The "try" function tests whether the code inside of it will evaluate.
> try(logarithm(2), silent = TRUE)[1]  # a function that does not exist
[1] "Error in try(logarithm(2), silent = TRUE) : \n  could not find
> try(log(2, e = 2), silent = TRUE)[1] # an argument that does not exist
[1] "Error in log(2, e = 2) : unused argument(s) (e = 2)\n"
> try(log 2, silent = TRUE)[1]         # forgetting the parentheses
[1] "Error: unexpected numeric constant in "try(log 2"
```

If you get an error, check to make sure that your code is correct and that your arguments make sense, etc.

Sometimes, `R` will evaluate code but give you a warning. For instance, if you try to take the logarithm of -2 (which is not defined), `R` will tell you that the result is not a number (`NaN`) and that you probably meant to do something else.

```
> log(-2)
[1] NaN
```

There are *four* weird values that you should be wary of if you see them in `R`. They could completely screw up subsequent code.

1. `NaN` (not a number; 0/0, sqrt(-2), log(-2))
2. `NA` (not applicable)
3. `Inf` (infinity)
4. `-Inf` (negative infinity)

All of the above have predefined values, and you can use them in expressions:

```
> 2 + NaN
[1] NaN
```

but I cannot figure out any reason you would ever want to do that.

1.5 Assignment

Usually, we need to give a particular value (or any other *object* that we will talk about soon) a name. There are *four* typical ways to assign values to names (as well as the more general `assign` function that is a bit beyond the scope of this book) - two of which make sense (and you should use), and two of which are used only by advanced `R` people. The most common assignment operators are the `<-` (less-than followed by a dash) and `=` (equals). The act of assignment *does not* produce any output. So, if you run:

```
> x <- 3  # assignment via the arrow
```

or:

```
> y = 3  # assignment via the equals
```

`R` does not tell you that it did anything. But if you type `x` or `y`, you will see the value:

```
> x
[1] 3
> y
[1] 3
```

You can also assign via the right arrow and the double arrow.

```
> 3 -> z
> z
[1] 3
> g <<- 5
> g
[1] 5
```

Basically `->` does the same thing as `<-` except with the value and name switched (and looks really weird). The double arrow `<<-` is much more complicated and only used by programmers, but effectively means: "I really really really want you to do this assignment."

The following are rules for how you should name variables:

1. They must start with a letter or dot (.)
    - Never start the names with a dot (.) - they're reserved for *important* things.
2. They *can* include letters, dots, underscores (_).
3. They *cannot* include space or mathematical characters (+, -, *, /)
4. They are case sensitive (`Abc` is different from `abc`).
5. There are certain names reserved for `R`, which you should not use.

Once we have assigned a value to a name, we can use functions on the name and `R` will act as though we used the function on the value. So rather than writing:

```
> log(10, base = 2)  # what we did before
[1] 3.321928
```

we can write:

```
> val <- 10
> base <- 2
> log(x = val, base = base)  # fancy dancy!
[1] 3.321928
```

and `R` will do the same thing. Note that we can call values by names that are arguments to functions *and not* affect the functionality of those functions.

1.6 Vectors in R

1.6.1 The Basics of Vectors

So, to review. In `R`, we can: perform arithmetic, follow the order-of-operations, use functions on values, assign values to names, and then use functions on those names rather than the values themselves. However, one might wonder:

> How does this help us do statistics? Data are usually more than one number, and if we always had to type every number in individually, `R` would not be very useful.

Luckily we can combine values into *vectors* of observations. So if we have a variable (say IQ scores) that has the values: 100, 125, 143, 165, 172, 110, 95 ... for particular people, we can combine those values using the *concatenate* function: `c`. This is one of the most important functions you will use in `R`.

```
> IQ.scores <- c(100, 125, 143, 165, 172, 110, 95)
```

Note that the arguments for `c` are `...`, which essentially means that we can enter an unlimited number of values into `c` and `R` will form a vector of all of them.

So, if we look at `IQ.scores`:

```
> IQ.scores
```

```
[1] 100 125 143 165 172 110  95
```

R will print out all of the values in our vector separated by spaces. We can also combine two vectors into a much bigger vector.

```
> v1 <- c(1, 2, 3, 4, 5)  # five happy scores
> v2 <- c(6, 7, 8, 9, 10) # five more happy scores
> vBig <- c(v1, v2)       # combine v1 and v2
> vBig                    # what are those ten scores
 [1]  1  2  3  4  5  6  7  8  9 10
```

We can make vectors of *more than* just numbers. There are several major types of vectors, three of which we will talk about now.

1. We can have vectors of numbers:

```
> v.numb <- c(1, 2, 3, 4, 5, 6)
> v.numb
[1] 1 2 3 4 5 6
```

2. We can have vectors of character strings (which are numbers or words contained in single or double quotes):

```
> v.char <- c("terry", "terry", "john", "michael", "graham", "eric")
> v.char
[1] "terry"   "terry"   "john"    "michael" "graham"
[6] "eric"
```

3. We can have vectors of logicals (true and false):

```
> v.logi <- c(TRUE, TRUE, FALSE, FALSE, TRUE)
> v.logi
[1]  TRUE  TRUE FALSE FALSE  TRUE
```

The function `mode` allows us to figure out whether our vector is a numerical, character, or logical vector.

```
> mode(v.numb)
[1] "numeric"
> mode(v.char)
[1] "character"
> mode(v.logi)
[1] "logical"
```

and if we mix two vectors, R will turn the combined vector into the most general of the bunch using the following rules:

Specific: logical --> numeric --> character :General

So, if we combine a logical vector with a numeric vector, R will turn the trues and falses into 1s and 0s.

```
> v.what <- c(v.logi, v.numb)
> v.what
 [1] 1 1 0 0 1 1 2 3 4 5 6
```

and if we combine a numeric vector with a character vector, R will surround the numbers with quotes.

```
> v.what2 <- c(v.numb, v.char)
> v.what2
 [1] "1"       "2"       "3"       "4"       "5"
 [6] "6"       "terry"   "terry"   "john"    "michael"
[11] "graham"  "eric"
```

> **Warning:** Mixing vector of two different types will frequently turn the resultant vector into a character vector, and arithmetic operations will consequently not work.

If desired, we can also *name* elements of a vector. It is a little odd how you do this. The `names` function is used on the left side of the assignment operator (`<-`) to assign a character vector as the names of another vector. But we can also retrieve the names of the character vector by using the `names` function by itself. For example:

```
> # Assign values to v.numb:
> v.numb <- c(1, 2, 3, 4, 5, 6)
> # Assign names to the elements of v.numb:
> names(v.numb) <- c("terry", "terry", "john", "michael", "graham", "eric")
```

assigns lovely names to the values 1 – 6. And if we type `v.numb`, we will see the values (as always), but we will also see the names above those values (not surrounded by quotes):

```
> v.numb
  terry   terry    john michael  graham    eric
      1       2       3       4       5       6
```

and then if we type the following:

```
> names(v.numb)
[1] "terry"   "terry"   "john"    "michael" "graham"
[6] "eric"
```

we retrieve the names as their own character vector:

```
> mode( names(v.numb) ) # the names are a character vector.
```

```
[1] "character"
```

Note: We can assign names to character, numeric, and logical vectors. And when we perform operations on those vectors (in the next section) the names will not interfere with those operations.

At this point, it might be helpful to reveal the meaning of that silly little `[1]` sign is on each line of your `R` code. A `[j]` indicates you the vector position (in this case, `j`) of the element directly to the right of it. If you only type a few elements, and they do not spill over to the next line, the vector element directly to the right of the first line is a `[1]` (it's the first element of the vector). But if you type lots of elements, you'll see that the row numbers correspond to the particular place of the element lying next to it. Just type in the following code (without the `>` and `+` of course), and count the elements if you do not trust me.

```
> c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
+   11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
+   21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
+   31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
+   41, 42, 43, 44, 45, 46, 47, 48, 49, 50)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
[19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
[37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

1.6.2 Operations on Vectors

`R` has two types of function operations:

1. Functions on the entire vector, and
2. Vectorized functions.

Usefully, `R` usually knows what you *want* to do (because logically awesome people programmed it). Functions on an entire vector take a vector (usually of numbers) and find useful things about it. For instance, we can calculate the "mean" of a set of numbers by using the `mean` function on a number vector.

```
> IQ.scores  <- c(100, 125, 143, 165, 172, 110, 95)
> mean(IQ.scores)   # the mean of the IQ scores
[1] 130
```

Alternatively, we can use other functions to calculate the mean a slightly more complicated way:

```
> length(IQ.scores)                # the number of IQ scores
[1] 7
> sum(IQ.scores)                   # the sum of the IQ scores
[1] 910
```

```
> sum(IQ.scores)/length(IQ.scores) # the mean another way!
[1] 130
```

I will talk about more of these functions in the upcoming chapters. Just note that they do exist, most are named what you would expect, and they are awesome.

Vectorized functions perform the operation on each value of a particular vector. For instance, the square-root, logarithm, and exponent functions will not act on the entire vector (whatever that means) but will act on each element separately.

```
> log(IQ.scores)  # the log of EACH
[1] 4.605170 4.828314 4.962845 5.105945 5.147494 4.700480
[7] 4.553877
> exp(IQ.scores)  # the exponent of EACH
[1] 2.688117e+43 1.935576e+54 1.270899e+62 4.556061e+71
[5] 4.996327e+74 5.920972e+47 1.811239e+41
> sqrt(IQ.scores) # the square-root of EACH
[1] 10.000000 11.180340 11.958261 12.845233 13.114877
[6] 10.488088  9.746794
```

Vectorization becomes really helpful when we want to combine information in two vectors. So let's say we have before and after weight scores and we want to calculate the difference between all of them. Then we only have to subtract the vectors:

```
> befor <- c(110, 200, 240, 230, 210, 150) # before dieting?
> after <- c(110, 190, 200, 220, 200, 140) # after dieting?
> befor - after  # the difference :)
[1]  0 10 40 10 10 10
```

We can also (element-wise) perform multiplication, division, and powers.

```
> befor + after # adding the weights together?
[1] 220 390 440 450 410 290
> befor * after # multiplying the weights?
[1] 12100 38000 48000 50600 42000 21000
> befor / after # dividing?  this really doesn't make sense, you know.
[1] 1.000000 1.052632 1.200000 1.045455 1.050000 1.071429
> befor ^ after # powers?  Steve, you've lost your mind!
[1] 3.574336e+224           Inf           Inf           Inf
[5]           Inf 4.495482e+304
```

Notice that `befor^after` resulted in many `Inf` values, only meaning that the numbers were too large for the computer to handle. They *exploded* will evaluating fun!

Vectorization also allows us to add a number to a vector. Wtf?! If we add a number to a vector, `R` knows that we want to add that number to every value of the vector and *not just* to one of the numbers.

```
> befor
[1] 110 200 240 230 210 150
```

```
> befor + 2
[1] 112 202 242 232 212 152
```

Because of vectorization, we can calculate the sample variance with ease.

```
> x <- c(2, 5, 4, 2, 9, 10)
> xbar <- mean(x)      # the mean of the scores
> N <- length(x)       # the number of scores
> x.dev  <- x - xbar   # deviation scores
> s.dev  <- x.dev ^ 2  # squared deviation scores
> ss.dev <- sum(s.dev) # sum of squared deviation scores
> ss.dev / (N - 1)     # the sample variance.
[1] 11.86667
```

And we can also combine many of the vectorized functions to calculate the sample variance in one step.

```
> sum( ( x - mean(x) )^2 ) / (length(x) - 1)
[1] 11.86667
```

In the above code chunk, we used a few functions on vectors (sum, mean, and length) and a few vectorizations (subtracting a number from a vector, squaring a vector). We will learn more later.

1.6.3 Other Ways to Create Vectors

There are several other ways to create vectors. The first method is the simple `scan` command. If you type `scan()` followed by "return," every element that you type after that (separated by spaces) will be attached to the vector. You can complete the vector by pushing "return" twice.

```
> x <- scan()
1: 1 2 3 4 5
6:
Read 5 items
> x
```

You can also create so-called "structured" vectors. The most common structured vectors are repeated values and sequences. You can create integer sequences using the `:` command in `R`.

```
> 1:10  # a vector of integers from 1 to 10
 [1]  1  2  3  4  5  6  7  8  9 10
```

The `:` command also works in reverse.

```
> 10:1  # a vector of integers from 10 to 1
 [1] 10  9  8  7  6  5  4  3  2  1
```

Or you can use the `rev` function on the original vector.

```
> rev(1:10)  # reverse the order of the 1, 2, 3, ... vector
 [1] 10  9  8  7  6  5  4  3  2  1
```

You can also create sequences (with more options) using the `seq` function. It has two forms, either:

```
seq(from, to, by)
```

or:

```
seq(from, to, length.out)
```

For instance, if you want to create a vector of the odd numbers between 1 and 9, you would type:

```
> # Every other number from 1 to 9
> seq(from = 1, to = 9, by = 2)
[1] 1 3 5 7 9
```

The *by* argument makes sure that `R` only chooses every other number to include in the vector. You can also use the `length.out` argument and `R` will automatically figure out the step-size needed to create a vector of that size.

```
> seq(from = 1, to = 9, length.out = 5) # ooh
[1] 1 3 5 7 9
> seq(from = 9, to = 1, length.out = 3) # ahh
[1] 9 5 1
> seq(from = 1, to = 2, length.out = 5) # magic
[1] 1.00 1.25 1.50 1.75 2.00
```

Another pattern vector that you might want to use is `rep`, which has one of the following forms:

```
rep(x, times)
```

Or:

```
rep(x, each)
```

And `R` will create a vector of many `x`'s stacked behind each other. We can repeat scalars:

```
> rep(1, times = 10)
 [1] 1 1 1 1 1 1 1 1 1 1
```

or numeric vectors:

```
> rep(1:3, times = 5)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

or even character vectors:

```
> rep(c("bob", "john", "jim"), times = 2)
[1] "bob"  "john" "jim"  "bob"  "john" "jim"
```

And if we use the `each` argument, rather than the `times` argument, `R` will repeat the first element `each` times before moving to the second element. For instance

```
> rep(1:3, each = 5)
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
```

now repeats "1" 5 times before starting to repeat "2".

1.7 Getting Help

Because `R` has a plethora of functions, arguments, syntax, etc., one might need help with figuring out the correction function to use and how to use it. There are a couple ways of getting help. The easiest thing to do is to type your function or question into google (along with the letter `R`) and see what pops up. For example, typing:

How do I calculate a mean in R?

will pull up pages, some of which might be useful, some of which might not be useful, and some of which might be ... umm ... mean, actually.

There are two ways of getting help directly through `R`.

1. If you know the function you want to use, you can type `help` followed by the function surrounded by quotes and parentheses:

   ```
   > help("mean")
   ```

   or, equivalently, you can type the function preceded by a question mark:

   ```
   > ?mean
   ```

2. If you do not know the function you want to use, you can `help.search` followed by the function surrounded by quotes and parentheses:

   ```
   > help.search("normal")  # what functions work with the normal dist?
   >                        # ugh - there are lots of them!?mean
   ```

1.8 In the Following Chapters

The following chapters will discuss much more of the nuts and bolts of `R`, including:

1. Building data frames and matrices from vectors.
2. Accessing individual elements of vectors.
3. Putting a specific value in a particular place of an already-existing vector.

- This goes by the phrase: "working with indices."

4. Creating logical values by testing vector elements.
5. Using logical values/vectors to help organize data.
6. Reading in data from external sources.
7. Using and installing `R` packages.
8. Much of statistics.

This probably seems overwhelming at the moment. `R` knowledge requires quite a bit of practice -- and there is a lot of material left before you are familiar enough with the program to do much without ripping your hair out. But the chapters will be written piece by piece, so that each topic builds on the last and is easily applied to statistical procedures.

1.9 Appendix: Functions Used

Here is a list of important functions from this chapter:

```
# Calculation
+
-
*
/
^

# Assignment
<-
=

# Basic Functions
sqrt(x)
sin(x)
exp(x)
log(x, base)

# Vectors
c(...)             # the concatenate function - important!
names(x)           # assign names OR extract names.
scan()             # create a vector by hand.
n:m                # integers from n to m (both numbers)
rev(x)             # reverse the order of a vector
seq(from, to, by)  # create a sequence
seq(from, to, length.out)
rep(x, times)      # repeat x some number of times
mode(x)            # what type of vector do we have?

# Operations on Vectors
mean(x, na.rm)     # don't worry about na.rm yet.
length(x)
sum(x, na.rm)      # don't worry about na.rm yet.

# Help
help(topic)          # if you know the function
?function            # identical to "help"
help.search(keyword) # if you only know a keyword
```

Chapter 2
Dataframes and Indices

Well last week, we showed you how to become a gynaecologist. And this week on 'How to Do It' we're going to show you how to play the flute, how to split an atom, how to construct a box girder bridge, how to irrigate the Sahara Desert and make vast new areas of land cultivatable, but first, here's Jackie to tell you all how to rid the world of all known diseases.
—Monty Python's Flying Circus - Episode 28

2.1 Packages

One of the benefits (and curses) of `R` is that it is designed with few built in functions but many (frequently user contributed) external packages. One day you might be able to write a package and help some sad data analyst in a time of need (*sigh*). Unfortunately, much of your time will be spent realizing that regular `R` cannot do something important and you need to install or load an external package to use a particular function. If the package is already downloaded (or comes with `R`) you can load it with the simple command:

```
library(pkg)
```

or ...

```
require(pkg)
```

I am not exactly sure the difference between `library` and `require`, but I tend to use `library` for standard use and `require` only in personally written functions (which we will cover much later). Some functions in one package conflict with functions in another package (i.e., they have the same name but do different things or work with different types of data), so `R` loads only the standard packages (e.g., `base`, which has most of the stuff we talked about last week, and `stats`, which includes – not surprisingly – basic statistical stuff).

The function `mvrnorm` is in the `MASS` package, which comes with the standard distribution but is not automatically loaded. We will not worry about details of the function

`mvrnorm` at the moment, but the main idea is that `mvrnorm` creates a dataset with vectors that correlate a specified amount. If you try to run the function `mvrnorm` or ask for help (about the function) without loading the `MASS` package, `R` will yell at you (very loudly and possibly even in red) that the object does not exist and there are no help pages:

```
> mvrnorm
> ?mvrnorm
```

But if you load the package first, then `R` will remember that the function exists ... `R` sometimes has very poor short term memory.

```
> library(MASS) # load the package first
> ?mvrnorm      # now the help file should show up
>               # and the function will work.
```

You can load any package with or without quotes, whichever you think is easier. There is a way of getting at the help file of a function without loading its package, by using `help` (and not `?`), and by adding the name of the package as an argument:

```
> # function name followed by package name (with or without quotes)
> help("read.spss", package = "foreign")
```

but it is usually easier (and less taxing on already compromised brain cells) to load the packages you want to use before trying to do things with their functions. Plus, packages are people too - and they like to be remembered for their "package-like" nature.

If you know of a package that you have on your computer but either do not know of functions in that package or do not remember the name of a function you need, you can do one of two things. The command:

```
library(help = pkg)
```

will give you a brief rundown of a package and its functions. For instance, by typing:

```
> library(help = "base") # the notorious base package - ooh!
```

you can see that much of what was discussed in the last chapter is located there. However, the information on functions is minimal, so a better strategy would be to go into the "Package-Manager" in one of the menus to access the help files directly.

If your computer does not have a function that you need, this function might exist in a package that you have not downloaded yet (google is your friend). Type what you want the function to calculate by `R` into google, and click on some of the pages that pop up. Many of those pages will be message boards of other people who have had the same problems, and the advice (if it exists) might be helpful. Yet other pages will be `R` help files from packages that you do not have. You will be able to tell `R` help files by the format of the page (comparing it to help files you have seen) or the words "R Documentation" on the far right of the page. On the far left of the page, the help file will display the function name and then the package in curly braces. If you look at the help file of `read.spss` again

```
> help("read.spss", package = "foreign")
```

you can see that it comes from the package `foreign`.

If you need to install a package that is contained in CRAN (the "Comprehensive R Archive Network," where most packages live a long and fruitful life), you can either click on "Package Installer" in one of the menu items (and then follow the directions), or type:

```
install.packages(pkg, dependencies = TRUE)
```

into `R` where `pkg` is the package name (in quotes). For instance, let's say that we found a function in the package `psych` that we want to use (which might be useful for our psych-i-ness). Then, to use that function, type:

```
> # installing the package onto our machine:
> install.packages("psych", dependencies = TRUE)


> # loading the package, so we can use its functions and data:
> library(psych)
```

Many of the packages also contain datasets that we can use to do analyses (go into "Data Manager" in the "Packages & Data" menu if you ever want to feel overwhelmed). If we want to use a particular dataset (by name) you only need to load the package in which it is located. Of course, datasets can be pretty big, so loading all of them at once might take a while and eat up lots of memory.

So, let's say we want to use the `salary` dataset in the package `alr3` (which you should not have on your machine). First, you check the package:

```
> library(alr3)
```

It does not exist, so you install the package.

```
> install.packages("alr3", dependencies = TRUE)
```

Once you finish installing it, you can load the package.

```
> library(alr3)
```

and use the salary dataset just by typing its name.

```
> salary  # trying to see what is in the salary dataset
   Degree Rank Sex Year YSdeg Salary
1       1    3   0   25    35  36350
2       1    3   0   13    22  35350
3       1    3   0   10    23  28200
4       1    3   1    7    27  26775
5       0    3   0   19    30  33696
6       1    3   0   16    21  28516
7       0    3   1    0    32  24900
8       1    3   0   16    18  31909
9       0    3   0   13    30  31850
10      0    3   0   13    31  32850
11      1    3   0   12    22  27025
```

12	1	2	0	15	19	24750
13	1	3	0	9	17	28200
14	0	2	0	9	27	23712
15	1	3	0	9	24	25748
16	1	3	0	7	15	29342
17	1	3	0	13	20	31114
18	0	2	0	11	14	24742
19	0	2	0	10	15	22906
20	0	3	0	6	21	24450
21	0	1	0	16	23	19175
22	0	2	0	8	31	20525
23	1	3	0	7	13	27959
24	1	3	1	8	24	38045
25	1	2	0	9	12	24832
26	1	3	0	5	18	25400
27	1	2	0	11	14	24800
28	1	3	1	5	16	25500
29	0	2	0	3	7	26182
30	0	2	0	3	17	23725
31	0	1	1	10	15	21600
32	0	2	0	11	31	23300
33	0	1	0	9	14	23713
34	0	2	1	4	33	20690
35	0	2	1	6	29	22450
36	1	2	0	1	9	20850
37	1	1	1	8	14	18304
38	1	1	0	4	4	17095
39	1	1	0	4	5	16700
40	1	1	0	4	4	17600
41	1	1	0	3	4	18075
42	0	1	0	3	11	18000
43	1	2	0	0	7	20999
44	1	1	1	3	3	17250
45	1	1	0	2	3	16500
46	1	1	0	2	1	16094
47	1	1	1	2	6	16150
48	1	1	1	2	2	15350
49	1	1	0	1	1	16244
50	1	1	1	1	1	16686
51	1	1	1	1	1	15000
52	1	1	1	0	2	20300

In previous versions of `R`, typing the name of the dataset would not load a data frame automatically. You *first* needed to type the function `data` followed by the dataset name before you could use the data.

```
> data(salary)  # loading the dataset (in alr3)
> head(salary)  # seeing the contents of the dataset
```

```
  Degree Rank Sex Year YSdeg Salary
1      1    3   0   25    35  36350
2      1    3   0   13    22  35350
3      1    3   0   10    23  28200
4      1    3   1    7    27  26775
5      0    3   0   19    30  33696
6      1    3   0   16    21  28516
```

But they changed the data loading mechanism for the current version of R. However, the data function allows you to load the data from a package that is not currently loaded by typing in the package name as the second argument (in quotes):

```
> data(salary, package = "alr3")
```

Of course, the salary package is already loaded (as is the alr3 package), so that this extra step does not seem to add any data loading benefit. And in any case, the dataset salary is of a form that we have not seen before ... it has *many* columns, and we only talked about vectors. What are those weird vertical things next to our wonderful vector?

2.2 Dataframes in R

In the previous chapter, we built vectors of scores (or words or trues and falses) using the concatenate function (c), and then assigned those vectors to names using an assignment operator (<- or =). Well, data are usually made up of more than one vector – we are usually interested in the relationships (models) among lots of things, and we would like to keep all of those things in one place. R has two additional objects that take pointy things (vectors) and combines them to make rectangulary things. They are called data.frames and matrices. The main difference between data.frames and matrices is that *all* of the elements of matrices will be of the same mode (e.g., all character, numeric, or logical), and data.frame can combine vectors of various modes. Let's say we have the three vectors, all of the same length:

```
> x <- c("bob", "pete", "banana", "jim", "orange", "black-suit",
+        "blue-suit", "buzzoff", "yadda")
> y <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
> z <- c(TRUE, FALSE, FALSE, TRUE, FALSE, FALSE, TRUE, FALSE, FALSE)
```

Note that x is a character vector, y is numeric, and z is logical. You can use the mode function to verify that.

```
> mode(x)
[1] "character"
> mode(y)
[1] "numeric"
> mode(z)
[1] "logical"
```

Next, combine x, y, and z into a data.frame using the ... um ... data.frame command:

```
> dat <- data.frame(meep = x, mope = y, zu = z)
```

The idea is simple. There are as many arguments as there are vectors that you want to combine together. All of the vectors *must* be the same length, but they can be of different modes, and the *names* of the variables should be the *names* of the arguments. So by writing `meep = x`, I am saying "take the vector `x` and give it the name `meep` in the `data.frame`."

```
> dat
        meep mope    zu
1        bob    1  TRUE
2       pete    2 FALSE
3     banana    3 FALSE
4        jim    4  TRUE
5     orange    5 FALSE
6 black-suit    6 FALSE
7  blue-suit    7  TRUE
8    buzzoff    8 FALSE
9      yadda    9 FALSE
```

But when you try to find out the *mode* of `dat`, it will not show up as numeric, character, or logical:

```
> mode(dat)
[1] "list"
```

It will show up as a list, which effectively means that different columns are (potentially) different modes. A `list` is a generic object in `R`, something that we will talk about more generally in a few chapters - and data frames are lists of vectors that all have the same length and are put into a nice form. Once we have a data frame, we can edit the values, add new values, make more variables etc. just by using the `edit` command.

```
> dat <- edit(dat)  # to work more naturally with data.frames
>                           # ... also use ''fix'' for perminant changes
```

We can also change the variable names using the `names` argument (just like we did for vectors), and we can name subjects/rows via the `rownames` argument.

```
> names(dat)    <- c("larry", "curly", "moe")
> rownames(dat) <- c("s1", "s2", "s3", "s4",
+                    "s5", "s6", "s7", "s8", "s9")
> dat
        larry curly   moe
s1        bob     1  TRUE
s2       pete     2 FALSE
s3     banana     3 FALSE
s4        jim     4  TRUE
s5     orange     5 FALSE
s6 black-suit     6 FALSE
s7  blue-suit     7  TRUE
s8    buzzoff     8 FALSE
s9      yadda     9 FALSE
```

If you forget whether the `names` argument renames the rows or columns of the data frame, remember that the *columns* are the variables, are more important, and the `names` argument changes the important names. The rows are the observations, and the `rownames` argument (being more specific) is subservient to the less specific `names`. As a side note, you could also change the variable names via the `colnames` function:

```
> colnames(dat) <- c("buzz", "lightyear", "tom")
> dat
         buzz lightyear   tom
s1        bob         1  TRUE
s2       pete         2 FALSE
s3     banana         3 FALSE
s4        jim         4  TRUE
s5     orange         5 FALSE
s6 black-suit         6 FALSE
s7  blue-suit         7  TRUE
s8    buzzoff         8 FALSE
s9      yadda         9 FALSE
```

So the rules are:

1. `colnames` always changes (or displays) the column names,
2. `rownames` always changes (or displays) the row names,
3. `names` only changes (or displays) the *important* or *obvious* names.

Once we have a data frame of variables (all in vector form), we might want to pull out particular vectors for personal use. There are many ways of extracting vectors, three of which we will talk about at some point in this chapter. The easiest method of using variables is by *attaching* a `data.frame` via the `attach` function. After we attach a data frame, the columns of the data frame become objects that we can call directly by name.

```
> dat          # what does the data frame look like?
         buzz lightyear   tom
s1        bob         1  TRUE
s2       pete         2 FALSE
s3     banana         3 FALSE
s4        jim         4  TRUE
s5     orange         5 FALSE
s6 black-suit         6 FALSE
s7  blue-suit         7  TRUE
s8    buzzoff         8 FALSE
s9      yadda         9 FALSE
> attach(dat)  # so we can call variables by name
> lightyear    # the second column of the data frame :)
[1] 1 2 3 4 5 6 7 8 9
```

And if we are done using the vectors of a `data.frame`, we should `detach` the data frame to free up its objects.

```
> detach(dat)  # removing the vectors associated with dat
> try(lightyear, silent = TRUE)[1]
[1] "Error in try(lightyear, silent = TRUE) : object 'lightyear' not found\n"
> # the vector/variable doesn't exist anymore :(
```

The next easiest method to extract vectors from a data frame is calling them by name using the $ operator. I like to call the $ operator the drawer opener. It is as though a data frame (or *any* list as we will find out later) is a cabinet filled with drawers, and the dollar sign opens the appropriate drawer. Use the following syntax: `data.frame$variable` ... for example:

```
> dat$lightyear        # pulls out the lightyear vector
[1] 1 2 3 4 5 6 7 8 9
> wee <- dat$lightyear # assigns that vector to "wee"
> mean(wee)            # calculates stuff on that vector!
[1] 5
```

Once you have particular objects (vectors and data frames, or (later) matrices and lists), sometimes you want to see what is contained in those objects without seeing the *entire* object. Even though our `data.frame` only has:

```
> nrow(dat)  # to find out the number of rows
[1] 9
```

rows and

```
> ncol(dat)  # to find out the number of columns
[1] 3
```

columns, many data sets are gigantic, and showing all of the data at one time would be annoying. To only show a few rows of the data frame, we can use the functions: `head` (and `tail`). Both `head` and `tail` depend on the particular object. For data frames (and matrices), those functions show the first (and last) few rows, but for vectors, they show the first (and last) few elements. For example:

```
> head(dat)        # the first six rows (default)
         buzz lightyear   tom
s1        bob         1  TRUE
s2       pete         2 FALSE
s3     banana         3 FALSE
s4        jim         4  TRUE
s5     orange         5 FALSE
s6 black-suit         6 FALSE
> head(dat, n = 3) # the first three rows (changing an argument)
     buzz lightyear   tom
s1    bob         1  TRUE
s2   pete         2 FALSE
s3 banana         3 FALSE
```

```
> tail(dat)        # the last six rows (default)
         buzz lightyear   tom
s4        jim         4  TRUE
s5     orange         5 FALSE
s6 black-suit         6 FALSE
s7  blue-suit         7  TRUE
s8    buzzoff         8 FALSE
s9      yadda         9 FALSE
> tail(dat, n = 3) # the last three rows (changing an argument)
        buzz lightyear   tom
s7 blue-suit         7  TRUE
s8   buzzoff         8 FALSE
s9     yadda         9 FALSE
```

shows rows for data frames, but

```
> head(wee)        # the first six elements
[1] 1 2 3 4 5 6
> tail(wee)        # the last six elements
[1] 4 5 6 7 8 9
```

shows elements for vectors.

Unfortunately, a funny thing happened on the way to creating our data frame. One of our vectors was originally a character vector:

```
> x       # show the original x
[1] "bob"        "pete"       "banana"     "jim"
[5] "orange"     "black-suit" "blue-suit"  "buzzoff"
[9] "yadda"
> mode(x) # it is a character vector!
[1] "character"
```

but when we extracted that vector from the data frame something strange happened:

```
> x2 <- dat$buzz  # the original x (in the data frame)
> x2              # what in the world is that?
[1] bob        pete       banana     jim        orange
[6] black-suit blue-suit  buzzoff    yadda
9 Levels: banana black-suit blue-suit bob buzzoff ... yadda
> mode(x2)        # and ... it's numeric? It doesn't look numeric!?!!
[1] "numeric"
```

This new fangled thing is called a "factor," and it is another kind of vector ... one that we have not talked about until now.

2.3 Factors in R

When you try to put a character vector into a data frame, `R` is like the mother who knows best:

You *think* that this vector should be a character vector. But because you are putting it into a data frame, it has got to be a variable. Why else would you put it in a "data" frame? You are not putting it into a "character" frame. But because it is in character form, you really mean that it is a nominal or categorical variable, and it is much easier if we just turn the levels of the nominal variable into numbers. So - there bubala - and you don't have to thank me.

A factor vector is really like a cross between a numeric vector and a character vector and acts like both. It has "values" (and "levels") - which are numbers, the 1, 2, 3, ... coding each category, and "labels," which are character strings indicating what each number represents. R basically treats it as a numeric variable that the user sees as character strings ... without the quotes. So let us say that we have 5 people in our data set, some of which are males and some of which are females. We can create a factor vector in a couple of ways.

1. Write the males and females into a character vector and tell R that we want this character vector to *really* be a factor vector:

```
> gender <- c("male", "female", "female", "male", "female")
> gender         # gender has quotes

[1] "male"   "female" "female" "male"   "female"

> mode(gender)   # and it is a character vector (duh!)

[1] "character"

> gender2 <- as.factor(gender)  # gender REALLY is a factor vector.
> gender2        # see!?!  now there are "levels" for each category,

[1] male   female female male   female
Levels: female male

>                # shown below the vector, and there are no quotes?
> mode(gender2)  # it is numeric!?  weird!

[1] "numeric"
```

2. Write the 1s and 2s into a numeric vector and tell R that we want this numeric string to *really* be a factor vector with appropriate labels:

```
> # 1 and 2 are easier to type than "man" and "woman":
> gender3 <- c(1, 2, 2, 1, 2)
> gender3                      # a standard numeric vector

[1] 1 2 2 1 2

> mode(gender3)                # see!?!

[1] "numeric"

> gender4 <- factor( gender3, labels = c("male", "female") )
> gender4                      # same as gender2 from before
```

```
[1] male   female female male   female
Levels: male female
```

Note that the `labels` argument takes a character string: The *first* element codes the 1s as "male" and the *second* element codes the 2s as "female" etc.

3. Or we could write a character vector and build a data frame from it, as we did before.

Note that in the first code chunk, we used `as.factor`. There are also `as.numeric`, `as.logical`, `as.character`, etc., which will change vector types (if possible). You probably will not have to worry about those too often, but be aware that they exist. Even though the *mode* of a factor vector is "numeric," its *class* is "factor."

```
> class(gender4)  # this is how we know it is a factor vector
[1] "factor"
```

The `class` argument is a little beyond the scope of this class (*snickers*). Basically, different objects have different classes, and certain functions will treat the objects differently depending on their class. For instance, when we (later in this book) want to perform a regression or ANOVA, the `class` of our vector reminds `R` *not* to treat the numbers as real numbers in-and-of-themselves, but as categories. And if we try to take the mean of a factor vector:

```
> mean(gender4)  # the mean of a categorical variable!??!
[1] NA
```

`R` reminds us that the mean of a categorical variable does not make any sense.

Now let's say we have a data frame composed of one factor and one numeric variable:

```
> # a data frame of gender and extraversion:
> dat2 <- data.frame(gender = gender4, extra = c(2.3, 4.5, 3.2, 4, 2))
```

and we are interested in *just* looking at properties of the males (or females!?). How can we extract elements of our `data.frame` without just printing the whole data frame and copying/pasting numbers into a new vector? Well, `R` allows us to access the indices of a vector easily and with little use of copy and paste.

2.4 Accessing Specific Indices

2.4.1 Comparing Vectors

There are a few things to know before we can work with indices. First, you need to know how to create a logical vector from a numeric or character vector. We want to tell `R` *which* rows/columns/elements to take, and using trues and falses for that purpose is nice. The logical operators you should know are as follows:

```
<    # less than (duh!)
<=   # less than or equal to
>    # greater than (double duh!)
```

```
>=   # greater than or equal to
==   # equal to (you NEED both equals)
!=   # not equal to
```

The only tricky logical operator (for those of you with no programming experience) is ==. Remember that = is an *assignment* operator, so if you write:

```
> x = 2  # ASSIGN 2 to x
```

R will **assign** the value "2" to the name x. Therefore, we need a different method of comparing two vectors or numbers, and the double equals serves that purpose. Now, if we type:

```
> x == 2  # IS x equal to 2?
[1] TRUE
```

R will not do *any* assignment and just print TRUE.

Remember the last time we talked about "vectorized" operations. Comparisons are "vectorized" (and even "matrix-ized" and "data frame-ized"). If we use the principle that the object on the left- hand -side is the object on which we want to do the comparison, and the object on the right-hand-side is what we want to compare it to, we can create a true/false vector pretty easily:

```
> x <- c(3, 3, 3, 1, 1, 2, 1, 5, 4, 5, 6, 7)
> x == 3  # which elements of x are equal to 3?
 [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
[10] FALSE FALSE FALSE
> x != 3  # which elements of x are not equal to 3?
 [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[10]  TRUE  TRUE  TRUE
> x >  3  # which elements of x are greater than 3?
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
[10]  TRUE  TRUE  TRUE
> x <= 3  # which elements of x are less than or equal to 3?
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
[10] FALSE FALSE FALSE
>         # ...
```

and we can also compare a vector to a vector:

```
> y <- c(3, 2, 3, 1, 2, 3, 2, 2, 2, 2, 3, 7)
> x == y  # which values of x are equal to the
 [1]  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
[10] FALSE FALSE  TRUE
>         # corresponding values of y?
> x >= y  # which values of x are greater than or equal to the
 [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE
[10]  TRUE  TRUE  TRUE
```

```
>         # corresponding values of y?
```

or even a `data.frame` to a `data.frame`:

```
> dat2 <- data.frame(gender = gender4, extra = c(2.3, 4.5, 3.2, 4, 2))
> dat3 <- data.frame(gender = gender2, extra = c(1, 4.5, 3.2, 1, 1))
> dat2 == dat3  # compare data frame elements directly!
     gender extra
[1,]   TRUE FALSE
[2,]   TRUE  TRUE
[3,]   TRUE  TRUE
[4,]   TRUE FALSE
[5,]   TRUE FALSE
```

Make sure to remember that the numeric comparisons (e.g., "greater than") only work with numeric variables, obviously.

```
> dat2 >= dat3 # ??? how can we compare categorical variables this way?
```

Second, you need to know how to build a logical vector from logical vectors. The logical comparisons that you should know are the following:

```
&  # "and":             comparing all of the elements
&& # "and":             comparing only the FIRST element
|  # "or" (inclusive): comparing all of the elements
|| # "or" (inclusive): comparing only the FIRST ELEMENT
!  # "not":            flip the TRUE's and FALSE's
```

The single `&` and `|` create a logical *vector* out of a logical *vector*. The double `&&` and `||` create a logical *scalar* by just comparing the first elements. For instance:

```
> x <- 1:10          # create a numeric vector
> x < 7              # which x's are less than 6
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
[10] FALSE
> x > 3              # which x's are greater than 3
 [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[10]  TRUE
> (x > 3) & (x < 7)  # is x greater than 3 AND less than 7?
 [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE
[10] FALSE
> (x > 3) && (x < 7) # what about only the FIRST element of x?
[1] FALSE
> (x > 3) | (x < 7)  # is x greater than 3 OR less than 7?
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
> (x > 3) || (x < 7) # what about only the FIRST element of x?
[1] TRUE
> x == 5             # is x equal to 5?
```

```
 [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
[10] FALSE
> x != 5             # is x NOT equal to 5? ... or:
 [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
[10]  TRUE
> !(x == 5)          # flip the signs of "is x equal to 5?"
 [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
[10]  TRUE
```

Sometimes, we want to compare each element of a vector to any number of elements. For instance, maybe we have two sets of 1 – 5s and we want to see if anybody is a 1 or a 2. Well, if we do this directly:

```
> x <- c(1:5, 1:5)
> x == c(1, 2)
 [1]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[10] FALSE
```

R just looks at the first two elements and ignores the rest. Weird! Now, we could look at each element individually and then combine the vectors with the logical | (or):

```
> (x == 1) | (x == 2)
 [1]  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
[10] FALSE
```

but this method can get cumbersome pretty quickly (if we have a lot of elements to check). A shortcut to checking any number of elements is to use the **%in%** command, which compares each element of vector 1 with all of the elements of vector 2:

```
> x %in% c(1, 2)
 [1]  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
[10] FALSE
```

Note that there are many functions that have this form (percent followed by command followed by percent), and even more functions that compare logical vectors and/or numeric vectors and/or character vectors, but we do not need to know them at present. We *do*, however, need to know how to extract indices from a vector and data frame.

2.4.2 Extracting Elements

R uses particular bracket/braces things for particular purposes. There are (effectively) four bracket/braces things to know:

```
()   # group the stuff inside together or
     # use a function on the stuff inside
{}   # put the stuff inside into one expression
[]   # extract elements of the stuff inside
[[]] # extract list elements of the stuff inside
```

At the moment, you really *don't* need to know about `[[]]`, and we haven't (yet) been introduced to `{}`. But differentiating between parentheses and square brackets can be tricky. Know that parentheses *tend* to be used for functions, and square brackets *tend* to be used for objects. Currently, we want to extract particular elements from vectors/data frames, so we will work with the square brackets. First, we need to know how to call square brackets. As it turns out, each dimension that we add to an object (rows, columns, third dimension, fourth dimension, trippy!) results in a comma being added to the square-i-ness of the bracket. So, for a vector (with one dimension, sort of), we will type something like:

```
vector[stuff to extract]
```

and for a data frame, we will (usually) type something like:

```
data.frame[rows to extract, columns to extract]
```

Now, this is a simplification–you can extract elements from data frames (or matrices, later) without a comma, but I will mostly use the dimension-as-comma setup to reduce confusion.

The easiest object to work with is a vector, and the methods of extracting elements from vectors generalize to higher dimensional objects. For vectors, the *numbers inside* the square brackets correspond to the element-place in the vector. So, if you have a vector of length 10, and you want to extract the *first* element, put a "1" inside square brackets:

```
> x <- c(2, 4, 3, 2, 5, 4, 3, 6, 5, 4)
> x[1]
[1] 2
```

and if you want to pull out the *fifth* element, put a "5" inside square brackets:

```
> x[5]
[1] 5
```

and if you want to pull out the *last* element, you can find out the number of elements with the `length` function, and put that number inside square brackets:

```
> x[length(x)]
[1] 4
```

We can also take more than one entry at a time, which is called "slicing." If `x` is a vector of data, and `ind` corresponds to a vector of indices (places) of `x`, then writing `ind` inside square brackets will form a *new* vector from the `x` elements at those places.

```
> ind <- 1:4  # a sequential vector from 1 -- 4
> ind         # double checking
[1] 1 2 3 4
> x[ind]      # the 1 -- 4 elements of x
[1] 2 4 3 2
```

We can also take an arbitrary vector, as long as it contains positive integers but does not contain a number greater than the length of the vector.

```
> x[c(1, 3, 7)]  # the 1st, 3rd, and 7th elements of x
[1] 2 3 3
```

As shown in the previous code chunk, by putting a vector of *positive* integers inside `x`, `R` *extracts* the indices corresponding to those integers. But if we put a vector of *negative* integers inside `x`, `R` will *remove* the indices corresponding to those integers. For example, `x[-1]` removes the first element of a vector.

```
> x     # all of the elements
 [1] 2 4 3 2 5 4 3 6 5 4
> x[-1] # get rid of the first element
[1] 4 3 2 5 4 3 6 5 4
```

And we can put *any* vector inside of `x` *as long as* the integers are all of the same sign.

```
> ind1 <- -1:4            # making an integer vector from -1 to 4
> ind1
[1] -1  0  1  2  3  4
> ind2 <- -(1:4)          # making an integer vector of -1 to -4
> ind2
[1] -1 -2 -3 -4
> try(x[ind1], silent = TRUE)[1]  # does not work because
[1] "Error in x[ind1] : only 0's may be mixed with negative subscripts\n"
>                                 # there are pos AND neg integers
> x[ind2]                 # removes the first four elements
[1] 5 4 3 6 5 4
```

An alternative method of extracting vector elements is by using logical vectors. The logical vectors must be of the *same length* as the original vector, and (obviously) the TRUEs correspond to the elements you want to extract whereas (obviously) the FALSEs correspond to to the elements you do not want to extract. For instance, if you have a logical vector:

```
> x2 <- 1:4
> logi <- c(TRUE, FALSE, FALSE, FALSE)
> x2[logi]     # only extract the first element
[1] 1
```

only extracts the "1" because the "true" corresponds to the first element of `x2`. We can do this with comparisons as well.

```
> x3 <- c(2, 2, 2, 3, 3, 3, 4, 4, 5, 6, 6, 6)
> x3[x3 > 2]  # extract the values where x3 is greater than 2
[1] 3 3 3 4 4 5 6 6 6
> x3[x3 <= 6] # extract the values where x3 is leq 6
 [1] 2 2 2 3 3 3 4 4 5 6 6 6
> x3[x3 == 4] # extract the values where x3 equals 4
```

```
[1] 4 4
```

Because the stuff inside the square brackets are logicals, we can use *any* logical vector and not just those that correspond to the vector itself.

```
> # A data frame of gender and extraversion:
> dat2 <- data.frame(gender = gender4, extra = c(2.3, 4.5, 3.2, 4, 2))
> gen <- dat2$gender    # put the gender vector in "gen"
> ext <- dat2$extra     # put the extraversion vector in "ext"
> ext[gen == "male"]    # extraversion scores of the males :)
[1] 2.3 4.0
> ext[gen == "female"]  # extraversion scores of the women :)
[1] 4.5 3.2 2.0
```

So what do the following do?

```
> ext[!(gen == "male")]    # ????
[1] 4.5 3.2 2.0
> ext[!(gen == "female")]  # ???? + 1
[1] 2.3 4.0
```

Oddly, we can *convert* a logical vector directly to indices directly by using the `which` function.

```
> which(gen == "male")         # which indices correspond to guys?
[1] 1 4
> which(c(TRUE, FALSE, TRUE))  # the locations of TRUE!
[1] 1 3
```

The `which` function makes it pretty easy to go back and forth between logicals and indices. For instance, we might want to know which participants are "male" - and the `which` function provides a direct method of finding them. Or we might feel more comfortable working with indices than logical vectors. Or we might have a vector of frequencies and need to determine the location corresponding to the maximum frequency. (Hint: Some things in R are not as straightforward as in other programs).

Of course we might have a data frame rather than a vector, and if we have a data frame and want to extract rows/columns, we do *exactly* the same thing as we did for vectors, but we include a comma separating the rows we want to take from the columns we want to take. Moreover, if we want to include *all* of the rows, we leave the row part blank ... and if we want to include *all* of the columns, we leave the column part blank.

```
> dat3 <- data.frame(gender = gender4,
+                    extra = c(2.3, 4.5, 3.2, 4, 2),
+                    stuff = c(1, 0, 1, 1, 0))
> dat3[c(1, 3), c(1, 2)]  # two rows and two columns
  gender extra
1   male   2.3
3 female   3.2
```

```
> dat3[c(1, 3), ]         # two rows and every column
  gender extra stuff
1   male   2.3     1
3 female   3.2     1
> dat3[ , c(1, 2)]        # every row and two columns
  gender extra
1   male   2.3
2 female   4.5
3 female   3.2
4   male   4.0
5 female   2.0
```

So back to the problem at the beginning of this section. How could we construct a new `data.frame` containing only "male" participants? Well, the easiest method is as follows.

```
> dat3[dat3$gender == "male", ] # only the male rows
  gender extra stuff
1   male   2.3     1
4   male   4.0     1
```

In the previous code chunk, I effectively wrote: "extract the gender variable, put TRUE when the gender variable is "male", and take out those rows from the `data.frame` `dat3`". You should play around with calling indices. But we also need to do a little bit of statistics/graphing before the chapter ends so that you can see the basic statistical capabilities of `R` before I induce boredom and you give up.

2.5 Descriptive Statistics and Graphing

For the following (brief) demonstration, we want to use the `sat.act` dataset in the `psych` package. First, we need to load the package and then the data inside the package

```
> library(psych)
> data(sat.act)
```

Using `head` we can see the first few rows of the data frame, and we can also see a description of the data frame with `?`:

```
> head(sat.act)
      gender education age ACT SATV SATQ
29442      2         3  19  24  500  500
29457      2         3  23  35  600  500
29498      2         3  20  21  480  470
29503      1         4  27  26  550  520
29504      1         2  33  31  600  550
29518      1         5  26  28  640  640


> ?sat.act
```

Apparently `ACT` is the ACT composite score for a particular age group. Also note that they did not code the gender variable in the dataset but revealed the coding in the help file. It might make sense to code gender ourselves:

```
> sat.act$gender <- factor( sat.act$gender, labels = c("male", "female") )
```

For categorical variables and/or other discrete variables, the `table` function is rather useful. If you pass `table` a vector, it tells you the number of times each of the possible values occurs in that vector. So

```
> table(sat.act$gender)
  male female
   247    453
```

tells you the number of men and women (there are many more women than men) and

```
> table(sat.act$education)
  0   1   2   3   4   5
 57  45  44 275 138 141
```

tells you the number of people at each education level. To make things simpler, let us just look at the ACT scores of the women:

```
> # - Put TRUE where sat.act$gender is "female" and FALSE otherwise,
> # - Keep the ACT scores of the TRUEs ... err ... the "females"
> act.fem <- sat.act$ACT[sat.act$gender == "female"]
```

Using the female data, we can calculate descriptive statistics, many of which were discussed in the previous chapter.

```
> mean(act.fem)    # the mean of the ACT scores
[1] 28.41722
> median(act.fem)  # the median of the ACT scores
[1] 29
```

The mean is close to the median, so the data is probably pretty symmetric. How would we know if the data was negatively or positively skewed from these two statistics?

```
> var(act.fem)  # the variance of the ACT scores
[1] 21.9782
> sd(act.fem)   # the standard deviation (square root of variance)
[1] 4.688091
> IQR(act.fem)  # the inner-quartile range of ACT scores
[1] 7
```

We could try to calculate the *actual* range:

```
> range(act.fem) # trying to calculate the range?
[1] 15 36
```

but R is weird and returns the minimum and maximum scores and *not* the difference between them. We can calculate that difference in a few ways

```
> x <- range(act.fem)
> x[2] - x[1]
[1] 21
```

or:

```
> diff(x)  # calculate the difference between the x elements
[1] 21
```

or:

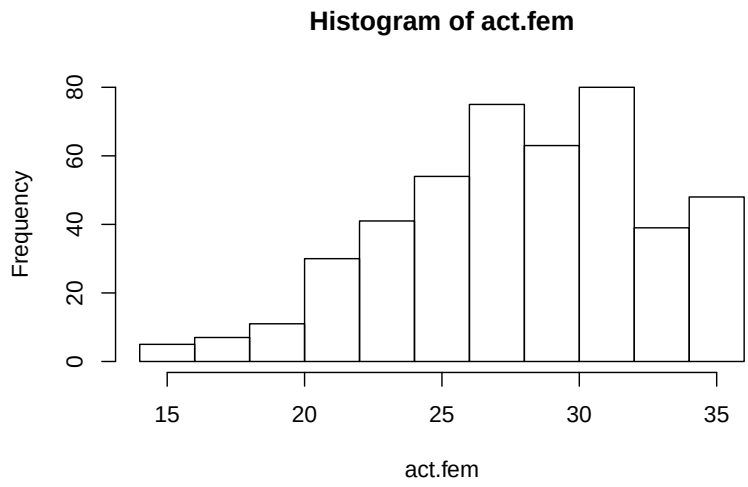
```
> max(x) - min(x)
[1] 21
```

Finally, we might want to visualize the scores, and R has lovely graphing capabilities. We will talk about graphing (in detail) in a future chapter because: This chapter is already *way* too long. The first major graph is a histogram, and the important arguments of the histogram are:

```
hist(x, breaks, freq)
```

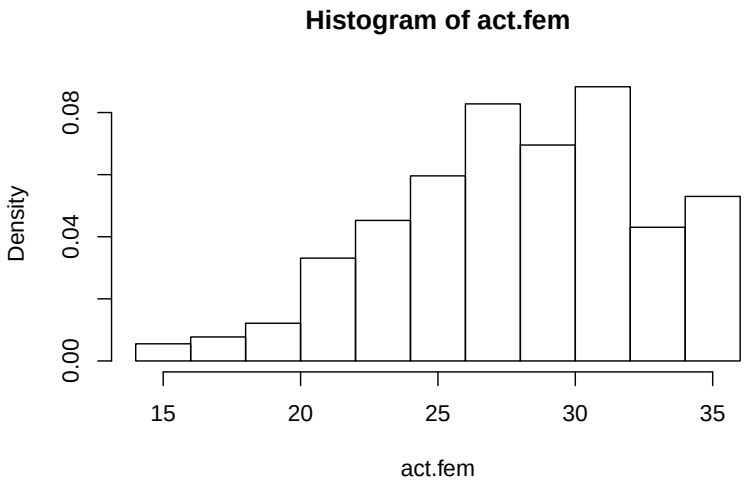
where x is a numeric vector, freq is a logical (TRUE/FALSE) indicating if "frequencies" or "probabilities" should be plotted on the y-axis, and breaks indicates the number of break points. If you ignore the last two arguments and just surround your vector with hist:

```
> hist(act.fem)  # a histogram of female scores on the ACT
```



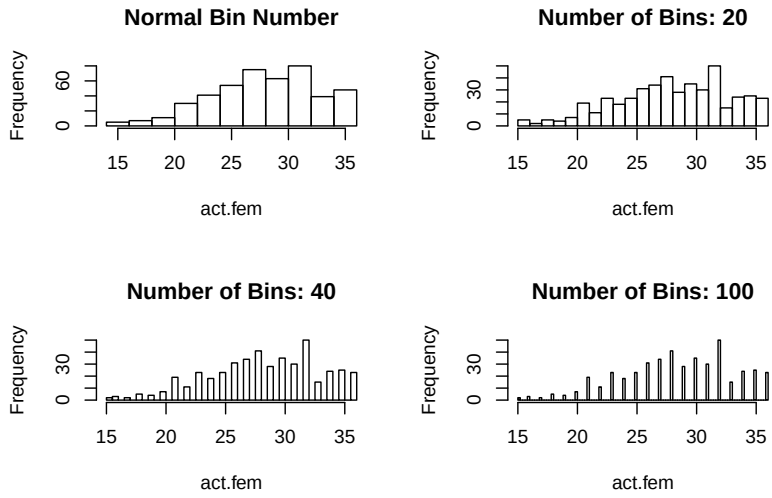
R will *automatically* set the bins, based on an algorithm, and plot the frequency on the y-axis (in most cases). You can set `freq = FALSE` to plot the probabilities rather than the frequencies:

```
> hist(act.fem, freq = FALSE)
```



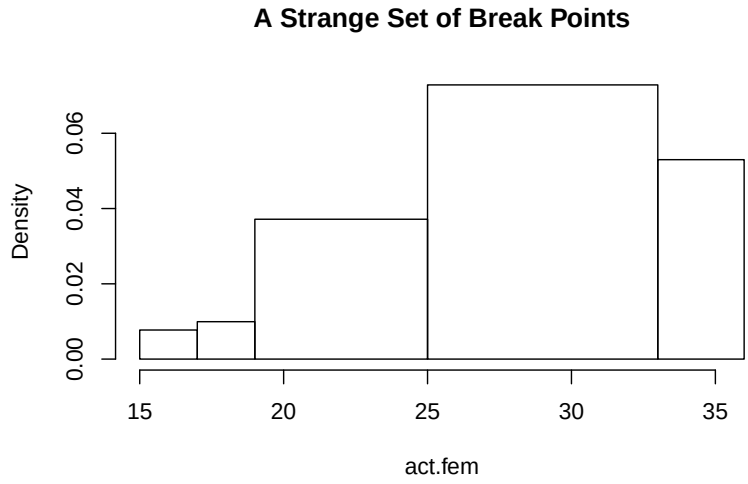
Magic!!?! And you can change the number and location of the bins in two ways: (1) Telling R the number of bins/breaks that you want, or (2) Setting the exact location of the breaks yourself. To tell R the number of bins/breaks you want, just plug a number into the breaks argument.

```
> # - Ignore the "par(mfrow)" command for now.
> # - Just copy/paste each of the histograms into your R session.
> par(mfrow = c(2, 2))
> hist(act.fem,
+      main = "Normal Bin Number")
> hist(act.fem, breaks = 20,
+      main = "Number of Bins: 20")
> hist(act.fem, breaks = 40,
+      main = "Number of Bins: 40")
> hist(act.fem, breaks = 100,
+      main = "Number of Bins: 100")
> par(mfrow = c(1, 1))
```



And to tell R *exactly where* to put the breaks, you supply a numeric vector of break locations (where the minimum score is the first element and the maximum score is the last).

```
> hist(act.fem, breaks = c(15, 17, 19, 25, 33, 36),
+      main = "A Strange Set of Break Points")
```



Of course, R often decides not to listen to your `breaks` argument and sets the number of bins itself. And in either case, the distribution of female scores appears slightly negatively skewed.

2.6 Appendix: Functions Used

Here is a list of important functions from this chapter:

```
# Packages
install.packages(pkg, dependencies = TRUE) # get a package
library(pkg)             # load a package
require(pkg)             # same as "library" with an odder name
library(help = "pkg")    # what is in the package "pkg"?
help(topic, package)     # help before loading the package
data(dataset, package)   # access a dataset in a package

# Dataframes
data.frame(...)          # turn the (vector) arguments into a data frame
names(x)                 # assign or extract IMPORTANT/OBVIOUS names
rownames(x)              # assign or extract names for the rows
colnames(x)              # assign or extract names for the columns
attach(df)               # call the vectors by name
df$drawer                # pick out one vector from a data frame
nrow(x), ncol(x)         # how many rows/columns does x have?
head(x, n), tail(x, n)   # the first/last n (rows/elements) of x

# Factors
factor(x, levels, labels) # make a factor vector with these labels
                          # (in order) coding these levels (in order)
as.factor(x)              # turn x into a factor vector

# Logical Operators
<, <=, >, >=, ==, !=     # compare numeric/character vectors
&, &&, |, ||, !          # compare logical vectors
x %in% y                 # is each x a part of the y vector?

# Indices
vector[stuff to extract] # either indices or TRUES/FALSES
data.frame[rows to extract, cols to extract]
which(x)                  # which are the indices corresponding to TRUE?

# Descriptive Statistics
length(x)                # length of the vector
sum(x, na.rm)            # sum of all elements in the vector
mean(x, na.rm)           # mean
median(x, na.rm)         # median (using standard definition)
var(x, na.rm)            # variance (which is it!?)
sd(x, na.rm)             # standard deviation
range(x)                 # the minimum and maximum score
min(x), max(x)           # also ... the minimum and maximum score
diff(x)                  # the difference between elements in a vector
hist(x, breaks, freq)    # plotting a histogram
```

Chapter 3
Matrices and Lists

Interviewer: *Good evening. Well, we have in the studio tonight a man who says things in a very roundabout way. Isn't that so, Mr. Pudifoot?*
Mr. Pudifoot: *Yes.*
Interviewer: *Have you always said things in a very roundabout way?*
Mr. Pudifoot: *Yes.*
Interviewer: *Well, I can't help noticing that, for someone who claims to say things in a very roundabout way, your last two answers have had very little of the discursive quality about them.*
—Monty Python's Flying Circus - Episode 26

3.1 Matrices in R

3.1.1 Building Matrices

R is very strange (if you haven't figured out by now). It has two *types of objects* to handle multidimensional arrays: data frames and matrices (there are also *actual* arrays, which are like multidimensional matrices, but most people cannot think in too many dimensions, so these are ill-advisable). Unlike matrices, data frames are like data *sets* and can have vectors of various modes. For instance, we could have a character vector, numeric vector, and logical vector, and put them *all* in a data frame.

```
> c.vec <- c("my", "mother", "told", "me", "to", "pick", "the")
> n.vec <- c(1, 2, 3, 4, 5, 6, 7)
> l.vec <- c(TRUE, FALSE, FALSE, TRUE, FALSE, FALSE, TRUE)
> d.frame1 <- data.frame(blah = c.vec, eek = n.vec, muck = l.vec)
> d.frame1  # prints the data frame (only the character vector change!)
    blah eek  muck
1     my   1  TRUE
2 mother   2 FALSE
3   told   3 FALSE
```

```
4     me   4  TRUE
5     to   5 FALSE
6   pick   6 FALSE
7    the   7  TRUE
```

Only the *character* vector changed (into a factor vector), mostly because vectors have irritating parents, as we talked about earlier. We can, of course, make sure that the character vector *does not* change into a factor vector by altering the (almost too obvious) `stringsAsFactors` argument to FALSE.

```
> d.frame2 <- data.frame(blah = c.vec, eek = n.vec, muck = l.vec,
+                        stringsAsFactors = FALSE)
> d.frame1[ , 1]  # no quotes, of course (and the levels thing)
[1] my     mother told   me     to     pick   the
Levels: me mother my pick the to told
> d.frame2[ , 1]  # the quotes are back baby - w00t!
[1] "my"     "mother" "told"   "me"     "to"     "pick"
[7] "the"
```

Even though the vector's parent thinks that it knows best, no vector is affected by the mode of other vectors in a data frame. The only relationship in construction between the vectors is their length -- they all must be of the same length because the entries in the data frame are supposed to represent a set of linked scores. Matrices (on the other hand) are like multidimensional (well ... two-dimensional, so not very many of the "multi"s) vectors, so all of the entries in a matrix must be of the same type. If we try to convert the data frame into a matrix (which is possible, given our futuristic, 21st century technology), *just like it did when we added a bunch of vectors together*, `R` will pick the most general mode and force all of the vectors to be of that type.

```
> mat1 <- as.matrix(d.frame1)
> mat1  # ooh - a matrix.
     blah     eek muck
[1,] "my"     "1" " TRUE"
[2,] "mother" "2" "FALSE"
[3,] "told"   "3" "FALSE"
[4,] "me"     "4" " TRUE"
[5,] "to"     "5" "FALSE"
[6,] "pick"   "6" "FALSE"
[7,] "the"    "7" " TRUE"
```

This is the thought process of `R` during the `mat1` trasformation:

- Factors do not make sense with matrices.
- We must convert the factor vector back into a character vector.
- Now the character vector is the most general mode of the bunch.
- So we must put quotes around the numbers and TRUEs/FALSEs.
- Yay!

R also keeps the names of the rows/columns the same after the conversion, but those names are deceptive. Because we no longer have a list (of which a data frame is a specific example), our columns are no longer *different* drawers of the same cabinet, so we can no longer open those drawers with the dollar sign (`$`) argument.

```
> # Note: the dollar sign is no longer valid?  Annoying!
> try(mat1$blah, silent = TRUE)
```

To call particular columns of the matrix, we can either select the *entire* column by `[ , column we want]` after the object name (just like for data frames).

```
> mat1[ , 2]     # the column of character numbers?
[1] "1" "2" "3" "4" "5" "6" "7"
```

We can also call the column, by name, inside the square brackets.

```
> mat1[ , "eek"]  # works too.
[1] "1" "2" "3" "4" "5" "6" "7"
```

The "name inside square bracket" works for data frames as well.

```
> d.frame1[ , "eek"]  # works, but I like dollar signs ...
[1] 1 2 3 4 5 6 7
>                     # they help with statistical poverty!
```

However, I *rarely* use the "name-inside-square-bracket" method, as (for some strange reason) I don't seem to trust that it will work. But if you prefer indexing vectors using their names, you can. You can even put *multiple names* in a vector, and pull out *multiple vectors* that way.

```
> mat1[ , c("blah", "eek")]
     blah     eek
[1,] "my"     "1"
[2,] "mother" "2"
[3,] "told"   "3"
[4,] "me"     "4"
[5,] "to"     "5"
[6,] "pick"   "6"
[7,] "the"    "7"
> mat1
     blah     eek muck
[1,] "my"     "1" " TRUE"
[2,] "mother" "2" "FALSE"
[3,] "told"   "3" "FALSE"
[4,] "me"     "4" " TRUE"
[5,] "to"     "5" "FALSE"
[6,] "pick"   "6" "FALSE"
[7,] "the"    "7" " TRUE"
```

Lest I get distracted with randomly pulling out vectors (which is possible on a dark and stormy Saturday such as this one), I should indicate how to create a *new* matrix ... from scratch. The general format is as follows:

```
matrix(vector, nrow, ncol, byrow = FALSE, dimnames = NULL)
```

I will go through each argument in turn:

1. `vector`: A vector of any type, although factors will be converted back to character vectors. If `vector` is not as long as the *actual* matrix, it will be repeated and repeated until the matrix is filled.
2. `nrow`: The number of rows in your matrix.
3. `ncol`: The number of columns in your matrix.
4. `byrow`: A logical (TRUE or FALSE) indicating whether the `vector` should fill up the first row, then the second row, then the third row ... *or* (defaultly) the first column, then the second column, then the third column, ...
5. `dimnames`: Do not worry about this argument.

So, if you want to create a 3×2 matrix of all 1s (because you are number 1 – go you), the easiest way to do so is as follows.

```
> # We only need to put one 1 because R recycles the 1s.
> # -- nrow = 3 because we want three rows,
> # -- ncol = 2 because we want two columns,
> # No need for a "byrow" argument because every element is the same!
> ones <- matrix(1, nrow = 3, ncol = 2)
> ones  # pretty :)
     [,1] [,2]
[1,]    1    1
[2,]    1    1
[3,]    1    1
```

We can also build a matrix of different numbers, but we must consider the `byrow = TRUE` argument to make sure that the entries are in the "correct" place.

```
> m.awesome1 <- matrix( c( 1,  2,  3,  4,
+                          5,  6,  7,  8,
+                          9, 10, 11, 12,
+                         13, 14, 15, 16), nrow = 4, ncol = 4)
> m.awesome1  # whoops - this is not what we want.
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
```

```
> m.awesome2 <- matrix( c( 1,  2,  3,  4,
+                          5,  6,  7,  8,
+                          9, 10, 11, 12,
+                         13, 14, 15, 16), nrow = 4, ncol = 4,
+                                          byrow = TRUE)
> m.awesome2  # much better
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   13   14   15   16
```

As specified earlier, to construct a matrix all of the same number, you should insert that number as a scalar into `matrix` and let `R` do the work of replicating. And if you want to build a more complicated matrix, you should write the vector in exactly the same form as you want the matrix to look, specify the number of rows and columns, and use the `byrow = TRUE` argument. Just make sure to put a vector in the first argument of `matrix`, but the vector that you use can be built in *any* of the multitude of vector-building methods discussed to this point, such as:

```
c(...)               # vectors the tedious way.
begin:end            # vectors the integer way.
seq(begin, end, by)  # vectors the sequence way.
rep(x, times)        # vectors the repetitive way.
```

Unlike data frames, which have a cryptic mode of `list`, the mode of a `matrix` is *exactly* the same as the mode of the vector inside, unless, of course, that vector is a factor vector, in which case the factor turns back into a character vector.

```
> mode(m.awesome1)  # a numeric ... matrix!
[1] "numeric"
> fact <- d.frame1[1:6 , "blah"]  # some of the factor vector
> m.weird <- matrix(fact, nrow = 2, ncol = 3, byrow = TRUE)
> fact              # fact is a factor vector
[1] my     mother told   me     to     pick
Levels: me mother my pick the to told
> m.weird           # m.weird is now a character vector
     [,1] [,2]     [,3]
[1,] "my" "mother" "told"
[2,] "me" "to"     "pick"
> mode(fact)        # the mode of fact is "numeric"
[1] "numeric"
> mode(m.weird)     # the mode of m.weird is "character"
[1] "character"
```

Once you've constructed a matrix, you might be wondering what to do with it. It does look pretty and rectangular, but that might not be enough.

3.1.2 Using Matrices for Things

One (not quite sure exactly *who* as Psychologists are nice and rarely reveal their disgruntled nature at having to construct matrices) might wonder *why* we are talking about matrices. Well, let's assume that we have a psychological measure with multiple questions, all of which are binary responses (i.e., 0 or 1). With multiple questions, we *can* indicate scores as columns of a data frame, but all of the questions are really all part of the *same* thing, namely the questionnaire. Therefore, it makes more sense to build a matrix representing the scores of participants to all of the items.

```
> scores <- matrix(c(0, 0, 1, 1,
+                    0, 1, 1, 1,
+                    0, 0, 0, 1,
+                    1, 1, 0, 0,
+                    1, 1, 1, 1,
+                    0, 0, 1, 1,
+                    0, 1, 1, 1,
+                    0, 0, 0, 1,
+                    1, 1, 0, 0,
+                    0, 0, 1, 1), nrow = 10, ncol = 4,
+                                 byrow = TRUE)
> colnames(scores) <- c("it1", "it2", "it3", "it4")
> rownames(scores) <- c("p1", "p2", "p3", "p4", "p5",
+                       "p6", "p7", "p8", "p9", "p10")
> scores  # 10 people (down the rows) and 4 items (across the columns)
    it1 it2 it3 it4
p1    0   0   1   1
p2    0   1   1   1
p3    0   0   0   1
p4    1   1   0   0
p5    1   1   1   1
p6    0   0   1   1
p7    0   1   1   1
p8    0   0   0   1
p9    1   1   0   0
p10   0   0   1   1
```

We can then use our (newfangled) matrix to calculate the total score of each person on the measure.

```
> rowSums(scores)  # creates a vector from a matrix
 p1  p2  p3  p4  p5  p6  p7  p8  p9 p10
  2   3   1   2   4   2   3   1   2   2
> colSums(scores)  # creates a vector from a matrix
it1 it2 it3 it4
  3   5   6   8
```

To find the total score of each person, I used two new functions, `rowSums` and `colSums`, which do pretty much what you should assume they will do (by their fairly explicit names)

... sum across the rows and columns of a matrix. You *can* use `rowSums` and `colSums` on data frames, but because data frames are mixed modes, these functions might not work very well.

```
> # But can we add a character to a number?
> try(rowSums(d.frame1), silent = TRUE)[1]
[1] "Error in rowSums(d.frame1) : 'x' must be numeric\n"
```

Summing the rows and columns implicitly *assumes* that every part of the sum is of the same type ... namely numeric. Therefore, by using a form that *assumes* everything is of one type, we reduce the number of potential errors. There are also corresponding functions to find the proportion of people/items that each person/item marked.

```
> rowMeans(scores)  # the proportion of items for each person
  p1   p2   p3   p4   p5   p6   p7   p8   p9  p10
0.50 0.75 0.25 0.50 1.00 0.50 0.75 0.25 0.50 0.50
> colMeans(scores)  # the proportion of persons for each item
it1 it2 it3 it4
0.3 0.5 0.6 0.8
```

The row sums, column sums, row means, and column means are numeric vectors, so that we can manipulate them *like* they are vectors. Remember from the first chapter that `R` has nice vectorized capabilities.

```
> x <- rowSums(scores)  # store the vector
> x + 1                 # add 1 to each element of the vector
 p1  p2  p3  p4  p5  p6  p7  p8  p9 p10
  3   4   2   3   5   3   4   2   3   3
> mat3 <- matrix(x, nrow = 2, ncol = 5, byrow = TRUE)
> mat3                  # a new matrix ... that doesn't make sense?
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    3    1    2    4
[2,]    2    3    1    2    2
```

As a side note, there is a generic function, called `apply` that can do *everything* that `rowSums` and `rowMeans` can do (although a bit slower), but we will not cover that (awesome) function until functions are covered more generally.

Anyway, now we have a matrix of test scores for people to items. See:

```
> scores  # A test!  Yippee!
    it1 it2 it3 it4
p1    0   0   1   1
p2    0   1   1   1
p3    0   0   0   1
p4    1   1   0   0
p5    1   1   1   1
p6    0   0   1   1
p7    0   1   1   1
p8    0   0   0   1
p9    1   1   0   0
p10   0   0   1   1
```

Glorious! Isn't it? Now let's say we want to calculate the split-half reliability of our assessment. We can first divide the items into two parts – the odd items and the even items (or, alternatively, the first half and the second half).

```
> ass1 <- scores[ , seq(from = 1, to = ncol(scores), by = 2)]  # odds
> ass2 <- scores[ , seq(from = 2, to = ncol(scores), by = 2)]  # evens
>
> # Note: We could write out the numbers directly: c(1, 3) and c(2, 4)
> #       but my method is more generic, and if we have a matrix of a
> #       LOT of columns, writing out all of the numbers can be painful,
> #       and it is easy to make mistakes.
```

Once we have the scores, we can find the *total* score for each person on each half of the test by using our newly discovered functions.

```
> tot1 <- rowSums(ass1) # total score to the odd items
> tot2 <- rowSums(ass2) # total score to the even items
```

We can find the *raw* correlation between each half of the assessment by using the soon-to-be discovered correlation function.

```
> u.reli <- cor(tot1, tot2)
> u.reli  # the uncorrected reliability estimate
[1] 0.5267866
```

But we must *correct* the correlation by the Spearman-Brown formula to estimate the reliability of the *entire* assessment.

```
> k <- 2  # two parts to the test
> c.reli <- (k * u.reli) / ( 1 + (k - 1) * u.reli )
> c.reli  # the corrected reliability estimate
[1] 0.6900592
```

> **Useful Advice**: Rather than plugging "2" into the equation, indicating that the full test is twice as long as each half, you should write a generic formula with `k` and assign the "2" to `k`. That way, if we want to estimate the reliability for a hypothetical test of even longer length, we would only have to change the value of `k` *outside* of the function.

```
> k <- 4  # changing one thing ...
> c.reli <- (k * u.reli) / ( 1 + (k - 1) * u.reli )
> c.reli  # ... is much easier than changing 2 ...
[1] 0.8166095
>         # ... or lightbulbs.
```

The major function that we used to calculate the split-half reliability is the `cor` (or correlation) function. If you compute the correlation between two vectors, `cor` will return the result as a single scalar value.

```
> v1 <- c(1, 2, 3, 4, 5, 6)
> v2 <- c(1, 3, 2, 5, 4, 4)
> cor(v1, v2)
[1] 0.7625867
```

However, if you hand `cor` a *matrix* (rather than two vectors), the function no longer returns the correlation as a scalar element, but rather computes *all possible correlations* between *all possible columns* of the matrix and returns the result as a matrix.

```
> M1 <- cbind(v1, v2)  # a matrix from vectors
> cor(M1)              # a matrix of correlations?
          v1        v2
v1 1.0000000 0.7625867
v2 0.7625867 1.0000000
```

Note that the above code chunk illustrates a *different* method of creating matrices. Rather than forming a matrix from a single vector, `cbind` (and `rbind`) forms a matrix by combining several vectors together. You should easily adapt to these functions because they have (pretty much) the same format as the `data.frame` function without the names argument: to form a matrix from a bunch of vectors, just put all of the vectors that you want to combine, in order, into the `cbind` or `rbind` function, and `R` does all the (binding) work.

```
> cbind(v1, v2)  # column bind
     v1 v2
[1,]  1  1
[2,]  2  3
[3,]  3  2
[4,]  4  5
[5,]  5  4
[6,]  6  4
> rbind(v1, v2)  # row bind
   [,1] [,2] [,3] [,4] [,5] [,6]
v1    1    2    3    4    5    6
v2    1    3    2    5    4    4
```

`cbind` and `rbind` are rather generic, because as long as the binds have the correct length and/or number of rows and/or number of columns, you can even bind *matrices* together.

```
> M1             # a matrix
     v1 v2
[1,]  1  1
[2,]  2  3
[3,]  3  2
[4,]  4  5
[5,]  5  4
[6,]  6  4
> cbind(M1, M1)  # a much bigger matrix
```

```
     v1 v2 v1 v2
[1,]  1  1  1  1
[2,]  2  3  2  3
[3,]  3  2  3  2
[4,]  4  5  4  5
[5,]  5  4  5  4
[6,]  6  4  6  4
```

Or even bind data frames to other data frames.

```
> cbind(d.frame1, d.frame2)
    blah eek  muck   blah eek  muck
1     my   1  TRUE     my   1  TRUE
2 mother   2 FALSE mother   2 FALSE
3   told   3 FALSE   told   3 FALSE
4     me   4  TRUE     me   4  TRUE
5     to   5 FALSE     to   5 FALSE
6   pick   6 FALSE   pick   6 FALSE
7    the   7  TRUE    the   7  TRUE
```

Although, `rbind`ing data frames is tricky, because the column names need to match in all of the binded data frames.

```
> names(d.frame2) <- c("blah2", "eek", "much")
> try(rbind(d.frame1, d.frame2), silent = TRUE)[1]  #won't work :(
[1] "Error in match.names(clabs, names(xi)) : \n names do not match
```

Surveying the landscape, we now we have several things that we can build in `R`:

- vectors,
- data frames,
- matrices

Is there any way to build an object of multiple types that are *not* all of the same length? Well, yes - of course there is, and those (amazing) objects are called lists.

3.2 Lists in R

Many functions will return an object with multiple elements, all of different types. For instance, if you run a linear regression (later in the book), `R` will give you some output that you might not find particularly useful.

```
> v1 <- c(1, 2, 3, 2, 3, 4, 5, 2, 3, 1)
> v2 <- c(2, 5, 4, 5, 4, 5, 7, 6, 4, 4)
> comp <- lm(v1 ~ v2)
> comp
```

```
Call:
lm(formula = v1 ~ v2)

Coefficients:
(Intercept)           v2
   -0.03659      0.57317
```

Once you run a regression (or *any* function), you might need to take some of the numbers in the output and do computation-y things with them, but `R` is stubborn. If you try to add 1 to `comp`, `R` will yell at you.

```
> try(comp + 1, silent = TRUE)[1]
[1] "Error in comp + 1 : non-numeric argument to binary operator\n"
```

The `comp` object is a list. A list is just an extrapolation of a data frame to odd object types. You can figure out what is *inside* the list by using the `names` function:

```
> names(comp)
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
```

And if you want to pull something out of the list, you can use the dollar sign operator.

```
> comp$coefficients  # to pull the coefficients out.
(Intercept)          v2
-0.03658537  0.57317073
```

Once you extract the coefficients inside the list, then you have a vector, and you can treat that vector like any other vector.

You can also create a list on your own (similar to how you create a data frame but by using the `list` rather than `data.frame` function). But unlike data frames, each object of the list can be of *any* type (even another list!).

```
> v1        # a vector
 [1] 1 2 3 2 3 4 5 2 3 1
> v2        # a vector
 [1] 2 5 4 5 4 5 7 6 4 4
> M1        # a matrix
     v1 v2
[1,]  1  1
[2,]  2  3
[3,]  3  2
[4,]  4  5
[5,]  5  4
[6,]  6  4
> d.frame1  # a data frame
```

```
    blah eek  muck
1     my   1  TRUE
2 mother   2 FALSE
3   told   3 FALSE
4     me   4  TRUE
5     to   5 FALSE
6   pick   6 FALSE
7    the   7  TRUE
> l.cool <- list(v1 = v1, v2 = v2, M = M1, d.frame = d.frame1)
> l.cool
$v1
 [1] 1 2 3 2 3 4 5 2 3 1

$v2
 [1] 2 5 4 5 4 5 7 6 4 4

$M
     v1 v2
[1,]  1  1
[2,]  2  3
[3,]  3  2
[4,]  4  5
[5,]  5  4
[6,]  6  4

$d.frame
    blah eek  muck
1     my   1  TRUE
2 mother   2 FALSE
3   told   3 FALSE
4     me   4  TRUE
5     to   5 FALSE
6   pick   6 FALSE
7    the   7  TRUE
```

When constructing a list, R thinks: "Well I would like to turn these objects into a data frame, but they have weird dimensions, so let's just put them together into one object to make it easier to call, but let's space the objects out rather than putting them together." You can (as always) figure out the names in your list.

```
> names(l.cool)
[1] "v1"      "v2"      "M"       "d.frame"
```

And you can also pick sub-objects out of your list by using the dollar sign operator.

```
> l.cool$d.frame              # the data frame stored in the list
    blah eek  muck
1     my   1  TRUE
2 mother   2 FALSE
```

```
3   told   3 FALSE
4     me   4  TRUE
5     to   5 FALSE
6   pick   6 FALSE
7    the   7  TRUE
> d.frame1 == l.cool$d.frame  # are they the same?  yup!
     blah  eek muck
[1,] TRUE TRUE TRUE
[2,] TRUE TRUE TRUE
[3,] TRUE TRUE TRUE
[4,] TRUE TRUE TRUE
[5,] TRUE TRUE TRUE
[6,] TRUE TRUE TRUE
[7,] TRUE TRUE TRUE
```

You can also turn a list into a vector by typing the `unlist` function: `R` will (usually, although not always) spread magical pixie dust to make each object a vector, combine each vector into a long vector, and make the type of that vector the most general type in your list.

```
> unlist(l.cool)
          v11           v12           v13           v14
            1             2             3             2
          v15           v16           v17           v18
            3             4             5             2
          v19          v110           v21           v22
            3             1             2             5
          v23           v24           v25           v26
            4             5             4             5
          v27           v28           v29          v210
            7             6             4             4
           M1            M2            M3            M4
            1             2             3             4
           M5            M6            M7            M8
            5             6             1             3
           M9           M10           M11           M12
            2             5             4             4
d.frame.blah1 d.frame.blah2 d.frame.blah3 d.frame.blah4
            3             2             7             1
d.frame.blah5 d.frame.blah6 d.frame.blah7  d.frame.eek1
            6             4             5             1
 d.frame.eek2  d.frame.eek3  d.frame.eek4  d.frame.eek5
            2             3             4             5
 d.frame.eek6  d.frame.eek7 d.frame.muck1 d.frame.muck2
            6             7             1             0
d.frame.muck3 d.frame.muck4 d.frame.muck5 d.frame.muck6
            0             1             0             0
d.frame.muck7
            1
```

Unfortunately, unlike matrices and data frames, a list is dimensionless. The elements of a list do have an order, though, so you can also extract an element by indicating the number of the element that you want to extract rather than calling it by name (with the dollar sign argument). There are two ways to do this: using the single brackets `[objects to extract]` or using the double brackets `[[object to extract]]`. The single bracket method keeps the result in list form.

```
> l2 <- l.cool[2]               # extract a vector
> l2                            # is still is a (short) list?
$v2
 [1] 2 5 4 5 4 5 7 6 4 4
> try(l2 + 2, silent = TRUE)[1] # adding doesn't work on lists
[1] "Error in l2 + 2 : non-numeric argument to binary operator\n"
> l.cool[c(1, 2)]  # but we can take out multiple list elements.
$v1
 [1] 1 2 3 2 3 4 5 2 3 1

$v2
 [1] 2 5 4 5 4 5 7 6 4 4
```

The double bracket method no longer results in a smaller list, but you can only extract *one thing at a time.*

```
> v3 <- l.cool[[2]] # extract a vector
> v3                # it is now a vector!
 [1] 2 5 4 5 4 5 7 6 4 4
> v3 + 2            # adding numbers now WORKS.  Yay!
 [1] 4 7 6 7 6 7 9 8 6 6
> l.cool[[c(1, 2)]] # taking out multiple elements does what?  Weird!
[1] 2
```

Speaking of lists, after you create a bunch of vectors and matrices and data frames and lists, you will have a lot of objects in your R system. When there are many objects stored in R, keeping track of the names becomes difficult. A very useful command to keep track of the objects that you have in R is `ls`, which stands for (appropriately) "list":

```
> ls() # specify ls followed by (), and R will let you know all
 [1] "ass1"       "ass2"       "c.reli"     "c.vec"
 [5] "comp"       "d.frame1"   "d.frame2"   "fact"
 [9] "k"          "l.cool"     "l.vec"      "l2"
[13] "m.awesome1" "m.awesome2" "m.weird"    "M1"
[17] "mat1"       "mat3"       "n.vec"      "ones"
[21] "scores"     "tot1"       "tot2"       "u.reli"
[25] "v1"         "v2"         "v3"         "x"
>      # of the objects that R has in memory.  The resulting names
>      # will be a character vector.  The world has come full circle.
>      # We have, yet again, arrived at a character vector.
```

Before I get "list"-less, once you have all of these (multitude of awesome) objects, you might want to replace particular elements without having to create a data frame or vector again. There are (of course) easy methods of replace elements, and the easiest method is similar to finding and extracting those elements in the first place.

3.3 Indices and Replacement

Let's say you have a vector:

```
> vec <- c(1, 3, 2, 4, 3, 5)
> vec
[1] 1 3 2 4 3 5
```

If you want to see the third entry of that vector, you just stick a "3" into brackets:

```
> vec[3] # show me the 3rd element
[1] 2
```

If you want to *replace* the third entry with a different number, you start your expression in the exact same way,

```
vec[3]
```

but rather than leaving the space to the left of `]` blank, you stick an assignment operator and the number that you want to assign next to `]`.

```
> vec         # show me the entire vector
[1] 1 3 2 4 3 5
> vec[3]      # show me the 3rd element
[1] 2
> vec[3] <- 3 # replace the 3rd element
> vec[3]      # the vector element has changed
[1] 3
> vec         # but the rest of the vector is the same
[1] 1 3 3 4 3 5
```

You can replace *multiple elements* in a similar way, by using multiple indices and slicing. Rather than just changing the 3rd element, let's change the 3rd and 4th elements of `vec` to the same number...

```
> vec               # show me the entire vector
[1] 1 3 3 4 3 5
> vec[c(3, 4)]      # show me the 3/4 element
[1] 3 4
> vec[c(3, 4)] <- 1  # replace both elements with 1
> vec[c(3, 4)]      # yup - they are both 1
[1] 1 1
> vec               # and the rest of the vector is the same
```

```
[1] 1 3 1 1 3 5
```

... or to a different number:

```
> vec                       # show me the entire vector
[1] 1 3 1 1 3 5
> vec[c(3, 4)]              # show me the 3/4 element
[1] 1 1
> vec[c(3, 4)] <- c(3, 4)   # replace with different things
> vec[c(3, 4)]              # now they are different
[1] 3 4
> vec                       # but the rest of the vector is the same
[1] 1 3 3 4 3 5
```

Replacement via indices is rather useful, especially when we reach `for` loops in a few chapters. A complicated use of `for` loops is to create a vector containing a sequence of integers.

```
> c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  # a difficult way
 [1]  1  2  3  4  5  6  7  8  9 10
> 1:10                              # an easy way
 [1]  1  2  3  4  5  6  7  8  9 10
> int <- NULL
> for(i in 1:10){                   # a "for loop" way:
+   int[i] <- i
+ }
> int
 [1]  1  2  3  4  5  6  7  8  9 10
```

The above `for` loop replaces the ith element of `int` with i, which just happens to be our fantastic integers (1 – 10).

Our replacement method also generalizes to matrices and data frames. For instance, if you have a matrix and want to replace the entire second column of that matrix with the number "2", you would just say: "Find all of the rows of the second column and put a 2 in their place."

```
> M2 <- cbind(v1, v2, v1, v2) # creating a matrix
> M2                          # a silly matrix
      v1 v2 v1 v2
 [1,]  1  2  1  2
 [2,]  2  5  2  5
 [3,]  3  4  3  4
 [4,]  2  5  2  5
 [5,]  3  4  3  4
 [6,]  4  5  4  5
 [7,]  5  7  5  7
 [8,]  2  6  2  6
 [9,]  3  4  3  4
[10,]  1  4  1  4
```

```
> M2[ , 2]                    # the second column of M2
 [1] 2 5 4 5 4 5 7 6 4 4
> M2[ , 2] <- 2               # replacing the second column with 2
> M2[ , 2]                    # yup - replaced
 [1] 2 2 2 2 2 2 2 2 2 2
> M2                          # the entire (new) matrix
      v1 v2 v1 v2
 [1,]  1  2  1  2
 [2,]  2  2  2  5
 [3,]  3  2  3  4
 [4,]  2  2  2  5
 [5,]  3  2  3  4
 [6,]  4  2  4  5
 [7,]  5  2  5  7
 [8,]  2  2  2  6
 [9,]  3  2  3  4
[10,]  1  2  1  4
```

And if you wanted to replace the entire third row with the number "3", you would do something similar.

```
> M2[3, ]       # the third row of M2
v1 v2 v1 v2
 3  2  3  4
> M2[3, ] <- 3  # replacing the third row with 2
> M2[3, ]       # yup - replaced
v1 v2 v1 v2
 3  3  3  3
> M2            # the entire (new) matrix
      v1 v2 v1 v2
 [1,]  1  2  1  2
 [2,]  2  2  2  5
 [3,]  3  3  3  3
 [4,]  2  2  2  5
 [5,]  3  2  3  4
 [6,]  4  2  4  5
 [7,]  5  2  5  7
 [8,]  2  2  2  6
 [9,]  3  2  3  4
[10,]  1  2  1  4
```

And if you wanted to replace the element in the fifth row and second column with the number "100", you would make sure to call only that element and not an entire row or column.

```
> M2[5, 2]        # the fifth row and second column
v2
 2
```

```
> M2[5, 2] <- 100 # replacing that element with 100
> M2[5, 2]        # yup - replaced
 v2
100
> M2              # the entire (new) matrix
      v1  v2 v1 v2
 [1,]  1   2  1  2
 [2,]  2   2  2  5
 [3,]  3   3  3  3
 [4,]  2   2  2  5
 [5,]  3 100  3  4
 [6,]  4   2  4  5
 [7,]  5   2  5  7
 [8,]  2   2  2  6
 [9,]  3   2  3  4
[10,]  1   2  1  4
```

Unsurprisingly, you can use logical vectors inside the bracket to replace elements rather than numbers. First, for vectors:

```
> vec                 # our vector
[1] 1 3 3 4 3 5
> vec[vec == 3] <- 2  # replace any element that is 3 with 2
> vec                 # no more 3s :(
[1] 1 2 2 4 2 5
> vec[vec <= 2] <- -1 # replace any element leq 2 with -1
> vec                 # everything is small !!
[1] -1 -1 -1  4 -1  5
```

Second, for matrices:

```
> M2                            # our matrix
      v1  v2 v1 v2
 [1,]  1   2  1  2
 [2,]  2   2  2  5
 [3,]  3   3  3  3
 [4,]  2   2  2  5
 [5,]  3 100  3  4
 [6,]  4   2  4  5
 [7,]  5   2  5  7
 [8,]  2   2  2  6
 [9,]  3   2  3  4
[10,]  1   2  1  4
> M2[M2[ , 2] == 100, 2] <- 99  # a complicated expression :)
> M2                            # but magic :)
      v1 v2 v1 v2
 [1,]  1  2  1  2
 [2,]  2  2  2  5
```

```
 [3,]  3  3  3  3
 [4,]  2  2  2  5
 [5,]  3 99  3  4
 [6,]  4  2  4  5
 [7,]  5  2  5  7
 [8,]  2  2  2  6
 [9,]  3  2  3  4
[10,]  1  2  1  4
```

The second line of the code chunk essentially says:

> Take the second column of the matrix. Mark TRUE when an element of that column is equal to 100 and FALSE otherwise. Replace the TRUEs (there is only one of them) corresponding to locations in the second column of the matrix with 99.

With a complicated expression, you should remember that the constituent parts are pretty simple, so just break the expression into simpler chunks, explain what those simpler chunks mean, and combine the chunks to explain the expression.

Finally, a matrix is *really* a really long vector, as strange as that sounds. The elements of the vector are stacked column by column. Rather than calling elements of a matrix with the standard:

```
matrix[rows to extract, columns to extract]
```

you can extract elements by indicating their ... vector place.

```
matrix[elements to extract]
```

Just keep in mind that R counts down the first column, then the second column, etc.. For example, if your matrix is 10×4 and you want to extract the first element in the second column, that element would be correspond to the 11th element of the matrix: 10 (for the entire first column) + 1 (for the first element of the second column).

```
> M2[1, 2]  # first element in second column
v2
 2
> M2[11]    # first element in second column?
[1] 2
```

Replacing matrix elements by calling the vector number is confusing and rarely used. However, frequently we will want to replace *all* of the elements of a matrix that are equal to some number with a different number. And because R keeps track of the vector place automatically, we can create a logical matrix and put that matrix inside single brackets next to our original matrix. For example, an easy way to replace all of the elements of a matrix that are equal to 2 with another number (say -1) is to create a TRUE/FALSE vector of the *entire matrix* and then use that TRUE/FALSE vector to replace the corresponding elements of the *entire matrix*.

```
> M2[M2 == 2] <- -1  # replace all 2s with -1s in the entire matrix!
> M2                 # magic.  Isn't it!?
      v1 v2 v1 v2
 [1,]  1 -1  1 -1
 [2,] -1 -1 -1  5
 [3,]  3  3  3  3
 [4,] -1 -1 -1  5
 [5,]  3 99  3  4
 [6,]  4 -1  4  5
 [7,]  5 -1  5  7
 [8,] -1 -1 -1  6
 [9,]  3 -1  3  4
[10,]  1 -1  1  4
```

Replacing elements of a matrix by using a logical matrix (in single brackets) is natural, but the only reason that this logical statement works is because a matrix *really* is a very long vector.

3.4 Appendix: Functions Used

Here is a list of important functions from this chapter:

```
# Matrices and Data Frames
data.frame(..., stringsAsFactors = FALSE)   # no factors!
matrix(vector, nrow, ncol, byrow, dimnames) # build a matrix
as.matrix(x)             # create a matrix from something else
cbind(...)               # puts a bunch of vec/mat as columns of a matrix
rbind(...)               # puts a bunch of vec/mat as rows of a matrix

# Operations on Matrices
rowSums(x), colSums(x)   # add the row/column elements
rowMeans(x), colMeans(x) # average the row/column elements

# Generic Operations
cor(x, y, use, method)   # find the correlation between vectors
cor(X, use, method)      # find a correlation matrix

# Lists
list(...)                # put things into a list
unlist(list)             # turn a list into a vector (usually)
list$drawer              # extract the drawer object from list
list[[index]]            # take out the "index" list object
list[objects to extract] # take out objects, and make a list
ls()                     # what objects do we have in R?

# Indices (Again)
vector[stuff to replace]                      <- replacement
data.frame[rows to replace, cols to replace] <- replacement
matrix[rows to replace, cols to replace]     <- replacement
matrix[stuff to extract]
matrix[stuff to replace]                      <- replacement
```

Chapter 4
Files and Tables

You probably noticed that I didn't say: "And now for something completely different" just now. This is simply because I am unable to appear in the show this week. Sorry to interrupt you.
—Monty Python's Flying Circus - Episode 19

4.1 Reading and Writing Files

4.1.1 Reading in External Data

In Chapter 2, I discussed loading data into `R` when those data already belong to some `R` package. We loaded data in the following way:

1. Load the `R` package containing the data using the `library` or `require` function.
2. Load the particular dataset using the `data` function.
3. Access particular variables using the name of the dataset followed by the dollar sign operator *or* attaching the dataset using the `attach` function and calling the variables by name.

For instance, if we want to use the `Ginzberg` Depression data in the `car` package, we would first determine whether or not the package exists on our computer.

```
> library(car)
```

Because `car` probably *does not exist* (although it is a useful package to download), we must then install the package,

```
> install.packages("car", dependencies = TRUE)
> library(car)
```

load the package into `R` once it is installed,

```
> library(car)
```

load the dataset into R once the package is loaded,

```
> data(Ginzberg)
```

and then play around with (or explore) the data.

```
> head(Ginzberg)               # what do the data look like?
  simplicity fatalism depression adjsimp adjfatal  adjdep
1    0.92983  0.35589    0.59870 0.75934  0.10673 0.41865
2    0.91097  1.18439    0.72787 0.72717  0.99915 0.51688
3    0.53366 -0.05837    0.53411 0.62176  0.03811 0.70699
4    0.74118  0.35589    0.56641 0.83522  0.42218 0.65639
5    0.53366  0.77014    0.50182 0.47697  0.81423 0.53518
6    0.62799  1.39152    0.56641 0.40664  1.23261 0.34042
> simp <- Ginzberg$simplicity  # accessing a variable by dollar sign.
> attach(Ginzberg)             # attaching the data.
> head(simplicity)             # now accessing a variable by name.
[1] 0.92983 0.91097 0.53366 0.74118 0.53366 0.62799
> head(simp)                   # they seem to be the same.
[1] 0.92983 0.91097 0.53366 0.74118 0.53366 0.62799
> identical(simplicity, simp)  # see :) and a new function to boot!
[1] TRUE
```

A major problem in the recently reviewed method is that the data *need* to be in a particular package in R before we can explore them. Most data that we need ... are usually not randomly inside of a package in R, as big and profound as the R machine actually is. One may wonder whether there a way of loading external data into R without having to type it? (You may ask). Well, of course, although loading data is not as easy as loading packages. The major function to load data into a `data.frame` is `read.table`, and its most important arguments are as follows:

```
read.table(file, header = FALSE, sep = "",
           na.strings = "NA", nrows = -1, skip = 0,
           fill = FALSE, stringsAsFactors)
```

The first argument is `file`: the particular dataset that you want to explore. If you have the dataset on your computer, you do not know exactly *where* it is, and you want to search for it, the function: `choose.files()` (in Windows) or `file.choose()` (on a Mac) will allow you to select your file interactively. The "choosing interactively" thing is probably the easiest way of finding and picking and loading a file. You can also directly indicate the location of the file using its path **in quotes**. Windows users can either use double backslashes (\\) to separate directories or a single forward slash (/). Mac/Unix users must use the single forward slash. For instance, if you are working with a Mac and your data file is labeled "data.txt" and located on your Desktop, using:

```
file = "~/Desktop/data.txt"
```

as your file path (in quotes) inside of `read.table` would *attempt* to load that particular file. In Windows, I think that the path you would need to get to the Desktop depends on your current location, but something like:

```
file = "C:/Documents and Settings/(username)/Desktop/data.txt"
```

or:

```
file = "C:\\Documents and Settings\\(username)\\Desktop\\data.txt"
```

should work. Notice the quotes surrounding the file name and the double (rather than single) backslash to access directories. Of course the particular path depends (in part) on where R is located. What in the world does that mean?! Well R works from a directory - you can see *which* directory by using the `getwd` function.

```
> getwd() # it's a function, so it needs parentheses ...
[1] "/Users/stevennydick/Documents/School, Work, and Expense... "
>         # but the function has no arguments ... strange ...
```

The directory depends on whether you open an R file (something with the ".R" extension) or the R application first. As you can see, I am working from the directory in which I am writing this book, which (due to my insane organizational scheme) is somewhere in the bowels of my harddrive. You can change the working directory by using the `setwd` function.

```
setwd(dir)
```

In `setwd(dir)`, the `dir` argument is of the same form as the `file` argument of `read.table` (quotes, single forward or double backward slashes, ...) without the *actual* file at the end of the extension.

```
file = "~/Desktop/data.txt" # includes the file, but
dir = "~/Desktop"           # only gets you to a directory
```

As a side note, `getwd` and `setwd`? Isn't R a poetic language!?? *sigh.* Anyway, back to setting directories. The easiest method of setting the working directory is to use the menu item "Change Working Directory," and navigate to where you want R to be located. And once R is located in the same directory as the file, you only need to write the file name (in quotes, of course) and not the entire path. For example, if you want to access the "data.txt" located on the Desktop, you originally had to type:

```
file = "~/Desktop/data.txt"
```

but it might be easier to type:

```
> o.dir <- getwd()          # saving the current directory
> setwd(dir = "~/Desktop")  # changing the dir to file location
> getwd()                   # making sure the dir is correct
[1] "/Users/stevennydick/Desktop"
```

and then set:

```
file = "data.txt"   # don't forget the .txt extension
```

Because all of the files that I need are in the folder in which this book is being written, I should change the working directory back to its original location. Luckily, I had already saved the original path as `o.dir`, so that I can easily type:

```
> setwd(dir = o.dir)
```

and I am back in business.

As a review, if a file is on your computer, the easiest method of accessing that file is by using the `file.choose` function. If the file is in the working directory of `R`, you can call the file name directly (in quotes and not forgetting the file extension). If the file *is not* in the working directory of `R`, you can specify the full path (in quotes, separating the directories by `/`), *or* you can set the working directory to the location of the file (using `setwd` or a menu item) and then directly call the name of the file.

You can also access files on the internet by using the `url` function. For instance, one of the datasets that I have used in a class that I taught is the "ADD" data discussed in the book *Statistical Methods for Psychology* by David C. Howell. The file is at the following address:

```
> # Make sure the path is in quotes.
> site = url("http://www.uvm.edu/~dhowell/methods7/DataFiles/Add.dat")
> # --> You do not need the url function, but it helps with clarity.
```

All you have to do is find the location of the text file online, copy the path of that text file, and assign `file` the path, in quotes, inside the `url` function. We will continue using this dataset in a few paragraphs.

The *second* argument of `read.table` is `header`. The argument, `header`, takes a logical value (TRUE or FALSE) indicating whether the file has a ... um ... header (tricky stuff), which is just a row prior to the data that contains the variable names. If your data *has* a header, and you set `header` to FALSE, then `R` will think that the variable names are observations.

The last of the *really important* arguments is `sep`, which tells `R` how the observations on each row are ... um ... separated. Most of the time, your data will be separated by white space, in which case you can leave `sep = ""`. However, a common method of creating a dataset is through Excel, saving the data as a ".csv" (comma-separated) file (try constructing a ".csv" file in your spare time - commas are fun and swoopy). When trying to read a "csv" file into `R`, you can either use the `read.csv` function (but who wants to memorize a *whole new function*) or set `sep = ","`. The remaining arguments of `read.table` *are* important, but you can read about what they do in your spare time.

Now let's complete our loading of the "ADD" data on the Howell website. The easiest strategy to load the data into `R` is directly via the link. The dataset is located at:

http://www.uvm.edu/~dhowell/methods7/DataFiles/Add.dat

and if you take a look at the data, note that there *is* a header row of variable names, and observations are separated by white space, so:

```
> read.table(file = site,
+            header = TRUE, sep = "")[ , -1]
```

```
   ADDSC Gender Repeat  IQ EngL EngG  GPA SocProb Dropout
1     45      1      0 111    2    3 2.60       0       0
2     50      1      0 102    2    3 2.75       0       0
3     49      1      0 108    2    4 4.00       0       0
4     55      1      0 109    2    2 2.25       0       0
5     39      1      0 118    2    3 3.00       0       0
6     68      1      1  79    2    2 1.67       0       1
7     69      1      1  88    2    2 2.25       1       1
8     56      1      0 102    2    4 3.40       0       0
9     58      1      0 105    3    1 1.33       0       0
10    48      1      0  92    2    4 3.50       0       0
11    34      1      0 131    2    4 3.75       0       0
12    50      2      0 104    1    3 2.67       0       0
13    85      1      0  83    2    3 2.75       1       0
14    49      1      0  84    2    2 2.00       0       0
15    51      1      0  85    2    3 2.75       0       0
16    53      1      0 110    2    2 2.50       0       0
17    36      2      0 121    1    4 3.55       0       0
18    62      2      0 120    2    3 2.75       0       0
19    46      2      0 100    2    4 3.50       0       0
20    50      2      0  94    2    2 2.75       1       1
```

would work. But, unfortunately, running the `read.table` function without assigning it to an object prints the data to your screen but does not save it as a data frame for future use.

```
> ls()   # list the objects in memory!
[1] "Ginzberg" "o.dir"    "simp"     "site"
>        # there do not appear to be any that relate to ADD... ugh!
```

To save a file as a `data.frame` in `R`, you need to actually assign the dataset as your own, aptly named object. For example:

```
> site <- url("http://www.uvm.edu/~dhowell/methods7/DataFiles/Add.dat")
> dat.add <- read.table(file = site,
+                       header = TRUE, sep = "")[ , -1]
> head(dat.add)  # looks nice :)
  ADDSC Gender Repeat  IQ EngL EngG  GPA SocProb Dropout
1    45      1      0 111    2    3 2.60       0       0
2    50      1      0 102    2    3 2.75       0       0
3    49      1      0 108    2    4 4.00       0       0
4    55      1      0 109    2    2 2.25       0       0
5    39      1      0 118    2    3 3.00       0       0
6    68      1      1  79    2    2 1.67       0       1
```

would work, and now we can attach the data frame and call the variables by name *or* use the dollar sign operator.

```
> ADD <- dat.add$ADDSC  # by dollar sign (as it IS a data frame)
> attach(dat.add)       # we can attach data frames
> head(dat.add)
  ADDSC Gender Repeat  IQ EngL EngG  GPA SocProb Dropout
1    45      1      0 111    2    3 2.60       0       0
2    50      1      0 102    2    3 2.75       0       0
3    49      1      0 108    2    4 4.00       0       0
4    55      1      0 109    2    2 2.25       0       0
5    39      1      0 118    2    3 3.00       0       0
6    68      1      1  79    2    2 1.67       0       1
> head(ADD)             # and they look the same.  Yippee!
[1] 45 50 49 55 39 68
```

You might be wondering what happens if we do not remember to let R know that our data contains a header row. Well if we forget to set `header` to TRUE, then our data frame would be as follows.

```
> site <- url("http://www.uvm.edu/~dhowell/methods7/DataFiles/Add.dat")
> dat.add2 <- read.table(file = site, sep = "")[ , -1]  # no header?
> head(dat.add2)                                        # uggggly!
     V2     V3     V4  V5   V6   V7   V8      V9     V10
1 ADDSC Gender Repeat  IQ EngL EngG  GPA SocProb Dropout
2    45      1      0 111    2    3 2.60       0       0
3    50      1      0 102    2    3 2.75       0       0
4    49      1      0 108    2    4 4.00       0       0
5    55      1      0 109    2    2 2.25       0       0
6    39      1      0 118    2    3 3.00       0       0
```

In `dat.add2`, there is an extra row of *new* variable names (`V1` through `V10`), and our *actual* variable names are observations in the data frame. Think about why pretending our variable names are observations is a bad idea. What is the "class" of the original dataset (numeric)? Now, what is the "class" of the new dataset where the variable names are now observations (factor)?

We can also save the data to the Desktop (using a right click and "Save As"). I saved my file as "Add.dat," and now I can read the file into R directly from my machine without having to access the internet.

```
> dat.add <- read.table(file = "~/Desktop/Add.dat", header = TRUE)
> head(dat.add)

  ADDSC Gender Repeat  IQ EngL EngG  GPA SocProb Dropout
1    45      1      0 111    2    3 2.60       0       0
2    50      1      0 102    2    3 2.75       0       0
3    49      1      0 108    2    4 4.00       0       0
4    55      1      0 109    2    2 2.25       0       0
5    39      1      0 118    2    3 3.00       0       0
6    68      1      1  79    2    2 1.67       0       1
```

Not surprisingly, the data frame is exactly the same due to being based on the website. Now that we can read files into R, can we get the data back out again? Of course we can! R is a read-write machine!

4.1.2 Writing Data to an External File

The most common method of writing R data into a file is the `write.table` command, with arguments rather similar to those in `read.table`.

```
write.table(x, file = " ", quote = TRUE, sep = " ",
            na = "NA", row.names = TRUE, col.names = TRUE)
```

In my setup, `x` is the particular object (usually a data frame) that we want to write to an external file, `file` is the name of the file, `quote` is a logical indicating whether we want factors/characters to be in quotes in the external file, `sep` tells R how the data should be spaced, and `row.names/col.names` indicates whether R should include the row names and/or column names (i.e., the participant and/or variable names) as a column and/or row of the external file. Usually, I set `row.names` to FALSE (because the row names are not interesting) but leave `col.names` as TRUE (because the column names are usually the variable names). As in `read.table`, you can let `file` be a path to a new directory or just a name (in quotes, with an extension), in which case the file will be written to your current working directory.

As an example, let's continue working with the "ADD" data.

```
> head(dat.add) # already loaded into R in the previous subsection.
  ADDSC Gender Repeat  IQ EngL EngG  GPA SocProb Dropout
1    45      1      0 111    2    3 2.60       0       0
2    50      1      0 102    2    3 2.75       0       0
3    49      1      0 108    2    4 4.00       0       0
4    55      1      0 109    2    2 2.25       0       0
5    39      1      0 118    2    3 3.00       0       0
6    68      1      1  79    2    2 1.67       0       1
```

If we want to look at the data *outside* of R, we can write it to an external ".txt" file, separating the values by spaces.

```
> # --> the object is dat.add,
> # --> we will save the object as "Add.txt" in the current directory,
> # --> factors (there are none) will not have quotes,
> # --> data will be separated by spaces,
> # --> we will remove the row names but keep the variable names.
> write.table(dat.add, file = "Add.txt",
+             quote = FALSE, sep = " ",
+             row.names = FALSE, col.names = TRUE)
```

If you go into your working directory, you should see the file "Add.txt," and if you double click that file, you should see the data. We can (of course) read the data *right* back into R.

```
> # The data looks exactly the same as it did before.
> dat.add3 <- read.table("Add.txt",
+                        header = TRUE, sep = "")
> head(dat.add3)  # magic!??
  ADDSC Gender Repeat  IQ EngL EngG  GPA SocProb Dropout
1    45      1      0 111    2    3 2.60       0       0
2    50      1      0 102    2    3 2.75       0       0
3    49      1      0 108    2    4 4.00       0       0
4    55      1      0 109    2    2 2.25       0       0
5    39      1      0 118    2    3 3.00       0       0
6    68      1      1  79    2    2 1.67       0       1
```

Unfortunately, a text file with a bunch of spaces is pretty difficult to parse, and we might want to edit our data in Excel. The easiest method of writing a file that is Excel compatible is by using the ".csv" extension and separating the data by commas.

```
> write.table(dat.add3, file = "Add.csv",
+             quote = FALSE, sep = ",",
+             row.names = FALSE, col.names = TRUE)
```

You should be able to read the "Add.csv" file back into Excel, either by double clicking the file or by double clicking Excel and opening the ".csv" file. Once you have the file in Excel, you can alter values, rename columns, etc., save the big table as a ".csv" and then read that big table *right* back into `R` by only changing the `sep` argument of `read.table`.

```
> dat.add3 <- read.table("Add.csv",
+                        header = TRUE, sep = ",")
```

Alternatively, you *can* use the functions `read.csv` and `write.csv` to do all of your comma-separated reading and writing, but I find using the `read.table` and `write.table` functions and change the `sep` argument is much easier. Due to the flexibility of `R`, you can read and write just about anything, including SPSS, SAS, Stata files. Functions allowing you to access those files are in the `foreign` library, as SPSS, SAS, and Stata are foreign concepts to the awesome `R` user.

When you want to save multiple `R` objects (including lists), reading and writing to files is a little tricker. However, it is only a *little* trickier, and requires a function with the most appropriate name: `dump`. To save a bunch of objects in one file, all you have to do is specify their names in a vector inside the `dump` function, and indicate the (desired) name of the file.

```
> x <- c(1, 2, 3, 4, 5)                       # a numeric vector
> y <- matrix(c("a", "b", "c", "d"), nrow = 2) # a character matrix
> z <- list(x, y)                             # a list
> dump(list = c("x", "y", "z"), file = "Awesome.txt")
> # Note: make sure the object names are in quotes.
>
> rm(x, y, z)               # removing x, y, z
> try(x, silent = TRUE)[1]  # does x exist? no!
[1] "Error in try(x, silent = TRUE) : object 'x' not found\n"
```

```
> try(y, silent = TRUE)[1]  # does y exist? no!
[1] "Error in try(y, silent = TRUE) : object 'y' not found\n"
> try(z, silent = TRUE)[1]  # does z exist? no!
[1] "Error in try(z, silent = TRUE) : object 'z' not found\n"
```

After dumping your objects into an external file, you can read those objects back into R using the source function, and even if you removed those objects from R's memory (using the rm function), they magically reappear. Oh objects! How I missed you dearly.

```
> source("Awesome.txt")
> x
[1] 1 2 3 4 5
> y
     [,1] [,2]
[1,] "a"  "c"
[2,] "b"  "d"
> z
[[1]]
[1] 1 2 3 4 5

[[2]]
     [,1] [,2]
[1,] "a"  "c"
[2,] "b"  "d"
```

Unfortunately, the dumped file does not have a nice format. It is (maybe) R readable, but it does not help humans with presentation, understanding, and analysis. An alternative method of saving *readable* data is by using the sink function. Basically the sink function diverts the input/output to an external file. A basic use of the sink function is to print a few objects for human consumption.

```
> sink("Awesome2.txt")  # the file you want to write stuff to
> print(x)              # the stuff you want to save,
> print(y)              # surrounded by the "print" function,
> print(z)              # ... <- later
> sink()                # an empty sink to close the con
```

After you sink the data into an external file, the data is copy and paste readable by humans, but it is nearly impossible to read the data back into R. The print function (by the way) is needed to actually place the objects into the file, though we will talk about print more later. Also, note that many R practitioners recommend saving R objects using the save (and *not* the dump) functions, as saveing is more reliable than dumping. However the format of the saved file will only be R compatible but completely unreadable by humans (unless they have sufficient decoding-computer-symbols-skills).

In any case, reading and writing data is kind of a pain in any computer language, so we will stop talking about it now and start talking about something completely different ... tables, categorical variables, and categorical relationships.

4.2 Categorical Variables in R

4.2.1 Tables and Boxplots

An initially tricky problem in R is trying to find the mode of a set of data. The most efficient method of calculating the mode is by using the `table` function. *Regardless* of the *type* of data, surrounding a vector with `table` results in a "vector" of counts at each level of the vector.

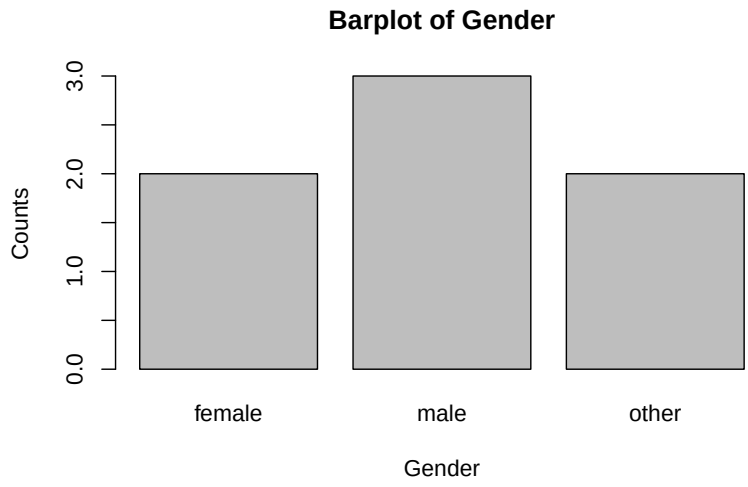
```
> dat1 <- c(1, 2, 3, 3, 3, 3, 2, 2, 5, 6, 7, 4, 2, 3)
> table(dat1)
dat1
1 2 3 4 5 6 7
1 4 5 1 1 1 1
```

As you can probably tell, the *names* of the vector elements are the possible values of the data, and the actual *elements* are the counts/frequencies for each value. We can also form tables from character vectors.

```
> dat2 <- c("male", "female", "female", "male", "other", "other", "male")
> table(dat2)
dat2
female   male  other
     2      3      2
```

R is pretty neat and sorts the vector either numerically (if your data are numbers) or alphabetically (if your data are character strings). Once your data are in `table` form, you can stick the table into a barplot (using `xlab` to label the x-axis, `ylab` to label the y-axis, and `main` to label the plot - these arguments work on *almost* any plot).

```
> barplot(table(dat2),
+         xlab = "Gender", ylab = "Counts", main = "Barplot of Gender")
```

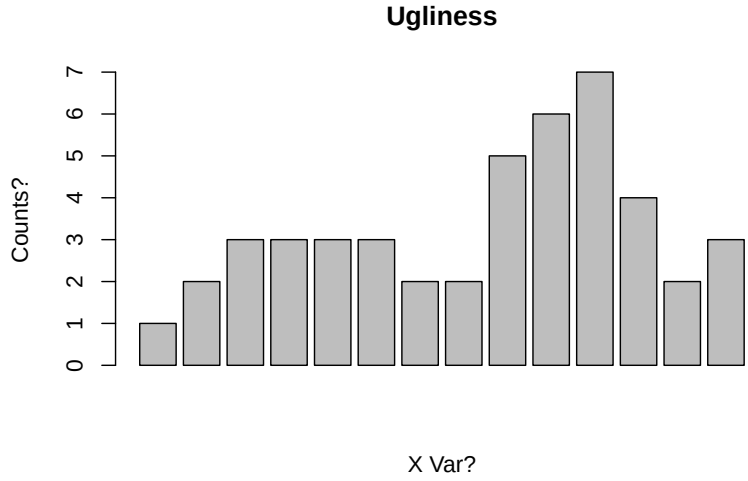


A common mistake in R-barplot construction is trying to use the original vector and *not* a numeric summary via the `table` function.

> **Warning**: When you construct a barplot, you must input your data as a vector. The *names* of that vector must refer to the unique values of the variable, and the *values* of that vector must refer to the counts. You *cannot* enter the vector itself but must use a summary of the vector, and the easiest way of obtaining a correct summary is via the `table` function.

```
> # --> Does not work with a character vector
> try(barplot(dat2,
+             xlab = "Gender?", ylab = "Counts?", main = "Ugliness"),
+     silent = TRUE)[1]
[1] "Error in -0.01 * height : non-numeric argument to binary operator\n"
```

```
> # --> Is not correct with a numeric vector
> barplot(dat1,
+         xlab = "X Var?", ylab = "Counts?", main = "Ugliness")
```



Frequently, you will need to work with proportions rather than counts. To work with proportions, you can insert your original (count) table into the `prop.table` function.

```
> g.tab <- table(dat2)     # to make it easier to call the table
> prop.table(g.tab)        # stick a table and NOT the original vector
dat2
   female      male     other
0.2857143 0.4285714 0.2857143
> sum( prop.table(g.tab) ) # should be 1.  Why?
```

```
[1] 1
```

You can also attach the total count of your data to the table by using the `addmargins` function.

```
> addmargins(g.tab)          # I'm not quite sure what to do with "Sum"
dat2
female   male  other    Sum
     2      3      2      7
```

And you can find the total count (i.e., the marginal count of the original variable) by using the `margin.table` function.

```
> margin.table(g.tab)
[1] 7
```

Of course, with one variable, the functions `sum` (on the table) or `length` (on the original vector) would also find the total count, and both have many fewer letters. Yet all of these recently discussed table functions generalize to two variables. For instance, let's load the data set `donner` in the package `alr3` (which you can download by typing `install.packages("alr3")` into your `R` session).

```
> data(donner, package = "alr3")
> head(donner)
                        Age Outcome    Sex Family.name Status
Breen_Edward_            13       1   Male       Breen Family
Breen_Margaret_Isabella   1       1 Female       Breen Family
Breen_James_Frederick     5       1   Male       Breen Family
Breen_John               14       1   Male       Breen Family
Breen_Margaret_Bulger    40       1 Female       Breen Family
Breen_Patrick            51       1   Male       Breen Family
```

The `donner` dataset has five variables: `Age`, `Outcome`, `Sex`, `Family.name`, and `Status`. `Outcome` refers to whether or not a particular person survived the westward migration, and `Status` indicates whether the person was a family member, hired, or single. We can use the `table` function to determine survival rates based on gender and family membership.

```
> # Coding Outcome as a factor ...
> # --> to make it easier to see survival status for particular people.
> donner$Outcome <- factor(donner$Outcome, levels = c(0, 1),
+                          labels = c("died", "survived"))
> # Attaching the dataset ...
> # --> to make it easier to call the variables by name:
> attach(donner)
> # Printing attributes of the data:
> table(Outcome, Status) # what are the rates by membership?
          Status
Outcome    Family Hired Single
  died         25    12      5
  survived     43     6      0
```

```
> table(Outcome, Sex)    # what are the rates by gender?
          Sex
Outcome    Female Male
  died         10   32
  survived     25   24
```

The R functions prop.table and addmargins fully generalize from the one-way table to summarize multiple categorical variables.

```
> # Saving certain tables as R objects:
> tab.stat <- table(Outcome, Status)  # to make it easier to call
> tab.sex  <- table(Outcome, Sex)     # to make it easier to call
> # Printing attributes of those tables:
> addmargins(tab.stat) # marginal frequencies of tab.stat
          Status
Outcome    Family Hired Single Sum
  died         25    12      5  42
  survived     43     6      0  49
  Sum          68    18      5  91
> addmargins(tab.sex)  # marginal frequencies of tab.sex
          Sex
Outcome    Female Male Sum
  died         10   32  42
  survived     25   24  49
  Sum          35   56  91
> prop.table(tab.stat) # as proportions?
          Status
Outcome        Family      Hired     Single
  died     0.27472527 0.13186813 0.05494505
  survived 0.47252747 0.06593407 0.00000000
> prop.table(tab.sex)  # as proportions?
          Sex
Outcome       Female      Male
  died     0.1098901 0.3516484
  survived 0.2747253 0.2637363
```

As before, the sum of all of the entries in the proportion table should sum to 1:

```
> sum( prop.table(tab.stat) ) # should be 1
[1] 1
> sum( prop.table(tab.sex) )  # should be 1
[1] 1
```

The function margin.table does not generalize in exactly the same manner as the addmargins and prop.table functions, at least to this point in the example. Rather than just having *one* argument, indicating the table in which to find the margins, the margin.table function has *two* arguments: (1) the table in which to find the margins, and (2) the particular margin to calculate. So, if one wants to calculate the "row margins,"

he or she should plug a "1" into the `margin` argument, and if one wants to calculate the "column margins," he or she should plug a "2" into the `margin` argument.

```
> margin.table(tab.stat, margin = 1)  # row margins
Outcome
    died survived
      42       49
> margin.table(tab.sex,  margin = 1)  # row margins
Outcome
    died survived
      42       49
> # These results should be the same ... are they?
```

... and ...

```
> margin.table(tab.stat, margin = 2)  # column margins
Status
Family  Hired Single
    68     18      5
> margin.table(tab.sex,  margin = 2)  # column margins
Sex
Female   Male
    35     56
> # These results (of course) should not be the same.
```

Any precocious statistics student might (at this point) be wondering: "what about the conditional distributions? How can we find the conditional distributions using these commands?" My immediate reaction is: "bugger off," as they are pretty easy to calculate based on the `addmargins` function.

> **Note**: Conditional distributions are found by dividing the joint counts by the (appropriate) marginal sums. So the conditional distribution of being female given that you died is 10/42: the number of female dead people (10) divided by the total number of dead people (42).

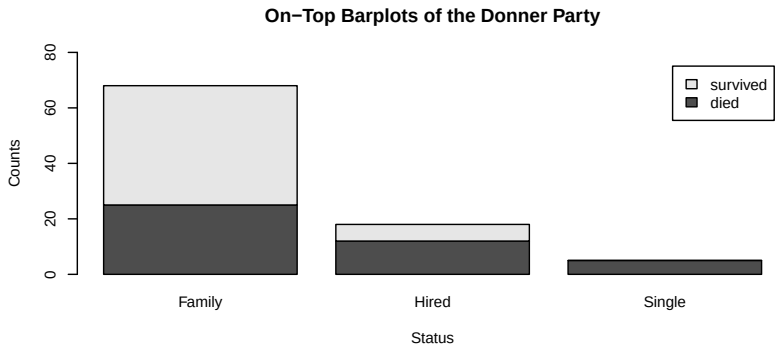
But lazy people (and I *am* one) can find conditional distributions/probabilities by using the `margin` argument (from before) in the `prop.table` function:

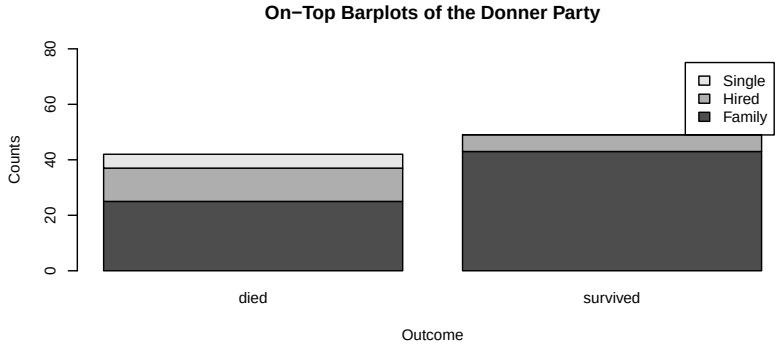
```
> prop.table(tab.stat, margin = 1) # conditional on rows
> prop.table(tab.sex, margin = 1)  # conditional on rows
> prop.table(tab.stat, margin = 2) # conditional on columns
> prop.table(tab.sex, margin = 2)  # conditional on columns
```

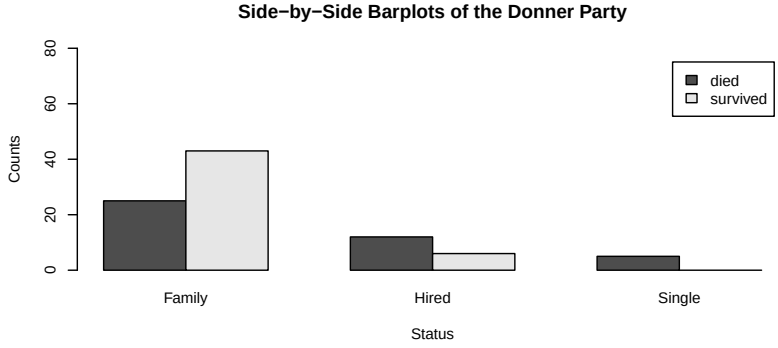
So the generalization of the `margin.table` and the `prop.table` is more similar than originally thought. In your spare time, you can figure out whether the rows and/or columns of these tables sum to 1.

You can also plot barplots using two-way tables (if you like). A few of your options are as follows.

```
> par(mfrow = c(3, 1))
> # 1) The columns of your data are on the x-axis:
> barplot(tab.stat, ylim = c(0, 80),
+         xlab = "Status", ylab = "Counts",
+         main = "On-Top Barplots of the Donner Party",
+         legend.text = TRUE)
> # 2) The rows of your data are on the x-axis:
> barplot(t(tab.stat), ylim = c(0, 80),
+         xlab = "Outcome", ylab = "Counts",
+         main = "On-Top Barplots of the Donner Party",
+         legend.text = TRUE)
> # 3) The bars are side-by-side rather than on top:
> barplot(tab.stat, ylim = c(0, 80),
+         xlab = "Status", ylab = "Counts",
+         main = "Side-by-Side Barplots of the Donner Party",
+         beside = TRUE, legend.text = TRUE)
> par(mfrow = c(1, 1))
```







I can never remember which part of the table goes on the x-axis and how the bars are segmented, so I usually end up trying a few things before I settle on a barplot that I like. Note that `t` is the "transpose" function and flips the rows and columns of a matrix (so that we can easily change the organization of the plot).

4.2.2 Bayes Theorem

A final task for the statistically savvy would be to use probabilities in Bayes theorem. Let's say we know the probability of dying given that one is female, the probability of surviving given that one is male, and the probability of being female. We can save the appropriate probabilities using indices in `R`.

```
> # We are conditioning on the columns (male/female):
> cond.sex <- prop.table(tab.sex, margin = 2)
> (p.d_fem  <- cond.sex[1, 1]) # probability of dying given female
[1] 0.2857143
> (p.s_male <- cond.sex[2, 2]) # probability of surviving given male
[1] 0.4285714
> # We also need the probability of being female:
> marg.sex <- margin.table(prop.table(tab.sex), margin = 2)
> (p.fem    <- marg.sex[1])    # probability of being female
   Female
0.3846154
```

Bayes Theorem states:

$$\Pr(B|A) = \frac{\Pr(A|B)\Pr(B)}{\Pr(A|B)P(B) + \Pr(A|\neg B)\Pr(\neg B)}$$

Therefore, if we wanted to find the probability of being female given that one has died (assuming that we cannot calculate this probability directly from a table), then we can use the information saved above. Letting A be died, $\neg A$ be survived, B be female, and $\neg B$ be male. Then

$$\begin{aligned}
\Pr(A|B) &= 0.29 \\
\Pr(A|\neg B) &= 1 - \Pr(\neg A|\neg B) = 1 - 0.43 = 0.57 \\
\Pr(B) &= 0.38 \\
\Pr(\neg B) &= 1 - \Pr(B) = 1 - 0.38 = 0.62
\end{aligned}$$

Finally, we can plug those (saved) numbers into R to find the inverse probability.

```
> p.fem_d <- (p.d_fem*p.fem) / (p.d_fem*p.fem + (1-p.s_male)*(1-p.fem))
> p.fem_d
   Female
0.2380952
```

Of course, if we have the entire set of data, we can check our calculation by using the proportion table to find the conditional probability in the other direction.

```
> cond.surv <- prop.table(tab.sex, margin = 1)
> cond.surv[1, 1]  # This should be the same as
[1] 0.2380952
> p.fem_d          # this ...
   Female
0.2380952
```

Methods based on Bayes' theorem are infiltrating (err ... becoming a more important part of) psychological data analysis, so you should be aware of the basic ideas even though this book will not pursue those ideas further.

Now that we have covered the basic principles and functions for R coding, in the next few chapters, I will introduce specialized functions that can assist in data analysis and explain methods used in writing one's own analysis routine.

4.3 Appendix: Functions Used

Here is a list of important functions from this chapter:

```
# Reading in External Data
read.table(file, header, sep, ...)  # read in a data frame
file.choose()       # find file - Mac/Windows
choose.files()      # fine file = Windows
url(description)    # open the internet connection
getwd()             # where is R currently located?
setwd(dir)          # change the location of R

# Writing External Data
write.table(x, file, quote, sep,   # write a table to an
            row.names, col.names)  # external file.
dump(list, file)    # put a bunch of objects in a txt file
source(file)        # read in R commands/functions/objects
sink(file)          # divert connection to an external file
sink()              # close the external connection
print(object)       # ... um ... print an object :)

# Listing Objects
ls(...)             # what objects are in R?
rm(...)             # remove objects from R's memory

# Categorical Variables
table(x, y)         # contingency table (if you include y)
                    # or counts (if you don't include y)
prop.table(tab, margin)   # joint or conditional proportions
margin.table(tab, margin) # sum across the "margin" of tab
addmargins(tab)     # add marginal counts to a table

# Graphing
barplot(tab,        # make a barplot with (maybe) a legend
        beside,
        legend.text, ... )

# Misc Functions
t(x)                # flip the rows and columns of x
```

Part II

More Advanced Features of R

Chapter 5
Samples and Distributions

> Waitress: *You can't have egg, bacon, Spam and sausage without the Spam.*
> Mrs. Bun: *Why not?!*
> Waitress: *Well, it wouldn't be egg, bacon, Spam and sausage, would it?*
> —Monty Python's Flying Circus - Episode 25

5.1 Sampling from User Defined Populations

A benefit to using R is the ability to play controller-of-the-universe (which, as it turns out, is my main draw to this particular statistical software). You can create a population and then take samples from the population. For instance, if you have a Bernoulli distribution where

$$f(x; p) = \begin{cases} p & x = 1 \\ 1 - p & x = 0 \end{cases}$$

then sampling from that distribution requires only need to setting up your population in a vector, indicating the probability of 0 and 1 in another vector, and using the `sample` function in R.

```
> set.seed(91010)
> p <- .8
> x <- c(0, 1)
> samp <- sample(x = x, size = 1000, replace = TRUE, prob = c(1 - p, p))
```

The first part of the above code chunk involved setting my seed (surprisingly, *not* a euphemism). When R simulates random numbers, it does not simulate them in a *fully* random fashion but "pseudo-randomly." Numbers are simulated according to an algorithm that takes a long time to complete so that the samples appear to be random. We frequently want to repeat a particular simulation in *exactly the same way*. That is to say, even though we want to "randomly sample," we might want a future "random sample" to be identical to a past "random sample." For example, if statisticians want to reproduce earlier research based on simulations, they often need to retrieve the *exact same* numbers. By setting

my seed (using the `set.seed` function with a particular integer inside), I can retrieve identical samples from my population.

```
> set.seed(888)  # setting the seed
> sample(x = c(0, 1), size = 10, replace = TRUE)
 [1] 0 0 0 1 1 0 0 0 0 1
> set.seed(888)  # setting the same seed
> sample(x = c(0, 1), size = 10, replace = TRUE)
 [1] 0 0 0 1 1 0 0 0 0 1
> set.seed(888)  # setting the same seed
> sample(x = c(0, 1), size = 10, replace = TRUE)
 [1] 0 0 0 1 1 0 0 0 0 1
> set.seed(999)  # setting a different seed
> sample(x = c(0, 1), size = 10, replace = TRUE)
 [1] 0 1 0 1 1 0 1 0 0 1
>
> # Notice that the last sample is different from the first three.
```

And this "seed setting" does not *just* work with a simple sample (try saying that ten times fast) of 0s and 1s but with *any random sample*.

After setting the seed, I then picked a particular probability and let `x` be a vector of 0 and 1. The probability that $x = 1$ is p, and the probability that $x = 0$ is $1 - p$. I finally ran the `sample` function, which takes four arguments:

1. `x`: A vector representing the population, or (at least) the unique values *in* the population.

2. `size`: The size of the sample one should take from the population. Notice that, in my example, `x` is a vector of length 2 and `size` is 1000 (much larger than 2). Therefore, the `x` that I used does not represent the entire population but the unique values in the population that units can take.

3. `replace`: A logical indicating whether one is sampling with or without replacement from the population. Generally, if `x` is a vector of the *entire* population, then `replace = FALSE`, but if `x` only indicates the *unique* values in the population then `replace = TRUE`.

4. `prob`: A vector (of the same length as `x`) representing the probability of selecting each value of `x`. If the user does not specify a vector for `prob`, then `R` assumes that each value is equally likely to be selected.

In our case, we sampled 1000 0s and 1s (with replacement) using specified probabilities for 0 and 1. Notice that because `x = c(0, 1)`, the *first* element of `prob` is the probability of selecting a 0 (or $1 - p$), and the *second* element of `prob` is the probability of selecting a 1 (or p).

After sampling scores, we should examine our sample.

```
> head(samp, n = 20)  # what does samp contain?
 [1] 1 1 1 0 1 1 1 1 1 0 1 0 1 1 1 1 0 1 0 1
```

```
> mean(samp)          # what is the average value of samp?
[1] 0.793
> var(samp)           # what is the variability of samp?
[1] 0.1643153
```

Considering that the expected value of a Bernoulli random variable is p (with variance $p(1-p)$), does our sample result in a mean and variance *close to expectation*?

```
> p                   # the expected value (close to mean(samp)?)
[1] 0.8
> p*(1 - p)           # the variance (close to var(samp)?)
[1] 0.16
```

Of course, we do not need to only sample *numbers*. We can sample strings corresponding to categories of a categorical variable.

```
> set.seed(9182)              # so we can repeat
> x <- c("heads", "tails")    # our possible outcomes
> # Determining the sample (as before):
> # --> Note that we did not specify "prob"
> # --> Both values are assumed equally likely.
> samp <- sample(x = x, size = 1000, replace = TRUE)
> head(samp, n = 20)         # a few values in our sample.
 [1] "tails" "tails" "heads" "heads" "heads" "heads" "tails"
 [8] "heads" "tails" "tails" "tails" "heads" "tails" "tails"
[15] "heads" "tails" "tails" "tails" "tails" "heads"
```

We can also check to make sure that the probability of heads is what we would expect based on the Bernoulli distribution.

```
> mean(samp == "heads")     # the mean of our sample
[1] 0.499
> var(samp == "heads")      # the variance of our sample
[1] 0.2502492
> p <- .5                   # the probability of scoring heads
> p                         # the expected value
[1] 0.5
> p*(1 - p)                 # the population variance
[1] 0.25
```

> **Note**: `samp == "heads"` turns a character vector into TRUEs and FALSEs, and `mean()`/`var()` turns the TRUEs into 1s and the FALSEs into 0s. This is a useful trick for how to find the probability (and variance) of being in a particular category.

We (of course) are not forced to only sample 0s and 1s (or "heads" and "tails"). We can construct `x` to be as long as we want. For instance, if `x` is a vector of $1 - 6$, then sampling values of `x` is like rolling dice.

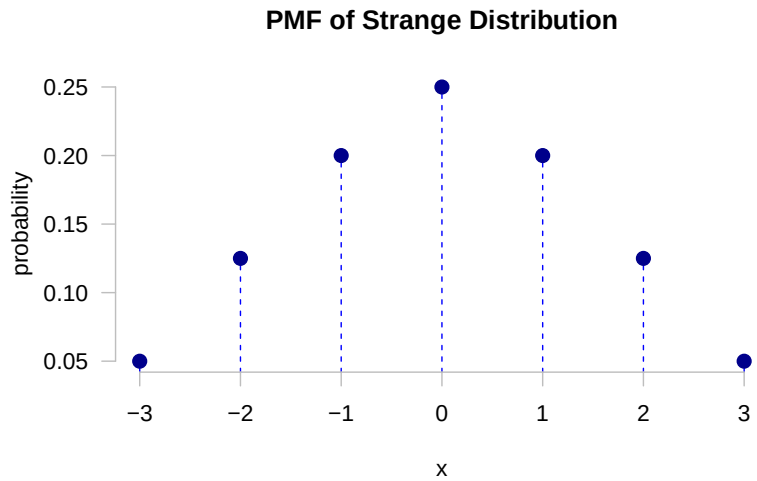
```
> set.seed(91353)
> x <- 1:6                                          # set the die!
> y1 <- sample(x, size = 1)                         # roll a die
> y2 <- sample(x, size = 1) + sample(x, size = 1)   # add two dice rolls.
> y3 <- sum( sample(x, size = 2, replace = TRUE) ) # add two dice rolls.
> # Display the values for each:
> y1
[1] 3
> y2
[1] 7
> y3
[1] 2
```

Or we can create a new distribution,

```
> x <- -3:3                          # the possible values
> p1 <- c(1, 2.5, 4, 5, 4, 2.5, 1) # a weird assortment of numbers
> p  <- p1 / sum(p1)                 # turning numbers into probabilities
> sum(p)                             # making sure we have a distribution
[1] 1
```

plot that distribution (using the `plot` function),

```
> plot(x = x, y = p,
+      xlab = "x", ylab = "probability",
+      main = "PMF of Strange Distribution",
+      type = "h", lty = c(2, 2), col = "blue", axes = FALSE)
> axis(1, col = "grey")
> axis(2, col = "grey", las = 1)
> points(x = x, y = p,
+        pch = 20, cex = 2, col = "darkblue")
```

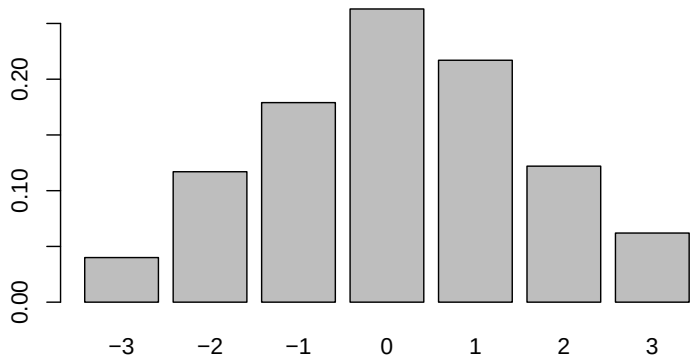


sample 1000 scores from the distribution,

```
> set.seed(974)
> samp2 <- sample(x, size = 1000, replace = TRUE, prob = p)
> head(samp2, n = 100)          # what does the sample look like?
  [1]  0  0 -2 -1 -2  2  2  1  0 -1 -2 -2  1  0  0 -1  0 -2
 [19]  1 -2  2 -2  0 -2  0  1  2  3  1  0 -2 -3  0  0  1  0
 [37] -2 -1  1 -1 -1  1 -1  0  2  2  1  1  0  3  1  0  1  1
 [55]  1  1  1  0  1  2  1  2  2 -1  1  0 -1  0 -2  0  0  3
 [73]  0 -1  1  1  2  3  0 -2  0  0 -2 -1  1  1  1 -1 -2 -2
 [91] -2  1  0  3  2 -3  1  2  0  1
```

and form a barplot of the sample.

```
> # Finding the counts and dividing by the counts by the length
> # to obtain proportions.  What is another way of doing this?
> # Hint: It was discussed in a previous chapter.
> barplot( table(samp2)/length(samp2) )
```



The barplot should look pretty similar to the original PMF plot.

In the above example, only two of the functions have I yet to describe: `plot` and `points`. Moreover, *most* of the arguments that I used in `plot` can be applied to *any* of the plot-type functions (such as `hist` or `barplot`). Not surprisingly, `x` and `y` are the values on the x-axis and y-axis, respectively. As we had discussed in a previous chapter, `xlab` and `ylab` are the x-axis and y-axis labels, and `main` is the title of the plot. The argument `type` is a character (surrounded by quotes) indicating the *type* of plot: `"p"` indicates "points", `"l"` indicates "lines", `"b"` indicates "both points and lines", `"h"` indicates "histogram like lines", and `"n"` results in an empty plot. There are other possible values for `type`, but the arguments just listed are the only ones that I ever use. Therefore, by including the argument `type = "h"`, `R` displays vertical lines from 0 to the (x, y) value on the plot. The

argument `lty` represents the *type* of line (1 corresponds to a solid line, 2 corresponds to a dashed line, etc.), `col` is (obviously) the color of the plot, `pch` is the point type (either a number without quotes or a character in quotes). You can see how to call `pch` in the `points` help file.

```
> ?points
```

Finally, `cex` is a number indicating the *size* of the points, relative to the default of `cex = 1`. Note that `points` is an "add-to-plot" function, the annoyance of which I discuss in a note near the end of this chapter. The other arguments/commands I will let you to explore on your own.

5.2 Sampling from Built-In Populations

5.2.1 Attributes of All Distributions

Fortunately, in case that you don't know how to magically create a population (which is possible unless you have a dragon with hallucinogenic capabilities), `R` contains built-in populations. Every distribution in `R` includes a set of four functions. Given a distribution with the name of "awesome" (as the one common fact to all distributions is that they are awesome), those functions are as follows.

```
dawesome(x, parameters)
pawesome(q, parameters, lower.tail)
qawesome(p, parameters, lower.tail)
rawesome(n, parameters)
```

The `parameters` "argument" refers to a sequence of parameters (with corresponding names) separated by commas. For all distributions, the function (1) `dawesome` calculates the density of the distribution at `x`. For discrete distributions, the "density" at `x` refers to the probability of scoring `x` on that distribution with those parameters. The function (2) `rawesome` takes a random sample (of size `n`) from the "awesome" distribution. The result of `rawesome` will be a vector of size `n` (similar to using the `sample` function with a vector of "all possible values" and `prob` corresponding to the result of `dawesome`). The functions (3) `pawesome` and (4) `qawesome` are a little bit more complicated. If you know the term "cumulative density function" (CDF), `pawesome` is the value of the CDF for a particular score, and `qawesome` is the inverse of `pawesome`. When `lower.tail = TRUE` (the default), `pawesome` finds the probability of having a score "at or below" `q`, and `qawesome` finds the value such that `p` is the probability of being "at or below" it. When `lower.tail = FALSE`, `pawesome` finds the probability of being "above" `p`, and `qawesome` finds the value such that `p` is the probability of being "above" it. If all of this is confusing, `pawesome` and `qawesome` are even more annoying when working with discrete distributions.

> **Warning**: When working with discrete random variables, `pawesome` with `lower.tail = FALSE` finds the probability of scoring *above* `q`. What this means is that if you are trying to find the probability of throwing *at least* 6 free-throws (out of 10 total free throws), then the `q` that will give you the correct probability is *not* 6 ... but 5!

Because the probability of being above a particular score is 1 minus the probability of being at or below that score, `pawesome` with `lower.tail = FALSE` is *exactly equal to* 1 minus the same `pawesome` but with `lower.tail = TRUE`.

Now that I introduced the generic "awesome" distribution, I probably should introduce *actual* distributions, all of which have `d`, `p`, `q`, and `r` functions (the names of the `parameters` are in parentheses).

- `norm(mean, sd)`: A normal distribution with a mean of `mean` and a standard deviation of `sd`.
- `binom(size, prob)`: A binomial distribution after sampling `size` Bernoulli random variables, all of which have probability `prob`.
- `unif(min, max)`: A (continuous) uniform distribution between `min` and `max`.
- `cauchy(location, scale)`: A Cauchy distribution with a location and scale parameter. You don't need to know what each of these are, but `location` is *like* a mean, and `scale` is *like* a standard deviation. A Cauchy distribution (oddly) does not have a mean nor a standard deviation.
- `t(df)`: A t-distribution with `df` degrees of freedom.
- `F(df1, df2)`: An F-distribution with numerator degrees of freedom of `df1` and denominator degrees of freedom of `df2`.
- `chisq(df)`: A χ^2 distribution with `df` degrees of freedom.

Note that all of the common distributions (including those described above) can be found with the following *R*-yelp:

```
> ?Distributions
```

5.2.2 Examples of Distributions in Action

The Binomial Distribution

Any set of the most important distributions include the binomial distribution (as an example of a discrete distribution) and the normal distribution (as an example of a continuous distribution). Pretend that we have a random variable (X) that is Bernoulli distributed with some probability p of observing a 1. Then if $Y = X_1 + \cdots + X_n$ is the sum of independent Bernoulli random variables, Y will have a Binomial distribution with probability p and sample size n. For instance, if a basketball player shoots free throws with $p = .60$, then (assuming that free throws are independent, which is probably not true) the number of successful free throws after a particular number of free throw attempts will be binomially distributed. For example, to find the probability of making *exactly* 6 out of 10 free throws, we can use the `dbinom` function.

```
> # The probability of making EXACTLY 6 out of 10 free throws:
> dbinom(x = 6, size = 10, prob = .60)
[1] 0.2508227
```

Notice how *even though* the probability of making *each* free throw is .60, there is a less-than 50% chance of throwing *exactly* 6 out of 10 free throws. We could also find the probability of making *exactly* 6 out of 10 free throws using the binomial distribution formula

$$f(y;\, n, p) = \binom{n}{y} p^y (1-p)^{n-y}$$

where $\binom{n}{y}$ is the number of ways to choose y out of n things. For example, $\binom{10}{6}$ is the number of ways to shoot 6 out of 10 free throws. You can either: make the first, miss the second, make the third, make the fourth, miss the fifth, make the sixth, make the seventh, miss the eighth, miss the ninth, and make the tenth ... or any other sequence of makes/misses such that the number of makes is 6. From basic discrete math, one finds $\binom{10}{6} = \frac{10!}{6!(10-6)!}$ resulting sequences of making 6 out of 10 things. To calculate $\binom{n}{y}$, we could try using the `factorial` function

```
> # Calculating the binom coefficient using factorials:
> ten.choose.six <- factorial(10)/(factorial(6)*factorial(10 - 6))
> ten.choose.six
[1] 210
```

but factorials tend to be rather large numbers, and `R` will *blow* up at a certain point.

```
> # KaBoom!
> factorial(1000)/(factorial(2)*factorial(1000 - 2))
[1] NaN
```

`R` actually includes a function to calculate $\binom{n}{y}$ without having to worry about the factorials.

```
> ten.choose.six2 <- choose(n = 10, k = 6)
> ten.choose.six2
[1] 210
```

The `choose` function will not blow up as easily with extreme values due to only indirectly evaluating the factorials.

```
> # Not really KaBoom!
> choose(n = 1000, k = 2)
[1] 499500
```

Using the `choose` function, we can also find the probability of making *exactly* 6 out of 10 free throws.

```
> choose(10, 6) * .6^6 * (1 - .6)^(10 - 6)  # the formula
[1] 0.2508227
> dbinom(x = 6, size = 10, prob = .6)       # the dbinom function
[1] 0.2508227
```

If we need the probability of making *at most* 3 out of 10 free throws, we can either sum a bunch of probabilities (using `dbinom`),

```
> # Sum a bunch of individual probabilities:
> { dbinom(x = 0, size = 10, prob = .60) +
+   dbinom(x = 1, size = 10, prob = .60) +
+   dbinom(x = 2, size = 10, prob = .60) +
+   dbinom(x = 3, size = 10, prob = .60) }
[1] 0.05476188
```

find a *vector* of probabilities and sum that vector,

```
> # The d/p/q/r functions are vectorized!  Cool!
> sum( dbinom(x = 0:3, size = 10, prob = .60) )
[1] 0.05476188
```

or use the `pbinom` function.

```
> pbinom(q = 3, size = 10, prob = .60)
[1] 0.05476188
```

If we want to find the probability of making *at least* 6 out of 10 free throws, we can also sum up individual probabilities,

```
> # Sum a bunch of individual probabilities:
> { dbinom(x = 6, size = 10, prob = .60) +
+   dbinom(x = 7, size = 10, prob = .60) +
+   dbinom(x = 8, size = 10, prob = .60) +
+   dbinom(x = 9, size = 10, prob = .60) +
+   dbinom(x = 10, size = 10, prob = .60) }
[1] 0.6331033
```

or find a *vector* or probabilities and sum that vector,

```
> # The d/p/q/r functions are STILL vectorized
> sum( dbinom(x = 6:10, size = 10, prob = .60) )
[1] 0.6331033
```

but if you use the `pbinom` function with `lower.tail = FALSE` (to find the probability of shooting at least a particular number of free throws), `R` does not behave.

```
> # NOT the probability of shooting at least 6 free throws:
> pbinom(q = 6, size = 10, prob = .60, lower.tail = FALSE)
[1] 0.3822806
```

What gives!??! The `pbinom` function with `lower.tail = FALSE` resulted in a probability *much lower* than summing up the individual probabilities. Unfortunately, the previous code chunk corresponds to the probability of shooting *more* than 6 free throws, which is really the probability of shooting *at least* 7 free throws. How annoying! To find the probability of shooting *at least* 6 free throws, we must subtract 1 in the `q` argument of `pbinom`.

```
> # The probability of shooting AT LEAST 6 free throws:
> pbinom(q = 5,     # q = 6 - 1 = 5.  Oy!
+        size = 10, prob = .60, lower.tail = FALSE)
[1] 0.6331033
>
> # And it gives us the correct result.
```

We can in turn use `qbinom` to find the score that corresponds to particular quantiles of the distribution. In fact, you might be wondering what `q` stands for in `qbinom`: quantile. Aren't those `R` people very creative? So, if we want to find the 90th percentile (or .9 quantile) of the binomial distribution, we should use the `qbinom` function.

```
> qbinom(p = .90, size = 10, prob = .60)
[1] 8
```

What exactly does the `R` output mean with respect to the binomial distribution? If we were to randomly sample an *infinite* number of times from this particular binomial distribution, then 90% of those samples would have 8 or fewer free throws made. To find the score corresponding to the top 10%, use the following code.

```
> qbinom(p = .10, size = 10, prob = .60, lower.tail = FALSE)
[1] 8
```

Of course, using `lower.tail = FALSE` for the `qbinom` function sometimes results in odd things happening. To find the score such that 10% of scores are at or above it, I recommend subtracting the percentage from 1 and using `lower.tail = TRUE`.

```
> # The score corresponding to top 10%
> qbinom(p = 1 - .10, size = 10, prob = .60) # safer to type this!
[1] 8
```

One might also wonder how to draw samples from the binomial distribution. Fortunately, there is a function (`rbinom`) that can sample lots of observations.

```
> set.seed(999)
> x <- rbinom(n = 1000, size = 10, prob = .60)
> head(x)
[1] 6 6 8 4 5 8
```

In our example, `x` is a vector of length 1000 with every element between 0 and 10. We can find the mean and variance of `x`.

```
> mean(x)
[1] 6.021
> var(x)
[1] 2.336896
```

We should check the simulated mean and variance against the theoretical values.

```
> n <- 10
> p <- .6
> n*p         # theoretical mean of binomial
[1] 6
> n*p*(1 - p) # theoretical variance of binomial
[1] 2.4
>
> # Are they close to the previous code chunk?
```

Alternatively, we can pretend that `x` is a population of $N = 1000$ scores and sample from `x` using the sample function.

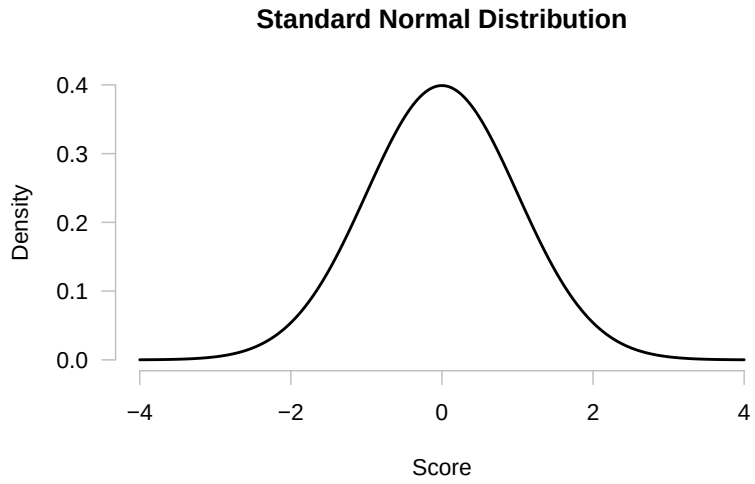
```
> set.seed(818)
> y <- sample(x, size = 10) # sampling from a sample?
```

In the above code chunk, I created a population (`x`) that is $N = 1000$ scores from a binomial distribution, and I *sampled* from that population (using the `sample` function) without replacement. It's as though our actual population is sort-of binomial and limited in size.

The Normal Distribution

The most important distribution in probability and statistics is (probably) the normal distribution. Because the normal distribution is continuous, `dnorm` *does not* indicate the probability of having a particular score but, rather, `dnorm` denotes the *height* of the density at a particular point. Therefore, with an exception for plotting the shape of a normal distribution (which I do quite a bit, as the normal distribution is quite pretty), you will rarely need to use the `dnorm` function. If you would like to plot a standard normal distribution, use the following commands (where `lwd` is the line width, similar to `cex` being the point size).

```
> x <- seq(-4, 4, by = .01)
> y <- dnorm(x, mean = 0, sd = 1)
> plot(x = x, y = y,
+      xlab = "Score", ylab = "Density",
+      main = "Standard Normal Distribution",
+      type = "l", lwd = 2, axes = FALSE)
> axis(1, col = "grey")
> axis(2, col = "grey", las = 1)
```



Unlike discrete variables, the probability of scoring at or below any point on a continuous distribution is 1 minus the probability of scoring at or above it (because the probability of having a score of exactly *anything* is identically 0: Go wacky knowledge from calculus!). So you do not need to worry about slightly correcting scores when finding the probability in the upper tail. If a variable is standard normally distributed, the probability of scoring below particular standard deviations is relatively straightforward (because the quantile of a standard normal distribution corresponds to a standard deviation).

```
> pnorm(q = -3, mean = 0, sd = 1) # below three sd (below the mean)
[1] 0.001349898
> pnorm(q = -2, mean = 0, sd = 1) # below two sd (below the mean)
[1] 0.02275013
> pnorm(q = -1, mean = 0, sd = 1) # below one sd (below the mean)
[1] 0.1586553
> pnorm(q =  0, mean = 0, sd = 1) # below zero sd
[1] 0.5
> pnorm(q =  1, mean = 0, sd = 1) # below one sd (above the mean)
[1] 0.8413447
> pnorm(q =  2, mean = 0, sd = 1) # below two sd (above the mean)
[1] 0.9772499
> pnorm(q =  3, mean = 0, sd = 1) # below three sd (above the mean)
[1] 0.9986501
```

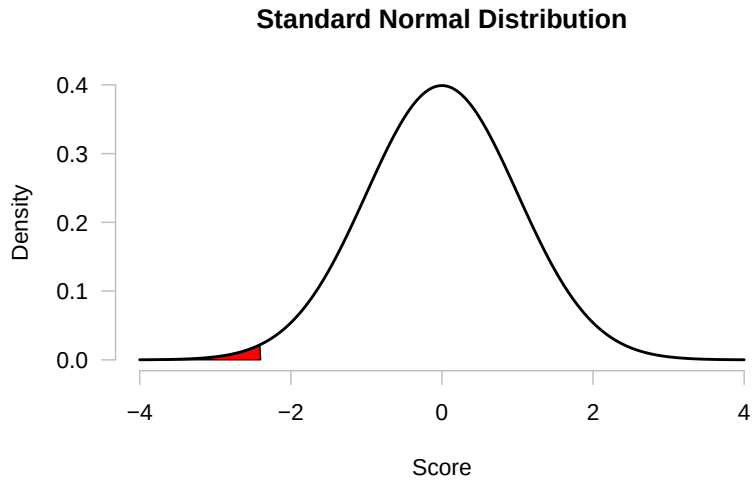
And the probability of scoring *between* particular standard deviations only involves subtracting areas.

```
> # Scoring within three standard deviations of the mean:
> pnorm(q = 3, mean = 0, sd = 1) - pnorm(q = -3, mean = 0, sd = 1)
[1] 0.9973002
```

```
> # Scoring within two standard deviations of the mean:
> pnorm(q = 2, mean = 0, sd = 1) - pnorm(q = -2, mean = 0, sd = 1)
[1] 0.9544997
> # Scoring within one standard deviation of the mean:
> pnorm(q = 1, mean = 0, sd = 1) - pnorm(q = -1, mean = 0, sd = 1)
[1] 0.6826895
```

One of the beauties of R is being able to *easily* find the probability of scoring below/between/beyond points on a normal distribution *regardless* of the mean and standard deviation. For instance, assume that a variable (X) is normally distributed with a mean of 60 and a standard deviation of 5. In introduction to statistics classes, finding the probability of scoring below a 48 requires calculating z-scores *and then* looking up the probability in the back of a book.

```
> # First calculate z-score:
> z <- (48 - 60)/5
> # Then look up probability in the back of the book:
> pnorm(q = z, mean = 0, sd = 1)  # Still easier in R
[1] 0.008197536
> # Do we use the smaller portion?  Or larger portion?
> # My head hurts, I probably should draw a picture, and
> # color in the area.
>
> x <- seq(-4, 4, by = .01)
> y <- dnorm(x, mean = 0, sd = 1)
> plot(x = x, y = y,
+      xlab = "Score", ylab = "Density",
+      main = "Standard Normal Distribution",
+      type = "n", axes = FALSE)
> axis(1, col = "grey")
> axis(2, col = "grey", las = 1)
> polygon(x = c(min(x), x[x < z], z),
+         y = c(y[which.min(x)], y[x < z], y[which.min(x)]),
+         col = "red")
> lines(x = x, y = y, lwd = 2)
> # OK, so the red is the smaller portion, so I must look
> # up smaller portion in the z-table in the back of the book.
> # I'm confused!
```



In R, finding the area under a normal distribution (regardless of the mean and standard deviation) can be done on one line.

```
> # Area below 48 on the appropriate normal distribution:
> pnorm(q = 48, mean = 60, sd = 5)
[1] 0.008197536
>
> # Gives exactly the same probability.  Cool!
```

Moreover, to find the probability of having a score between 48 and 65, you just have to subtract without worrying about first converting to z-scores.

```
> pnorm(q = 65, mean = 60, sd = 5) - pnorm(q = 48, mean = 60, sd = 5)
[1] 0.8331472
```

You don't need to bother about whether one scores is below the mean and the other score is above the mean. You must only know that (making sure not to change `lower.tail` to FALSE), the area returned by `pnorm` is in the "below-the-score" direction. Therefore, finding the probability of having a score between two values requires only typing in the following.

```
pnorm(q = largest score, ... ) - pnorm(q = smallest score, ... )
```

Easy peasy! And to find the probability of having an score greater than 125 for a variable that is (sort of) normally distributed with a mean of 100 and a standard deviation of 15, you can use the `pnorm` function.

```
> # Use lower.tail = FALSE to get the upper part of the dist:
> pnorm(q = 125, mean = 100, sd = 15, lower.tail = FALSE)
[1] 0.04779035
```

R is much easier than having to calculate things, color in areas, and use the back of a book.

You can also easily find particular cutoff scores that bracket certain areas on a normal distribution. For instance, if a test is normally distributed with a mean of 70 and a standard deviation of 8, then the cutoff corresponding to the top 10% of students is the following.

```
> # One way is to use the upper portion:
> qnorm(p = .10, mean = 70, sd = 8, lower.tail = FALSE)
[1] 80.25241
> # The other way is to use the lower portion:
> qnorm(p = 1 - .10, mean = 70, sd = 8)
[1] 80.25241
>
> # Approximately a score of 80!  Nice!
```

Alternatively, we can use the qnorm function to find the scores bracketing the middle 50% of students on a test.

```
> p <- .50  # proportion we want to bracket.
> qnorm(p = (1 + p)/2, mean = 70, sd = 8) # score for the top 25%
[1] 75.39592
> qnorm(p = (1 - p)/2, mean = 70, sd = 8) # score for the bottom 25%
[1] 64.60408
```

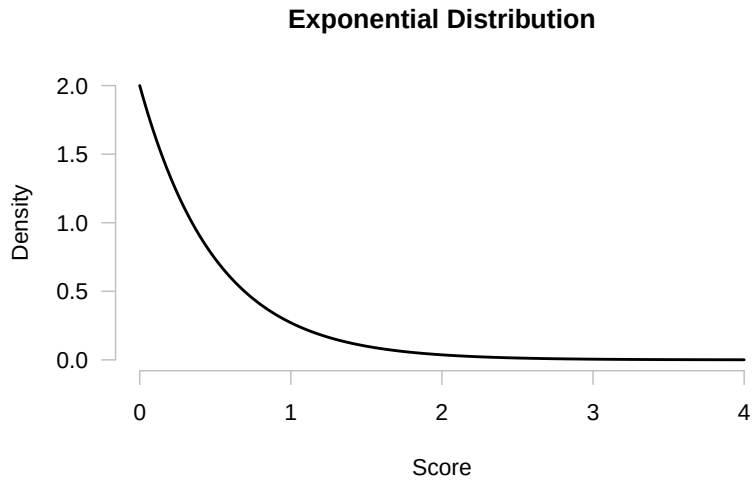
The code from the the above code chunk can be extended to bracket *any* percentage of a normal distribution. For example, to bracket the middle 95% (◡̈):

```
> p <- .95  # proportion to bracket.
> qnorm(p = (1 + p)/2, mean = 70, sd = 8)
[1] 85.67971
> qnorm(p = (1 - p)/2, mean = 70, sd = 8)
[1] 54.32029
```

Checking Distribution Normality

Given a particular sample, one might need to determine whether that sample comes from a normal population. An easy method of determining how similar a population is to being normal requires finding the percentage of scores between 1, 2, and 3 standard deviations of a sample. For example, an exponential distribution is heavily right skewed.

```
> x <- seq(0, 4, by = .01)
> y <- dexp(x, rate = 2)
> plot(x = x, y = y,
+      xlab = "Score", ylab = "Density",
+      main = "Exponential Distribution",
+      type = "l", lwd = 2, axes = FALSE)
> axis(1, col = "grey")
> axis(2, col = "grey", las = 1)
```



If you had a sample from an exponential distribution, you could empirically test whether the sample was close to being normally distributed.

```
> set.seed(678)
> samp3 <- rexp(n = 1000, rate = 2)  # a distribution you don't know.
```

The process is as follows. First, determine the mean and standard deviation of the sample.

```
> m.s3  <- mean(samp3) # the mean of our sample
> sd.s3 <- sd(samp3)   # the sd of our sample (with N - 1 in denom)
```

Second, put TRUEs and FALSEs in the correct places by determining the sample values above or below 1, 2, and 3 standard deviations from the mean.

```
> samp3.1sd <- (samp3 >= m.s3 - 1*sd.s3) & (samp3 <= m.s3 + 1*sd.s3)
> samp3.2sd <- (samp3 >= m.s3 - 2*sd.s3) & (samp3 <= m.s3 + 2*sd.s3)
> samp3.3sd <- (samp3 >= m.s3 - 3*sd.s3) & (samp3 <= m.s3 + 3*sd.s3)
```

Finally, find the proportion of sample values between 1, 2, and 3 standard deviations from the mean.

```
> sum(samp3.1sd)/length(samp3)
[1] 0.867
> sum(samp3.2sd)/length(samp3)
[1] 0.951
> sum(samp3.3sd)/length(samp3)
[1] 0.982
```

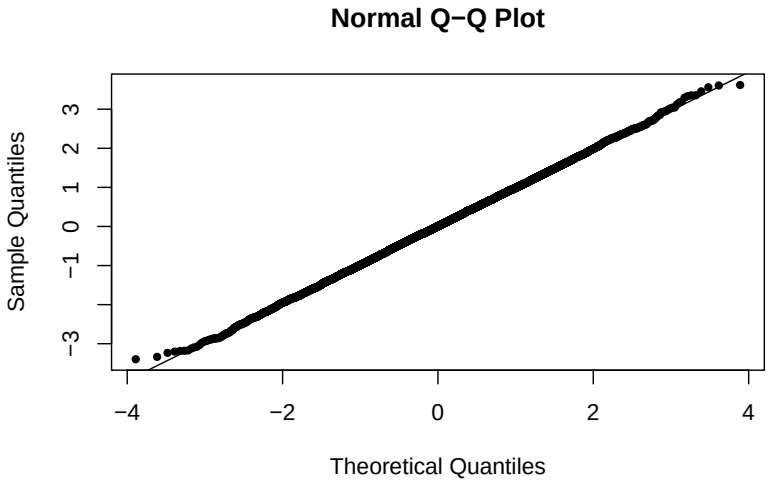
Because those proportions dramatically differ from the 68.3%, 95.4%, and 99.7% between standard deviations on a normal distribution, our sample is not close to being normally distributed.

If you had a sample from a normal distribution, then the proportion of that sample between 1, 2, and 3 standard deviations of the mean should be close to expectation.

```
> set.seed(1234)
> samp4 <- rnorm(10000, mean = 0, sd = 1)
> # First, determining the mean and sd of our sample:
> m.s4  <- mean(samp4)  # the mean of our sample
> sd.s4 <- sd(samp4)    # the sd of our sample
> # Second, put TRUEs/FALSEs into the correct place:
> samp4.1sd <- (samp4 >= m.s4 - 1*sd.s4) & (samp4 <= m.s4 + 1*sd.s4)
> samp4.2sd <- (samp4 >= m.s4 - 2*sd.s4) & (samp4 <= m.s4 + 2*sd.s4)
> samp4.3sd <- (samp4 >= m.s4 - 3*sd.s4) & (samp4 <= m.s4 + 3*sd.s4)
> # Third, find the proportion of TRUEs:
> sum(samp4.1sd)/length(samp4)  # between -1 and 1 sd
[1] 0.6825
> sum(samp4.2sd)/length(samp4)  # between -2 and 2 sd
[1] 0.9554
> sum(samp4.3sd)/length(samp4)  # between -3 and 3 sd
[1] 0.9971
```

An alternative method of determining whether a sample arises from a normal population involves constructing quantile-quantile plots. Luckily, constructing quantile plots is rather easy in R. Comparing a particular sample to quantiles of the normal distribution can be done effectively with two functions.

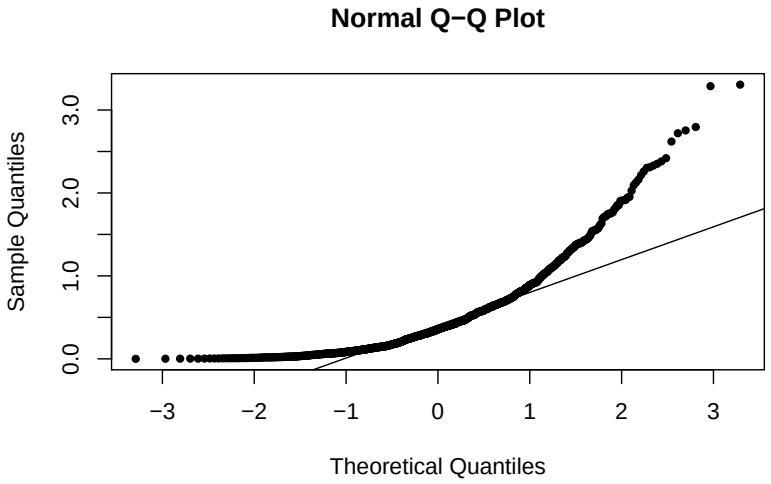
```
> qqnorm(samp4, pch = 20)  # first plotting
> qqline(samp4)            # then drawing a line on top of the plot
```



Important: Several plotting functions, including `qqline`, `points`, `lines` plot *on top* of an existing plot. If you try to use those functions *without* the graphing device being open, they will give you an error, and if you try to use those functions *after closing* the graphing device, they will give you an error. Therefore, make sure to call `plot` or `qqnorm` first, and then *without closing the plot*, call the subsequent functions `points`, `lines`, or `qqline`.

Because the sample `samp4` was generated from normal distribution, the points lie pretty much on top of the q-q line. However, `samp3` was sampled from an exponential distribution (with positive skew), so the corresponding q-q points lie above the q-q line.

```
> qqnorm(samp3, pch = 20)
> qqline(samp3)
```



An interesting question is how skew and kurtosis affect the shape of normal quantile-quantile plots. I do not have time, energy, nor space to go over that at the moment, but you can experiment on your own with some flesh-and-blood friends and the rather intrusive (but knowledgable) friend: google.

5.3 Appendix: Functions Used

Here is a list of important functions from this chapter:

```
# Sampling
set.seed(seed)                   # fix the "random" numbers
sample(x, size, replace, prob)   # sample from a population
plot(x, y, ... )                 # plot y against x
points(x, y, ... )               # add points to a plot

# Built-in Populations #

# Generic functions (applying to all)
dawesome(x, params)              # the density (or probability)
pawesome(q, params, lower.tail) # prob at or below (or above)
qawesome(p, params, lower.tail) # value associated with pawesome
rawesome(n, params)              # random sample from distribution

# Specific distributions
norm(mean, sd)                   # normal distribution
binom(size, prob)                # binomial distribution
unif(min, max)                   # uniform distribution
cauchy(location scale)           # Cauchy distribution
t(df)                            # t distribution
F(df1, df2)                      # F distribution
chisq(df)                        # chi-square distribution

# Populations and probabilities
factorial(x)           # x*(x - 1)*(x - 2)*...*(1)
choose(n, k)           # how many ways to pick k out of n things?
qqnorm(y)              # plot a quantile-quantile plot
qqline(y)              # draw a line ON TOP of an existing plot
```

Chapter 6

Control Flow and Simulation

Man: *Um, is this the right room for an argument?*
Mr. Vibrating: *I've told you once.*
Man: *No you haven't.*
Mr. Vibrating: *Yes I have.*
Man: *When?*
Mr. Vibrating: *Just now.*
Man: *No you didn't.*
Mr. Vibrating: *I did.*
Man: *Didn't.*
Mr. Vibrating: *Did.*
Man: *Didn't.*
Mr. Vibrating: *I'm telling you I did.*
Man: *You did not.*
Mr. Vibrating: *Oh I'm sorry, just one moment. Is this a five minute argument or the full half hour?*
—Monty Python's Flying Circus - Episode 29

6.1 Basic Simulation

6.1.1 for Loops in R

One of the reasons that `R` is better (yes, better) than other statistical software is due to its flexibility. Anybody can write code, add code, modify code, destroy code, and maybe even hug code (when lonely due to hanging around too much non-code). And not surprisingly, the aspect of `R` *most* emblematic of its statistical-software-that-could mantra is its ability to act like a programming language. In fact, `R` uses the `S` programming language, written by John Chambers, and thus, has very similar (though probability more intuitive) control-flow methods to C.

To take advantage of the full power of the `S` language, you need to know how to use (at least) the following commands: `for`, `if`, `else`, and `while`, although `function`, `repeat`, `break`, and `next` are also useful. The function `function` is *so* useful that it will be covered in a separate chapter, but the basis for most simulations (even those in

functions) is via `for` loops[1]. The basic format of a `for` loop is the following.

```
for(name in set.of.values){

  commands

} # END for name LOOP
```

In the above "pseudo-code", `name` is *any* valid name of an `R` object. Usually we reserve simple names, such as `i`, `j`, or `k` for `name`, but *any* valid `R` name is legal. `set.of.values` is either a *vector* or a *list*. If `set.of.values` is a vector, then `name` will take the next value of the vector during the next iteration of the `for` loop. If `set.of.values` is a list, then `name` will take the next sub-object of the list during the next iteration. The `set.of.values` part of the `for` loop is usually a vector of integers (such as `1:100`), although any vector or list would work. The final part of the `for` loop is `commands`, which are *any* set of commands, although they usually contain the object `name` somewhere. The set of commands tends to be enclosed in curly braces.

> **Important**: To loop over a set of commands, you usually need to enclose those commands in curly braces (i.e., `{`, `}`). Otherwise, `R` will only loop over the first line of your set of commands, and the `for` loop will probably not do what you want. Moreover, as `for` loops are more complex than simple `R` statements, you really should be writing your code in a separate file and entering the code into `R` once you are confident enough in its error free-ness.

A final, useful trick when constructing your own `for` loops is called the `NULL` vector: a vector of *no* length with *nothing* in it.

```
> vec <- NULL  # a NULL vector!
> length(vec)  # see - it has no length.  Weird!?
[1] 0
```

`R` will *stretch* the `NULL` vector to as long as needed. For instance, if you try to put an element in the fifth place of the `NULL` vector, `R` will increase the size of the `NULL` vector to 5, put the element in the 5th place, and fill the rest of the vector with `NA`'s.

```
> vec[5] <- 1  # putting an element in the 5th place
> vec
[1] NA NA NA NA  1
```

The `NULL` vector is useful because we can create an object (the `NULL` vector) to store values of a simulation, but we do not need to know *at the outset of the simulation* how many values we will need to store.

To illustrate the beauty of `for` loops, the following code chunk contains a brief and typical set of commands.

[1] Unlike previous chapters, several code chunks in this and future chapters use an empty continuation prompt rather than the default `+` prompt. I change the continuation prompt for long code chunks so that you can easily copy and paste the code into your own `R` session.

```
> int <- NULL     # a NULL vector!
> for(i in 1:10){ # setting the for loop!
+
+   int[i] <- i   # our commands
+
+ }               # ending the for loop!
```

In the above code chunk, I first created a `NULL` vector to store values. I then set a `for` loop: (a) `i` is our `name`, and during each iteration of the `for` loop, the *object* `i` will take the next value of the `set.of.values` vector; and (b) `1:10` (a vector of 1 – 10) is our the `set.of.values` vector. So, during the first iteration of the `for` loop, `i` takes the value 1; during the second iteration of the `for` loop, `i` takes the value 2, ... etc. Why? Because 1 is the first value in our `set.of.values` vector, 2 is the second, ... etc. Once I set the parameters of the `for` loop, I put a simple command inside curly braces: at each cycle of the `for` loop, `R` puts the object `i` into the `i`th place of the vector `int`. What in the world does this mean? Well, during the first iteration, the object `i` is 1, so the value 1 goes into the 1st place of `int`; during the second iteration, the object `i` is 2, so the value 2 goes into the 2nd place of `int`; etc. After putting 1 in the 1st place, 2 in the 2nd place, etc., the contents of `int` should be pretty obvious.

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

Just to demonstrate that a `for` loop can take *any* vector as its `set.of.values`, try to figure out what the following code is doing on your own.

```
> # A set of bizarre code:
> wha <- matrix(NA, nrow = 10, ncol = 4)
> v1  <- c(1, 3, 4, 6, 7, 10)
> v2  <- c("a", "b", "d", "c")
> for(i in v1){
+ for(j in v2){
+
+   wha[i, which(v2 == j)] <- j
+
+ }} # END for LOOPS
> # Looking at our for loop results.
> wha
      [,1] [,2] [,3] [,4]
 [1,] "a"  "b"  "d"  "c"
 [2,] NA   NA   NA   NA
 [3,] "a"  "b"  "d"  "c"
 [4,] "a"  "b"  "d"  "c"
 [5,] NA   NA   NA   NA
 [6,] "a"  "b"  "d"  "c"
 [7,] "a"  "b"  "d"  "c"
 [8,] NA   NA   NA   NA
 [9,] NA   NA   NA   NA
[10,] "a"  "b"  "d"  "c"
```

Note: As is evident from the above code chunk, `for` loops can be nested within other `for` loops. If a `for` loop set is nested, the iterations cycle from the inside out, completing the `j` cycle before moving to the second element of the `i` cycle.

6.1.2 Constructing Sampling Distributions

Many basic simulations in statistics have a similar theme: calculate a sampling distribution of some statistic and determine its distribution. The outline of this procedure will not vary ... very much.

```
reps <- 10000          # large number of samples to take
N    <- 10             # sample size for EACH replication
vec  <- NULL           # vector to store statistics

for(i in 1:reps){      # repeating the samples "reps" times

  samp <- rdist(N, ... ) # a sample from the dist
  vec[i] <- stat(samp)   # calculating the statistic

} # END for i LOOP
```

You must always specify: (1) the number of samples to take (the closer to infinite, the better, within time constraints; I usually take around 10,000 samples, which is about as close to infinity as I can count); (2) the size of *each* sample (here $N = 10$, although you will have to change N depending on the question); and (3) an empty vector to hold statistics. Then you will set your `for` loop (where `i` iterates over the replications), sample from a particular distribution, calculate a statistic, and put that statistic in the `i`th place of the empty vector. Once you repeat the `for` loop many times, you will then examine the distribution of statistics and determine whether the central limit theorem was made up by delusional probabilists without computing power.

A simple example of a sampling distribution is determining the mean/variance/shape of a distribution of sample means when $N = 10$ while sampling from a standard normal distribution.

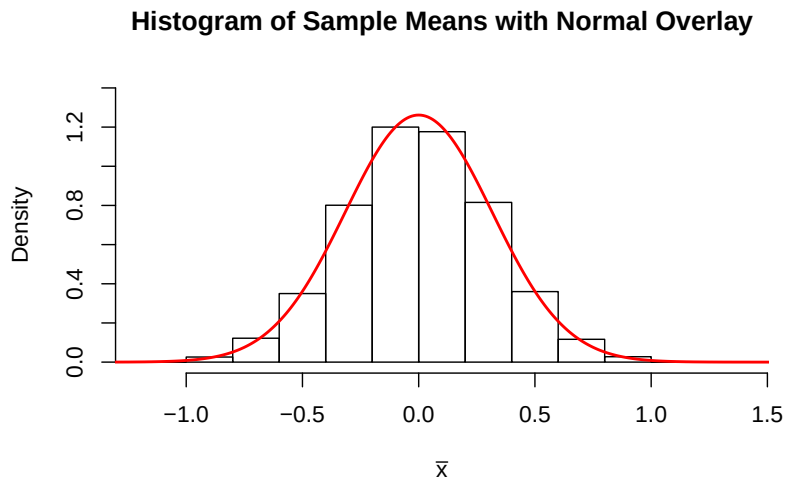
```
> set.seed(9102)      # so things can be replicated
> reps <- 10000       # large-ish number of samples
> N    <- 10          # sample size for EACH replication
> xbar <- NULL        # vector to store sample means
> for(i in 1:reps){   # repeating "reps" times
+
+ # Sampling from a standard normal distribution:
+   samp.i <- rnorm(n = N, mean = 0, sd = 1)
+
+ # Calculating the sample mean and putting it in the ith place:
+   xbar[i] <- mean(samp.i)
+
+ } # END for i LOOP
```

After simulating a sampling distribution, we can compare the "empirical" mean/variance to the "theoretical" mean/variance of this particular sampling distribution.

```
> mean(xbar)  # SIMULATED mean of the sampling dist
[1] 0.0006293143
> var(xbar)   # SIMULATED variance of the sampling dist
[1] 0.09927399
> sd(xbar)    # SIMULATED standard deviation
[1] 0.3150777
> ######################################################
>
> 0           # THEORETICAL mean of the sampling dist
[1] 0
> 1/N         # THEORETICAL variance of the sampling dist
[1] 0.1
> 1/sqrt(N)   # THEORETICAL standard deviation
[1] 0.3162278
```

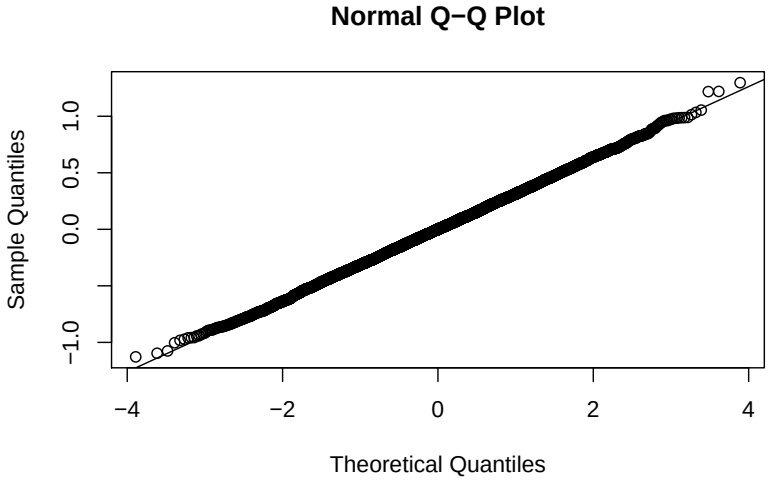
And we can construct a histogram to determine the similarity of our empirical sample and a normal distribution.

```
> x <- seq(-4, 4, by = .01)
> y <- dnorm(x, mean = 0, sd = 1/sqrt(N))
> hist(xbar, freq = FALSE,
+      xlab = expression(bar(x)), ylab = "Density",
+      main = "Histogram of Sample Means with Normal Overlay",
+      ylim = c(0, 1.4))
> lines(x = x, y = y, lwd = 2, col = "red")
```



And we can draw a `qqnorm` plot to determine the similarity of our sampling distribution with a normal distribution.

```
> qqnorm(xbar)    # the qqnorm plot
> qqline(xbar)    # the qqnorm line in the plot
```



The only challenging part of the above code chunks is the function `expression` inside `histogram`. Unfortunately, the `expression` function is rather tricky, but for *simple* symbols (as in `bar(x)` or `mu`), the plotted output of `expression` will be as expected. In any case, the `qqnorm` plot looks pretty much on a line, so I would say that our sampling distribution is close to being normally distributed.

Not surprisingly, the central limit theorem eventually works for non-normally distributed variables. And we can also examine sampling distributions in these cases. An example of a heavily skewed distribution is the χ^2 distribution with 2 degrees of freedom.

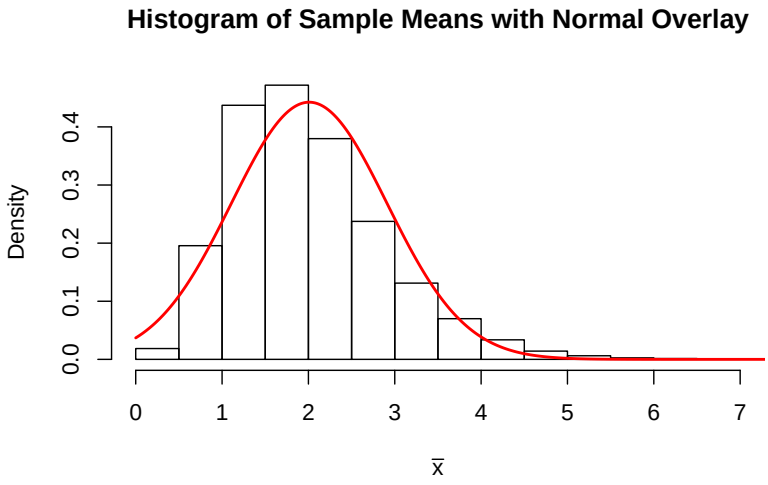
```
> set.seed(9110)     # so things can be replicated
> reps <- 10000      # large-ish number of samples
> N    <- 5          # sample size for EACH replication
> xbar <- NULL       # vector to store sample means
> for(i in 1:reps){  # repeating "reps" times
+
+ # Sampling from a chi-square distribution:
+   samp.i <- rchisq(n = N, df = 2)
+
+ # Calculating the sample mean and putting it in the ith place:
+   xbar[i] <- mean(samp.i)
+
+ } # END for i LOOP
```

Note: Seriously, these simulations should not be very difficult. All you have to do is change `N` to your sample size, change (perhaps) the distribution that

you are sampling from, and change (perhaps) the sample statistic. Everything else is *exactly* the same. I just copied and pasted the earlier code chunk.

And we can show that our sampling distribution is *not* normally distributed by looking at histograms and `qqplot`s (as before). Note that figuring out the theoretical mean and variance of our sampling distribution requires knowing the mean and variance of the population, and we can estimate μ and σ^2 by taking a *very large* sample from the original population[2].

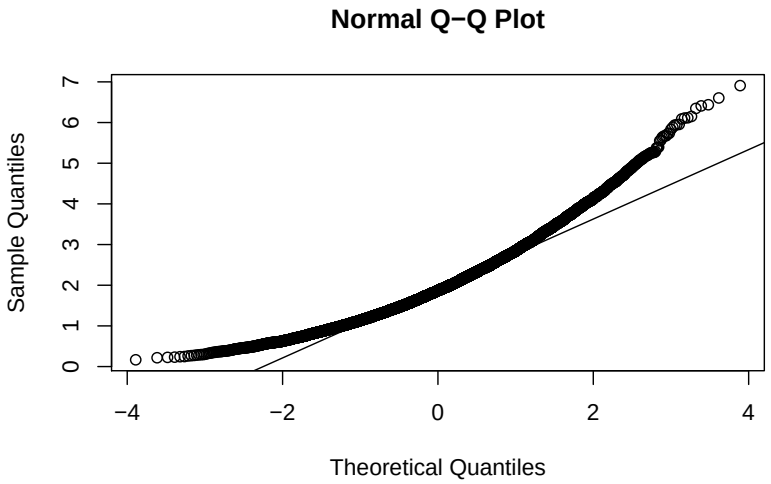
```
> popn <- rchisq(n = 100000, df = 2)
> x    <- seq(0, 10, by = .01)
> y    <- dnorm(x, mean = mean(popn), sd = sd(popn)/sqrt(N))
> hist(xbar, freq = FALSE,
+      xlab = expression(bar(x)), ylab = "Density",
+      main = "Histogram of Sample Means with Normal Overlay")
> lines(x = x, y = y, lwd = 2, col = "red")
```



For this particular distribution with this sample size, the normal distribution does not appear to work very well. We can *more easily* see the problems in our approximation by using a `qqnorm` plot.

```
> qqnorm(xbar)   # curvy and ugly!
> qqline(xbar)   # positively skewed.  see?
```

[2]The theoretical mean of a χ^2 random variable is its df, and the theoretical variances is $2 \times df$.



Sampling distributions do not (of course) just apply to the sample mean. You might also want to calculate a *random* statistic with a particular sample size, determine if that statistic is normally distributed-ish, and then compare the distribution of that statistic to a theoretical non-normal distribution. For example, if X is sampled from a standard normal distribution and $N = 4$, then

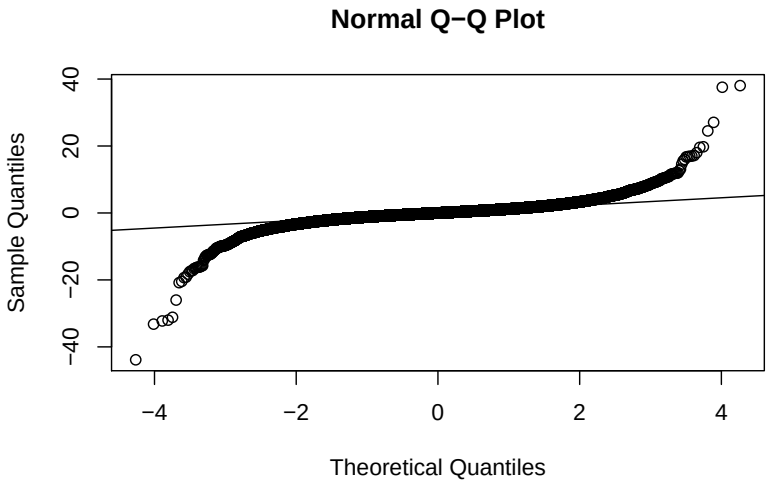
$$T = \frac{\bar{X}}{S_x/\sqrt{N}}$$

will be t-distributed with $df = 3$. First, simulating a sampling distribution.

```
> set.seed(91011)    # so things can be replicated
> reps <- 50000      # large-ish number of samples
> N    <- 4          # sample size for EACH replication
> T <- NULL          # vector to store sample means
> for(i in 1:reps){  # repeating "reps" times
+
+ # Sampling from a standard normal distribution:
+   samp.i <- rnorm(n = N, mean = 0, sd = 1)
+
+ # Calculating our t-statistic and putting it in the ith place
+   T[i] <- mean(samp.i) / (sd(samp.i) / sqrt(N))
+
+ } # END for i LOOP
```

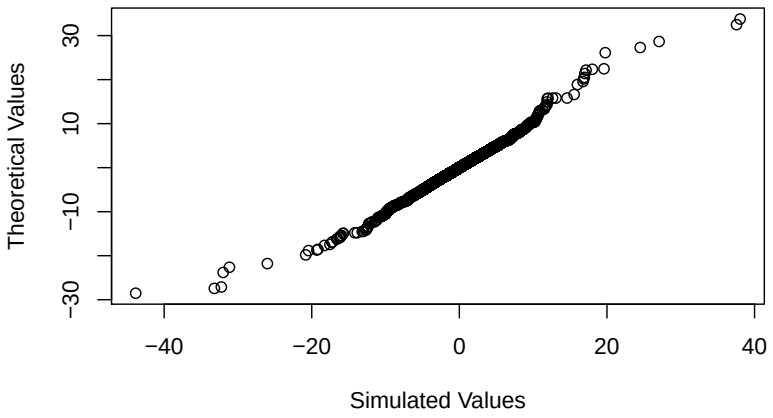
Then, examining a quantile plot relative to a normal distribution.

```
> qqnorm(T)  # qqnorm of t-distribution
> qqline(T)  # very long tails - can't you tell!?
```



And, finally, comparing our sampling distribution to the appropriate t-distribution.

```
> set.seed(123485)
> qqplot(x = T, y = rt(n = reps, df = 3),
+        xlab = "Simulated Values",
+        ylab = "Theoretical Values")
```

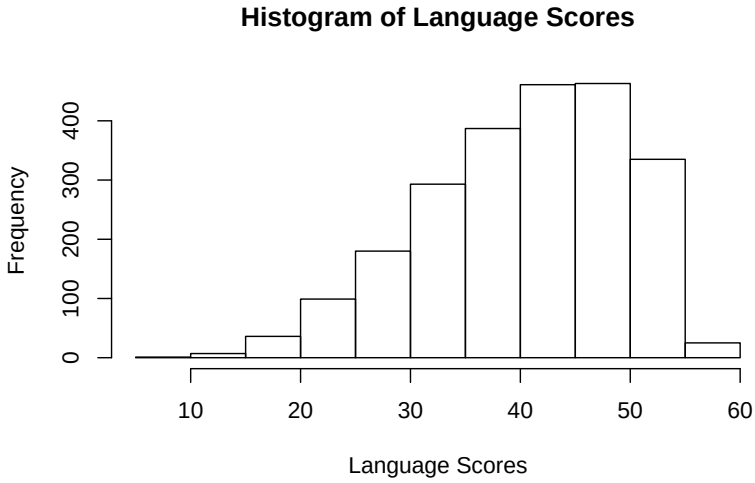


Even though our simulation is not a perfect t-distribution, the q-qT values are closer to a line than the q-qnorm values. Note that I put two vectors into `qqplot`: (a) the simulated sampling distribution, and (b) a random sample from the theoretical distribution. My method of plugging a random sample into `qqplot` is not the most appropriate—it would be better if we directly plugged in quantiles to the `qqplot` function—but both methods

lead to similar conclusions, and a random sample from the t-distribution is easier to understand from a coding perspective.

One can also create a sampling distribution from a set of scores by sampling *with replacement* from those scores. The only change to the previous code is specifying the vector and using the `sample` function rather than a distribution function.

```
> library(MASS)          # the library where the data is stores
> data(nlschools)        # school data (we are interested in lang)
> lang <- nlschools$lang # language scores on a test
> hist(lang,
+      xlab = "Language Scores", ylab = "Frequency",
+      main = "Histogram of Language Scores")
```



The original distribution is clearly negatively skewed, but we can still take samples of various sizes.

```
> set.seed(182420)
> samp.10  <- NULL
> samp.100 <- NULL
> samp.all <- NULL
> for(i in 1:10000){
+
+   samp.10[i]  <- mean(sample(lang, size = 10, replace = TRUE))
+   samp.100[i] <- mean(sample(lang, size = 100, replace = TRUE))
+   samp.all[i] <- mean(sample(lang, size = length(lang), replace = TRUE))
+
+ } # END for i LOOP
> # The mean of the sampling distributions (shouldn't change much).
> mean(samp.10)
[1] 40.96612
```

```
> mean(samp.100)
[1] 40.92599
> mean(samp.all)
[1] 40.9307
> # The variance of the sampling distributions (should decrease).
> var(samp.10)
[1] 8.312437
> var(samp.100)
[1] 0.8076201
> var(samp.all)
[1] 0.03538113
```

One of the uses of sampling from our sample (with replacement) of size equal to the original sample (`samp.all` in the above code chunk) is to estimate reasonable locations for the population mean if we do not know the theoretical sampling distribution. This method (called "bootstrapping") is an alternative, robust procedure for finding p-values and confidence intervals.

6.2 Other Control Flow Statements

6.2.1 The Beauty of while and if

The while Loop in R

The other two bits of control flow useful for our purposes are `while` and `if/else`. Both of these have a pretty similar form. The format of the `while` loop is as follows.

```
while(logical){

  commands

}
```

In the above "pseudo-code", `logical` is a logical *value* (either TRUE or FALSE); if `logical` is TRUE, then the `while` loop repeats, and if `logical` is FALSE, then the `while` loop terminates. Of course, the `while` loop only makes sense if one of the `commands` inside of the `while` loop has an opportunity to change the `logical` statement to FALSE. Otherwise, the `while` loop repeats indefinitely (save for silly people "turning off computers" or "ending programs").

The `while` loop is also useful in simulation and distribution construction. For instance, pretend that we have a Bernoulli random variable (with some p) and want to simulate the distribution of "number of samples/flips/whatever before one success". Examples include the number of coin flips that turn up tails before *one* turns up heads, or the number of missed basketball shots before one make. Simulating this distribution can be easily done via the `while` loop.

```
> set.seed(90210)   # setting a seed
> reps  <- 10000    # number of replications
```

```
> sim.5 <- NULL     # vector to store observations
> for(i in 1:reps){ # repeat reps times

  # First, set k = 0 and suc to FALSE
    k   <- 0
    suc <- FALSE

    while(!suc){      # while suc is not TRUE (i.e., until a success)

  # Sample one observation from a binomial distribution:
      tmp.i <- rbinom(1, size = 1, prob = .5) # <- the success rate is .5!

      if(tmp.i == 1){ # if that observation is a 1, we have a success!
        suc <- TRUE
      } else{
        suc <- FALSE  # otherwise, we have failed (miserably)
        k <- k + 1    # and add 1 to k
      } # END if STATEMENTS

    } # END while LOOP

  # Finally, put the sum until the first success in the ith place.
    sim.5[i] <- k

  } # END for i LOOP
```

The if/else Statement in R

The `while` loop made sure that we kept counting trials until a success. Yet *checking* for that success needed another control-flow type function, the `if` statement.

```
if(logical){

  commands

} else{

  commands

}
```

The above "pseudo-code" is one possibility for the `if` statement, but `else` part might not even be needed, so that the following is another (common) possibility.

```
if(logical){

  commands
```

```
}
```

In the former set of statements, if the logical inside of `if` is TRUE, then `R` will run the commands inside the first set of curly braces, but if the logical inside of `if` is FALSE, then `R` will run the commands inside the second set of curly braces. In the latter set of statements, if the logical inside of `if` is TRUE, then `R` will run the commands inside the first set of curly braces, and regardless of the TRUE or FALSE nature of the logical, `R` will run the commands following the `if` statement.

> **Important**: Both the `while` loop and the `if` statement take logical *values*: TRUE or FALSE. Moreover, if you put a vector inside `while` or `if`, `R` will only look at the first element in that vector to decide whether (or not) to run the loop. You can (and should) use comparisons (e.g., `>`, `<=`, `!`) to get those logical values.

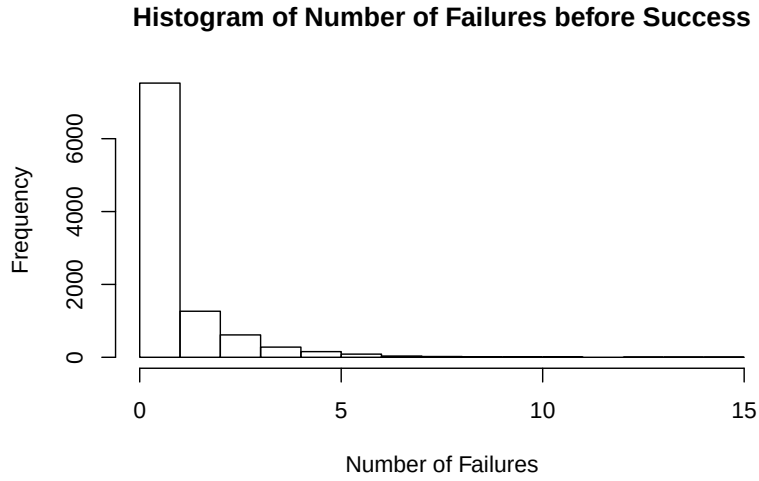
Complicated `while`, `if`, `else`, and `for` statements can be better understood by breaking them down into constituent parts.

- `set.seed(90210)` sets the seed, so that I can repeat the exact same simulation in the future, `reps <- 10000` sets the number of replications to a rather large number, and `sim.5 <- NULL` builds an empty vector to store values. All of these we have discussed earlier in this chapter.
- `for(i in 1:reps)` sets the `for` loop to repeat `reps` times such that the first element of `i` is 1, the second element of `i` is 2, etc.
- `k <- 0` makes sure that the count starts at 0. Note that at *every cycle* through the loop, `k` (the count of failures before a success) will be reset to 0.
- `suc <- FALSE` makes sure that we start out the simulation (for each iteration) on a failure. If `suc` did not equal FALSE, we would never satisfy the logical statement in the `while` loop, we would never simulate *any* value, and we could not get off the ground.
- `while(!suc)` forces us to repeat whatever is inside the `while` curly braces until `suc` is TRUE. Thus, the main objective of the `while` loop is determining when to assign TRUE to `suc` so that we can exit the `while` loop and start the next iteration of the `for` loop.
- `tmp.i <- rbinom(1, size = 1, prob = .5)` samples 1 observation from a binomial distribution with $N = 1$. By setting $N = 1$, we are effectively sampling from a Bernoulli distribution.
- `if(tmp.i == 1)` checks to see if we have a success. If so (yay for us), we set `suc <- TRUE`, which satisfies the exit condition of the `while` loop. Why? Because we only repeat the `while` loop as long as `suc` is FALSE.
- `else` determines what happens if we do not have a success. If we don't have a success, then we keep `suc` equal to FALSE, and we add another failure to `k` (so that `k <- k + 1`). Each time through the `while` loop, when we fail to have a success, another value is added to `k`, so that by the time that the while loop is exited, `k` equals the number of failures that occurred before the success that allowed us to leave the loop.

- `sim.5[i] <- k` sets the `i`th value of `sim.5` to `k` (the number of failures before a success), and then we set `i` equal to `i + 1`, and we repeat the loop again.

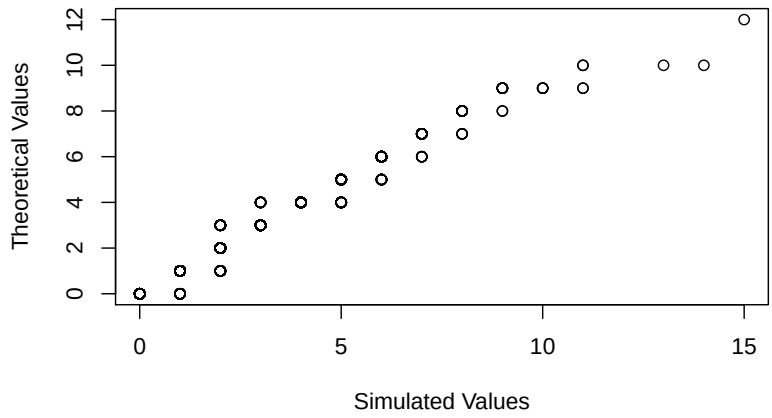
We can plot a histogram of our odd distribution and determine its shape.

```
> hist(sim.5,
+      xlab = "Number of Failures", ylab = "Frequency",
+      main = "Histogram of Number of Failures before Success")
```



Our crazy distribution is clearly *not* normally distributed. However, a standard distribution is generated in the fashion above: the geometric distribution. And forming a `qqplot` of our sample against a random sample from a geometric distribution, both samples should align.

```
> set.seed(234238)
> qqplot(x = sim.5, y = rgeom(n = reps, prob = .5),
+        xlab = "Simulated Values",
+        ylab = "Theoretical Values")
```



You should change the success rate from $p = .5$ to other values (e.g., $p = .6$ for a basketball player, or $p = .01$ for a rather rare event, such as a small lottery), and determine the distribution of the above statistic (`sim.6` and `sim.01`).

6.2.2 A Strange Sampling Distribution

I will end this chapter with a simulation from *Using R for Introductory Statistics* (Verzani, 2005, p. 180). A brief description of the logic behind the simulation will be provided, but you should determine (on your own time) the actual result. Pretend that we have a square with length 2 and centered at the origin. Then the area of the square is $2 \times 2 = 4$. Moreover, a circle of radius 1 (with area $\pi \times 1^2 = \pi$) will touch the edges of the square. And taking the ratio of "circle area" to "square area" results in $\pi/4$. Finally, knowing that all of the points within a circle of radius 1 satisfies the constraint that $x^2 + y^2 \leq 1$, a simple (but rather inefficient) simulation is as follows.

```
> set.seed(1920)
> N        <- 1000
> x        <- runif(N, min = -1, max = 1)
> y        <- runif(N, min = -1, max = 1)
> circ.pts <- x^2 + y^2 <= 1
> magic    <- 4 * mean(circ.pts)
> magic
[1] 3.132
```

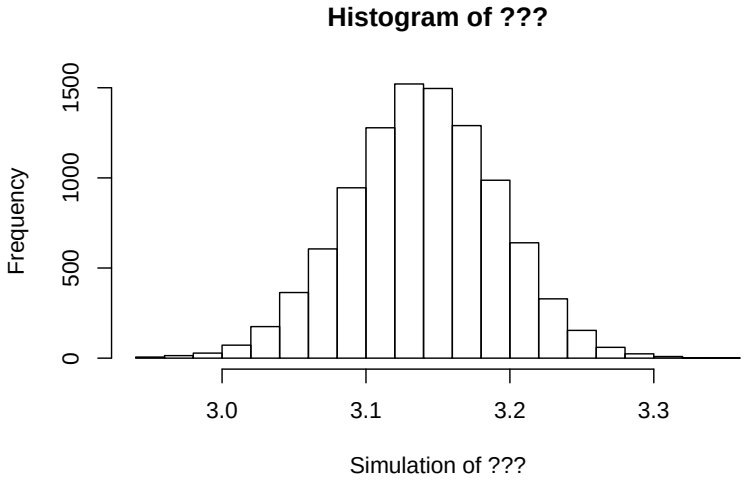
As always, we can place our simulation inside of a `for` loop to determine the sampling distribution of our statistic.

```
> set.seed(19203)
> reps   <- 10000
> N      <- 1000
> s.cool <- NULL
```

```
> for(i in 1:reps){
+
+   tmp.x     <- runif(N, min = -1, max = 1)
+   tmp.y     <- runif(N, min = -1, max = 1)
+
+   s.cool[i] <- 4 * mean(tmp.x^2 + tmp.y^2 <= 1)
+
+ } # END for i LOOP
```

And we also determine the mean, standard deviation, and shape of our sampling distribution.

```
> mean(s.cool)
[1] 3.14154
> sd(s.cool)
[1] 0.05215702
> hist(s.cool,
+      xlab = "Simulation of ???",
+      ylab = "Frequency",
+      main = "Histogram of ???")
```



Try the above code chunk with $N = 5000$, 10000, 50000, and describe what happens to `s.cool`.

6.3 Appendix: Functions Used

Here is a list of important functions from this chapter:

```
# Control Flow

## for loop ##
for(name in set.of.values){  # loop over indices
  commands
}

## if/else statements ##
if(logical){                 # if something holds, evaluate stuff
  commands
} else{                      # if not, then evaluate other stuff
  commands
}

## while loop ##
while(logical){              # while something holds, evaluate stuff
  commands
}

# More Graphs
hist(x, freq = FALSE, ... )   # hist with densities
lines(x, y, ... )             # draw lines on existing graph
expression(...)               # draw symbols in plots
qqplot(x, y, ... )            # compare the dists of x and y
```

Chapter 7

Functions and Optimization

Good evening. Tonight on "It's the Mind", we examine the phenomenon of déjà vu, that strange feeling we sometimes get that we've lived through something before, that what is happening now has already happened tonight on "It's the Mind", we examine the phenomenon of déjà vu, that strange feeling we sometimes get that we've... Anyway, tonight on "It's the Mind", we examine the phenomenon of déjà vu, that strange ...

—Monty Python's Flying Circus - Episode 16

7.1 Functions in R

7.1.1 An Outline of Functions

As explained in chapter 1, `R` is a functional programming language. Practically everything done in `R` is through various functions. Code that you would not even consider functions (e.g., addition, multiplication, pulling out objects, assignment) are functionally programmed. Therefore, being able to write functions is necessary for understanding `R`, so this chapter is designed to introduce you to rudimentary/basic functional programming. If you (for some strange reason) want to write your own packages and/or take advantage of generic functions in your own code, such as `summary`, `print`, and `plot`, you would need to learn the object oriented structure of `R` (I recommend Matloff, 2011 for a useful intro into `R` programming).

The basic format of a function is similar to much of `R`.

```
function(arguments){

  commands

  return statement

} # END FUNCTION
```

Functions are rather flexible, so that the above "psuedo-code" is the only form common to most of them, and even so, not all functions will have `arguments`, not all functions will have a `return statement`, and many functions will not even return anything. Yet I find it useful to diagram the "typical" function, bearing in mind that some functions will not follow the "typical" format. To declare a function, you first write `function` (which is the `function` function ... oy!) and then surround the `arguments` of the function in parentheses. The `arguments` are the intended user-inputs of the function (i.e., the *names* of stuff the user might want to change about how the function works) separated by commas. Each argument is usually of *one* of the following forms:

- A name *without* a default value (e.g., `data`).
  - If there is a name without a default value, the function writer often intends for the user to define the value him/herself and `R` will error if the user forgets.
- A name *with* a default value following an equal sign (e.g., `mean = 0`)
  - If there is a name with a default value, and if the user does not change this particular value when running the function, the argument will default to the default value.
  - An example of a default argument is `mean = 0` inside the `pnorm`, `qnorm`, or `rnorm` functions. If you do not indicate that `mean` should be different from 0, `R` will assume that your normal distribution should have a population mean of 0.
- A name with a *vector of possible values* following an equal sign (e.g., `method = c("pearson", "kendall", "spearman")` in the `cor` function).
  - Even if a particular argument defaults to a particular character string (e.g., `method` defaults to "`pearson`" inside `cor`), the function writer will often list all of the possible inputs in a vector so that the user knows the legal values of an argument.

Following initialization of the function, `commands` is a series of legal `R` commands/functions, separated by new lines and/or semi-colons (i.e., `;`) and surrounded by curly braces.

> **Note**: If you want `R` to perform a series of commands, regardless of whether or not they are inside of a function, `for` loop, `if` statement, set of curly braces, etc., you can either put each command on its own line or separate the series of commands with a semi-colon. Because `R` is generally not sensitive to white space, the semi-colon tells `R` when to stop evaluating one command and start evaluating the next.

When terminating a function, you generally want the function to output *something*. The default behavior of `R` is to return the last line of the function before the second curly brace. If the last line of a function is an assignment (i.e., something that includes `<-`), `R` will return the value *invisibly*, meaning that if you run a function without assigning its output to an `R` object, the output will be hidden from view, and all you will see is the next command prompt. If the last line of a function is a declaration of an object (e.g., `x` if `x` is visible to the function), then `R` will return the value *visibly*, meaning that if you

run a function without assigning its output to an object, R will display the output. Most of the functions that you have used thus far (including `mean`, `sd`, `rnorm`, etc.) display output visibly. If you want R to potentially return something *before* the last curly brace, you can type

```
return(x)      # x is the object you want to return
```

anywhere in the function, as long as R knows the value of `x` by that point, to return the object `x` visibly, or

```
return(invisible(x)) # x is the object you want to return
```

to return the object `x` invisibly.

> **Advice**: If you are not sure what R does when you run a function or write a function or write a `for` loop or ... really anything you want to do in R, run a few tests, try to predict the output, and determine if the output is what you predicted.

Just citing the theoretical underpinnings of functions is not particularly useful. More usefully, I can demonstrate R's functional capabilities with a few examples. The following is (almost) the simplest function that I could write.

```
function(x){

  x + 2

}
```

In fact, this (rather useless) function (unless for some reason, you really like adding "2" to things) is hopefully simple enough to illustrate the basic functional format. And all of the principles from the following examples will apply to (much) more complicated functions. The first thing that you could do with your new found function is write it without assigning it to anything.

```
> function(x){
+
+   x + 2
+
+ } # END FUNCTION
function(x){

  x + 2

}
```

And R will (happily) print the function back to you, but it will *not* calculate anything nor save it to an R object. So, you might be wondering: "How might I be able to write a function *and* run the function given some data?" One possible method of running the

function is by surrounding the function with curly braces followed by the arguments of the function inside parentheses.

```
> { function(x){
+
+   x + 2
+
+ } }(x = 3)  # if x = 3, then x + 2 = 5 for the win!
[1] 5
```

And R (cheerfully) prints the number 5. Yet this method is rather silly, as you would have to re-load the *entire* function every time you wanted to re-run it. Most R function writers will want to assign functions to object names (functions *are* objects just like objects *are* functions ... trippy!) so they can call the *names* of the functions as proxies for the function themselves. So we will assign the function to the name `blah`.

```
> blah <- function(x){
+
+   x + 2
+
+ } # END blah FUNCTION
```

Now every time we run the function `blah` with a different `x` value, R will add 2 to `x`. As long as `x` is an object for which the "add" operator works, R will oblige your functional desires.

```
> # Add 2 to: a numeric scalar, a numeric vector, and a numeric matrix
> blah(x = 3)
[1] 5
> blah(x = c(1, 2, 3))
[1] 3 4 5
> blah(x = matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2, byrow = TRUE))
     [,1] [,2]
[1,]    3    4
[2,]    5    6
```

But if you try to run the function `blah` with an `x` object for which the "add" operator does not make sense, R will either error, warn you of its misdeeds, or give nonsensical results.

```
> # An error because we cannot add to a letter.
> try(blah(x = "s"), silent = TRUE)[1]
[1] "Error in x + 2 : non-numeric argument to binary operator\n"
```

We might not be satisfied with the original construction of the `blah` function. One might think it wise to set a default value to the `x` argument of `blah`.

```
> blah <- function(x = 2){
+
+   x + 2
+
+ } # END blah FUNCTION
```

Now we can run the function without hand picking a value for `x` because `R` will automatically sets `x = 2`.

```
> # To run the function "defaultly":
> # --> type the function name followed by two parentheses.
> blah()
[1] 4
```

> **Important**: To run a function, you always need to surround your arguments in parentheses. If the function has default values for all of its arguments, and if you want to use all of the defaults, you only need to write the function name followed by two parentheses with nothing in them. We already saw two parentheses with nothing in them for the `ls` function described in chapter 4.

But functions are not useful just in their calculating abilities. We could also assign the output of `blah` to a different object rather than *just* displaying on the second line of the function. Let's call the object `y`. Then if we run the function without assigning the function run to another object, `R` will not display anything.

```
> blah <- function(x){
+
+   y <- x + 2
+
+ } # END blah FUNCTION
> # Run the function without assignment.
> blah(3)         # R doesn't display anything.
> # Run the function with assignment.
> eek <- blah(3)  # R doesn't display anything.
> # Call eek by itself
> eek             # R displays 5 ... weird!
[1] 5
```

But if we assign the function run to `eek` and then type `eek` into our `R` session, `R` displays the result of the function call. Finally, if we type `return` anywhere inside our function, `R` will usually return whatever is inside of `return` rather than the last line of the function.

```
> ####### NO RETURN #########
> blah <- function(x){
+
+   y <- x + 2
+   z <- x + 3
+   z
+
+ } # END blah FUNCTION
> blah(3)  # R returns 6 (z) and NOT 5 (y)
[1] 6
> ############################
> ######## A RETURN #########
```

```
> blah <- function(x){
+
+   y <- x + 2
+   return(y)   # return before the last line.
+
+   z <- x + 3
+   z
+
+ } # END blah FUNCTION
> blah(3)  # now R returns 5 (y) and NOT 6 (z)
[1] 5
> ############################
```

Most of the time, you will write functions for two purposes:

1. Simple functions that take a vector, matrix, or even a bunch of arguments and return a scalar value.
2. Complex functions that take a bunch of arguments and return a list.

The former functions are used to summarize data (such as the mean, variance, standard deviations) and optimize a range of specifications with respect to some objective function. The latter functions are used to perform a set of actions and return descriptions about that set.

7.1.2 Functions and Sampling Distributions

R functions (as is *almost anything* in R) are best understood by providing and breaking down examples. In the last chapter, I discussed creating sampling distributions using `for` loops. Because the generic structure of a sampling distribution is always the same, but because sampling distributions frequently take many lines of code, one might save time and space by putting the general "sampling distribution" outline inside of a function.

```
> sampDist <- function(dist = c("norm", "unif", "t", "chisq"),
                       stat = mean, size = 10, reps = 10000, ... ){

  # Getting the appropriate distribution:
    if(dist == "norm"){
      rdist <- function(n) rnorm(n, ... )
    } else if(dist == "unif"){
      rdist <- function(n) runif(n, ... )
    } else if(dist == "t"){
      rdist <- function(n) rt(n, ... )
    } else if(dist == "chisq"){
      rdist <- function(n) rchisq(n, ... )
    } else{
      stop("pick a legal dist: norm, unif, t, chisq")
    } # END if STATEMENTS
```

```
# Indicating a NULL vector to store samples:
  samp <- NULL

# Repeat this a lot (reps number) of times.
  for(i in 1:reps){

# Each time:
#  - take a sample from the distribution
#  - calculate your statistic on the sample
    samp[i] <- stat( rdist(n = size) )

  } # END for i LOOP

# Put the samples and mean/sd of the samples into a list:
  out <- list(mn.samp = mean(samp), sd.samp = sd(samp), samp = samp)

# Return the list to R!
  return(out)

} # END sampDist FUNCTION
```

The `sampDist` function is designed to calculate a sampling distribution of some statistic given a particular distribution, sample size, and number of replications. The `sampDist` function also takes the following arguments.

- `dist`: The particular population distribution. The options are: "norm" for normal distribution, "unif" for uniform distribution, "t" for t-distribution, and "chisq" for χ^2 distribution. Any other input for `dist` will result in an error.
- `stat`: The statistic of which the sampling distribution is composed. `stat` can be *any* function that returns a scalar argument.
- `size`: The size of *each* sample.
- `reps`: The number of replications (i.e., the number of samples to take) to construct the sampling distribution. Note that more samples are always better as sampling distributions are theoretically infinite.
- `...`: A placeholder for user specified arguments. In our case, `...` represents the parameters of the distribution. For instance, if you wanted to change the mean and standard deviation of the normal distribution function, you would set `mean = 2` in the `...` portion of the function call, and `R` would know to change the mean to 2. For this function, `...` is useful because every distribution refers to a different set of parameters (e.g., the normal distribution is parameterized with `mean` and `sd` whereas the uniform distribution is parameterized with `min` and `max` and the t-distribution is parameterized `df`). Using `...` prevents the function writer from having to specify *all* of the arguments for *all* of the possible distributions.

After setting the arguments, the first few lines of the function are intended to determine the population distribution. For instance, if `dist == "norm"`, then `R` assigns `rdist` (a

generic name) the function `rnorm` with the user specified parameters. And for the rest of the function, `rdist` is synonymous with `rnorm`. Cool! `R` then declares an empty vector, `samp`, in which to store statistics. Finally, `R` repeatedly samples from the distribution, calculates statistics on the distribution, and assigns those statistics to the `i`th place of `samp`. Unlike building a sampling distribution by only using `for` loops, `rdist` is not *really* a function in `R` but is assigned an already existing distribution, and `stat` is not *really* a function to calculate a statistic in `R` but is assigned an already existing function. Thus, by indicating that `stat = median` rather than `stat = mean`, `R` will construct a sampling distribution of the median. Once the `for` loop is finished, the last few lines of the function are designed to create a `list` of interesting statistics (e.g., the mean, standard deviation, and vector of observations) and return that list to the user. And after saving the function call into an object, the user can extract the mean, standard deviation, and vector of observations with the `$` operator.

```
> set.seed(78345)
> s.dist1 <- sampDist(dist = "norm", stat = median, size = 20)
> # What sub-objects are saved in s.dist1?
> names(s.dist1)
[1] "mn.samp" "sd.samp" "samp"
> s.dist1$mn.samp  # pulling out the sampling distribution mean.
[1] -0.0008891166
```

Of course, sampling distribution lectures usually start with properties of the sampling distribution of the mean when sampling from a normal population. And who cares about the `...` arguments when you will always be sampling from a normal distribution where the parameters are `mean` and `sd`?

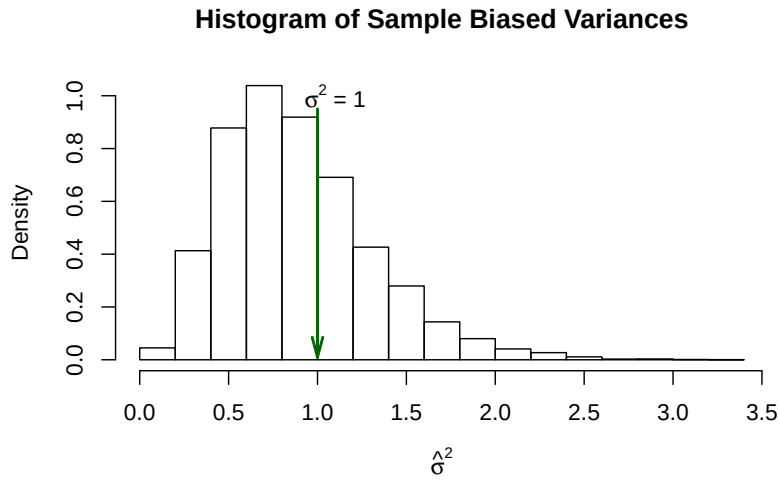
```
> sampNorm <- function(stat = mean, size = 10, reps = 10000,
                       mean = 0, sd = 1){

  # Indicating a NULL vector to store samples:
    samp <- NULL

  # Repeat this a lot (reps number) of times.
    for(i in 1:reps){

  # Each time:
  #  - take a sample from the distribution
  #  - calculate your statistic on the sample
      samp[i] <- stat( rnorm(n = size, mean = mean, sd = sd) )

    } # END for i LOOP

  # Put the samples, and mean/sd of the samples into a list:
    out <- list(mn.samp = mean(samp), sd.samp = sd(samp), samp = samp)

  # Return the list to R!
    return(out)

  } # END sampNorm FUNCTION
```

Unlike `sampDist`, `sampNorm` always samples from a normal distribution. In both functions, the `stat` argument is a *function* designed to calculate a statistic, and one can set `stat` to typical functions, such as the mean, median, and variance.

```
> set.seed(666777888)
> s.dist2 <- sampNorm(stat = mean, size = 10)
> s.dist2$mn.samp
[1] -0.003181224
> s.dist2$sd.samp
[1] 0.3143386
> s.dist3 <- sampNorm(stat = median, size = 10)
> s.dist3$mn.samp
[1] -0.003582825
> s.dist3$sd.samp
[1] 0.3761127
> s.dist4 <- sampNorm(stat = var, size = 10)
> s.dist4$mn.samp
[1] 1.00184
> s.dist4$sd.samp
[1] 0.4726863
```

But sometimes, you might want to construct your own statistic function and use that function as `stat` inside `sampNorm`.

```
> # A function to calculate the biased variance:
> biasVar <- function(x){
+
+   N <- length(x)
+   return( sum( (x - mean(x))^2 )/N )
+
+ } # END biasVar FUNCTION
> # Using the biased variance as our "stat":
> set.seed(92939495)
> s.dist5 <- sampNorm(stat = biasVar, size = 10)
> # Showing that the variance is biased by using a histogram.
> # (Figure out the use of "arrows" and "text" on your own).
> hist(s.dist5$samp, freq = FALSE,
+      xlab = expression(hat(sigma)^2), ylab = "Density",
+      main = "Histogram of Sample Biased Variances")
> arrows(x0 = 1.0,  x1 = 1,
+        y0 = 0.95, y1 = 0.01,
+        length = .15, angle = 15, col = "darkgreen", lwd = 2)
> text(x = 1.1, y = 1.0,
+      labels = expression(paste(sigma^2, " = 1", sep = "")))
```



Note that the `stat` function can be *anything* that returns a scalar value.

Now that you know how to write functions, frequently, you will need to provide formatted output so that users can understand the contents of those functions. Several functions in `R` are designed to format output for printing, some of which are based off of similar C functions, but I will only describe the most basic functions that most `R` programmers use.

7.1.3 Formatting Output

Printing and writing allow an `R` programmer to format the output of `R` functions so that the end-user can see interesting/relevant calculations. And the two major methods of printing/writing are

```
print(x, digits, ...)
```

and:

```
cat(..., sep = " ")
```

The print Function

The `print` function takes an object and prints it on the next line, and the `cat` function takes a series of values and character strings, pastes them together into one long character string, and displays that long character string. Many of the advanced `R` functions use `print` and `cat` statements to concisely display results.

```
> x <- c(1, 2, 3)  # it's easy as 1-2-3!
> print(x)
[1] 1 2 3
```

The above code assigns the vector `c(1, 2, 3)` to `x` and displays it on the line after the `print` statement. Now you might be thinking: "Steve, your code is silly! I could have just typed `x`, and the values 1, 2, 3 would have also been displayed." I agree with your precociousness! However, there are many benefits to using the `print` function rather than just typing the variable you want to print. For example, you can assign an object *inside* of a `print` statement, and `R` will appropriately assign the object and print the result of that assignment at the same time. The print + assign technique is great for those of you who like to write all of his/her code on one line (keep in mind that `R` does prefer if you press "Return" every once in a while).

```
> # See, it prints, and everything is on one line!
> print(x <- c(1, 2, 3))
[1] 1 2 3
```

Note that *only one* command can go inside of a particular `print` statement. If you try to print *two* (or more) things, separated by semi-colons, `R` will yell at you and refuse to print them.

```
> # Hear the yell of illegalness!
> print(x <- c(1, 2, 3); y <- c(4, 5, 6))
Error: unexpected ';' in "try(print(x <- c(1, 2, 3);"
```

> **Note**: You can only surround one assignment/object/thing inside of a print statement. Even though one can separate `R` assignments with semi-colons, `R` can only print (via `print`) one thing at a time.

Another use of `print` is to indicate the desired number of printed digits, simply by adding the `digits` argument to the `print` statement. One often simulates data that has (nearly) an infinite number of digits, but he/she usually wants to (on the fly) reduce the number of displayed digits to something much less than infinity!

```
> # If we print without setting the digits:
> set.seed(1983)
> print(x <- rnorm(10))
 [1] -0.01705205 -0.78367184  1.32662703 -0.23171715
 [5] -1.66372191  1.99692302  0.04241627 -0.01241974
 [9] -0.47278737 -0.53680130
> # But sometimes three digits is enough:
> set.seed(1983)
> print(x <- rnorm(10), digits = 3)
 [1] -0.0171 -0.7837  1.3266 -0.2317 -1.6637  1.9969  0.0424
 [8] -0.0124 -0.4728 -0.5368
```

Of course, you could also change the default `R` options to print fewer digits.

The cat Function

But if you want to print more than pre-formatted objects, you must design your own printing using the `cat` function. Unlike `print` (which can print entire matrices, lists, vectors, etc.), `cat` always prints a concatenated string of letters.

```
> x <- matrix(c(1, 2,
+               3, 4), nrow = 2, byrow = TRUE)
> print(x)  # prints all of x, as a nice matrix
     [,1] [,2]
[1,]    1    2
[2,]    3    4
> cat(x)    # wtf!??? turns it into a vector? annoying?!
1 3 2 4
```

The `print` statement printed the appropriate matrix, whereas the `cat` function turned the matrix into a vector and *sort of* printed the vector. And unless one specifies a new line *inside* of `cat`, `R` will put the command prompt right next to the end of the printing ... on the *same* line. The commands (also called escape sequences) to "enter", "tab", and other fantastical things, are similar to other computer languages:

```
"\n":       New Line
"\t":       Horizontal Tab
"\v":       Vertical Tab
"\a":       Bell
"\'":       Double Quote
```

You only must include one of those escape sequences (in quotes) for `R` to perform the appropriate action.

```
> cat(x, "\n")  # prints x and goes onto the next line
1 3 2 4
```

Notice that the previous code chunk highlights the layout of printing inside of a `cat` function. The `cat` function will concatenate several `R` objects (separated by commas) into one printed statement.

```
> set.seed(818)
> x <- 2        # some number
> y <- 1        # another number
> p <- rnorm(1) # another (random) number
> # A very complicated cat statement:
> cat("\tMy", "mother", "told", "me", x, "pick", "the",
+     "very", "best", y, "and", p, "is", "it.\n")
        My mother told me 2 pick the very best 1 and -0.2373162 is it.
>
> # - The \t at the beginning tabs over,
> # - The \n at the end puts the command prompt on the next line,
> # - Everything else is separated by commas, including objects.
```

By default, `cat` separates all of the objects/words that it prints by a space, but you can alter the spacing of `cat` by adding a `sep` argument to the end of the function.

```
> cat("My", "mother", p, "\n", sep = "")     # no separation
Mymother-0.2373162
```

```
> cat("My", "mother", p, "\n", sep = ",")    # comma separation
My,mother,-0.2373162,
> cat("My", "mother", p, "\n", sep = " ")    # space separation
My mother -0.2373162
> cat("My", "mother", p, "\n", sep = "\n")   # new line separation
My
mother
-0.2373162
> cat("My", "mother", p, "\n", sep = "\t")   # tab separation
My        mother        -0.2373162
> cat("My", "mother", p, "\n", sep = " x ")  # umm ... ???
My x mother x -0.2373162 x
```

Note that you can enter: (1) a sequence of letters (or escape sequences) within quotes; and (2) an R object without quotes. Make sure not to accidentally put an R object within quotes nor character strings without quotes.

Now harken back to a few paragraphs ago (... *harkening*). I indicated that a typical use for `print` and `cat` is to display text inside of an R function. By adding a few lines of code inside the sampling distribution function, the end user will be better attuned to the resulting output.

```
> ## STUFF FROM EARLIER ##
> sampNorm2 <- function(stat = mean, size = 10, reps = 10000,
                        mean = 0, sd = 1){

    samp <- NULL

    for(i in 1:reps){
      samp[i] <- stat( rnorm(n = size, mean = mean, sd = sd) )
    }

    out <- list(mn.samp = mean(samp), sd.samp = sd(samp), samp = samp)
  ############################

  ## STUFF FROM NOW ##

  # Now, we can print the mean and the standard deviation:
  cat("\n\n\t", "Sampling Dist Mean: ", round(mean(samp), 3), sep = "")
  cat("\n\t"  , "Sampling Dist SD:   ", round(sd(samp), 3),   sep = "")

  # And we can print a few vector elements:
  cat("\n\n  ", "A Few Observations:\n")
  print(samp[1:6], digits = 2)

  # And adding some more space:
  cat("\n")

  ############################
```

```
  ## STUFF FROM EARLIER ##
    return(out)

  }
> ##########################
```

By calling the *new* `sampNorm` function, `R` displays pretty output even before we analyze the output ourselves.

```
> set.seed(1827)
> mod.samp <- sampNorm2(stat = mean, size = 20)
        Sampling Dist Mean: 0.003
        Sampling Dist SD:   0.223

   A Few Observations:
[1]  0.0083  0.4794  0.0422 -0.0224  0.1542 -0.1507
```

Designing print statements inside of functions is more art than science and usually involves manipulating `cat` and `print` functions to force the output to display something reasonably attractive to the mass of non-aesthetically devoid programmers. `R` programmers also use trickery to achieve pretty printing whenever you call an object by name (and not just when you run the function), but advanced printing is for another book.

The paste Function

A device useful for assisting printing, assignment, saving, etc. is the `paste` function. I find `paste` solves many of my `R` manipulation needs. The `paste` function takes a bunch of objects, strings, etc., just like `cat` (or `c` for that matter), and combines them into character strings. `paste` behaves differently depending on the inserted objects.

- A bunch of character strings/numbers separated by commas results in:
  - $\rightarrow$ A very long string.
  - $\rightarrow$ Separated by whatever you put in `sep`.

```
> # A bunch of possible options:
> w <- paste("a", "b", "c", 2, "d", 4, sep = "")
> x <- paste("a", "b", "c", 2, "d", 4, sep = " ")
> y <- paste("a", "b", "c", 2, "d", 4, sep = ",")
> z <- paste("a", "b", "c", 2, "d", 4, sep = "\n")
> # Printing z does not print the new lines, as "\n" is added:
> print(z)

[1] "a\nb\nc\n2\nd\n4"

> # You can print the new lines by surrounding z with cat:
> cat(z)
```

```
a
b
c
2
d
4
```

- A vector of whatever and a bunch of character strings separated by commas results in:
  - → Each vector element separately pasted to the sequence of character strings.
  - → Separated by whatever you put in `sep`.

```
> x <- c(1, 2, 3, 4, 5)
> # A vector and strings/numbers in whatever order you want:
> mo <- paste(x, "s", sep = "")
> po <- paste("s", x, sep = "")
> bo <- paste(x, "s", "t", 1, 2, sep = ",")
> # See the wackiness:
> mo
[1] "1s" "2s" "3s" "4s" "5s"
> po
[1] "s1" "s2" "s3" "s4" "s5"
> bo
[1] "1,s,t,1,2" "2,s,t,1,2" "3,s,t,1,2" "4,s,t,1,2"
[5] "5,s,t,1,2"
```

- A bunch of vectors, all of the same length results in:
  - → Individual vector elements pasted together.
  - → Separated by whatever you put in `sep`.

```
> x <- c(1, 2, 3, 4, 5)
> y <- c("a", "b", "c", "d", "e")
> z <- c(7, 8, 9, 10, 11)
> # A bunch of vectors (the same length) put together:
> peep <- paste(x, y, sep = "")
> beep <- paste(x, y, z, sep = "p")
> # See the wackiness:
> peep
[1] "1a" "2b" "3c" "4d" "5e"
> beep
```

```
[1] "1pap7"  "2pbp8"  "3pcp9"  "4pdp10" "5pep11"
```

Each modification of the `paste` function results in slightly different output.

There are many benefits to using the `paste` function, but one of the simplest uses is being able to easily label the rows and columns of a matrix.

```
> # X is a 4 x 2 matrix (subjects x variables):
> X <- matrix(c(1, 2,
+               3, 4,
+               5, 6,
+               7, 8), nrow = 4, ncol = 2)
> # Easily identify individual subjects/variables using paste:
> colnames(X) <- paste("v", 1:2, sep = "")
> rownames(X) <- paste("s", 1:4, sep = "")
> # See - awesomeness:
> X
   v1 v2
s1  1  5
s2  2  6
s3  3  7
s4  4  8
```

Of course, `paste` can also assist in naming the variables in a `data.frame` and elements of a single vector.

A slightly more complicated use of the `paste` function is to take a two vectors, combine both vectors together, and then collapse the new vector into one character string. You can combine and collapse vectors of character strings by using the `collapse` argument of `paste` just as you would the `sep` argument.

```
> # x is a vector:
> x <- c(1, 2, 3, 4)
> # Without collapse, the output would be a vector:
> paste(x, "s", sep = "")
[1] "1s" "2s" "3s" "4s"
> # But collapse connects the vector elements:
> paste(x, "s", sep = "", collapse = ",")
[1] "1s,2s,3s,4s"
```

By using the `collapse` argument: *first*, `paste` creates a character vector by combining several character vectors together (separating them by whatever is in the `sep` argument); and *second*, `paste` creates a character string by combining all of the elements of the character vector together (separating them by whatever is in the `collapse` argument).

Now why would any self respecting person want to know how to write and format `R` functions? Well, much of statistical analysis require finding maximums or minimums of stuff. And much of the time, the optimum value cannot be solved for analytically. Not surprisingly, `R` has reasonable optimization mechanisms, although if one needs to optimize a tricky function, he/she is better served by hand programming that function in a different language.

7.2 Simple Optimization

Most of statistics is concerned with finding the *best* of something, usually the best estimator for a particular parameter, and we (people who do statistics) frequently define "best" in terms of how close it is, on average, to the parameter. Trying to find the distance between an estimate and its parameter is a minimization problem (or a maximization problem if we quantify "best" using other means). Because we can never perfectly predict the parameter, we try to find an estimate that is close enough: we (1) pick some parameter estimation function (called "loss function" by really cool statisticians); and (2) try to pick the estimator that minimizes the chosen function. If all of this is a tad bit over your head, just note that minimization/maximization of functions is rather important to the applied statistician.

`R` has reasonable, although not necessarily the best nor most efficient, optimization algorithms: `optim`, `nlm`, `nlmimb`, `trust`, etc., most of which are rather difficult to understand and apply to practical problems. A (mostly complete) list of optimization functions can be accessed on the following webpage.

http://cran.r-project.org/web/views/Optimization.html

Even though I still get confused as to how to choose and use optimization functions, I stick by the following rule of thumb.

> **Rule of Thumb**: If I am trying to optimize a function with multiple parameters or arguments, I will use `optim` or `nlm`; if I am trying to optimize a (smooth, err ... nice) function with *only one* parameter or argument, I will use `optimize`; and if I am trying to optimize a very odd function, something more specific, not smooth, and/or with constraints, I will try to dig through the above webpage to find a better optimization routine.

And if one optimization routine does not seem to work, takes very long, or results in solutions that do not make sense, I will try another one.

Although many optimization functions exist in `R`, I will briefly outline only the most basic optimization algorithm, `optimize`, which has the following form.

```
optimize(f, interval, lower, upper, maximum = FALSE, ... )
```

`optimize` uses (effectively) a bisection search routine, which (effectively) sounds like gibberish to me but impresses people that I don't know at the two or three parties (per decade) that I attend. Of course, saying "a bisection search routine" too often has the unfortunate consequence of not resulting in many party invitations.

In `optimize`, `f` is a function, either user specified or something already existing in `R`. The only limit on the `f` function is that one of the function inputs must be a scalar value, and the function must return a scalar. `R` will then try to find the function input that results in the minimum (or maximum) output. Then one needs to set boundaries for the optimization search. `interval` is a vector of two values, indicating the minimum and maximum of the search, whereas `lower` and `upper` are the minimum and maximum scalar value of the search (and one needs to only include either `interval` or both of `lower`/`upper`). `maximum` is a logical value (TRUE or FALSE) indicating whether `R` should search for the minimum of the function *or* the maximum of the function. Finally, `...` is the annoying `...` argument, which (effectively) represents *all* of the arguments to the `f`

function, *called by name* and *separated by commas* (except for the scalar input supposed to result in the maximum or minimum output).

I can see (err ... read ... err ... hypothesize about) your confusion, so I will go over some examples of when to use `optimize`. Pretend that you have a simple function that returns $x^2 + 1$ given a scalar value `x`.

```
> x.sq <- function(x){
+
+   x^2 + 1
+
+ } # END x.sq FUNCTION
```

The `x.sq` function is a rather nice function, if I do say so myself. $f(x) = x^2 + 1$ is entirely continuous (and differentiable), has a minimum at $x = 0$ and evaluates to a minimum of $y = 1$. We can double check our assertion as to the minimum of $f(x)$ by plugging the function into `optimize`. Because `x.sq` only has one argument (which we want to maximize over), we do not need to enter anything into the `...` part of `optimize`.

```
> optimize(f = x.sq, lower = -2, upper = 2, maximum = FALSE)
$minimum
[1] -5.551115e-17

$objective
[1] 1
```

And `R` quickly tells us that the minimum (of `x`) is a value close to 0 (annoying computing error) with an objective (i.e., y) of 1 ... exactly as predicted.

> **Important**: When you are using optimization functions, or (really) any function with function arguments, you must put a *function* as the argument. What that means is that if you put `f = x^2`, `R` would give you an error because `x^2` is a combination of object and function. However, if you put `f = function(x) x^2`, then `R` would display the appropriate answer. Thus the most reliable course of action is to write an entire function prior to using `optimize` and stick that function (without quotes) into the function argument.

Note that the `optimize` function essentially subverts all of first semester, differential calculus by algorithmically finding the minimums/maximums of smooth functions.

```
> x.sq2 <- function(x){
+
+   4*x^2 + 6*x
+
+ } # END x.sq2
> optimize(f = x.sq2, lower = -2, upper = 2, maximum = FALSE)
$minimum
[1] -0.75

$objective
[1] -2.25
```

But there is a better reason for optimization in statistics. Pretend that we want to find the mode of a continuous distribution. We know that `d` (followed by the distribution) gives the density at any point. And for the most basic, continuous distributions, the densities are nice and smooth. Therefore, we can find the `x` that maximizes the density (and therefore is the mode of the distribution) using `optimize`.

The easiest distribution to maximize is the normal distribution. Because the normal distribution is symmetric and unimodal, the mean and mode are equivalent. So by setting `f` to `dnorm` in the `optimize` function, we should end up with the mean/mode as the maximum.

```
> optimize(f = dnorm, lower = -4, upper = 4, maximum = TRUE)
$maximum
[1] 3.330669e-16

$objective
[1] 0.3989423
```

Note the two differences between the most recent use of `optimize` and the previous examples. First, `dnorm` is a built in `R` function, which takes a scalar (or vector) input `x` and returns the density at that (or those) point(s). Second, rather than using `maximum = FALSE`, as we did to find the minimum of the `x.sq` and `x.sq2` function, the mode of a distribution is a maximum. Therefore, only by changing the argument `maximum = TRUE` will `R` search for the correct value.

For fun, we could find the maximum of *other* normal distributions. How could we experience this bursting of pleasure? We would have to take advantage of the `...` argument to insert our custom mean and standard deviation as *new* values of `mean` and `sd`.

```
> # 1) Standard arguments: f, lower, upper, maximum.
> # 2) Specific arguments: mean, sd.
> optimize(f = dnorm, lower = -4, upper = 4, maximum = TRUE,
+          mean = 2, sd = 3)
$maximum
[1] 2.000001

$objective
[1] 0.1329808
```

And despite rounding error, `R` provides the correct result.

Using `optimize`, we can also find the mode of a less familiar distribution, such as the χ^2 distribution, by using a different density function. In the case of χ^2 we would use the `dchisq` function. The idea behind using `dchisq` is exactly the same as behind using `dnorm`, but rather than specifying `mean` and `sd` (as for the normal), we should specify `df`.

```
> optimize(f = dchisq, lower = 0, upper = 10, maximum = TRUE,
+          df = 5)
$maximum
[1] 2.999987
```

```
$objective
[1] 0.1541803
```

And the value corresponding to the maximum of this particular χ^2 distribution is approximately 3. Note that the mean and mode are different for χ^2 distributions. The mean of the χ^2 distribution with $df = 5$ is approximately 5, but the mode is approximately 3.

```
> set.seed(987)
> mean( rchisq(n = 1000000, df = 5) )
[1] 4.999766
```

As always, to use `optimize`, we stick the function we want to maximize/minimize into `f`, the lower and upper values for our search into `lower` and `upper`, whether we are minimizing or maximizing into `maximum`, and the remaining arguments of the function (specified by name and separated by commas) following `maximum`. You might be wondering how I knew the lower/upper values of the optimal search. Much of the time, I use "experience" to pick appropriate values, but if you are not sure, "trial and error" might be the best option. Or you can evaluate a range of values, plot them, and try to figure out approximately where the min/max of the plot is located.

We can (of course) save the `optimize` run into another `R` object and pull out particular values using the `$` operator.

```
> # Running (and saving) the chi-square optimization.
> m.chi <- optimize(f = dchisq, lower = 0, upper = 10, maximum = TRUE,
+                   df = 5)
> m.chi$maximum    # pulling out the maximum (x)
[1] 2.999987
> m.chi$objective  # pulling out the density at the maximum (y)
[1] 0.1541803
```

Try to experiment more with `optimize` on your own.

7.3 A Maximum Likelihood Function

One of the uses of optimization is to return the maximum of a likelihood function. The following code takes a vector of data, a specification of *which* parameter to calculate the likelihood with respect to (either the mean or variance), and a vector of user-defined estimates of that parameter.

```
> ########################
> # Likelihood Function #
> ########################
>
> # This function is designed to take:
> # - a vector of observations from a N(0, 1) dist,
> # - a character indicating the estimated parameter, and
> #   - it assumes that the var is the biased var if est the mean
> #   - it assumes that the mean is xbar if est the var
> # - a vector of (numerical) estimates of the unknown parameter.
```

```
>
> # And return:
> # - a list of interesting estimation points, and
> # - the log-likelihood corresponding to those estimation points.
>
> likNorm <- function(dat, which = c("mean", "var"), ests){

  #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
  # Arguments:                                                      #
  # - dat   - a vector of observations                              #
  # - which - the estimated parameter                               #
  # - ests  - a vector of estimates of that par                     #
  #                                                                 #
  # Values:                                                         #
  # - ests     - the original estimates                             #
  # - ests.lik - the log.lik at the ests                            #
  # - MLE      - the maximum likelihood estimate                    #
  # - MLE.lik  - the log.lik at the MLE                             #
  # - eval     - a vector of values for the log.lik                 #
  # - eval.lik - the log.lik at the "eval" vector                   #
  #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#

  ## 1. A FEW STATISTICS OF OUR DATA ##
    N     <- length(dat)         # the number of obs
    xbar  <- mean(dat)           # the sample mean
    v.dat <- var(dat)*(N - 1)/N  # the MLE (biased) var

  ## 2. CALCULATING THE LOG-LIK WITH RESPECT TO MU ##
    if(which == "mean"){

  # First, writing a simple function to find the log-likelihood:
      log.lik <- function(eval, dat){

        lik.i <- dnorm(x = dat, mean = eval, sd = sqrt(v.dat))
        sum( log(lik.i) )

      } # END log.lik FUNCTION

  # Second, picking a vector of values to evaluate the likelihood:
      eval <- seq(-4 / sqrt(N), 4 / sqrt(N), by = .01)

  # Third, finding the likelihood for our vector of observations:
  #  a) at our estimates, and
  #  b) at the evaluation points.
      ests.lik <- sapply(X = ests, FUN = log.lik, dat = dat)
      eval.lik <- sapply(X = eval, FUN = log.lik, dat = dat)

  # Fourth, finding the MLE
```

```
    max.lik <- optimize(f = log.lik,
                        lower = min(eval), upper = max(eval),
                        maximum = TRUE, dat = dat)
    MLE     <- max.lik$maximum    # the maximum
    MLE.lik <- max.lik$objective  # the log-lik at the maximum

## 3. CALCULATING THE LOG-LIK WITH RESPECT TO SIGMA^2 ##
  } else if(which == "var"){

# First, writing a simple function to find the log-likelihood:
    log.lik <- function(eval, dat){

      lik.i <- dnorm(x = dat, mean = xbar, sd = sqrt(eval))
      sum( log(lik.i) )

    } # END log.lik FUNCTION

# Second, picking a vector of values to evaluate the likelihood:
    eval <- seq(qchisq(p = .01, df = N - 1)/(N - 1),
                qchisq(p = .99, df = N - 1)/(N - 1),
                by = .01)

# Third, finding the likelihood for our vector of observations:
#  a) at our estimates, and
#  b) at the evaluation points.
    ests.lik <- sapply(X = ests, FUN = log.lik, dat = dat)
    eval.lik <- sapply(X = eval, FUN = log.lik, dat = dat)

# Fourth, finding the MLE
    max.lik <- optimize(f = log.lik,
                        lower = min(eval), upper = max(eval),
                        maximum = TRUE, dat = dat)
    MLE     <- max.lik$maximum    # the maximum
    MLE.lik <- max.lik$objective  # the log-lik at the maximum

## 4. GIVING AN ERROR IF THE INPUT WAS INCORRECT ##
  } else{

          stop("you must pick either 'mean' or 'var' for 'which'")

  } # END if STATEMENTS

## 5. PRINTING SOME PRETTY THINGS ##
  cat("\nYou found the MLE for the ", which, ":",
      "\n\n", sep = "")
  cat("  The value that maximizes the likelihood function: ", round(MLE, 3),
      "\n",   sep = "")
  cat("  The log-likelihood at the maximum: ", round(MLE.lik, 3),
```

```
        "\n\n", sep = "")

  ## 6. PUTTING INTO A LIST AND RETURNING ##
    out <- list(ests = ests, ests.lik = ests.lik,
                MLE  = MLE,  MLE.lik  = MLE.lik,
                eval = eval, eval.lik = eval.lik,
                which = which)

    return(out)

  } # END lik.norm FUNCTION
```

After loading the function into R, you first need to simulate data from a normal distribution, pick the parameter you want to estimate, and estimate that parameter in various ways, putting all of those estimates into the same vector.

```
> set.seed(289734)
> dat  <- rnorm(30)        # 30 draws from a standard normal dist.
> ests <- c(var(dat), 4.2) # estimates of the variance.
> # Running the likelihood/maximization function:
> l.mod <- likNorm(dat = dat, which = "var", ests = ests)
You found the MLE for the var:

  The value that maximizes the likelihood function: 1.363
  The log-likelihood at the maximum: -47.212
```

The `l.mod` object contains various sub-objects, the names of which are revealed by using the `names` function.

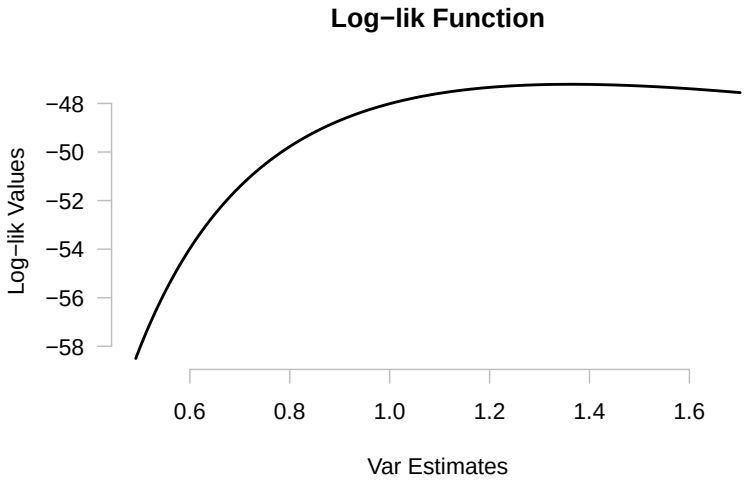
```
> names(l.mod)
[1] "ests"     "ests.lik" "MLE"      "MLE.lik"  "eval"
[6] "eval.lik" "which"
```

Note that `ests` is the vector of estimates, `ests.lik` is the log-likelihood values at the estimates, `MLE` is the maximum likelihood estimate (using `optimize` ⌣̈), `MLE.lik` is the log-likelihood value at the MLE, `eval` is a vector of evaluation points near the maximum, `eval.lik` are the log-likelihood values at the evaluation points, and `which` is a character string indicating whether we maximized the likelihood with respect to the mean or variance. We can pull each of the sub-objects out of the `l.mod` object using the `$` operator.

```
> l.mod$ests           # the estimates (that we plugged in)
[1] 1.40986 4.20000
> l.mod$ests.lik       # the log-lik at the estimates
[1] -47.22052 -53.96180
> l.mod$MLE            # the maximum likelihood estimate
[1] 1.362881
> l.mod$MLE.lik        # the log-lik at the MLE
[1] -47.21199
```

We can even plot the log-likelihood function by pulling out `eval` and `eval.lik` and connecting the dots.

```
> plot(x = l.mod$eval, y = l.mod$eval.lik,
+      xlab = "Var Estimates", ylab = "Log-lik Values",
+      main = "Log-lik Function",
+      type = "l", lwd = 2, axes = FALSE)
> axis(1, col = "grey")
> axis(2, col = "grey", las = 1)
```



Most of the useful optimization problems in statistics will take the form of `likNorm`, so play around with `likNorm` to better understand the logic of function construction and optimization. Only then can you become an R-master.

7.4 Appendix: Functions Used

Here is a list of important functions from this chapter:

```
# Functions
function(arguments){  # a function to do cool stuff
  commands
  return statement
}
...                   # the "dot-dot-dot" argument
return(x)             # return "x" from the function
invisible(x)          # don't display "x" (weird!)

# Optimization
optimize(f, lower, upper, maximum)  # find the min or max of the
                                    # function f
sapply(X, FUN, ...)                 # perform the function on each
                                    # element of a vector/matrix

# Printing and Formatting
print(x, digits)           # print an object
cat(..., sep)              # write some words
paste(..., sep, collapse)  # put some characters together

# Misc Plotting
plot(x, y,              # connect the dots to form a line
     type = "l", lwd)
```

Chapter 8
The "ply" Functions

Inspector Tiger: *Now someone has committed a murder here, and that murderer is someone in this room. The question is ... who?*
Colonel Pickering: *Look, there hasn't been a murder.*
Inspector Tiger: *No murder?*
All: *No.*
Inspector Tiger: *Oh, I don't like it. It's too simple, too clear cut. I'd better wait.*
—Monty Python's Flying Circus - Episode 11

8.1 Functions of a Systematic Nature

8.1.1 Introduction to the "ply" Functions

Statisticians often need to evaluate the same function, systematically, across an entire matrix, vector, list, etc. Some of the functions that we have already talked about are examples of applying specific (smaller) functions systematically over a vector or a matrix. For instance, `rowSums` takes a matrix, calculates the sum of each *row* of that matrix, and outputs a vector of row sums (and, likewise, `colSums` sums the columns).

```
> # Making sure the simulation can be repeated:
> set.seed(9182)
> # Generating 0/1 values from a Bernoulli dist with p = .7:
> X <- matrix(rbinom(40, size = 1, prob = .7), nrow = 8)
> X  # Which values did we generate?
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    1    0    1    1
[2,]    0    0    0    1    0
[3,]    1    1    0    1    1
[4,]    1    1    1    1    1
[5,]    1    0    1    0    0
[6,]    1    1    1    0    1
[7,]    0    1    0    0    1
```

```
[8,]    1    1    1    0    1
> # Calculating the row sums and col sums of X:
> rowSums(X)
[1] 3 1 4 5 2 4 2 4
> colSums(X)
[1] 5 6 4 4 6
```

By using `rowSums` or `colSums`, `R` is performing the same mini-function (i.e., the *sum* function) on each row or column.

Another example of repeatedly performing the same function across a matrix or vector is via vectorized functions. For example, as discussed in chapter 1, taking the logarithm of a vector of elements outputs a vector of the log of each element.

```
> x <- c(1, 2, 3, 4, 5) # vector of elements
> log(x)                # log of each element
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
```

In using `log` with a vector of inputs, `R` is performing the same mini-function (i.e., the *logarithm* function) on each, individual element.

One might wonder whether `R` contains a general method of systematically performing the same mini-function (*any* function) across rows of a matrix, across columns of a matrix, or on elements of a vector/matrix. As in every `R` desire you might have, the answer is (of course) "yes." I will discuss three of those functions: `sapply`, `apply`, and `tapply`, although two other functions (`lapply` and `mapply`) are also frequently used. Not surprisingly, due to the wonderful coherence of names, many `R` users refer to these series of functions as the "ply" functions (or the "pply" functions if you really like "p"'s). Each of the `ply` functions takes a specific type of `R` object, performs a mini, user-specified function in a specific manner, and returns a particular result.

1. `sapply`: Takes a (usually) vector or matrix, performs a function on each element of that object, and returns (usually) a vector of calculations.
2. `apply`: Takes (in general) a matrix, performs a function on the rows or columns of that matrix, and returns (usually) a vector of calculations.
3. `tapply`: Takes two or more vectors (one indicating score and the others indicating group membership), performs a function within each group, and returns (usually) a vector of calculations.

The `sapply` function is the easiest to understand, so I will discuss it first.

8.1.2 The sapply Function

The function `sapply` has the most basic structure of the "ply" family of functions. The basic form of the `sapply` function is as follows.

```
sapply(X, FUN, ... )
```

In the above set-up...

- `X`: A vector or matrix of scores. `X` can be of any `mode` (logical, numeric, character) as long as the mini-function allows an object of that mode.
- `FUN`: A function (either already in `R` or user specified) that takes a scalar input and returns (in general) a scalar output. The function can have multiple arguments, but one of those arguments (usually the first one) is of the same form as the individual values of `X`.
- `...`: The (annoying) dot-dot-dot argument. Essentially, the `...` "argument" allows for a mini-function that takes multiple arguments. One of those arguments must be the values of `X`, but the remaining arguments can be specified by name and separated by commas in place of `....` We will rarely ever use the `...` argument, so I will provide one example of how it works and ignore it thenceforth.

Perhaps an example might help alleviate confusion. Pretend that you have the following (incredibly useful) function.

```
> woah <- function(x){

    if(x <= 2){
      return("mouse")
    } else if(x > 2 & x <= 4){
      return("cat")
    } else{
      return("elephant")
    } # END if STATEMENTS

  } # END woah FUNCTION
```

The `woah` function returns: (1) "mouse" if the value of `x` is less than or equal to 2; (2) "cat" if the value of `x` is greater than 2 and less than or equal to 4; and (3) "elephant" if the value of `x` is greater than 4. Due to multiple `if` statements, the `woah` function can only take scalar elements rather than entire vectors. In fact, if you try to run this function on a vector of elements, `R` will only look at the first element, ignore the remaining elements, and issue a warning.

```
> # A vector of scores.
> boo <- c(-1, 0, 1, 2, 3, 4, 5, 6, 7, 8)
>
> # Trying to find out the associated animal:
> try(woah(boo), silent = TRUE)[1]
[1] "mouse"
Warning in if (x <= 2) { :
  the condition has length > 1 and only the first element
```

A solution to determining the animal associated with each element of the vector is to use the `sapply` function.

```
> sapply(boo, FUN = woah)  # the animal of each number!
 [1] "mouse"    "mouse"    "mouse"    "mouse"    "cat"
 [6] "cat"      "elephant" "elephant" "elephant" "elephant"
```

Hooray!

> **Note**: Even though `sapply` works with standard, vectorized functions (such as addition, logarithms, square roots, distribution densities, etc.), it is most appropriate when the function is based off a complicated sequence of non-vectorized statements. Because `R` is based on vectorization, using `sapply` instead of a vectorized function can result in a drastic loss in efficiency.

In our example, `sapply` looked at the first element of `boo`, determined the animal associated with the number located in the first element (based on the `if`/`else` statements), and put the corresponding animal as the first element of the result; then `sapply` looked at the second element of `boo` and repeated the entire process. Without `sapply`, your only recourse in determining the animal associated with each number of `boo` would be via a `for` loop.

```
> # Start with a null vector:
> vec <- NULL
> # Repeat the for loop for each element of boo:
> for(i in 1:length(boo)){
+
+   vec[i] <- woah(boo[i])
+
+ } # END for i LOOP
> # See what is in the vector:
> vec
 [1] "mouse"    "mouse"    "mouse"    "mouse"    "cat"
 [6] "cat"      "elephant" "elephant" "elephant" "elephant"
```

Yet `sapply` is easier, clearer, and quicker to write. Note that `sapply` also works with standard, built-in functions, such as the square root function.

```
> sapply(boo, FUN = sqrt)
 [1]      NaN 0.000000 1.000000 1.414214 1.732051 2.000000
 [7] 2.236068 2.449490 2.645751 2.828427
```

And even though `R` gives a warning (the square-root of a negative number is not applicable in $\mathbb{R}^1$), the square-root calculation is still attempted on each element of `boo`.

We could also try to `sapply` each element of an $n \times m$ dimensional matrix.

```
> # Turning boo into a matrix:
> boo2 <- matrix(boo, nrow = 2)
> boo2
     [,1] [,2] [,3] [,4] [,5]
[1,]   -1    1    3    5    7
[2,]    0    2    4    6    8
> sapply(boo2, FUN = woah)
 [1] "mouse"    "mouse"    "mouse"    "mouse"    "cat"
 [6] "cat"      "elephant" "elephant" "elephant" "elephant"
```

But when you use `sapply` on a matrix, `R` initially turns the matrix into a vector (putting the first column of the matrix as the first and second elements of the vector, the second column of the matrix as the third and fourth elements of the vector, etc. This is called "stacking a matrix column-by-column.") and then performs the mini-function on each element of the newly defined vector[3]. Therefore, when you `sapply` across the elements of a matrix, you must physically convert the vector that `R` outputs back into a matrix for subsequent calculations and displays.

```
> matrix(sapply(boo2, FUN = woah), nrow = nrow(boo2))
     [,1]    [,2]    [,3]  [,4]       [,5]
[1,] "mouse" "mouse" "cat" "elephant" "elephant"
[2,] "mouse" "mouse" "cat" "elephant" "elephant"
```

As mentioned earlier, the mini-functions entered into the `FUN` argument of `sapply` need not only have a single argument.

```
> woah2 <- function(x, y){

    if(x <= 2){
      return(paste("mouse-", y, sep = ""))
    } else if(x > 2 & x <= 4){
      return(paste("cat-", y, sep = ""))
    } else{
      return(paste("elephant-", y, sep = ""))
    } # END if STATEMENTS

  } # END woah2 FUNCTION
```

Now, when we set `x` as an element and `y` as practically anything, `R` first determines the animal associated with `x` and then `paste`s that animal to `y`.

```
> woah2(x = 2, y = "meep")
[1] "mouse-meep"
```

We can (of course) still use `R` to translate each element of `boo` to an animal, but we also must specify the value of `y` somewhere in `...` land.

```
> # - X is still boo,
> # - Our function is now woah2 (with x and y),
> # - We want the value of y to be "meep", so we indicate the value
> #   (by name) somewhere after we declare our function, in the ...
> sapply(boo, FUN = woah2, y = "meep")
 [1] "mouse-meep"    "mouse-meep"    "mouse-meep"
 [4] "mouse-meep"    "cat-meep"      "cat-meep"
 [7] "elephant-meep" "elephant-meep" "elephant-meep"
[10] "elephant-meep"
```

The `...` argument(s) work(s) similarly in all of the `ply` functions.

[3] `R` actually turns the matrix into a list, uses the `lapply` function on the list, and then `unlist`s the result. See the help page for a more thorough discussion of `lapply`.

8.1.3 The apply Function

Unlike `sapply`, `apply` takes a matrix, performs calculations on rows or columns of the matrix, and then returns the result as (usually) a vector of elements. The basic outline of `apply` is as follows.

```
apply(X, MARGIN, FUN, ... )
```

Notice how the arguments of `apply` are basically the same as the arguments of `sapply`, only now we must let `R` know the `MARGIN` over which the mini-function is to be `apply`-ed. In the `apply` function (in contrast to `sapply`), in the `apply` function, `X` will usually be a matrix of a certain dimension, and `FUN` will usually be a mini-function taking a vector of elements and returning a scalar (with optional arguments that can be put into `...` land)[4]. Unlike `sapply`, when `apply`-ing a mini-function to dimensions of a matrix, we must tell `R` over which dimension the mini-function should be `apply`d. Remember that you can either use `rowSums` to sum the rows of a matrix *or* `colSums` to sum the columns. `MARGIN` essentially tells `R` whether we want to do the `rowFunction` (if `MARGIN = 1`) or the `colFunction` (if `MARGIN = 2`). And `MARGIN` will usually be set to a scalar number (1 for applying the function to the rows of a matrix and 2 for applying the function to the columns).

Even though `rowSums` and `colSums` are more efficient than `apply`, we can use `apply` to get the same result.

```
> # Same matrix generation as before:
> set.seed(9182)
> X <- matrix(rbinom(40, size = 1, prob = .7), nrow = 8)
> # Row and column sums:
> apply(X = X, MARGIN = 1, FUN = sum)  # rowSum alternative
[1] 3 1 4 5 2 4 2 4
> apply(X = X, MARGIN = 2, FUN = sum)  # colSum alternative
[1] 5 6 4 4 6
```

Unlike `rowSums` and `rowMeans`, `R` does not provide a built-in function to calculate the row (or column) median or variance. Using the `apply` function, those calculations become rather straightforward.

```
> # Generating a different matrix:
> set.seed(8889)
> Y <- matrix(rnorm(40), nrow = 8)
> # We can find the rowSums and rowMeans with and without apply.
> apply(X = Y, MARGIN = 1, FUN = sum)  # option difficult: apply
[1]  2.555496548  0.234035967 -0.570690649  2.027959251
[5] -0.009911829 -1.744059059 -2.323881535  1.515020629
> rowSums(x = Y)                       # option easy: rowSums
```

[4]The `apply` function can actually take (as input) an higher-dimensional array of values and output a higher dimensional array (depending on the function and `...` arguments). Generalizations of the `apply` function will not be needed for the remainder of this book, which is why I'm putting it into a footnote. Go footnote power!

```
[1]  2.555496548  0.234035967 -0.570690649  2.027959251
[5] -0.009911829 -1.744059059 -2.323881535  1.515020629
> apply(X = Y, MARGIN = 1, FUN = mean) # option difficult: apply
[1]  0.511099310  0.046807193 -0.114138130  0.405591850
[5] -0.001982366 -0.348811812 -0.464776307  0.303004126
> rowMeans(x = Y)                      # option easy: rowMeans
[1]  0.511099310  0.046807193 -0.114138130  0.405591850
[5] -0.001982366 -0.348811812 -0.464776307  0.303004126
> # But the median and variance of the rows need apply.
> apply(X = Y, MARGIN = 1, FUN = median)  # easy peasy!
[1]  0.9758727 -0.2273004 -0.3643722  0.2650768 -0.6377596
[6] -0.5514250 -0.9664563 -0.0197963
> apply(X = Y, MARGIN = 1, FUN = var)     # easy peasy!
[1] 1.6558789 2.2930474 1.0154483 0.5901349 0.9842446
[6] 0.3537784 1.0736577 2.1269546
```

Important: `apply`, `sapply`, and `tapply` (later) take a function in the `FUN` argument. The desired function is (generally) *not in quotes* and must either be the name of an already existing function, the name of a function that you created, or a function written specifically inside the `FUN` argument.

As in `sapply`, one of the beauties of `apply` is being able to take a user-written mini-function, and to systematically `apply` that function to dimensions of a matrix. For instance, `R` does not provide a function to find the mode of a set of data, but writing our own `s.mode` function is relatively straightforward.

```
> s.mode <- function(x){

  # First, create a table of counts.
    tab <- table(x)

  # Second, pull out the name corresponding to the max count.
    mod <- names(tab)[which.max(tab)]

  # Finally, if the original values are numbers:
  # --> turn them back into numbers.
  # If the original values are characters:
  # --> keep them characters.
    if( is.numeric(x) ){
      return(as.numeric(mod))
    } else{
      return(mod)
    } # END if STATEMENTS

  } # END s.mode FUNCTION
```

If you remember, `boo` is a vector of scores, and `woah` (inside of the `sapply` function) determines the animal associated with each score.

```
> boo3 <- c(-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8)
> wee <- sapply(boo3, FUN = woah)  # finding the animals
> wee                              # indicating the animals
 [1] "mouse"    "mouse"    "mouse"    "mouse"    "mouse"
 [6] "mouse"    "cat"      "cat"      "elephant" "elephant"
[11] "elephant" "elephant"
```

Clearly, `"mouse"` is the mode of `wee` (there are more mice than anything else), so running `s.mode` on `wee` will result in a confirmation of what we already know.

```
> s.mode(wee)
[1] "mouse"
```

But `s.mode` takes a vector of data and returns a scalar element, which is just the sort of mini-function that we can plug into the `FUN` argument of `apply`. Therefore, by using `apply`, we can easily find the mode of each row of a matrix.

```
> wee2 <- matrix(wee, nrow = 4, byrow = TRUE)
> # mouse is the mode of rows 1/2,
> # cat is the mode of row 3,
> # and elephant is the mode of row 4.
> wee2
     [,1]       [,2]       [,3]
[1,] "mouse"    "mouse"    "mouse"
[2,] "mouse"    "mouse"    "mouse"
[3,] "cat"      "cat"      "elephant"
[4,] "elephant" "elephant" "elephant"
> apply(wee2, MARGIN = 1, FUN = s.mode)  # magical, isn't it!
[1] "mouse"    "mouse"    "cat"      "elephant"
```

You can `apply` over rows (or columns) of a matrix using *any* function that takes a vector of inputs and returns a scalar output.

8.1.4 The tapply Function

The final of the "ply" family of functions that I will discuss in this chapter is `tapply` and is structured according to the following diagram.

```
tapply(X, INDEX, FUN, ... )
```

Unlike the other two "ply" functions, `tapply` is intuitively helpful in calculating summaries for introductory statistics classes. In fact, the `tapply` function makes calculating within-group statistics easy and intuitively obvious. The only reason that I decided to discuss `sapply` and `apply` was as an introduction to the logic of `tapply`. The difference difference between `tapply` and `sapply` is the `INDEX` argument. For the rest of the book, `INDEX` takes a vector of the same length of `X`, indicating the category or group to which each member of `X` belongs. For instance, let's load the `anorexia` data in the `MASS` package.

```
> library(MASS)
> data(anorexia)
> head(anorexia)
  Treat Prewt Postwt
1  Cont  80.7   80.2
2  Cont  89.4   80.1
3  Cont  91.8   86.4
4  Cont  74.0   86.3
5  Cont  78.1   76.1
6  Cont  88.3   78.1
```

The `anorexia` data contains people in one of three "treatment" groups: those in the control group, those in the `CBT` group, and those in the `FT` group. We can verify the unique groups by using the `levels` function.

```
> levels(anorexia$Treat) # the unique levels of the treatment factor
[1] "CBT"  "Cont" "FT"
```

The entire `anorexia` data set contains *three* vectors: (1) a numeric vector indicating scores on the weight variable before the treatment; (2) a numeric vector indicating scores on the weight variable after the treatment; and (3) a factor or *grouping* variable associating a particular score to a specific condition. We call the latter an *index* because it tells us which scores belong together. And that *index* is what we need to use as `INDEX` in the `tapply` function.

```
> attach(anorexia)
> tapply(X = Postwt, INDEX = Treat, FUN = mean)  # group means
     CBT     Cont       FT
85.69655 81.10769 90.49412
> tapply(X = Postwt, INDEX = Treat, FUN = sd)    # group sds
     CBT     Cont       FT
8.351924 4.744253 8.475072
```

When using the `tapply` function with an `INDEX` argument, `R` knows to perform the mini-function specified in `FUN` within each group, returning the whole shebang as a vector of within-group calculations. For example, ANOVA (in a future chapter) becomes remarkably easy when taking advantage of `tapply`.

```
> # First - simple statistics per group.
> (xbar.g <- mean(Postwt))
[1] 85.17222
> (xbar.j <- tapply(X = Postwt, INDEX = Treat, FUN = mean))
     CBT     Cont       FT
85.69655 81.10769 90.49412
> (s2.j   <- tapply(X = Postwt, INDEX = Treat, FUN = var))
     CBT     Cont       FT
69.75463 22.50794 71.82684
> (n.j    <- tapply(X = Postwt, INDEX = Treat, FUN = length))
```

```
 CBT Cont   FT
  29   26   17
> # Second - Sums of Squares
> (SSB <- sum( n.j * (xbar.j - xbar.g)^2 ))
[1] 918.9869
> (SSW <- sum( (n.j - 1) * s2.j ))
[1] 3665.058
> (SST <- sum( (Postwt - xbar.g)^2 ))
[1] 4584.044
>
> # ... and then it's easy to calculate MS and F :)
```

Without using `tapply`, finding parts of an ANOVA table by using basic `R` calculations is nearly impossible to systematize.

8.2 Appendix: Functions Used

Here is a list of important functions from this chapter:

```
# Applying and Tapplying
sapply(X, FUN, ...)           # apply to each element
apply(X, MARGIN, FUN, ...)    # apply to row or column
tapply(X, INDEX, FUN, ...)    # apply to group members
levels(x)                     # unique levels/groups of a factor
```

Part III

Using R for Statistics

Chapter 9

Introduction to Statistical Inference in R

Pointed sticks? Ho, ho, ho. We want to learn how to defend ourselves against pointed sticks, do we? Getting all high and mighty, eh? Fresh fruit not good enough for you eh? Well I'll tell you something my lad. When you're walking home tonight and some great homicidal maniac comes after you with a bunch of loganberries, don't come crying to me!
—Monty Python's Flying Circus - Episode 4

All of the procedures that we have discussed to this point in the book are useful for computer programming. And many algorithms designed in C (or any other well-established programming language) can also be designed (in some fashion) in R. However, R was *built* for statistics, so much of Rs programming capabilities are best served as handmaidens to statistical analyses. Over the next few chapters, I will describe the basics of statistical analysis using R. But this description will only be a brief introduction. To better understand how to use R for statistics, pick up one of the more specialized books (e.g., Faraway, 2005; Fox & Weisberg, 2010; Pinheiro & Bates, 2009; Venables & Ripley, 2012) or plethora of free guides on the R website:

http://cran.r-project.org/other-docs.html

9.1 Asymptotic Confidence Intervals

Basic statistical inference usually involves several, related steps. I will discuss two of those steps: interval estimation and hypothesis testing. The third step to statistical inference, point estimation, was briefly discussed a few chapters ago alongside likelihood functions.

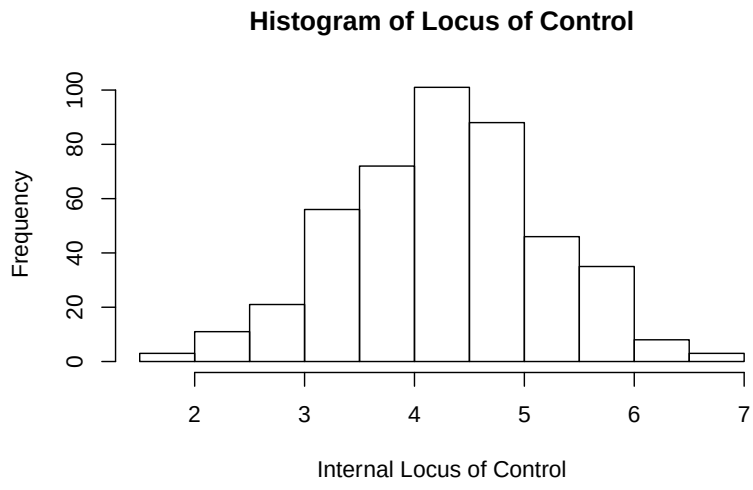
Classical interval estimation is somewhat more intuitive than hypothesis testing, so I will discuss it first. The process for calculating a normal-based confidence interval is rather straight forward. In fact, you should already know much of the requisite code. For an example calculation, load the `debt` dataset in the `faraway` package.

```
> # If you haven't installed "faraway", first install it.
```

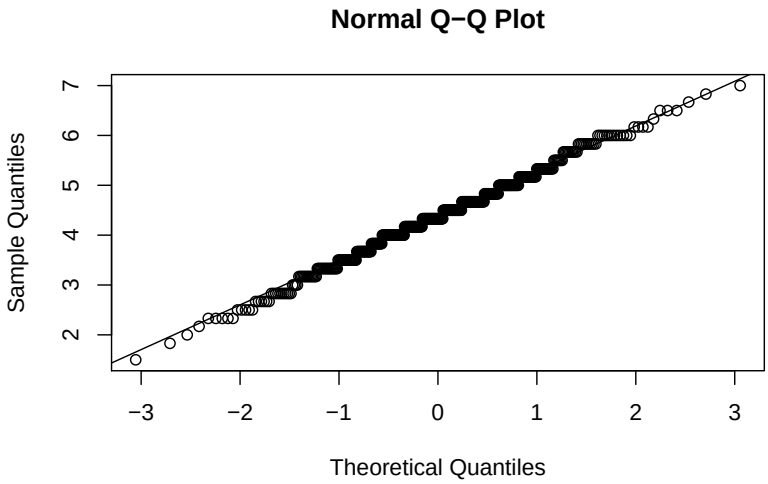
```
> library(faraway)  # loading the package "faraway"
> data(debt)        # loading the data "debt" inside "faraway"
> head(debt)        # looking at the first few rows of "debt"
  incomegp house children singpar agegp bankacc bsocacc
1        2     3        1       0     2       1      NA
2        5     2        3       0     2       1      NA
3        1     1        0       0     3      NA      NA
4        3     3        0       0     4       1       0
5        5     2        2       0     2       1       0
6        3     3        0       0     4       1       0
  manage ccarduse cigbuy xmasbuy locintrn prodebt
1      5        2      1       1     2.83    2.71
2      4        3      0       1     4.83    3.88
3      2        2      0       0     3.83    3.06
4      5        2      0       1     4.83    4.29
5      5        3      0       1     3.17    3.82
6      4        2      0       1     3.83    3.06
```

A histogram of the `locintrn` variable is somewhat normally distributed, and the qqpoints lie along the `qqline`. Therefore, using normal theory to calculate confidence intervals might be justified, especially given the large sample size (go CLT!).

```
> hist(debt$locintrn,
+      xlab = "Internal Locus of Control",
+      ylab = "Frequency",
+      main = "Histogram of Locus of Control")
```




```
> qqnorm(debt$locintrn)
> qqline(debt$locintrn)
```



The normal-based confidence interval for a single mean has the following formula:

$$CI = \bar{x} \pm z_{(1+\gamma)/2} \left(\frac{\sigma}{\sqrt{N}} \right)$$

where $\bar{x}$ is the sample mean, γ is the desired confidence level (usually $\gamma = .95$), $z_{(1+\gamma)/2}$ is the $(1+\gamma)/2$th quantile of the standard normal distribution, σ is the population standard deviation, and N is the sample size. The sample mean ($\bar{x}$) and sample size (N) are found by examining the data.

```
> loc  <- debt$locintrn[!is.na(debt$locintrn)]  # remove the NAs
> (xbar <- mean(loc))                           # sample mean
[1] 4.364527
> (N    <- length(loc))                         # sample size
[1] 444
```

Note that the `locintrn` sub-object of `debt` contains `NA` values (i.e., missing observations). `R` does not really know how to handle missing data without user specified input. For instance, if you construct a vector with `NA` values and try to take simple statistics, `R` does not know the actual values of the missing data, and those actual values are integral to the ultimate calculation.

```
> x <- c(1, 2, 3, 4, 5, NA)  # vector with NAs
> mean(x)                    # mean of vector with NAs?
[1] NA
```

One option to get rid of `NA`s is to change the `na.rm` argument (inside `mean`, `var`, `sd`, `median`, etc.) to `TRUE`.

```
> mean(x, na.rm = TRUE)  # remove NAs before calculating the mean
[1] 3
```

But sometimes the better option is to construct a new vector, `y`, that contains only the observations of `x` (by finding all of the values that *aren't* `NA` using the `is.na` function and negating it with `!`).

```
> y <- x[!is.na(x)]      # remove the NAs of x
> mean(y)                # the mean of the purged data
[1] 3
```

One reason that constructing a new vector might be more optimal than using `na.rm = TRUE` is because the new vector contains the correct number of observations (so that the `length` function will work correctly on `y`).

After calculating $\bar{x} = 4.365$ and $N = 444$, we still must find σ and $z_{(1+\gamma)/2}$ before we can calculate our normal-theory-based confidence interval. Assume that the population standard deviation is known and $\sigma = 1$ (due to metaphysical speculation). Then we only need to find the $(1 + \gamma)/2$th quantile of the standard normal distribution. Making sure that $\gamma \in [0, 1]$, we can find the corresponding quantile using the following procedure.

```
> # Several confidence levels:
> gamma1 <- .95
> gamma2 <- .99
> gamma3 <- .90
> gamma4 <- .752
> # Several critical values:
> ( z1 <- qnorm( (1 + gamma1)/2 ) ) # the 95%   quantile
[1] 1.959964
> ( z2 <- qnorm( (1 + gamma2)/2 ) ) # the 99%   quantile
[1] 2.575829
> ( z3 <- qnorm( (1 + gamma3)/2 ) ) # the 90%   quantile
[1] 1.644854
> ( z4 <- qnorm( (1 + gamma4)/2 ) ) # the 75.2% quantile
[1] 1.155221
```

Each confidence level results in a slightly different quantile and, therefore, leads to a slightly different confidence interval.

```
> sig <- 1
> ( CI1 <- xbar + c(-1, 1) * z1 * sig/sqrt(N) )  # 95% CI
[1] 4.271511 4.457543
> ( CI2 <- xbar + c(-1, 1) * z2 * sig/sqrt(N) )  # 99% CI
[1] 4.242284 4.486770
> ( CI3 <- xbar + c(-1, 1) * z3 * sig/sqrt(N) )  # 90% CI
[1] 4.286466 4.442588
> ( CI4 <- xbar + c(-1, 1) * z4 * sig/sqrt(N) )  # 75.2% CI?
[1] 4.309703 4.419351
```

The major `R` trick that I used to calculate symmetric confidence intervals in one step is multiplying the half-width $z_{(1+\gamma)/2}\left(\frac{\sigma}{\sqrt{N}}\right)$ by `c(-1, 1)` to simultaneously obtain the lower *and* upper values.

9.2 Hypothesis Tests of a Single Mean

Rather than finding a range of plausible values for a population parameter, data analysts are often interested in testing whether the population parameter is *statistically* different from a pre-specified null value. Ignoring the arguments against the frequentist version of statistical inference (e.g., Armstrong, 2007; Cohen, 1994), the simplest inferential procedures test the location of a single, population mean (μ). And the two most basic tests for this particular research question are the z-test and t-test, each with specific assumptions that may or may not be satisfied.

One difficulty in explaining the one-sample z-test (and one-sample t-test) is finding an appropriate (and realistic) research question. A typical research question for actually using one sample tests is for assessing long standing claims with well-defined null values. For instance, a few decades ago, researchers collected data designed to determine whether the average body temperature was actually equal to 98.6°, as conventional wisdom (and mothers/doctors) suggests (Mackowiak, Wasserman, & Levine, 1992). You can access the data using the following set of `R` commands.

```
> site <- "http://ww2.coastal.edu/kingw/statistics/R-tutorials/text/normtemp.txt"
```

```
> temp.dat <- read.table(url(site), header = FALSE)
> head(temp.dat) # what the data look like
    V1 V2 V3
1 96.3  1 70
2 96.7  1 71
3 96.9  1 74
4 97.0  1 80
5 97.1  1 73
6 97.1  1 75
> nrow(temp.dat) # the number of observations
[1] 130
```

The first column of `temp.dat` contains body temperature (in sorted order) of 130 observations. Rather than using the *entire dataset*, we will pretend that a random sample of $N = 20$ observations constitues our collection of data.

```
> set.seed(83234)
> ( temp.vec <- sample(temp.dat$V1, size = 20, replace = FALSE) )
 [1] 97.1 98.6 98.2 98.0 99.2 98.4 96.7 98.8 98.6 98.5 98.7
[12] 97.8 98.0 98.1 96.3 98.2 98.2 98.0 97.1 98.6
> # Notice that temp.vec is a much smaller "sample" of data.
```

9.2.1 One-Sample z-Tests

One-sample z-tests require several pieces of information.

1. The null and alternative hypotheses, which provides:
   - The population mean under the null hypothesis: μ_0.

- Whether we are performing a one-tailed or two-tailed test.

2. Other data-based information (e.g., nuisance parameters):

   - The sample mean: $\bar{x}$.
   - The sample size: N.
   - The population standard deviation: σ.

3. How to make a decision regarding our hypotheses:

   - The desired Type I error rate: α.

In the current section, we will pretend that the *entire* sample of $N = 130$ observations is the entire population, so that the standard deviation taken from the entire sample is "truth."

```
> ( sigma <- sd(temp.dat$V1) ) # our population standard deviation
[1] 0.7331832
```

Our experiment was designed to disconfirm the notion that the average body temperature in the population is $\mu = 98.6$, so we will use 98.6 as our null hypothesis mean – the thing that we are trying to provide evidence *against*. And without a strong alternative hypothesis, we should perform a two-tailed test to cover both of our options.

$$\begin{aligned} \mathrm{H}_0 &: \mu = 98.6 \\ \mathrm{H}_1 &: \mu \neq 98.6 \end{aligned}$$

And assuming $\alpha = .05$ (even though a more conservative Type I error rate might be more appropriate), we can perform the test. Assuming that the null hypothesis is true,

$$z = \frac{\bar{x} - \mu_0}{\frac{\sigma}{\sqrt{N}}}$$

is normally distributed with a mean of 0 and a standard deviation of 1. One method of making a decision is to calculate z and then find the probability that $|Z| > |z|$ (i.e., assuming that the null hypothesis is true) more extreme than z.

```
> # First, find the necessary statistics:
> xbar <- mean(temp.vec)
> N    <- length(temp.vec)
> # Second, calculate z using the formula:
> ( z <- (xbar - 98.6)/(sigma/sqrt(N)) )
[1] -3.324291
> # Then find the p-value by:
> # -- taking the absolute value of z,
> # -- finding the area in the upper tail, and
> # -- multiplying that area by 2.
> ( p.z <- 2*pnorm(abs(z), lower.tail = FALSE) )
[1] 0.0008864368
```

And because $z = -3.324$, so that $|z| = 3.324$, the probability of finding $Z > |z|$ is 0.0004432, and therefore, the probability of finding $|Z| > |z|$ is 0.0008864. Because $p < \alpha$, we would reject H_0 and claim evidence that the mean body temperature in the population is not equal to 98.6°.

Because R stores an infinite number of normal distributions, we do not even need to transform our data to z before finding the p-value.

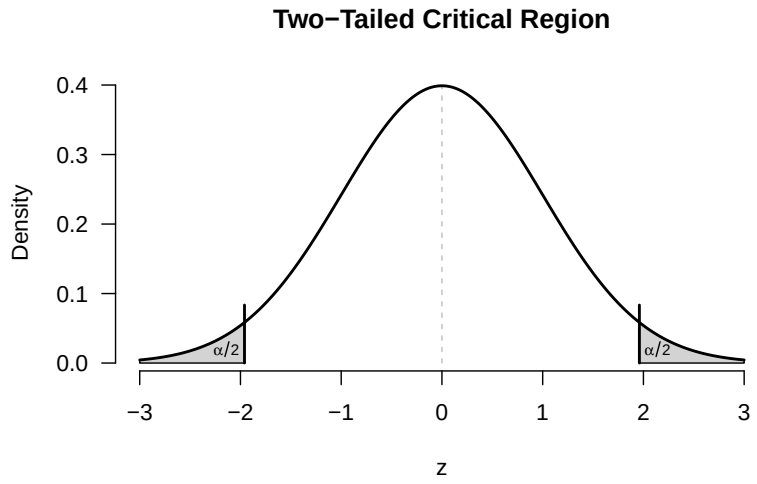
```
> # Calculating p without calculating z first:
> p.z <- 2*pnorm(abs(xbar - 98.6),
+                mean = 0, sd = sigma/sqrt(N), lower.tail = FALSE)
> p.z
[1] 0.0008864368
```

Notice that both p-values are identical due to linear transformations not changing the shape of a distribution.

We could also test our hypotheses using critical values in lieu of p-values. Finding critical values is identical to finding $z_{(1+\gamma)/2}$ (which I talked about in the confidence interval section). Why are these things equivalent? Well, we need to find the z of a standard normal distribution such that the proportion of area *greater* than z *plus* the proportion of area less than $-z$ is equal to α. The value of z such that $\alpha/2$ is the area above z is the $(1-\gamma)/2$ quantile of the distribution. These critical values are the endpoints of the center region in the following plot.

```
> alpha <- .05
> # a) The normal distribution evaluation:
> x <- seq(-3, 3, by = .01)
> y <- dnorm(x)
> # b) Plotting and labeling:
> plot(x, y,
+      xlab = "z", ylab = "Density",
+      main = "Two-Tailed Critical Region",
+      type = "n", axes = FALSE)
> axis(1)
> axis(2, las = 1)
> # c) Drawing the mean:
> lines(x = c(0, 0), y = c(0, max(y)),
+       lty = 2, col = "grey")
> # d) Coloring and drawing the critical values:
> polygon(x = c(min(x), x[x < qnorm(alpha/2)], qnorm(alpha/2)),
+         y = c(0, y[x < qnorm(alpha/2)], 0),
+         col = "lightgrey")
> polygon(x = c(qnorm(1 - alpha/2), x[x > qnorm(1 - alpha/2)], max(x)),
+         y = c(0, y[x > qnorm(1 - alpha/2)], 0),
+         col = "lightgrey")
> lines(x, y, type = "l", lwd = 2)
> # e) Text corresponding to alpha/2 and alpha/2:
> text(x = c(qnorm(alpha/2) - .18, qnorm(1 - alpha/2) + .18),
+      y = c(.02, .02),
```

```
+      labels = c(expression(alpha/2), expression(alpha/2)),
+      cex = .7)
> # f) Lines corresponding to the critical values:
> segments(x0 = c(qnorm(alpha/2), qnorm(1 - alpha/2)),
+          y0 = c(0, 0),
+          y1 = c(dnorm(qnorm(alpha/2)) + .025,
+                 dnorm(qnorm(1 - alpha/2)) + .025),
+          lwd = 2)
```



And finding z such that the proportion of area greater than z is equal to $\alpha/2$.

```
> # Our alpha level (again):
> alpha <- .05
> # Our z-critical values using our alpha level:
> ( z.crit <- qnorm(c(alpha/2, 1 - alpha/2),
+                   mean = 0, sd = 1) )
[1] -1.959964  1.959964
> # Reject the null hypothesis?
> (z < z.crit[1]) | (z > z.crit[2])
[1] TRUE
> # Retain the null hypothesis?
> (z > z.crit[1]) & (z < z.crit[2])
[1] FALSE
```

Another option would have been to find the critical x-values (rather than the critical z-values) and determine if our sample mean was inside the critical x-region.

One should always follow up hypothesis tests with post hoc procedures. Harking back to the beginning of this chapter, we can easily calculate a confidence interval for the population mean of body temperature.

```
> # Our silly little statistics:
> xbar  <- mean(temp.vec)
> sigma <- sd(temp.dat$V1)
> N     <- length(temp.vec)
> # Our confidence level:
> gamma <- 1 - alpha
> # Our confidence interval (symmetric!):
> CI.z <- xbar + c(-1, 1)*qnorm( (1 + gamma)/2 )*sigma/sqrt(N)
> CI.z
[1] 97.73367 98.37633
```

I will leave the corresponding lower-tailed test:

$$\begin{aligned} \mathrm{H}_0 &: \mu \geq 98.6 \\ \mathrm{H}_1 &: \mu < 98.6 \end{aligned}$$

or upper tailed test:

$$\begin{aligned} \mathrm{H}_0 &: \mu \leq 98.6 \\ \mathrm{H}_1 &: \mu > 98.6 \end{aligned}$$

as an exercise to the reader. Of course, much of the time, statisticians do not know the population standard deviation (σ), and the sample size (N) is not quite in asymptopia. The one-sample t-test was developed to answer questions about the population mean but without the unrealistic assumption of a known population variance.

9.2.2 One-Sample t-Tests

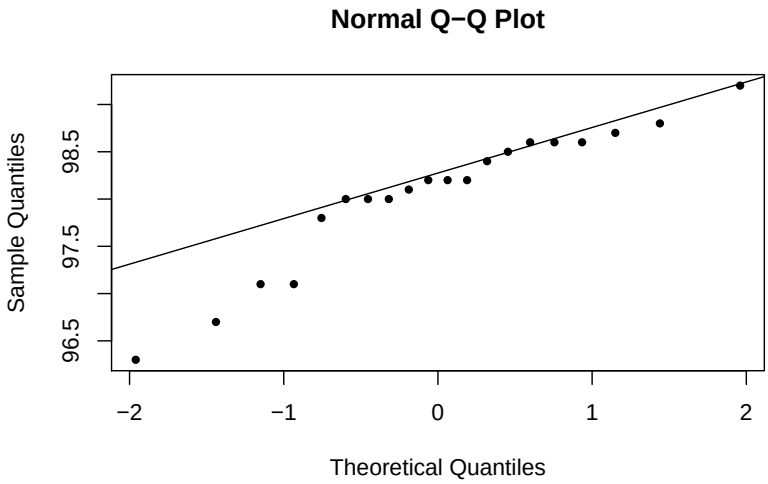
The t-based test procedure in `R` is nearly identical to the z-based test procedure. As always, we must specify a null and alternative hypotheses, and because we are using the same data as before, we will keep the same hypotheses,

$$\begin{aligned} \mathrm{H}_0 &: \mu = 98.6 \\ \mathrm{H}_1 &: \mu \neq 98.6, \end{aligned}$$

and we will use the same Type I error rate of $\alpha = .05$.

The t-test assumes that our data are randomly sampled from a normal distribution. Procedures exist to test for population normality, but visual techniques (e.g., `qqnorm` plots) are usually sufficient.

```
> qqnorm(temp.vec, pch = 20)  # is it normal?
> qqline(temp.vec)            # it should be on the line.
```



Unfortunately, the data do not appear to align with the `qqline`. Normally, I would choose a different method of analysis (e.g., the sign test, Wilcoxen test, bootstrapping, etc.) but we are only using these data as an illustration of how to perform basic analyses.

Using the t-test Formula

The one-sample t-statistic can be calculated with

$$t_{df} = \frac{\bar{x} - \mu_0}{\frac{s_x}{\sqrt{N}}},$$

where $df = N - 1$ and $s_x = \sqrt{\frac{\sum(x-\bar{x})^2}{N-1}}$. Because `R` knows the `t` family of functions (`dt`, `pt`, `qt`, and `rt`), the actual t-based procedure is not much different to the one-sample z-test.

```
> # First, find the necessary statistics:
> xbar <- mean(temp.vec)
> s_x  <- sd(temp.vec)
> N    <- length(temp.vec)
> # Second, calculate t using the formula:
> ( t <- (xbar - 98.6)/(s_x/sqrt(N)) )
[1] -3.299206
> # Then find the p-value by:
> # -- taking the absolute value of t,
> # -- finding the area in the upper tail of t with df = N - 1, and
> # -- multiplying that area by 2.
> ( p.t <- 2*pt(abs(t), df = N - 1, lower.tail = FALSE) )
[1] 0.003772094
```

And because $t = -3.299$, so that $|t| = 3.299$, the probability of finding $T > |t|$ is 0.001886, and therefore, the probability of finding $|T| > |t|$ is 0.0037721. Because $p < \alpha$, we would

reject H_0 and claim evidence that the mean body temperature in the population is not equal to 98.6°. Notice that I basically copied and pasted the conclusion from the one-sample z-test. We only calculated a slightly different statistic and used a slightly different distribution.

We can also easily calculate a t-based confidence interval.

```
> # Our silly little statistics:
> xbar <- mean(temp.vec)
> s_x  <- sd(temp.vec)
> N    <- length(temp.vec)
> # Our confidence level:
> gamma <- .95
> # Our confidence interval (symmetric!):
> CI.t <- xbar + c(-1, 1)*qt((1 + gamma)/2, df = N - 1)*s_x/sqrt(N)
> # How do our confidence intervals compare?
> CI.z  # a z (normal) based confidence interval.
[1] 97.73367 98.37633
> CI.t  # a t (t) based confidence interval.
[1] 97.70925 98.40075
```

Notice that the t-based confidence interval is slightly wider than the confidence interval calculated using asymptotic normal theory.

Using the t.test Function

Fortunately for R users everywhere, there exists a *function* (in R) that calculates t-statistics, p-values, and confidence intervals all in one step. For the one-sample t-test, the outline of the `t.test` function is as follows.

```
t.test(x,
       alternative = c("two.sided", "less", "greater"),
       mu = 0, conf.level = .95)
```

In the above description, `x` is a vector of data, `alternative` is the direction of rejection, `mu` is the null hypothesis mean, and `conf.level` corresponds to the desired confidence level. The `t.test` function defaults to performing a one-sample, two-sided t-test against the null hypothesis that $\mu = 0$ and using a confidence level of $\gamma = .95$ (which is equivalent to $\alpha = 1 - .95 = .05$). If we test the population mean of body temperature using `t.test` but keep the defaults, then R dutifully informs us that there is evidence to believe that the mean temperature of the population is not equal to 0.

```
> ( t.mod1 <- t.test(x = temp.vec) ) # keeping the defaults
        One Sample t-test

data:  temp.vec
t = 593.5846, df = 19, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 97.70925 98.40075
```

```
sample estimates:
mean of x
   98.055
```

How exciting!??! We should (of course) change `mu` to the actual null hypothesis mean.

```
> # The t-test automatically in R:
> ( t.mod2 <- t.test(temp.vec, mu = 98.6) )
        One Sample t-test

data:  temp.vec
t = -3.2992, df = 19, p-value = 0.003772
alternative hypothesis: true mean is not equal to 98.6
95 percent confidence interval:
 97.70925 98.40075
sample estimates:
mean of x
   98.055
> # Compare with our previous calculation:
> CI.t # our (calculated) t-based confidence interval
[1] 97.70925 98.40075
> p.t  # our (calculated) t-based p-value
[1] 0.003772094
```

And the automatic t-test resulted in exactly the same confidence interval and p-value as our earlier results. We can also change the confidence level to anything: .99, .90, .942, .42, and `R` will alter the width of the interval to match the desired confidence level.

```
> t.test(temp.vec, mu = 98.6, conf.level = .99)
        One Sample t-test

data:  temp.vec
t = -3.2992, df = 19, p-value = 0.003772
alternative hypothesis: true mean is not equal to 98.6
99 percent confidence interval:
 97.5824 98.5276
sample estimates:
mean of x
   98.055
> t.test(temp.vec, mu = 98.6, conf.level = .90)
        One Sample t-test

data:  temp.vec
t = -3.2992, df = 19, p-value = 0.003772
alternative hypothesis: true mean is not equal to 98.6
90 percent confidence interval:
 97.76936 98.34064
sample estimates:
```

```
mean of x
   98.055
> t.test(temp.vec, mu = 98.6, conf.level = .942)
        One Sample t-test

data:  temp.vec
t = -3.2992, df = 19, p-value = 0.003772
alternative hypothesis: true mean is not equal to 98.6
94.2 percent confidence interval:
 97.72174 98.38826
sample estimates:
mean of x
   98.055
```

Often, you might choose to perform one-sided hypothesis tests (given a particular theory about the true mean of the population). For instance, you might believe that the average body temperature was overestimated in the 19th century (using their primitive temperature-extraction methods). To find p-values corresponding to lower-tailed (or upper-tailed) alternative hypotheses, change `alternative` to `"less"` (or `"greater"`). But be warned: when `alternative` is not equal to `two.sided`, `R` also returns a one-sided confidence interval.

```
> t.test(temp.vec, mu = 98.6, alternative = "less")    # -Inf? Really!
        One Sample t-test

data:  temp.vec
t = -3.2992, df = 19, p-value = 0.001886
alternative hypothesis: true mean is less than 98.6
95 percent confidence interval:
     -Inf 98.34064
sample estimates:
mean of x
   98.055
> t.test(temp.vec, mu = 98.6, alternative = "greater") #  Inf? Really
        One Sample t-test

data:  temp.vec
t = -3.2992, df = 19, p-value = 0.9981
alternative hypothesis: true mean is greater than 98.6
95 percent confidence interval:
 97.76936      Inf
sample estimates:
mean of x
   98.055
```

As described in an earlier chapter, the `t.test` function is one of those model-based `R` functions that calculates many values, returns those values in a list (with particular names) and displays useful stuff (using `print` and `cat` statements) to the user. When a

function returns a list of objects, one should determine the names of those objects, the purpose of each object, how the objects were calculated from the data, and whether (or not) those objects are needed for further analyses.

```
> names(t.mod2)       # what is in mod2?
[1] "statistic"   "parameter"   "p.value"     "conf.int"
[5] "estimate"    "null.value"  "alternative" "method"
[9] "data.name"
> t.mod2$statistic    # our t-statistic
        t
-3.299206
> t.mod2$parameter    # our degrees of freedom?
df
19
> t.mod2$p.value      # the p-value from our test
[1] 0.003772094
> t.mod2$conf.int     # the confidence interval vector
[1] 97.70925 98.40075
attr(,"conf.level")
[1] 0.95
> t.mod2$estimate     # the x-bar in our sample
mean of x
   98.055
> t.mod2$null.val     # our null hypothesis mean
mean
98.6
```

Of course, even though we followed the one-sample t-test procedures on the temperature data, we have evidence to believe that temperature is not normally distributed in the population. The normal quantiles did not match the empirical quantiles from our data! Can R determine the robustness of a particular statistical method? Of course - by using the same control-flow, programming principles discussed earlier in the book.

The Robustness of the One-Sample t-Test

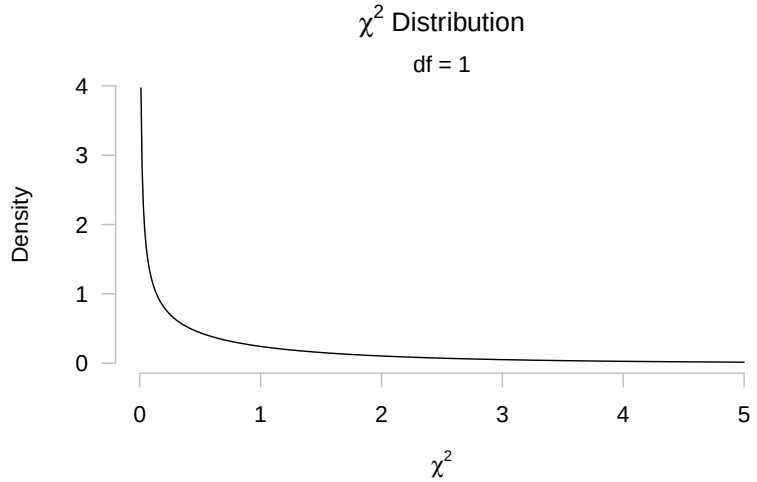
With respect to statistical testing, the meaning of the term "robustness" refers to how close the true Type I error rate (α) is to the Type I error set by the researcher (α_0) in the face of certain violations. An outline of the general "robustness" procedure can be illustrated when there are no violations. Assume that the data arise from a (standard) normally distributed population and that the null hypothesis is true. Then we can continuously sample from this population with a fixed sample size, and determine the proportion of p-values less than $\alpha = .05$. Because the null hypothesis is always true, the proportion of p-values less than $\alpha = .05$ should be .05.

```
> set.seed(827824)
> # Parameters of our simulation:
> reps  <- 10000 # the number of replications
```

```
> N     <- 10    # the sample size drawn from the population
> # A vector to hold t-test based p-values:
> pval  <- NULL
> # Repeating this simulation lots of times:
> for(i in 1:reps){
+
+ # Sampling from a normal dist and finding a p-value:
+   x       <- rnorm(N, mean = 0, sd = 1)
+   pval[i] <- t.test(x,
+                     mu = 0, alternative = "two.sided")$p.value
+
+ } # END for i LOOP
> alpha <- .05
> mean(pval < alpha)
[1] 0.0492
```

Is the proportion of p-values less than $\alpha = .05$ close to .05 when the population is normal? We can also determine the robustness of the one-sample t-test when the population is extremely skewed. An example of an extremely skewed distribution is the χ^2 distribution with $df = 1$.

```
> x <- seq(0, 5, by = .01)
> y <- dchisq(x, df = 1)
> plot(x, y,
+      xlab = expression(chi^2), ylab = "Density",
+      main = expression(paste(chi^2, " Distribution", sep = "")),
+      type = "l", axes = FALSE)
> mtext("df = 1")
> axis(1, col = "grey")
> axis(2, col = "grey", las = 1)
```



Using the $\chi^2_{df=1}$ distribution as our population, we can determine the proportion of p-values less than $\alpha = .05$ when repeatedly performing one-sample t-tests.

```
> set.seed(83490)
> # Parameters of our simulation:
> reps  <- 10000 # the number of replications
> N     <- 10    # the sample size drawn from the population
> # The ACTUAL population mean (or close to it):
> mu    <- mean(rchisq(100000, df = 1))
> # A vector to hold t-test based p-values:
> pval  <- NULL
> # Repeating this simulation lots of times:
> for(i in 1:reps){
+
+ # Sampling from the a skewed dist and finding a p-value:
+   x       <- rchisq(N, df = 1)
+   pval[i] <- t.test(x,
+                     mu = mu, alternative = "two.sided")$p.value
+
+ } # END for i LOOP
> alpha <- .05
> mean(pval < alpha)
[1] 0.1393
```

What happens to the proportion of p-values less than $\alpha = .05$ when we violate assumptions?

The following code is a generalization of the last two simulations to a variety of skewed or heavy tailed distributions. You just have to indicate the population (either normal, t-distributed, or χ^2), the size of each sample, the number of replications, the researcher determined α level, and distributional parameters (`df` for `dt` and `dchisq` and `mean`/`sd` for `dnorm`).

```
> ##########################################
> # Robust t Function to Normal Violations #
> ##########################################
>
> # This function is designed to take:
> # - a true population,
> #   - "norm" is a normal distribution,
> #   - "t" is t distribution (heavy tails),
> #   - "chisq" is chi-squared distribution (skewed),
> # - the size of each sample for testing,
> # - the number of test replications,
> # - the researcher set alpha level, and
> # - parameters of the true population (in the ...)
> #   - "norm" takes mean and sd,
> #   - "t" and "chisq" take df.
>
> # And return:
```

```
> # - the true Type I error rate.
>
> robustTNorm <- function(popn = c("norm", "t", "chisq"),
                          N = 5, reps = 10000, alpha = .05, ... ){

  # The ACTUAL population mean (or close to it):
    mu <- mean( get(paste("r", popn, sep = ""))(n = 100000, ... ) )

  # A vector to hold t-test based p-values:
    pval <- NULL

  # Repeating this simulation lots of times:
    for(i in 1:reps){

  # Sampling from the specified dist and finding a p-value:
      x    <- get(paste("r", popn, sep = ""))(n = N, ... )
      pval[i] <- t.test(x,
                        mu = mu, alternative = "two.sided")$p.value

    } # END for i LOOP

    return( mean(pval < alpha) )

  } # END robustTNorm FUNCTION
```

Because the null hypothesis is always true, the proportion of p-values less than α should be equal to α. Here are a few examples to illustrate how to use the function. You should play around with the `robustTNorm` function on your own time.

```
> set.seed(2723)
> robustTNorm(popn = "norm",  N = 10, mean = 0, sd = 1)
[1] 0.0477
> robustTNorm(popn = "t",     N = 14, df = 3)
[1] 0.0398
> robustTNorm(popn = "chisq", N = 20, df = 4)
[1] 0.0691
```

9.2.3 One-Sample Power Calculations

Unfortunately (or fortunately, if you need to publish) the alternative hypothesis is usually true. However, due to the insensitivity of particular tests, researchers frequently fail to detect the true alternative. Therefore, every researcher should make sure that his/her studies have enough power to detect meaningful effects (or else those studies were a big waste of time/money/energy). Determining the power of a study is easiest in the context of z-tests and uses the following pieces of information.

1. The null and alternative hypotheses, which provides:

- The population mean under the null hypothesis: μ_0.
- Whether we are performing a one-tailed or two-tailed test.

2. The actual information about the population:
   - The *true* population mean: μ.
   - The population standard deviation: σ.
3. The sample size: N.
4. How to make a decision regarding our hypotheses:
   - The desired Type I error rate: α.

For consistency, we will keep $\sigma = 0.733$, set $\alpha = .05$, and pretend that our future sample is of size $N = 20$ (the same as our current sample). We will further pretend that we need to calculate power for a one-tailed test with an alternative hypothesis of $\mu < 98.6$.

$$\begin{aligned} \mathrm{H}_0 &: \mu \geq 98.6 \\ \mathrm{H}_1 &: \mu < 98.6 \end{aligned}$$

Unfortunately, to perform a power analysis, researchers also need to also know the *true* population mean. Granted that you will never actually *have* the true population mean (otherwise, there would be no reason to conduct hypothesis testing), but you can think of the *true* population mean in the form of a "what-if"

> If the true population mean is actually at this location, what is the probability that we would be able to detect a difference (and reject our null hypothesis) in a random sample of data?

Researchers want to determine the power to detect a minimally relevant effect—an effect that is not just real but actually meaningful—and the value inserted as the true mean represents this minimal difference. For illustration, pretend that the true mean body temperature is $\mu = 98.4$ (or, in translation, we want to detect an effect if the true mean body temperature is less than 98.4°).

Calculating the power of a one-sample z-test requires two simple steps.

1. Determine the critical score (x) based on our null hypothesis distribution (with $\mu = 98.6$) such that we will reject anything below this critical score.
2. Find the probability of being below x given the "true" alternative distribution (with $\mu = 98.4$).

```
> alpha <- .05  # Our significance level
> sigma         # The true standard deviation
[1] 0.7331832
> N             # The actual sample size
[1] 10
```

```
> # Our null and alternative hypothesis means:
> mu0 <- 98.6
> mu1 <- 98.4
> # Finding x-crit (using the H0 dist with mu = mu0):
> ( x.crit <- qnorm( p = alpha,
+                    mean = mu0, sd = sigma/sqrt(N) ) )
[1] 98.21864
> # Finding power (using the H1 dist with mu = mu1):
> ( pow.z  <- pnorm( q = x.crit,
+                    mean = mu1, sd = sigma/sqrt(N) ) )
[1] 0.2170375
```

Power analysis for a one-sample t-test is not as straightforward as for the z-based test. The trick to calculating power for the one-sample t-test is to use a non-central t-distribution with non-centrality parameter.

$$\lambda = \frac{(\mu - \mu_0)}{\frac{\sigma}{\sqrt{N}}}$$

You can think of the non-central t-distribution as the true alternative distribution on the t-scale. The non-centrality parameter is similar to the "mean" of the test statistic given a particular alternative distribution and sample size. If you knew the population standard deviation (and were consequently performing a z-test), then λ would be the mean of your z-test statistic given μ, σ, and N. Therefore, we can also calculate the power of a z-test using the following simple steps.

1. Determine the critical value (z) such that we will reject anything below this critical score.
2. Find the probability of being below z given a normal distribution with a mean of $\frac{(\mu-\mu_0)}{\frac{\sigma}{\sqrt{N}}}$ and a standard deviation of 1.

```
> # Power on the standard normal scale:
> ( z.crit <- qnorm( p = alpha ) )
[1] -1.644854
> ( pow.z  <- pnorm( q = z.crit,
+                    mean = (mu1 - mu0)/(sigma/sqrt(N)) ) )
[1] 0.2170375
> # ... same answers, of course.
```

Unlike the z-test, where we can work in both the z and x scales, the t-distribution is easiest to work in the scale of the test statistic. Therefore, we should determine the power of a one-sample t-test using the following steps.

1. Determine the critical value (t) such that we will reject anything below this critical score.
2. Find the probability of being below t given a t-distribution with non-centrality parameter $\lambda = \frac{(\mu-\mu_0)}{\frac{\sigma}{\sqrt{N}}}$.

```
> alpha <- .05  # Our significance level
> sigma         # The true standard deviation
[1] 0.7331832
> N             # The actual sample size
[1] 10
> # Our null and alternative hypothesis means:
> mu0 <- 98.6
> mu1 <- 98.4
> # Finding t-crit:
> ( t.crit <- qt( p = alpha, df = N - 1 ) )
[1] -1.833113
> # Finding power (using the H1 dist with mu = mu1):
> ( pow.t  <- pt( q = t.crit, df = N - 1,
+                 ncp = (mu1 - mu0)/(sigma/sqrt(N)) ) )
[1] 0.1987425
```

Fortunately, `R` has a function (`power.t.test`) that will figure out any aspect of a power calculation (power, sample size, $\mu_0 - \mu$, population standard deviation) given any of the other aspects. Using `power.t.test` is pretty simple.

```
power.t.test(n = NULL, delta = NULL, sd = 1, sig.level = 0.05,
             power = NULL,
             type = c("two.sample", "one.sample", "paired"),
             alternative = c("two.sided", "one.sided"))
```

Replace *three* of the `NULL`s, indicate whether the alternative hypothesis is directional (`"one.sided"`) or non-directional (`"two.sided"`), set `type = "one.sample"`, and `R` will solve for the missing value in the power calculation.

```
> power.t.test(n = N, delta = mu0 - mu1,
+              sd = sigma, sig.level = .05,
+              power = NULL,                  # <-- we want this!
+              type = "one.sample",
+              alternative = "one.sided")

     One-sample t test power calculation

              n = 10
          delta = 0.2
             sd = 0.7331832
      sig.level = 0.05
          power = 0.1987425
    alternative = one.sided
```

Notice that the `power.t.test` function returns the same power as hand calculations (0.199). Sometimes researchers will know the desired power and want to solve for the needed sample size to achieve that power given a minimally-relevant mean difference. Because the t-distribution depends on sample size (through degrees of freedom), calculating that sample size by hand is tedious and requires iteration. Using the `power.t.test` function makes simple but tedious calculations much easier to find.

9.3 Appendix: Functions Used

Here is a list of important functions from this chapter:

```
# Confidence Intervals
is.na(x)                # returns TRUE if the value is NA
mean(x, na.rm = TRUE)   # remove the NAs before calculating the mean
var(x, na.rm = TRUE)    # remove the NAs before calculating the var
sd(x, na.rm = TRUE)     # remove the NAs before calculating the sd
median(x, na.rm = TRUE) # remove the NAs before calculating the median

# Hypothesis Tests:
t.test(x,                              # an automatic t-test
       alternative,
       mu, conf.level)
power.t.test(n = NULL, delta = NULL,   # t-test power calculations
             sd = 1, sig.level = 0.05,
             power = NULL,
             type = "one.sample",
             alternative = c("two.sided", "one.sided"))
```

Chapter 10

Two-Samples t-Tests

Mousebender: *It's not much of a cheese shop, is it?*
Wensleydale: *Finest in the district!*
Mousebender: *Explain the logic underlying that conclusion, please.*
Wensleydale: *Well, it's so clean, sir!*
Mousebender: *It's certainly uncontaminated by cheese...*
—Monty Python's Flying Circus - Episode 33

10.1 More t-Tests in R

10.1.1 Independent Samples t-Tests

Pretend that you have collected data on two groups of students: one group who drinks regularly, and one group who never drinks at all, and you want to determine whether the drinking population performs differently in a test of ability. Assume that the following scores are a random sample of $N = 22$ students, $n_1 = 12$ belonging to the alcohol group and $n_2 = 10$ belonging to the no-alcohol group.

```
> d.yes <- c(16, 20, 14, 21, 20, 18, 13, 15, 17, 21, 18, 15)
> d.no  <- c(18, 22, 21, 17, 20, 17, 23, 20, 22, 21)
```

A priori, students who drink alcohol should have a reduction in reading ability compared to those who abstain. This prior belief must be expressed before examining the data, as in the following set of hypotheses.

$$\mathrm{H}_0 : \mu_{\text{yes}} \geq \mu_{\text{no}}$$
$$\mathrm{H}_1 : \mu_{\text{yes}} < \mu_{\text{no}}$$

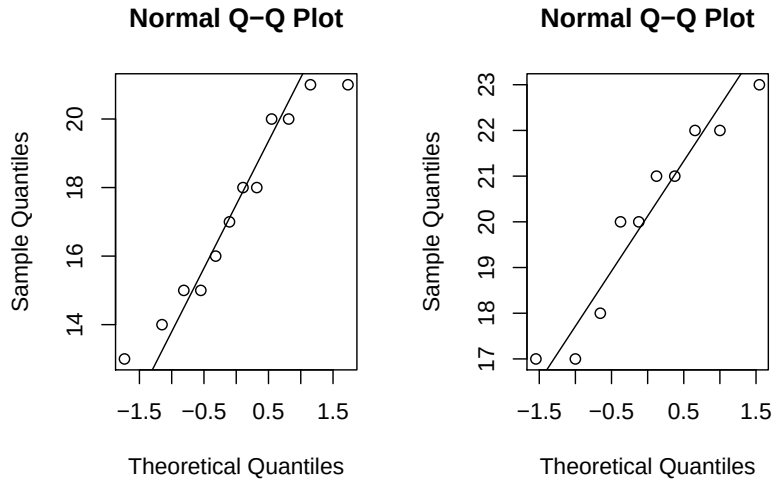
Of course, we do not usually have prior information, in which case the following set of hypotheses would test the more typical situation of whether the population means of two groups are different.

$$\mathrm{H}_0 : \mu_{\text{yes}} = \mu_{\text{no}}$$
$$\mathrm{H}_1 : \mu_{\text{yes}} \neq \mu_{\text{no}}$$

Checking t-Test Assumptions

When testing whether the population means of these (independent groups) are different, one generally performs an independent samples t-test. Of course, the independent samples t-test also arrives with a few assumptions: normality within each population, homogeneity of variance (i.e., equal population variances), and independence of observations. Remember that normality is assumed whenever one uses the t-distribution, as the t-distribution is a sampling distribution from a normally distributed population. The simplest method of checking normality is by using `qqnorm` plots within each of the groups.

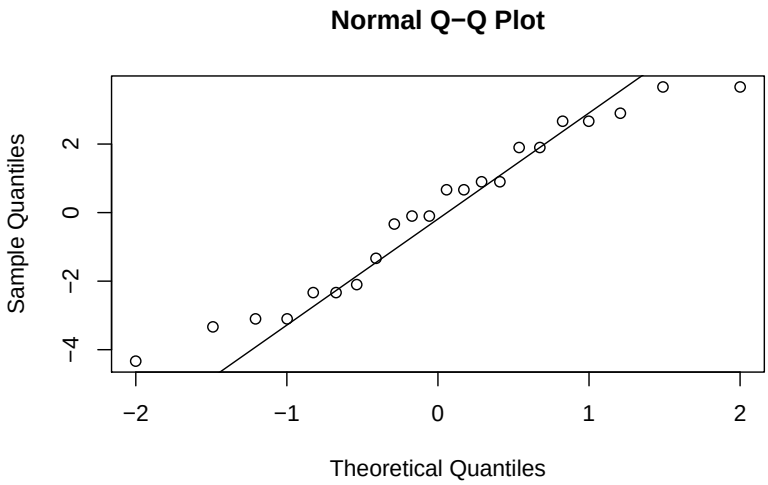
```
> par(mfrow = c(1, 2))
> qqnorm(d.yes)  # qqnorm plot for one group
> qqline(d.yes)
> qqnorm(d.no)   # qqnorm plot for the other group
> qqline(d.no)
> par(mfrow = c(1, 1))
```



But because both groups only have 10 observations, departures from normality are difficult to detect. Note that `par` sets graphical parameters, and the `mfrow` argument of `par` allows for showing multiple plots on the same screen. The `mfrow` argument must be specified as a vector of two values: (1) the number of rows of plots, and (2) the number of columns of plot. So by setting `mfrow = c(1, 2)`, `R` will show one row and two columns of plots. `mfrow` is so named because it forces the plotting screen to be filled by rows rather than by columns.

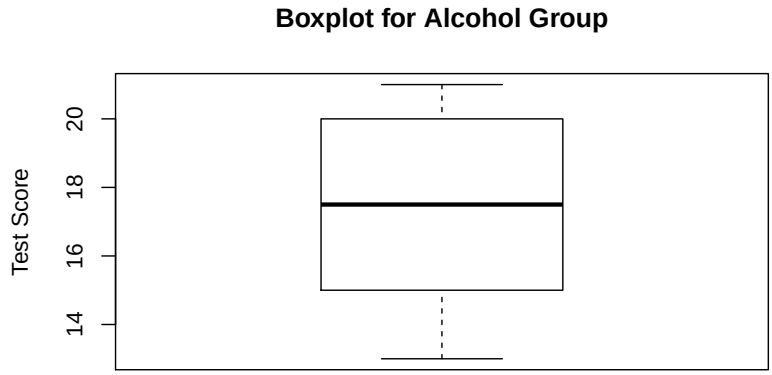
One could also check normality of populations by pooling the observations within each group. Because each group (potentially) has a different population mean, you should mean center the observations within each group before combining them into one, big pot.

```
> # Mean centering the observations and pooling them:
> d.all <- c(d.yes - mean(d.yes), d.no - mean(d.no))
> qqnorm(d.all)  # qqnorm plot for both groups together
> qqline(d.all)
```



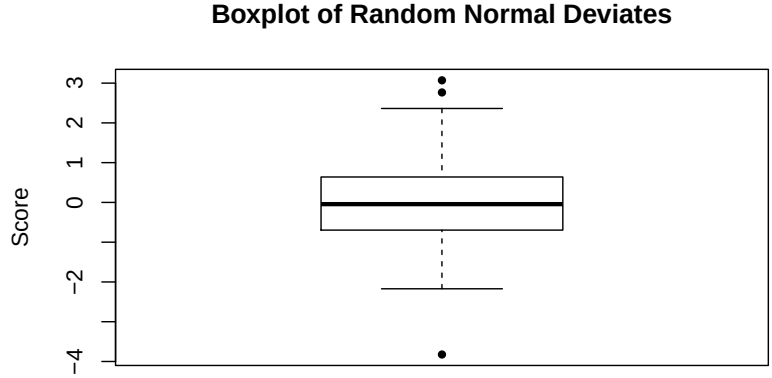
Even though the modified `qqnorm` plot does not appear to lie on the `qqline`, pooling data is only justified if both populations have the same variance. Of course, the classical independent samples t-test procedures *assumes* that both populations have the same variance (estimating that variance using a weighted average of sample variances–the so-called "pooled variance"). A simple check of whether both populations have the same variance is to construct side-by-side boxplots of the within-group scores. To construct a single boxplot, you would put a vector of values into the `boxplot` function and let `R` work its magic.

```
> boxplot(d.yes,
+         ylab = "Test Score",
+         main = "Boxplot for Alcohol Group")
```



Because R has magical fairies drawing pictures and calculating distributional quantiles, your boxplot arrives factory fresh with whiskers of the correct length and outliers as dots added to the top and bottom of the plot.

```
> set.seed(183)
> x <- rnorm(400)
> boxplot(x,
+         ylab = "Score",
+         main = "Boxplot of Random Normal Deviates",
+         pch = 20)
```



Constructing side-by-side boxplots is not as straightforward as constructing a single boxplot, but uses something that will be relevant over the final chapters in this book:

formulas. Whenever one predicts a criterion from one or more predictors, he/she must use R approved notation. The formula

```
y ~ x
```

tells R that y is a function of x. In a linear regression context, the dependence of a criterion on one or more predictors is obvious. However, an independent samples t-test is also, implicitly, a function of scores on an predictor variable. And by writing

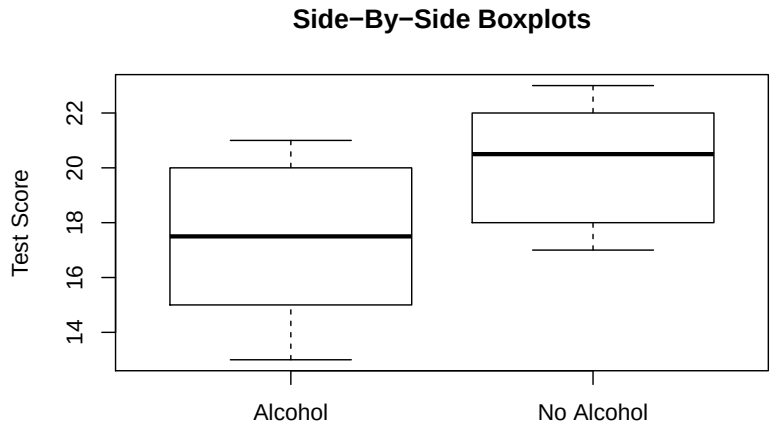
```
scores ~ grp
```

R knows that scores is a vector of scores for each person, and grp is a factor vector indicating group membership. And if we reject the null hypothesis, then the score of a particular person depends, in some way, on his/her group. If we did not think that test score depended on whether (or not) someone consumed alcohol, then we would have no reason to perform a t-test. We should first let scores and grp be equal to the appropriate numeric and factor vectors.

```
> # Constructing two vectors of data:
> #  1) A vector of scores for each person, and
> #  2) A vector of group membership associated with the scores.
> ( scores <- c(d.yes, d.no) )
 [1] 16 20 14 21 20 18 13 15 17 21 18 15 18 22 21 17 20 17
[19] 23 20 22 21
> ( grp    <- factor( c( rep(1, length(d.yes)), rep(2, length(d.no)) ),
+                     labels = c("Alcohol", "No Alcohol") ) )
 [1] Alcohol    Alcohol    Alcohol    Alcohol    Alcohol
 [6] Alcohol    Alcohol    Alcohol    Alcohol    Alcohol
[11] Alcohol    Alcohol    No Alcohol No Alcohol No Alcohol
[16] No Alcohol No Alcohol No Alcohol No Alcohol No Alcohol
[21] No Alcohol No Alcohol
Levels: Alcohol No Alcohol
```

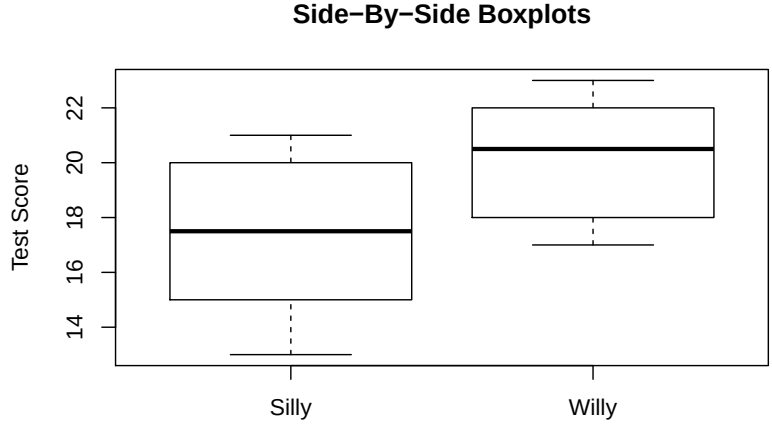
And then we can display side-by-side boxplots of test scores on group membership by using the formula syntax.

```
> boxplot(scores ~ grp,
+         ylab = "Test Score",
+         main = "Side-By-Side Boxplots",
+         pch = 20)
```



The default behavior of `boxplot` is to label group by its factor name (inside the `grp` vector), but one can easily change almost anything in `R` by altering a few arguments. And the `names` argument in `boxplot` allows one (if he/she so desires) to change this unfortunate default.

```
> boxplot(scores ~ grp,
+         names = c("Silly", "Willy"),
+         ylab = "Test Score",
+         main = "Side-By-Side Boxplots",
+         pch = 20)
```



Using the t-Test Formula

After checking the normality and homogeneity of variance assumptions, we can run the classic independent samples t-test by calculating

$$t_{df} = \frac{(\bar{x}_1 - \bar{x}_2) - (\mu_1 - \mu_2)_0}{s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}$$

where $df = n_1 + n_2 - 2$ and $s_p^2 = \frac{(n_1-1)s_1^2+(n_2-1)s_2^2}{n_1+n_2-2}$. After finding the appropriate t-statistic, one can easily calculate probabilities and make decisions by using the t-test procedure of the previous chapter. The most general method of calculating group-level statistics is by using the `tapply` command discussed a few chapters ago.

```
> # Sample size, mean, and variance within groups:
> n.j    <- tapply(X = scores, INDEX = grp, FUN = length)
> xbar.j <- tapply(X = scores, INDEX = grp, FUN = mean)
> s2.j   <- tapply(X = scores, INDEX = grp, FUN = var)
> # Calculating the pooled variance:
> s2.p   <- sum( (n.j - 1) * s2.j )/(sum(n.j - 1))
> # Finding the t-statistic:
> t <- (xbar.j[1] - xbar.j[2])/sqrt( s2.p * (sum(1/n.j)) )
```

Of course, you can also calculate the t-statistic the much more self-explanatory (but less cool) way.

```
> # Calculating simple statistics
> n1     <- length(d.yes)  # the number of people in each group
> n2     <- length(d.no)
> s2.1   <- var(d.yes)     # the variances of each group
> s2.2   <- var(d.no)
> xbar.1 <- mean(d.yes)    # the means of each group
> xbar.2 <- mean(d.no)
> # Calculating the pooled variance:
> s2.p2 <- ( (n1 - 1)*s2.1 + (n2 - 1)*s2.2 )/(n1 + n2 - 2)
> # Finding the t-statistic:
> t2 <- (xbar.1 - xbar.2)/sqrt( s2.p2 * (1/n1 + 1/n2) )
```

And you will find that (not surprisingly) both t-statistics are identical.

```
> t   # a t-statistic the cool way
  Alcohol
-2.578778
> t2  # a t-statistic the less cool way
[1] -2.578778
```

Finally, we should find the p-value corresponding to our test statistic. Assume that we are performing a one-tailed test and that the numerator of our test statistic subtracts "Non Alcohol" from "Alcohol" (as in the t-statistic calculated above). Because $\mu_{\text{yes}} < \mu_{\text{no}}$ is the resulting alternative hypothesis, $\mu_{\text{yes}} - \mu_{\text{no}} < 0$ is an equivalent way of writing our alternative hypothesis, and we should look in the lower tail of the t-distribution with $df = 20$.

```
> ( p.t <- pt(t2, df = n1 + n2 - 2) )
[1] 0.008965179
```

And because $t = -2.579$, the probability of finding $T < t$ is 0.009. Therefore, $p < \alpha = .05$, so we would reject H_0 and claim evidence that students who consume alcohol perform worse (on average) on a specific test than those who do not consume alcohol. We could have also calculated a two-tailed test (using $\mu_{\text{yes}} \neq \mu_{\text{no}}$ as our alternative hypothesis) by multiplying the p-value by 2 to account for area in the upper tail. Or we could have found critical values and compared our obtained t-statistic to those critical values. The logic behind all of these additions is explained in the previous chapter.

If we did not want to assume that the groups had the same population variance, we should modify our t-statistic (and resulting degrees of freedom) using the Welch-Satterthwaite correction. The new t-statistic is

$$t_{df} = \frac{(\bar{x}_1 - \bar{x}_2) - (\mu_1 - \mu_2)}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}},$$

where $df = \nu'$ are our corrected degrees of freedom and can be calculated using the following formula[5].

$$\nu' = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1 - 1} + \frac{\left(\frac{s_2^2}{n_2}\right)^2}{n_2 - 1}}$$

Both the t-statistic and modified degrees of freedom are much easier to calculate using the vector of variances.

```
> # Our modified degrees of freedom (see the formula):
> ( nu   <- ( sum(s2.j/n.j)^2 )/( sum( (s2.j/n.j)^2/(n.j - 1) ) ) )
[1] 19.89977
> # Our modified t-statistic:
> ( t.wc <- (xbar.j[1] - xbar.j[2])/sqrt( sum(s2.j/n.j) ) )
  Alcohol
-2.642893
> # The associated p-value:
> ( p.wc <- pt(t.wc, df = nu) )
    Alcohol
0.007823715
```

[5]The Welch-Satterthwaite correction is based off of the approximate degrees of freedom for a linear combination of variance terms. Let $Q = \sum_j k_j s_j^2$. Then Q is approximately χ^2 distributed with approximate degrees of freedom

$$\nu^{\star\prime} = \frac{Q^2}{\sum_j \frac{(k_j s_j^2)^2}{df_j}}$$

where df_j is the degrees of freedom corresponding to the jth variance. If $k_j = \frac{1}{n_j}$, then $\nu^{\star\prime}$ simplifies to ν' of the Welch-Satterthwaite correction (Satterthwaite, 1946).

The modified degrees of freedom are $\nu' = 19.9$ as compared to our original degrees of freedom of $df = 20$; the modified t-statistic is -2.643; and the modified (one-tailed) p-value is 0.008. The modified degrees of freedom are slightly lower than before, the modified p-value is also slightly lower, and (in either case) we would have rejected our null hypothesis.

Using the t.test Function

One can also follow the independent samples t-test steps using the built-in `t.test` function rather than calculating everything by hand. We only need to change a few of the arguments. The general form of the *brand spanking new* t-test function is

```
t.test(x, y, var.equal = FALSE, ... )
```

with `...` symbolizing all of the `t.test` stuff applicable to the one-sample t-test (discussed in the previous chapter), including `alternative`, `mu`, and `conf.level`. In the above function, `x` stands for a vector of data from one group, `y` stands for a vector of data for the other group, and `var.equal` is a logical value indicating whether homogeneity of variance should be assumed. One can also specify a t-test in `R` by using the "formula" construction.

```
t.test(formula, var.equal = FALSE, ... )
```

`R` assumes (by default) that the within-group variances are not equal, sets `var.equal` to FALSE, and carries out the Welch-Satterthwaite t-test.

> **Remember:** The Welch-Satterthwaite degrees of freedom correction is the default in `R`. To perform the classical t-test, you must (yourself) set `var.equal` to TRUE. Thus, even though the classical independent samples t-test requires homogeneity of variance, `R` knowns that most researchers will (most of the time) want to use the more robust procedure.

Before discussing the robustness of the independent samples t-test to violations of homogeneity of variance, I should first present a few examples of performing a t-test on the already discussed research question.

```
> # The original direction (using both input methods):
> ( i.m11 <- t.test(x = d.yes, y = d.no,
+                   var.equal = TRUE, alternative = "less") )
        Two Sample t-test

data:  d.yes and d.no
t = -2.5788, df = 20, p-value = 0.008965
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
       -Inf -0.9162861
sample estimates:
mean of x mean of y
 17.33333  20.10000
```

```
> ( i.m12 <- t.test(scores ~ grp,
+                   var.equal = TRUE, alternative = "less") )
        Two Sample t-test

data:  scores by grp
t = -2.5788, df = 20, p-value = 0.008965
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
       -Inf -0.9162861
sample estimates:
   mean in group Alcohol mean in group No Alcohol
                17.33333                 20.10000
> # The two-sided direction:
> ( i.m2 <- t.test(scores ~ grp,
+                  var.equal = TRUE, alternative = "two.sided") )
        Two Sample t-test

data:  scores by grp
t = -2.5788, df = 20, p-value = 0.01793
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -5.0046126 -0.5287207
sample estimates:
   mean in group Alcohol mean in group No Alcohol
                17.33333                 20.10000
> # Not assuming equality of variance (both tests):
> ( i.m3 <- t.test(scores ~ grp,
+                  var.equal = FALSE, alternative = "less") )
        Welch Two Sample t-test

data:  scores by grp
t = -2.6429, df = 19.9, p-value = 0.007824
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
      -Inf -0.960734
sample estimates:
   mean in group Alcohol mean in group No Alcohol
                17.33333                 20.10000
> ( i.m4 <- t.test(scores ~ grp,
+                  var.equal = FALSE, alternative = "two.sided") )
        Welch Two Sample t-test

data:  scores by grp
t = -2.6429, df = 19.9, p-value = 0.01565
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -4.9510266 -0.5823067
sample estimates:
```

```
mean in group Alcohol mean in group No Alcohol
             17.33333                 20.10000
```

Check to make sure that the p-values given via the `t.test` function match those calculated by hand.

```
> # Using t.test:
> i.m11$p.value     # Less than, no correction
[1] 0.008965179
> i.m3$p.value      # Less than, correction
[1] 0.007823715
> # Using hand calculations:
> p.t               # Less than, no correction
[1] 0.008965179
> p.wc              # Less than, correction
    Alcohol
0.007823715
```

After determining whether or not to reject the null hypothesis, you should follow up your t-calculation with confidence intervals (and/or effect size estimates). A simple method of obtaining confidence intervals on the difference between means is extracting those confidence intervals from the `t.test` function (making sure to specify `alternative = "two-sided"` or else `R` will output one-sided confidence intervals).

```
> i.m2$conf.int # confidence interval without correction
[1] -5.0046126 -0.5287207
attr(,"conf.level")
[1] 0.95
> i.m4$conf.int # confidence interval with correction
[1] -4.9510266 -0.5823067
attr(,"conf.level")
[1] 0.95
```

But you could just as easily calculate those confidence intervals by hand. Assuming homogeneity of variance, the confidence interval on the difference between two population means has the following formula.

$$CI = (\bar{x}_1 - \bar{x}_2) \pm t_{n_1+n_2-2;\,(1+\gamma)/2}\sqrt{s_p^2\left(\frac{1}{n_1}+\frac{1}{n_2}\right)}$$

Without assuming homogeneity of variance, the resulting confidence interval on the difference between two population means has the following formula.

$$CI = (\bar{x}_1 - \bar{x}_2) \pm t_{\nu';\,(1+\gamma)/2}\sqrt{\frac{s_1^2}{n_1}+\frac{s_2^2}{n_2}}$$

where ν' is the modified degrees of freedom. Both confidence interval formulas are simple rearrangements of the standard independent samples t-test formulas.

```
> gamma <- .95
> # Pooled variance confidence interval:
> { (xbar.1 - xbar.2) +
+    c(-1, 1) * qt((1 + gamma)/2, df = sum(n.j - 1)) *
+               sqrt(s2.p * (sum(1/n.j))) }
[1] -5.0046126 -0.5287207
> # Welch confidence interval:
> { (xbar.1 - xbar.2) +
+    c(-1, 1) * qt((1 + gamma)/2, df = nu) * sqrt(sum(s2.j/n.j)) }
[1] -4.9510266 -0.5823067
```

To calculate the confidence interval for a single mean, you must first decide whether the homogeneity of variance assumption is likely to be satisfied. If you *do not* assume homogeneity of variance, then the single mean confidence interval is exactly the same as the confidence interval from the one-sample t-test. However, by assuming that s_1^2 and s_2^2 are estimating a common population variance, σ^2, then s_p^2 provides a better estimate of σ^2 than either s_1^2 or s_2^2 (because s_p^2 is based off of more observations). Therefore, the general formula for a (homogeneity of variance assumed) single mean confidence interval is similar to that from the one-sample t-test, but with s_p^2 replacing s_1^2 or s_2^2 and $df = n_1 + n_2 - 2$ replacing $df_1 = n_1 - 1$ or $df_2 = n_2 - 2$.

```
> # Single mean confidence interval for group one:
> { xbar.1 +
+   c(-1, 1) * qt((1 + gamma)/2, df = sum(n.j - 1)) * sqrt(s2.p/n.j[1]) }
[1] 15.82451 18.84216
> # Single mean confidence interval from group two:
> { xbar.2 +
+   c(-1, 1) * qt((1 + gamma)/2, df = sum(n.j - 1)) * sqrt(s2.p/n.j[2]) }
[1] 18.44717 21.75283
```

Note that even when the confidence interval on the difference between means does not include 0, the single-mean confidence intervals can still overlap.

The Robustness of the Independent Samples t-Test

Finally, one might wonder whether the independent samples t-test is robust to certain violations of assumptions. Because I have already discussed robustness as applied to the normal population assumption, I will focus on assessing the robustness of the independent samples t-test to violations of homogeneity of variance. We can first assume that the sample sizes are equal and determine the percentage of p-values less than $\alpha = .05$ using both the classic independent samples t-test and the Welch-Satterthwaite correction.

```
> # Setting the seed and determining the number of replications.
> set.seed(9183)
> reps    <- 10000
> # Setting the sample size and within-group standard deviations.
> n1      <- 10
> n2      <- 10
> sig1    <- 1
```

```
> sig2    <- 1
> # Building empty vectors to store p-values.
> a.typic <- NULL
> a.welch <- NULL
> # Repeating a simulation lots of times.
> for(i in 1:reps){
+
+ # Each time, generating two vectors of data from a normal distribution:
+   x1 <- rnorm(n = n1, mean = 0, sd = sig1)
+   x2 <- rnorm(n = n2, mean = 0, sd = sig2)
+
+ # Calculating both t-tests, and pulling out the p-values
+   a.typic[i] <- t.test(x1, x2,
+                        var.equal = TRUE,
+                        alternative = "two.sided")$p.value
+   a.welch[i] <- t.test(x1, x2,
+                        var.equal = FALSE,
+                        alternative = "two.sided")$p.value
+
+ } # END for i LOOP
> # Figuring out the proportion of p-values less than alpha:
> alpha <- .05
> mean(a.typic < alpha)
[1] 0.0491
> mean(a.welch < alpha)
[1] 0.0477
```

The above code is the "ideal" scenario, in that the sample sizes are equal and the population variances are identical. We can write a function (similar to that of the last chapter) to determine robustness given any $n_1 \neq n_2$ and $\sigma_1^2 \neq \sigma_2^2$.

```
> ##########################################
> # Robust t Function to Hetero of Variance #
> ##########################################
>
> # This function is designed to take:
> # - the population standard deviations,
> #   - sig1 is the sd of group 1
> #   - sig2 is the sd of group 2
> # - the size of each sample,
> #   - n1 is the sample size of group 1
> #   - n2 is the sample size of group 2
> # - the number of test replications, and
> # - the researcher set alpha level.
>
> # And return:
> # - the true Type I error rate using:
> #   - the classic independent samples t-test,
> #   - the Welch corrected independent samples t-test.
```

```
>
> robustTVar <- function(sig1 = 1, sig2 = 1, n1 = 10, n2 = 10,
                         reps = 10000, alpha = .05){

  # Vectors to store t-test based p-values.
    a.typic <- NULL
    a.welch <- NULL

  # Repeating a simulation lots of times.
    for(i in 1:reps){

  # Generating two vectors of data from a normal distribution:
      x1 <- rnorm(n = n1, mean = 0, sd = sig1)
      x2 <- rnorm(n = n2, mean = 0, sd = sig2)

  # Calculating both t-tests, and pulling out the p-values
      a.typic[i] <- t.test(x1, x2,
                           var.equal = TRUE,
                           alternative = "two.sided")$p.value
      a.welch[i] <- t.test(x1, x2,
                           var.equal = FALSE,
                           alternative = "two.sided")$p.value

    } # END for i LOOP

  # Figuring out the proportion of p-values less than alpha:
   return(list( classic.t = mean(a.typic < alpha),
                welch.t   = mean(a.welch < alpha) ) )

  } # END robust.t.var FUNCTION
```

If the t-test is robust to assumptions, the proportion of p-values less than α should be approximately equal to α. What happens if we change n_1, n_2, σ_1^2 and σ_2^2?

```
> set.seed(2347907)
> # Both tests should give the same value if the sds/ns are equal:
> robustTVar(sig1 = 1, sig2 = 1, n1 = 10, n2 = 10)
$classic.t
[1] 0.0515

$welch.t
[1] 0.0502
> # And both tests are usually OK as long as the ns are equal:
> robustTVar(sig1 = 1, sig2 = 2, n1 = 10, n2 = 10)
$classic.t
[1] 0.0522

$welch.t
[1] 0.0481
```

```
> robustTVar(sig1 = 1, sig2 = 4, n1 = 10, n2 = 10)
$classic.t
[1] 0.0603

$welch.t
[1] 0.0497
> # But things get weird when the sample sizes change:
> robustTVar(sig1 = 1, sig2 = 4, n1 = 20, n2 = 10)
$classic.t
[1] 0.1588

$welch.t
[1] 0.0479
> robustTVar(sig1 = 1, sig2 = 4, n1 = 40, n2 = 10)
$classic.t
[1] 0.2927

$welch.t
[1] 0.0476
```

You should explore the `robust.t.var` function on your own time.

10.1.2 Paired Samples t-Tests

If our data do not arise from independent groups, then you shouldn't perform an independent samples t-test. A typical (frequently designed) violation of independence is gathering pairs of observations that are linked across groups. For instance: before/after, twin studies, or parent/children data. If you violate independence in this systematic way, you can perform a paired samples t-test as a counterpart to the independent samples t-test. For instance, assume that we have the following set of data.

```
> id       <- 1:13
> pre.swb  <- c(3, 0, 6, 7, 4, 3, 2, 5, 4, 3, 4, 2, 7)
> post.swb <- c(5, 7, 10, 14, 10, 6, 8, 5, 9, 10, 8, 6, 8)
> dat.swb  <- data.frame(id, pre.swb, post.swb)
> dat.swb
   id pre.swb post.swb
1   1       3        5
2   2       0        7
3   3       6       10
4   4       7       14
5   5       4       10
6   6       3        6
7   7       2        8
8   8       5        5
9   9       4        9
10 10       3       10
11 11       4        8
```

```
12 12       2        6
13 13       7        8
```

The variable `pre.swb` stands for subjective well-being before counseling, and the variable `post.swb` stands for subjective well-being (of the same person) after counseling. Because we have pre/post scores for the same person, you would find a moderate correlation between pre- and post- scores, and the independence (across groups) assumption of the independent samples t-test does not hold.

```
> cor(pre.swb, post.swb)
[1] 0.4941769
```

With the above data, one can also use the `t.test` function (as before) to perform a test of our hypotheses that subjective well-being improves after counseling.

$$\begin{aligned} \mathrm{H}_0 &: \mu_{\text{pre}} \geq \mu_{\text{post}} \\ \mathrm{H}_1 &: \mu_{\text{pre}} < \mu_{\text{post}} \end{aligned}$$

But rather than using formula (`scores ~ grp`) as we did in the independent samples t-test, we instead list our vectors as separate variables in the `t.test` function. We also must inform `R` that it should perform a paired samples t-test by setting the argument `paired` to TRUE. And that's it!

```
> # Testing mean of pre below mean of post:
> p.m2 <- t.test(x = pre.swb, y = post.swb, paired = TRUE,
+                alternative = "less")
> p.m2  # what were are really interested in calculating
        Paired t-test

data:  pre.swb and post.swb
t = -6.6853, df = 12, p-value = 1.121e-05
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
      -Inf -3.159275
sample estimates:
mean of the differences
              -4.307692
```

Of course, a paired samples t-test is just a one-sample t-test on difference scores. Therefore, we only need to find the difference scores to calculate the paired samples t-test statistic by hand.

```
> ## 1 ## Forming difference scores:
> d.diff <- pre.swb - post.swb
> ## 2 ## Calculating the sample mean, variance, and size:
> xbar.d <- mean(d.diff)    # the mean of the difference scores
> var.d  <- var(d.diff)     # the variance of the difference scores
> N.d    <- length(d.diff)  # the number of difference scores
> ## 3 ## Forming our t-statistic:
> ( t.d  <- (xbar.d - 0)/sqrt(var.d/N.d) ) # a one-sample t-test
```

```
[1] -6.685326
> ( p.d <- pt(t.d, df = N.d - 1) )          # lower.tailed test
[1] 1.121498e-05
> ## 4 ## Calculating a 95% confidence interval:
> gamma <- .95
> CI.d  <- { xbar.d +
+            c(-1, 1)*qt( (1 + gamma)/2, df = N.d - 1 )*sqrt(var.d/N.d) }
> CI.d
[1] -5.711611 -2.903773
```

As the paired samples t-test is identical in calculation to the one samples t-test, all of the requisite material was covered in the previous chapter.

10.2 Appendix: Functions Used

Here is a list of important functions from this chapter:

```
# Independent t-Tests
y ~ x                          # y is a function of x
t.test(x, y,                   # a more general t-test
       var.equal,
       alternative,
       mu, conf.level)
t.test(formula, ...)

# Paired t-Tests
t.test(x, y,                   # perform a paired-samples t-test
       paired = TRUE, ... )

# Graphing
par(mfrow = c(row, col))       # plot more than one on a screen
boxplot(x, ...)                # construct boxplot(s)
boxplot(formula, names, ...)
```

Chapter 11

One-Way ANOVA

Well, er, yes Mr. Anchovy, but you see your report here says that you are an extremely dull person. You see, our experts describe you as an appallingly dull fellow, unimaginative, timid, lacking in initiative, spineless, easily dominated, no sense of humour, tedious company and irrepressibly drab and awful. And whereas in most professions these would be considerable drawbacks, in chartered accountancy, they're a positive boon.
—Monty Python's Flying Circus - Episode 10

11.1 One-Way ANOVA in R

11.1.1 Setting up the ANOVA

An ANOVA tests the difference in population means between multiple groups. Assuming that all of the group sizes are equal, you might see the data used for an ANOVA in a matrix.

```
> # Made up data:
> DV.mat <- matrix(c(1.94, 3.01, 3.93, 4.10,
+                    0.55, 3.59, 2.90, 1.90,
+                    1.04, 2.13, 2.11, 3.40,
+                    3.24, 1.72, 3.13, 2.70,
+                    2.30, 1.81, 3.86, 3.89,
+                    0.74, 2.31, 4.12, 4.00), ncol = 4, byrow = T)
> colnames(DV.mat) <- c("A", "B", "C", "D")
> # What do the data look like?
> DV.mat
        A    B    C    D
[1,] 1.94 3.01 3.93 4.10
[2,] 0.55 3.59 2.90 1.90
[3,] 1.04 2.13 2.11 3.40
[4,] 3.24 1.72 3.13 2.70
[5,] 2.30 1.81 3.86 3.89
```

```
[6,] 0.74 2.31 4.12 4.00
```

In our example, the columns of the matrix represent different environmental conditions, and an individual element is the persistance time (in minutes) of a mouse, assigned to one of the conditions, on an exercise wheel.

Alternatively, you could see the data for an ANOVA in the form of two vectors: one of the dependent variable, and one of the independent factors.

```
> DV.vec <- c(DV.mat)
> IV.vec <- factor(c( col(DV.mat) ), labels = colnames(DV.mat))
```

The only command that you might not yet be familiar with is `col`, which takes a matrix and indicates the column of each entry. Therefore, `c(DV.mat)` puts the dependent variable matrix into one vector with columns stacked on top of each other, `col(DV.mat)` indicates the column of each entry of the dependent variable, and `c( col(DV.mat) )` then puts the column numbers (i.e., the independent variable condition) into one vector with columns stacked on top of each other. Neat!

You also might see the data used for an ANOVA in the form of one vector for *each* factor level.

```
> # Vector for each independent variable condition:
> DV.A <- c(1.94, 0.55, 1.04, 3.24, 2.30, 0.74)
> DV.B <- c(3.01, 3.59, 2.13, 1.72, 1.81, 2.31)
> DV.C <- c(3.93, 2.90, 2.11, 3.13, 3.86, 4.12)
> DV.D <- c(4.10, 1.90, 3.40, 2.70, 3.89, 4.00)
```

And even though you can do the hand calculations directly on those four vectors, you must combine then into one *big* vector for the built-in ANOVA functions to work.

```
> # The same DV and IV vectors as before.
> DV.vec2 <- c(DV.A, DV.B, DV.C, DV.D)
> IV.vec2 <- factor( c( rep(1, length(DV.A)),
+                       rep(2, length(DV.B)),
+                       rep(3, length(DV.C)),
+                       rep(4, length(DV.D)) ) )
```

Notice that building the dependent variable vector and independent factor vector was much easier starting out with a matrix than with vectors of individual scores. An alternative construction (which works when the sample sizes are unequal per group) is starting out by putting the dependent variable into a `list` and using the `unlist` command to combine each group into one long vector.

> **Important**: When you are trying to run a One-Way ANOVA in `R` using the built in functions (`aov` or `lm`), you *must* enter two vectors separated by the formula operator (`~`). One of those vectors (on the left size of `~`) must be a numeric vector of scores on the dependent variable, and the other vector (on the right side of `~`) must be a factor vector of scores on the independent variable. If the independent variable/group membership vector is *not* a factor, then you might end up with strange results.

We will use our data to perform an ANOVA two ways, once by R and once using the aov function. The hypotheses for each ANOVA will be the same, that is "all of the population means are equal" versus "at least one population mean differs from at least one other population mean."

$$\begin{aligned} &\mathrm{H}_0 : \mu_1 = \mu_2 = \mu_3 = \mu_4 \\ &\mathrm{H}_1 : \text{At least one } \mu \text{ is different.} \end{aligned}$$

The following code will be identical ways of answering that question using R.

11.1.2 Using the ANOVA Formula

After checking ANOVA assumptions, either by graphical or numeric methods (see the previous chapter), you are ready to perform an ANOVA. To calculate the ANOVA directly in R, it is easiest to work with the long vector of scores on the dependent variable, long vector of group membership on the independent variable, and the tapply function. Alternatively, you can use the original matrix and the apply function, or the vectors of scores and the mean/var/length functions, but these are slightly more cumbersome to employ.

The first step in ANOVA calculations is finding the (weighted) grand mean and individual group means.

```
> # Find the group means and the grand mean:
> xbar_j <- tapply(X = DV.vec, INDEX = IV.vec, FUN = mean)
> xbar_G <- mean(DV.vec)
```

Remember that tapply takes a vector of scores, a vector of index values (e.g., to which group each scores belongs), and a function to perform on the set of scores within each level of the index (e.g., within each group).

We also will need our within-group variances and within-group sample sizes for further calculations, and, not surprisingly, the exact same tapply function will be easiest to use here as well.

```
> # Find the within-group standard dev and sample size:
> s2_j <- tapply(X = DV.vec, INDEX = IV.vec, FUN = var)
> n_j  <- tapply(X = DV.vec, INDEX = IV.vec, FUN = length)
```

We will use the group means, grand means, group variances, and group sample sizes to calculate *three* sums of squares in a One-Way ANOVA: the Sums of Squares Between (SSB), Sums of Squares Within (SSW), and Sums of Squares Total (SST). The easiest sums of squares to calculate is $SST = \sum_{i,j}(x_{ij} - \bar{x}_G)^2$, which is just like the numerator of a sample variance calculation.

```
> SST <- sum( (DV.vec - xbar_G)^2 )
```

We could have also found the sample variance of DV.vec (using the var function) and multiplied it by the total number of observations minus 1. Another sums of squares to calculate is $SSB = \sum_{j=1}^{g} n_j(\bar{x}_j - \bar{x}_G)^2$, which is relatively simple using the (already calculated) output of the tapply function.

```
> SSB <- sum( n_j*(xbar_j - xbar_G)^2 )
```

The final sums of squares is $SSW = \sum_{j=1}^{g} \sum_{i=1}^{n_j} (x_{j_i} - \bar{x}_j)^2 = \sum_{j=1}^{g} [(n_j - 1)s_j^2]$, which is a generalization of the numerator of a pooled variance. Similar to SSB, SSW is rather easy to calculate in `R` by taking advantage of the `tapply` function.

```
> SSW <- sum( (n_j - 1)*s2_j )
```

As a check, we know that the sums of squares are summative, so $SST = SSB + SSW$.

```
> SSB             # sums of squares between
[1] 12.10658
> SSW             # sums of squares within
[1] 14.898
> SSB + SSW       # adding up the sums of squares
[1] 27.00458
> SST             # the total sums of squares
[1] 27.00458
> # Are the sums of squares summative?  Check! Yippee!  Stats doesn't lie!
```

There are also three sets of degrees of freedom, each of them corresponding to a sums of squares, $df_B = g - 1$, $df_W = N - g$, and $df_T = N - 1 = df_B + df_W$, where $N = \sum_{j=1}^{g} n_j$. With enough motivation, we could figure out the degrees of freedom by hand (and by counting), but I am a bit too lazy for complicated tasks if I have `R` to solve my problems for me.

```
> ( dfB <- length(xbar_j) - 1 )
[1] 3
> ( dfW <- sum(n_j) - length(xbar_j) )
[1] 20
> ( dfT <- sum(n_j) - 1 )
[1] 23
```

The final two calculations in an ANOVA are the mean squares and the F-statistic. Note that $MSB = SSB/df_B$, $MSW = SSW/df_W$ and $F = MSB/MSW$.

```
> ( MSB <- SSB/dfB ) # mean squares between
[1] 4.035528
> ( MSW <- SSW/dfW ) # mean squares within
[1] 0.7449
> ( F   <- MSB/MSW ) # F-statistic
[1] 5.417543
```

And as the F-distribution has two types of degrees of freedom (numerator degrees of freedom corresponding to df_B and denominator degrees of freedom corresponding to df_W), we can find the p-value by using the `pf` function. Note that the F test from an ANOVA is typically a one-sided (upper-tailed) test because we care about mean divergences that are *greater* than chance.

```
> (p <- pf(q = F, df1 = dfB, df2 = dfW, lower.tail = FALSE))
[1] 0.00681569
```

As $p = 0.007 < .05 = \alpha$, we would reject the null hypothesis, and claim that we have evidence that mice in different environmental conditions did not all spend the same amount of time, on average, on the exercise wheel. We could have also tested our hypotheses by finding an F-critical value for which to compare our F_{obt} test statistic.

```
> # Setting alpha and finding F_crit:
> alpha <- .05
> ( F_crit <- qf(p = alpha, df1 = dfB, df2 = dfW, lower.tail = FALSE) )
[1] 3.098391
> # Should we reject our null hypothesis?
> F > F_crit
[1] TRUE
```

11.1.3 Using the aov Function

An alternative (easier) method of performing an ANOVA is by using the `aov` command in `R`. The `aov` command is rather simple for a One-Way ANOVA:

```
aov(formula, data)
```

where the `formula` is the same type of formula described in the independent samples t-test chapter (but with more than two groups in the factor vector), and the optional `data` argument is the name of the `data.frame` containing the predictor and response variables. Because we assigned our data to individual vectors (and not to sub-objects of a data frame), we should not include the `data` argument.

```
> mod.aov <- aov(DV.vec ~ IV.vec)
```

But if our data were actually part of a `data.frame`, then we could indicate our variables *by name* (without using `attach` to attach variables of the data frame) if we let `aov` know in which `data.frame` to look for those variables.

```
> # Put the variables into a dataframe:
> dat <- data.frame(DV = DV.vec, IV = IV.vec)
> # If we don't indicate the data.frame:
> #  - we can't call the variables by name, but
> #  - we must indicate the data frame with the $ operator
> try(mod.aov2 <- aov(DV ~ IV), silent = TRUE)[1] # error!
[1] "Error in eval(expr, envir, enclos) : object 'DV' not found\n"
> mod.aov2 <- aov(dat$DV ~ dat$IV)                # no error!
> # If we indicate the data frame in the "data" argument:
> #  - we can call the variables by name.
> mod.aov3 <- aov(DV ~ IV, data = dat)
```

After forming an `aov` object, we can obtain summary statistics (using `summary.aov` or `summary`). Note that typing the object name (`mod.aov`) not surrounded by the `summary` function results in minimal output without much needed information. The limited information when entering `mod.aov` into the `R` console is due to several, hidden, `print` (or `cat`) statements that `R` programmers wrote into runs from the `aov` object, as explained in a previous chapter.

```
> summary.aov(mod.aov)  # which is identical to...
            Df Sum Sq Mean Sq F value  Pr(>F)
IV.vec       3  12.11   4.036   5.418 0.00682 **
Residuals   20  14.90   0.745
---
Signif. codes:
0
> summary(mod.aov)      # in this case!
            Df Sum Sq Mean Sq F value  Pr(>F)
IV.vec       3  12.11   4.036   5.418 0.00682 **
Residuals   20  14.90   0.745
---
Signif. codes:
0
```

And the `IV.vec` row tells us the between stuff, whereas the `Residuals` row tells us the within/error stuff. Unfortunately, `R` does not provide the SST, but you can determine SST by adding up the rows of the ANOVA table.

A *third* method of obtaining the identical ANOVA table is through the `lm` function.

```
lm(formula, data, ... )
```

which will be discussed more thoroughly in the next chapter. The syntax for `lm` is very similar to that for `aov`.

```
> mod.lm <- lm(DV.vec ~ IV.vec)
```

To obtain an ANOVA table from `lm`, we must use `anova` rather than the `summary` function.

```
> anova(mod.lm)
Analysis of Variance Table

Response: DV.vec
          Df Sum Sq Mean Sq F value   Pr(>F)
IV.vec     3 12.107  4.0355  5.4175 0.006816 **
Residuals 20 14.898  0.7449
---
Signif. codes:
0
```

As is fairly obvious from the similarities in coding, regression analysis and ANOVA procedures are different manifestations of the same analysis.

11.1.4 ANOVA Power Calculations

Because ANOVA generally includes more than two groups (unless your ANOVA is silly), the alternative hypothesis is generally more true than for the t-tests of the previous several chapters. More groups means that there is less likelihood that all (population) group means are the same. But researchers still frequently fail to detect the true alternative hypothesis *because* more groups also (generally) implies that more people are needed to detect an effect. Thus, researchers should still estimate the optimal sample size for detecting a reasonable effect lest the entire data collection operation be for naught. Determining the power of an ANOVA depends on similar information as to that from the simple z and t-tests.

1. The actual information about the population:
   - The *true* effect size: f.
   - The within-group variance: σ^2.
2. The within-group sample sizes: n_j.
3. The number of groups: g.
4. How to make a decision regarding our hypotheses:
   - The desired Type I error rate: α.

The easiest method of performing ANOVA-based power calculations in `R` is to use one of two pre-packaged functions: (1) `power.anova.test` (in the stats package); and (2) `pwr.anova.test` (in the `pwr` package). The arguments for `power.anova.test` are as follows.

```
power.anova.test(groups = NULL, n = NULL,
                 between.var  = NULL, within.var = NULL,
                 sig.level = 0.05, power = NULL)
```

In the `power.anova.test` function, `groups` is the number of groups in the One-Way ANOVA, `n` is the number of observations per group, `between.var` is the between group variance (with $g - 1$ in the denominator, weird), `within.var` is the within group variance, `sig.level` is the Type I error rate, and `power` is the power of the test. As in `power.t.test`, you pass to `power.anova.test` *all but one* of the arguments, and the power function will return the missing argument.

> **Important**: The `power.anova.test` function can only be used for a *balanced, One-Way ANOVA with assumed common within-group variances.*

Assume that we have four groups with true population means $\mu_1 = 120$, $\mu_2 = 121$, $\mu_3 = 123$, and $\mu_4 = 124$. Moreover, assume that we know $\sigma^2 = 50$. Then if we want to find out the power ("... probability of rejecting a false null hypothesis ...") to detect an effect if $n_j = 15$ people per group and $\alpha = .05$, simply use the `power.anova.test` function.

```
> mu.j   <- c(120, 121, 123, 124)
> sigma2 <- 50
> power.anova.test(groups = length(mu.j),
+                  n = 15,
+                  between.var = var(mu.j),
+                  within.var = sigma2,
+                  sig.level = .05,
+                  power = NULL)
     Balanced one-way analysis of variance power calculation

         groups = 4
              n = 15
    between.var = 3.333333
     within.var = 50
      sig.level = 0.05
          power = 0.257895

NOTE: n is number in each group
```

If $g = 4$ and $n = 15$, then we have very little power to detect this fairly small effect. But increase the number of people per group $\rightarrow$ increased power!

```
> power.anova.test(groups = length(mu.j),
+                  n = 30,
+                  between.var = var(mu.j),
+                  within.var = sigma2,
+                  sig.level = .05,
+                  power = NULL)
     Balanced one-way analysis of variance power calculation

         groups = 4
              n = 30
    between.var = 3.333333
     within.var = 50
      sig.level = 0.05
          power = 0.5031345

NOTE: n is number in each group
```

Most a-priori power calculations seek to determine the sample size needed for a particular (pre-specified) power. For example, if you want to figure out the within-group sample size needed for a power of $1 - \beta = .90$, then simply leave `n = NULL` and include `.90` as the value of power.

```
> ( mod.pwr <- power.anova.test(groups = length(mu.j),
+                               n = NULL,
+                               between.var = var(mu.j),
+                               within.var = sigma2,
+                               sig.level = .05,
+                               power = .90) )
```

```
     Balanced one-way analysis of variance power calculation

         groups = 4
              n = 71.84254
    between.var = 3.333333
     within.var = 50
      sig.level = 0.05
          power = 0.9

NOTE: n is number in each group
```

Therefore, you need at least 72 people. Wow! Those are pretty large groups.

An alternative function used to perform power calculations is `pwr.anova.test` in the `pwr` package with the following setup.

```
pwr.anova.test(k = NULL, n = NULL,
               f  = NULL,
               sig.level = 0.05,
               power = NULL)
```

Now, `k` is the number of groups (same as `groups` from before), `n` is the sample size per each group (go balanced designs), `f` is the effect size (which combines `between.var` and `within.var`), and everything else is the same. As in `power.anova.test`, you should pass all but one argument, and then `pwr.anova.test` will return the final answer. The only tricky argument in `pwr.anova.test` is the aggregated effect size (see Cohen, 1988):

$$f = \sqrt{\frac{\sum_j \frac{n_j}{N}(\mu_j - \mu)^2}{\sigma^2}}$$

This effect size combines the within group variance and the between group variance. Therefore, specifying `f` prevents you from having to specify both of `between.var` *and* `within.var` (go simplifications!). So to use `pwr.anova.test`, we must first calculate f before plugging things into power function.

```
> g  <- length(mu.j)
> n1 <- 15
> N1 <- g*n1
> f1 <- sqrt(n1*sum((mu.j - mean(mu.j))^2)/(N1*sigma2))
```

And then we can plug everything into `pwr.anova.test`.

```
> library(pwr)
> pwr.anova.test(k = g, n = n1,
+                f = f1,
+                sig.level = .05,
+                power = NULL)
     Balanced one-way analysis of variance power calculation

              k = 4
              n = 15
```

```
              f = 0.2236068
      sig.level = 0.05
          power = 0.257895

NOTE: n is number in each group
```

Notice that `pwr.anova.test` returns the exact same power calculated value as `power.anova.test` given the same inputs. We can also repeat the (exact same) power calculations with the other within group sample size.

```
> n2 <- 30
> N2 <- g*n2
> f2 <- sqrt(n2*sum((mu.j - mean(mu.j))^2)/(N2*sigma2))
> pwr.anova.test(k = g, n = n2,
+                f = f2,
+                sig.level = .05,
+                power = NULL)
     Balanced one-way analysis of variance power calculation

              k = 4
              n = 30
              f = 0.2236068
      sig.level = 0.05
          power = 0.5031345

NOTE: n is number in each group
```

Unfortunately, finding the sample size needed for a particular power calculation seems impossible at first glance. To find `n`, we must plug something into the `f` argument, but the within-group sample size is *part of the calculation* of `f`! One way out of this bind is to realize that $\frac{n}{N}$ can be thought of as the "proportion of total observations within each group." Because the proportion of scores within a group will always be the same if the number of groups are the same (assuming a balanced design), we can redefine this proportion as $p = \frac{n}{N} = \frac{1}{g}$. Therefore, to find the sample size needed for a particular power, add a prior step to calculate `p` before `f` before `n`.

```
> p  <- 1/g
> f3 <- sqrt(p*sum((mu.j - mean(mu.j))^2)/(sigma2))
> pwr.anova.test(k = g, n = NULL,
+                f = f3,
+                power = .90)
     Balanced one-way analysis of variance power calculation

              k = 4
              n = 71.84254
              f = 0.2236068
      sig.level = 0.05
          power = 0.9

NOTE: n is number in each group
```

And (not surprisingly) the needed sample size is identical to that from `power.anova.test`. The `pwr` package uses the f definition of effect size rather than the `power.anova.test` definition, as f is more generalizable to complicated designs.

11.1.5 Post-Hoc Procedures

After making an inferential decision, you should then perform follow up procedures, including estimating effect sizes, constructing confidence intervals for the mean of each group, and comparing pairs of means using post-hoc tests.

Effect Sizes

The two basic effect sizes for a One-Way ANOVA, eta-squared (η^2) and omega-squared (ω^2), are pretty easy to calculate in `R`. Unfortunately, I do not know of any function to automatically calculate those effect sizes. So you must calculate effect sizes either by `R` calculations or by pulling things (hopefully useful things) out of the ANOVA object.

The first effect size is called η^2 and is defined as

$$\eta^2 = \frac{SSB}{SST}$$

η^2 describes the proportion of variability in the *sample* accounted for by category membership. An alternative to η^2 is ω^2, which is calculated by

$$\omega^2 = \frac{SSB - df_B \times MSE}{SST + MSE}$$

for a One-Way ANOVA. Unlike η^2, ω^2 adjusts the "proportion of variance accounted for" by the complicatedness of the experimental design ... err ... the number of categories and observations within each category. Note that η^2 is similar to R^2 from linear regression, whereas ω^2 is similar to adjusted R^2. These topics will come up again in the next chapter when examining correlations and simple linear regressions. To calculate η^2 and ω^2, first run the ANOVA (which we did earlier), `anova` the ANOVA, and then pull out the sub-objects.

```
> (mod.anova <- anova(mod.aov))
Analysis of Variance Table

Response: DV.vec
          Df Sum Sq Mean Sq F value   Pr(>F)
IV.vec     3 12.107  4.0355  5.4175 0.006816 **
Residuals 20 14.898  0.7449
---
Signif. codes:
0
> names(mod.anova)
[1] "Df"      "Sum Sq"  "Mean Sq" "F value" "Pr(>F)"
```

> **Note**: In the dark ages of R, you could not run an ANOVA and conveniently pull out sums of squares, degrees of freedom, etc. To pull out these sub-objects, you first had to run lm (rather than aov), then anova your regression object, and then pull out the appropriate SS or df. However, R has grown up, and the writers/coders have made it easier to manipulate the aov function. Oddly, this ease of use makes running ANOVAs through R using aov pretty redundant. For example, performing a summary of the aov object or an anova of the aov object results in identical displays. Yet, for all their apparent similarities, the summary object has nothing in it, whereas the anova object includes all of your ANOVA table parts as sub-objects. Weird!

To extract parts of an ANOVA, you must: (1) Run aov (or lm) to fit the model; and then (2) Run anova on the aov object. We're in meta-ANOVA land here, apparently.

```
> (SS <- mod.anova$Sum)
[1] 12.10658 14.89800
> (df <- mod.anova$Df)
[1]  3 20
> (MS <- mod.anova$Mean)
[1] 4.035528 0.744900
```

The SS, df, and MS for a One-Way ANOVA in R include the "between" stuff followed by the "within" stuff. However, to get the "total" stuff, we still must sum things up ourselves.

```
> (eta2   <- SS[1]/sum(SS))
[1] 0.4483159
> (omega2 <- (SS[1] - df[1]*MS[2])/(sum(SS) + MS[2]))
[1] 0.3557502
```

Note that ω^2 is a bit smaller than η^2 due to the small sample size and fairly large number of categories.

Confidence Intervals and Plots

ANOVA-based confidence intervals are simple generalizations of the independent samples t-test confidence intervals using MSW as the estimation of the common within-group variance (σ^2), and df_W in place of df.

For a single mean, the formula for the confidence interval is

$$CI = \bar{x}_j \pm t_{N-g;\,(1+\gamma)/2} \cdot \sqrt{\frac{MSW}{n_j}}$$

where γ is the confidence level.

```
> gamma <- .95
> ( CI.A <- { mean(DV.A) +
+             c(-1, 1)*qt((1 + gamma)/2, df = dfW)*sqrt(MSW/length(DV.A)) } )
[1] 0.9000123 2.3699877
```

```
> ( CI.B <- { mean(DV.B) +
+             c(-1, 1)*qt((1 + gamma)/2, df = dfW)*sqrt(MSW/length(DV.B)) } )
[1] 1.693346 3.163321
> ( CI.C <- { mean(DV.C) +
+             c(-1, 1)*qt((1 + gamma)/2, df = dfW)*sqrt(MSW/length(DV.C)) } )
[1] 2.606679 4.076654
> ( CI.D <- { mean(DV.D) +
+             c(-1, 1)*qt((1 + gamma)/2, df = dfW)*sqrt(MSW/length(DV.D)) } )
[1] 2.596679 4.066654
```

And some (but not all) of the confidence intervals overlap.

Alternatively, we can form a vector of means/lengths to simultaneously find a vector of lower bounds and a vector of upper bounds for all of the groups.

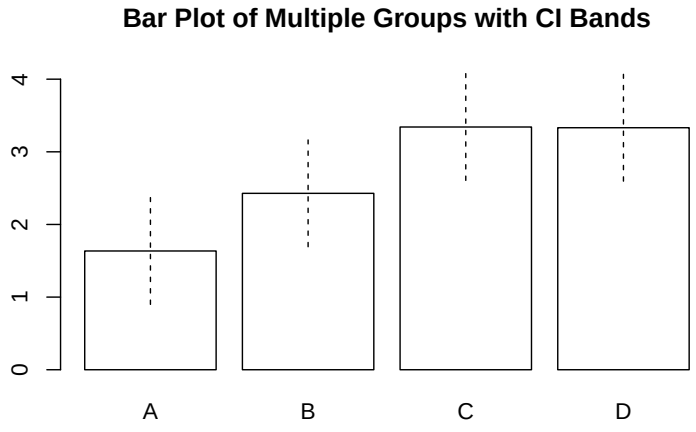
```
> (CI.l <- xbar_j - qt((1 + gamma)/2, df = dfW)*sqrt(MSW/n_j))
        A         B         C         D
0.9000123 1.6933457 2.6066790 2.5966790
> (CI.u <- xbar_j + qt((1 + gamma)/2, df = dfW)*sqrt(MSW/n_j))
       A        B        C        D
2.369988 3.163321 4.076654 4.066654
```

Finally, because we have the following vectors corresponding to each group

- A vector of group means: `xbar_j`.
- A vector of group lengths: `n_j`.
- A vector of group variances: `s2_j`.
- A vector of CI lower bounds: `CI.l`.
- A vector if CI upper bounds: `CI.u`.

we can visually compare the groups using a `barplot` with confidence interval bands.

```
> x.bar <- barplot(xbar_j, ylim = c(0, max(CI.u)),
+                  names = colnames(DV.mat),
+                  main = "Bar Plot of Multiple Groups with CI Bands",
+                  col = "white")
> segments(x0 = x.bar, x1 = x.bar,
+          y0 = CI.l,  y1 = CI.u,
+          lty = 2)
```



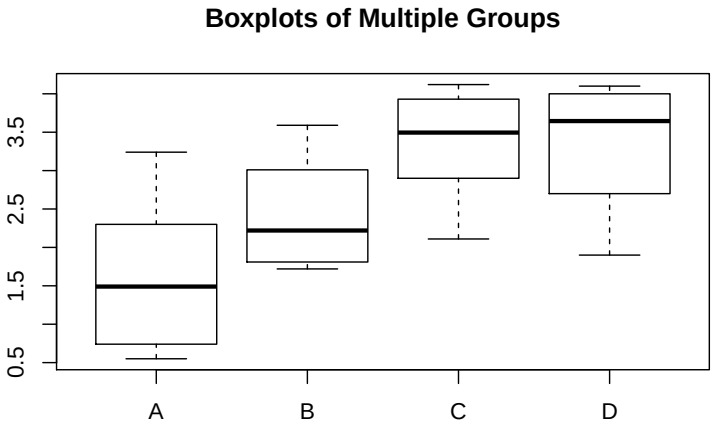
In the above code chunk, `barplot` made a plot of each group mean, and assigning the `barplot` to `x.bar` saved the midpoint of each bar. The other plotting command, `segments`, is like `lines` or `points` or `abline`, in that the output of `segments` is plotted on top of an existing plot. In this case, `segments` takes four main arguments:

1. `x0`: A vector of values on the x-axis where the segments should start to be drawn.
2. `y0`: A vector of values on the y-axis where the segments should start to be drawn.
3. `x1`: A vector of values on the x-axis where the segments should be finished.
4. `y1`: A vector of values on the y-axis where the segments should be finished.

And then `R` connects the first vectors of (x, y) values with the second vectors of (x, y) values. The `lty = 2` resulted in dashed (rather than solid) lines.

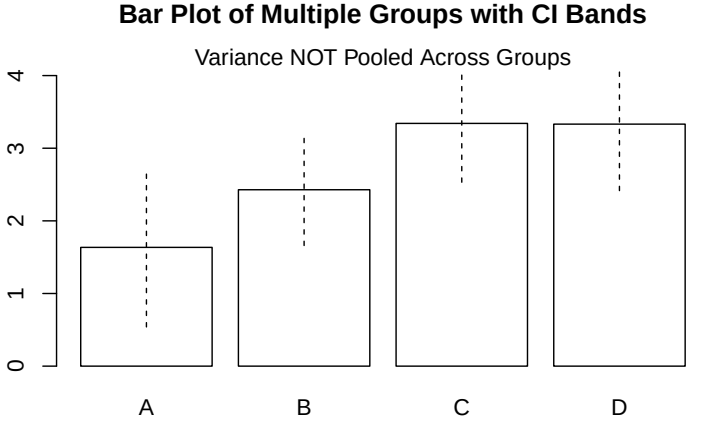
Alternatively, we could have constructed boxplots in exactly the same way that we constructed them in the two-samples t-tests chapter, by using the formula operator.

```
> boxplot(DV.vec ~ IV.vec,
+         main = "Boxplots of Multiple Groups",
+         names = colnames(DV.mat))
```



Because the homogeneity of variance assumption might be violated, a more appropriate visual comparison would be barplots of the variables with confidence bounds constructed from individual variance estimates rather than the MSW.

```
> # Barplots with confidence bounds.
> x.bar2 <- barplot(xbar_j, names = colnames(DV.mat),
+                   main = "Bar Plot of Multiple Groups with CI Bands",
+                   ylim = c(0, max(CI.u)), col = "white")
> mtext("Variance NOT Pooled Across Groups")
> segments(x0 = x.bar2, x1 = x.bar2,
+          y0 = xbar_j - qt((1 + gamma)/2, df = n_j - 1)*sqrt(s2_j/n_j),
+          y1 = xbar_j + qt((1 + gamma)/2, df = n_j - 1)*sqrt(s2_j/n_j),
+          lty = 2)
```



We also would want to construct confidence intervals on the difference between means rather than the means themselves. Assuming homogeneity of variance, the confidence interval formula for the difference between means is

$$CI = (\bar{x}_i - \bar{x}_j) \pm t_{N-g;\,(1+\gamma)/2} \cdot \sqrt{MSW\left(\frac{1}{n_i} + \frac{1}{n_j}\right)}$$

As an example, we can compare groups A and B without correcting for multiple testing.

```
> # The first part: the difference between means.
> (CI.AB <- {mean(DV.A) - mean(DV.B) +
+
+ # The second part: -/+ the critical value.
+              c(-1, 1)*qt((1 + gamma)/2, df = dfW)*
+
+ # The third part: the standard error (using MSW).
+                        sqrt(MSW*(1/length(DV.A) + 1/length(DV.B)))})
[1] -1.8327629  0.2460962
```

If we do *not* assume homogeneity of variance, then the easiest method of constructing confidence intervals is by using the `t.test` function but setting `var.equal` to FALSE. Of course, neither procedure is optimal because performing multiple t-tests or constructing multiple confidence intervals results in inflated Type I error rates (or inaccurate overall confidence interval coverage rates). To account for the family-wise error rate, we should correct our post-hoc procedures for multiple comparisons.

Post-Hoc Tests

Once you reject the null hypothesis in an ANOVA, you should perform post-hoc (or a priori) comparisons. Several of the post-hoc tests do not depend on whether or not we rejected the null hypothesis. Of those, the *easiest* comparison to do in `R` is the Tukey test.

```
> ( mod.tuk <- TukeyHSD(mod.aov, conf.level = .95) )
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = DV.vec ~ IV.vec)

$IV.vec
          diff        lwr      upr     p adj
B-A  0.7933333 -0.6013684 2.188035 0.4052637
C-A  1.7066667  0.3119649 3.101368 0.0131422
D-A  1.6966667  0.3019649 3.091368 0.0137367
C-B  0.9133333 -0.4813684 2.308035 0.2878407
D-B  0.9033333 -0.4913684 2.298035 0.2966874
D-C -0.0100000 -1.4047018 1.384702 0.9999970
```

The `TukeyHSD` function outputs the difference between the groups, the lower value of the (Tukey corrected) confidence interval, the upper value of the (Tukey corrected) confidence interval, and the adjusted p-value. You can easily perform the set of Tukey comparisons

even without the `TukeyHSD` function. The Tukey HSD test is identical to a standard independent samples t-test, but Tukey HSD uses the MSW as the pooled variance and has as its test statistic the absolute value of t multiplied by $\sqrt{2}$. Of course, the Tukey HSD test uses a different distribution to determine critical values--the Tukey distribution (or studentized range distribution), but `R` provides functions for calculating the appropriate quantiles/probabilities of the Tukey distribution (`ptukey` and `qtukey`). First, finding the p-value for a particular pair of comparisons.

```
> # The first part to the Tukey HSD is calculating the t-statistic:
> L.AB   <- mean(DV.A) - mean(DV.B)
> Lse.AB <- sqrt(MSW*(1/length(DV.A) + 1/length(DV.B)))
> t.AB   <- L.AB/Lse.AB
> # The second part to the Tukey HSD is:
> #  - Taking the absolute value of the t-statistic,
> #  - Multiplying the t-statistic by the square root of 2, and
> #  - Finding the probability of AT LEAST that value using ptukey.
> ptukey(q = sqrt(2)*abs(t.AB),
+        nmeans = length(xbar_j), df = dfW,
+        lower.tail = FALSE)
[1] 0.4052637
```

But it might be easier to find the critical value of the Tukey distribution, divide the critical value by $\sqrt{2}$, and compare the (modified) critical value to the absolute value of pairwise t-statistics.

```
> # The critical value (again, strangely, dividing by sqrt(2))
> q.crit <- qtukey(p = .05,
+                  nmeans = length(xbar_j), df = dfW,
+                  lower.tail = FALSE)/sqrt(2)
> # Compare the critical value to the absolute value of t:
> q.crit    # Is this smaller,
[1] 2.798936
> abs(t.AB) # than this?
[1] 1.592089
```

Another multiple comparison correction uses the Bonferroni p-value in correcting p-values with the `pairwise.t.test` function.

```
> ( mod.bon <- pairwise.t.test(x = DV.vec, g = IV.vec,
+                              p.adjust.method = "bonferroni",
+                              pool.sd = TRUE) )
        Pairwise comparisons using t tests with pooled SD

data:  DV.vec and IV.vec

  A     B     C
B 0.762 -     -
C 0.016 0.490 -
D 0.017 0.509 1.000
```

```
P value adjustment method: bonferroni
```

The function `pairwise.t.test` (using the argument `pool.sd = TRUE` and setting `p.adjust.method` to `"bonferroni"`) performs all pairs of t-tests (assuming a common variance for all groups) and adjusts the p-value of each test by the Bonferroni correction. Rather than adjusting the specified Type I error rate down (so that $\alpha_B = \alpha/k$, where k is the number of comparison), `pairwise.t.test` multiplies individual p-values by the number of comparisons.

As in the Tukey HSD procedure, you can easily perform the Bonferroni correction by using simple functions in `R`. The three values you will need are as follows.

1. The mean squares within (MSW)
2. The degrees of freedom within (df_W)
3. The number of comparisons we will make (k)

We had already found MSW and df_W in the omnibus ANOVA table.

```
> MSW
[1] 0.7449
> dfW
[1] 20
```

Alternatively, we could have pulled both MSW and df_W from the `anova.lm` object (which is often easier than working with the `summary.aov` object).

```
> # Saving the anova of the lm object.
> aov.lm <- anova(mod.lm)
> # Pulling out the degrees of freedom and mean squares.
> aov.lm$Mean[2]
[1] 0.7449
> aov.lm$Df[2]
[1] 20
```

Therefore, need only determine the number of comparisons we plan on making. If we decide to test differences between all pairs of groups, then we would have $k = \binom{g}{2} = \frac{g!}{2!(g-2)!} = \frac{g(g-1)}{2}$ possible t-tests. In our case, $g = 4$, so $k = \frac{4(3)}{2} = 6$ pairs of comparisons. Thus $\alpha_B = \alpha/k$, and if $\alpha = .05$, then $\alpha_B = \alpha/k = .05/6 \approx .0083$. Once we have our modified α_B, we can perform an independent samples t-test using α_B as the modified, per comparison, α-level.

```
> # The mean difference:
> L.CD    <- mean(DV.C) - mean(DV.D)
> # The standard error of the mean difference:
> Lse.CD  <- sqrt(MSW*(1/length(DV.C) + 1/length(DV.D)))
> # The t-statistic:
> t.CD    <- L.CD/Lse.CD
> # Either compare the two-tailed p-value to alpha_B.
> p     <- 2*pt(abs(t.CD), df = dfW, lower.tail = FALSE)
> alphB <- .05/6
> alphB     # Is this larger,
```

```
[1] 0.008333333
> p         # than this?
[1] 0.9841877
> # Or compare the two-tailed critical value to our t-statistic.
> t.crit <- qt(alphB/2, df = dfW, lower.tail = FALSE)
> t.crit    # Is this smaller,
[1] 2.927119
> abs(t.CD) # than this?
[1] 0.02006835
```

You can use the above logic to build modified confidence intervals based on the Bonferroni critical value (only replacing $t_{df_W;\,(1+\gamma)/2}$ with $t_{df_W;\,(1-(1-\gamma)/(2k))}$). But what if you want to determine if pairs of groups are different from other pairs of groups? Or if the means of three groups is different from the mean of a new, fancy experimental group? Simple paired comparisons cannot answer either of these questions. A more general method of performing post-hoc analyses is by means of linear contrasts.

Linear Contrasts

Let's pretend (a priori) that we expect the mean of groups "A" and "B" to be similar and the mean of groups "C" and "D" to be similar, but groups "A" and "B" to be different than groups "C" and "D". Then simple paired comparisons (or paired confidence intervals) will not provide this information. A more general method of comparing groups in an ANOVA is by means of linear contrasts. A linear contrast is defined as $L = a_1\mu_1 + a_2\mu_2 + \ldots a_g\mu_g$ such that $\sum_j a_j = 0$. In our simple example, we want to tests whether groups "A" and "B" differ from groups "C" and "D". Therefore, the contrast coefficients are as follows.

	A	B	C	D
L	-1	-1	$+1$	$+1$

These contrast coefficients correspond to the following hypotheses.

$$\begin{aligned} \mathrm{H}_0 &: -\mu_1 - \mu_2 + \mu_3 + \mu_4 = 0 \\ \mathrm{H}_1 &: -\mu_1 - \mu_2 + \mu_3 + \mu_4 \neq 0 \end{aligned}$$

Once we define a contrast, we can test the significance of the contrast using a modified t-test with the following formula:

$$t_{df} = \frac{\hat{L} - L}{\sqrt{MSW \sum_j \frac{a_j^2}{n_j}}}$$

with $df = N - g$. Notice that this contrast obeys the standard form of a t-test. The numerator is the estimate of the contrast minus the hypothetical contrast (given a true null hypothesis), and the denominator is the standard error of the contrast. Figuring out the pieces to plug into this function is pretty straightforward when using the `tapply` function (repeating earlier calculations to illuminate exactly what is happening).

```
> ## BASIC STATISTICS ##
> (xbar_j <- tapply(DV.vec, IV.vec, FUN = mean))
       A        B        C        D
1.635000 2.428333 3.341667 3.331667
> (n_j    <- tapply(DV.vec, IV.vec, FUN = length))
A B C D
6 6 6 6
> (s2_j   <- tapply(DV.vec, IV.vec, FUN = var))
        A         B         C         D
1.0887100 0.5349767 0.5954967 0.7604167
> ## PARTS FOR CONTRAST ##
> a_j  <- c(-1, -1, 1, 1)
> (dfW  <- sum( n_j - 1 ))
[1] 20
> (MSW  <- sum( (n_j - 1)*s2_j )/dfW)
[1] 0.7449
> (Lhat <- sum(a_j*xbar_j))
[1] 2.61
```

And then actually performing the test is pretty straightforward given the pieces.

```
> (t <- Lhat/sqrt((MSW*sum(a_j^2/n_j))))
[1] 3.703711
> (p <- 2*pt(abs(t), df = dfW, lower.tail = FALSE))
[1] 0.001405022
```

Therefore, $p = 0.0014 < .05 = \alpha$ means that we would reject the null hypothesis and claim to have evidence that the linear contrast is different from 0.

An alternative method of testing the significance of a contrast is by means of an F-statistic. The resulting ANOVA divides the SSL by MSW and, therefore, uses an F distribution with 1 degree of freedom in the numerator. The formula for this contrast sums of squares is

$$SSL = \frac{n\hat{L}^2}{\sum_j a_j^2} = \frac{n\big(\sum_j a_j \bar{x}_j\big)^2}{\sum_j a_j^2}$$

or, in R:

```
> (SSL <- (n_j[1]*Lhat^2)/(sum(a_j^2)))
       A
10.21815
```

Note that if you were to calculate $F = \frac{SSL}{MSW}$, you would find that $F = t^2$,

```
> (F <- SSL/MSW)
       A
13.71748
```

```
> t^2
[1] 13.71748
```

which immediately follows from F being a direct rearrangement of the t^2 formula. Why (you may ask) do we need to worry about SSL given that we can already test the contrast by means of a t-test? Well, I'm glad you asked (I think, assuming that you asked and that I know this information)! It turns out that if you performed a particular set of contrast that were set up in a particular way, then the SSL from those contrasts would add up to SSB from the ANOVA. So one could think of contrasts as usefully dividing the variance between groups. The rules for this educated breakdown is that the total number of contrasts must be $g-1$, the sum of the coefficients across each contrast must be 0, and the sum of the cross-product of coefficients across any two contrasts must also be 0. If these stipulations are satisfied, you have an "orthogonal" set of contrast. An example of an orthogonal set of contrasts is as follows.

	A	B	C	D
L1	-1	-1	$+1$	$+1$
L2	-1	$+1$	0	0
L3	0	0	-1	-1

Notice that all of our stipulations hold. There are $g-1=4-1=3$ contrasts, and

```
> a1 <- c(-1, -1, 1, 1)
> a2 <- c(-1, 1, 0, 0)
> a3 <- c(0, 0, -1, 1)
> sum(a1)
[1] 0
> sum(a2)
[1] 0
> sum(a3)
[1] 0
> sum(a1*a2)
[1] 0
> sum(a1*a3)
[1] 0
> sum(a2*a3)
[1] 0
```

Therefore $SSB = SSL_1 + SSL_2 + SSL_3$. We should (of course) check this claim.

```
> ## CONTRASTS ##
> (Lhat1 <- sum(a1*xbar_j))
[1] 2.61
> (Lhat2 <- sum(a2*xbar_j))
[1] 0.7933333
> (Lhat3 <- sum(a3*xbar_j))
[1] -0.01
```

```
> ## SUMS OF SQUARES ##
> (SSL1 <- (n_j[1]*Lhat1^2)/(sum(a1^2)))
       A
10.21815
> (SSL2 <- (n_j[1]*Lhat2^2)/(sum(a2^2)))
       A
1.888133
> (SSL3 <- (n_j[1]*Lhat3^2)/(sum(a3^2)))
    A
3e-04
> ## CHECK ##
> SSL1 + SSL2 + SSL3
       A
12.10658
> SSB
[1] 12.10658
```

Magic, isn't it!??!

Not surprisingly, several `R` functions allow you to automatically test the significance of contrasts. One commonly used function is `fit.contrast` in the `gmodels` package. There are four arguments for `fit.contrasts`: `model`, `varname`, and `coef`, and `conf.int`. `model` is an ANOVA fit from the `lm` or `aov` functions, `varname` is the name of the category membership vector (in quotes), `coef` is a vector (for one contrast) or matrix (for multiple contrasts), each row of which indicates the contrast coefficients, and `conf.int` is a number between 0 and 1 designating the confidence interval coverage rate. Note that `conf.int` must be specified *by the user* or `R` will not calculate a confidence interval for your contrast.

Let's fit a contrast on the previously run ANOVA:

```
> library(gmodels)
> mod.aov <- aov(DV.vec ~ IV.vec)
> fit.contrast(mod.aov, varname = "IV.vec",
+              coef = c(-1, -1, 1, 1), conf.int = NULL)
                       Estimate Std. Error t value Pr(>|t|)
IV.vec c=( -1 -1 1 1 )     2.61     0.7047   3.704 0.001405
```

Notice that the name of the row returned by `fit.contrast` is pretty ugly. One method of making reasonable looking names is by turning `coef` into a one-row matrix (using `rbind`) and then naming that row.

```
> fit.contrast(mod.aov, varname = "IV.vec",
+              coef = rbind(' -A-B+C+D' = c(-1, -1, 1, 1)),
+              conf.int = NULL)
                Estimate Std. Error t value Pr(>|t|)
IV.vec -A-B+C+D     2.61     0.7047   3.704 0.001405
```

And you can fit all of the contrasts by `rbind`ing the of the coefficients together.

```
> fit.contrast(mod.aov, varname = "IV.vec",
+              coef = rbind(' -A-B+C+D' = c(-1, -1, 1, 1),
+                           ' -A+B'     = c(-1, 1, 0, 0),
+                           '     -C+D' = c(0, 0, -1, 1)),
+              conf.int = NULL)
                Estimate Std. Error  t value Pr(>|t|)
IV.vec -A-B+C+D   2.6100     0.7047  3.70371 0.001405
IV.vec -A+B       0.7933     0.4983  1.59209 0.127047
IV.vec     -C+D  -0.0100     0.4983 -0.02007 0.984188
```

With multiple tests, you should probably correct your comparisons for an inflated family-wise error error. Unfortunately, `fit.contrast` does not provide methods of adjusting p-values for multiple comparisons. The easiest method of p-value correction would be to take the p-values supplied by `fit.contrast` and multiply them by the number of contrasts to arrive at Bonferroni-modified p-values. Of course, we would have no reason to correct for multiple comparisons if the family-wise error rate did not drastically escalate after each test. In the final section of this chapter, I discuss the consequence of multiple comparisons on the nominal FWE.

11.2 ANOVA and Family-Wise Error Rates

A major reason for performing an ANOVA rather than several independent samples t-test is due to the inflated α rate across a series of comparisons. One might want to know the degree of inflation and whether supposed corrections work as expected. The following code takes the number of groups (g), researcher specified Type I error rate (α), number of replications, population means and standard deviations of each group, and sample size of each group, and it estimates the family-wise error rate using independent samples t-tests versus an ANOVA.

```
> ######################
> # MultTest Function #
> ######################
>
> # This function is designed to take:
> # - a scalar indicating the number of groups,
> # - a scalar indicating the (specified) type I error rate (alpha),
> # - a scalar indicating the number of replications,
> # - a scalar/vector indicating the population means per group,
> #   - a scalar if all groups have the same population means
> #   - a vector if the groups have different population means
> # - a scalar/vector indicating the population sds per group, and
> # - a scalar/vector indicating the sample size per group.
>
> # And return:
> # - a vector of null hypothesis rejections using an ANOVA
> # - a vector of null hypothesis rejections using a t-test
> #   - each vector will be TRUE for reject/FALSE for not reject
> #   - the t-test vector is based on ANY rejection in all comparisons
>
```

```
> MultTest <- function(g = 2, alph = .05, reps = 1000,
                       mu = 0, sig = 1, n = 10){

  #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
  # Arguments:                                                         #
  # - g     - the number of groups                                     #
  # - alph  - the (specified) type I error rate                        #
  # - reps  - the number of replications/sims                          #
  # - mu    - the population mean per group                            #
  # - sig   - the population sd per group                              #
  # - n     - the population sample size per group                     #
  #                                                                    #
  # Values:                                                            #
  # - F.rej - rejections (TRUE/FALSE) using ANOVA                      #
  # - t.rej - rejections (TRUE/FALSE) using t-tests                    #
  #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#

  ## 1. A FEW STATISTICS OF OUR DATA ##
    mu   <- rep(mu,  length.out = g)  # the pop means
    sig  <- rep(sig, length.out = g)  # the pop sds
    n    <- rep(n,   length.out = g)  # the pop samp sizes

  ## 2. BUILDING THE FACTOR VECTOR ##

  # Building a vector to indicate the number of levels per group:
    ind    <- factor( rep(1:g, times = n) ) # a factor vector

  # Building empty vectors to store acceptance/rejectance:
    F.rej <- rep(FALSE, times = reps)
    t.rej <- rep(FALSE, times = reps)

  # We want to repeat this simulation a lot of times!
    for(i in 1:reps){

  ## 3. SIMULATING THE OBSERVATIONS ##

  # Simulating n_i observations for each group (into a list):
      dep  <- apply( cbind(n, mu, sig), MARGIN = 1,
                     FUN = function(x)
                       rnorm(n = x[1], mean = x[2], sd = x[3])
                   )

  # Unlisting the observations so that they are in a vector:
      dep  <- c( unlist(dep) )


  ## 4. THE ANOVA ##
      p.F      <- anova( lm(dep ~ ind) )$"Pr(>F)"[1]  # p-value
```

```
    F.rej[i] <- p.F < alph                          # p < alpha?


## 5. THE t-TEST ##

# Perform a t-test on all pairs of conditions.
    p.t <- pairwise.t.test(x = dep, g = ind,
                           p.adjust.method = "none",
                           pool.sd = TRUE)$p.value

    t.rej[i] <- any(p.t[!is.na(p.t)] < alph)  # is ANY p < alpha?

  } # END for i LOOP


## 6. RETURNING STUFF ##
  out <- list(F.rej = F.rej, t.rej = t.rej)

  return(out)

} # END MultTest FUNCTION
```

Once loaded, actually using the function is rather straightforward. For instance, to compare the ANOVA to all possible t-tests when there are only two groups, you would type in the following.

```
> set.seed(23407)
> mod1 <- MultTest(g = 2) # g = 2 for TWO GROUPS.
```

And if you were to type `names(mod1)`, you would find that there are two sub-objects in the `mod1` object: `F.rej` and `t.rej`. We can check the first few entries of each sub-object by using the `head` function.

```
> head(mod1$F.rej)  # what is in the F.rej subobject?
[1] FALSE FALSE FALSE FALSE FALSE FALSE
> head(mod1$t.rej)  # what is in the t.rej subobject?
[1] FALSE FALSE FALSE FALSE FALSE FALSE
```

`F.rej` and `t.rej` are `TRUE`/`FALSE` vectors indicating the number of times one would reject the null hypothesis that all group means are equal (using the specified α rate). We can find the proportion of rejections in either case by taking the mean of each sub-object.

```
> mean(mod1$F.rej)  # prop of rejections for using F
[1] 0.034
> mean(mod1$t.rej)  # prop of rejections for using t
[1] 0.034
```

And *both* proportion of rejections are close to $\alpha = .05$. Because the F-test is a generalization of the independent samples t-test to more than two groups, the proportion of

rejections *should* be identical when there are only two groups. However, when $g > 2$, the overall proportion of rejections using an ANOVA will be lower than multiple t-tests.

You have many options for how to use the `MultTest` function. For instance, we can check the overall Type I error rate when the groups have difference sample sizes of different standard deviations. We can also use `MultTest` if the groups have different sample sizes, just by changing the `n` argument to a vector indicating the specific group sample sizes.

```
> set.seed(280394)
> # Changing the sample sizes per groups:
> mod2 <- MultTest(g = 2, n = c(9, 15))
> lapply(mod2, FUN = mean)  # apply a function to list elements
$F.rej
[1] 0.055

$t.rej
[1] 0.055
> # Changing the standard devs per groups:
> mod3 <- MultTest(g = 2, sig = c(1, 3))
> lapply(mod3, FUN = mean)  # apply a function to list elements
$F.rej
[1] 0.067

$t.rej
[1] 0.067
```

And we can even use `MultTest` if the group have different means by changing the `mu` argument to a vector indicating the specific group means.

```
> mod4 <- MultTest(g = 2, mu = c(0, 1))
> lapply(mod4, FUN = mean)  # apply a function to list elements
$F.rej
[1] 0.581

$t.rej
[1] 0.581
```

By changing the means, the proportions of rejections is no longer the true Type I error rate but the *power* of the test. Of course, one does not gain by using the ANOVA procedure to compare two groups. Our interest resides in the false rejection rate when there are more than two groups.

```
> mod5 <- MultTest(g = 4)
> lapply(mod5, FUN = mean)
$F.rej
[1] 0.058

$t.rej
[1] 0.203
```

And as was suspected, increasing the number of groups drastically inflates the overall Type I error rate when using t-tests but does not affect the overall proportion of false rejections when using an ANOVA.

The `MultTest` function can be easily adjusted to determine the proportion of (overall) false rejections when using the Tukey, Bonferroni, or any number of Post-Hoc procedures. If one were to alter the `MultTest` function, he/she would find that the Bonferroni procedure tended to overcorrect and the Tukey procedure does not optimally correct when the group sample sizes are not all equal. The specific code for altering the `MultTest` function is left as an exercise.

11.3 Appendix: Functions Used

Here is a list of important functions from this chapter:

```
# ANOVA in R
col(mat)                    # what are the column numbers?
pf(q, df1, df2, lower.tail) # probability in F distribution
qf(p, df1, df2, lower.tail) # quantile in F distribution
aov(formula, data)          # perform an ANOVA
summary.aov(aov.mod)        # summarize our ANOVA in a nice way
lm(formula, data)           # perform a regression (ANOVA)
anova.lm(lm.mod)            # summarize a regression as an ANOVA

# ANOVA Power Calculations
library(pwr)                   # for the pwr.anova.test function
power.anova.test(groups, n,    # ANOVA power 1
                 between.var,
                 within.var,
                 sig.level,
                 power)
pwr.anova.test(k, n,            # ANOVA power 2
               f,
               sig.level,
               power)

# Confidence Intervals
barplot(height, names)      # draw a barplot
segments(x0, y0,
         x1, y1, ... )      # draw short lines on existing plots

# Post-Hoc Tests
TukeyHSD(aov.mod, conf.level)      # Tukey comparisons please :)
ptukey(q, nmeans, df, lower.tail)  # probability in Tukey dist
qtukey(p, nmeans, df, lower.tail)  # quantile in Tukey dist
pairwise.t.test(x, g,              # t-tests with adjusted p-values
                p.adjust.method,
                pool.sd, alternative)

# Contrasts
library(gmodels)             # for the fit.contrast function
fit.contrast(model, varname, # automatic linear contrasts :)
             coef, conf.int)

# Applying and Tapplying
lapply(X, FUN, ...)         # apply to each (list) element
```

Chapter 12

Correlation and Simple Linear Regression

Arthur Nudge: *Eh? know what I mean? Know what I mean? Nudge, nudge! Know what I mean? Say no more! A nod's as good as a wink to a blind bat, say no more, say no more!*
Man: *Look, are you insinuating something?*
Arthur Nudge: *Oh, no no no no... yes.*
Man: *Well?*
Arthur Nudge: *Well, you're a man of the world, squire... you've been there, you've been around.*
Man: *What do you mean?*
Arthur Nudge: *Well, I mean, you've done it... you've slept... with a lady...*
Man: *Yes...*
Arthur Nudge: *What's it like?*
—Monty Python's Flying Circus - Episode 3

Most people think "Statistical Analysis" includes several sub-parts: (1) modeling data, (2) point estimation, (3) interval estimation, (4) testing hypotheses, (5) making decisions, and (6) asymptotic inference (i.e., what happens when $N \rightarrow \infty$). Statisticians care about all six areas, but social scientists (including psychologists) care primarily about modeling complex phenomena, and regression analysis initiates the building of models. Because of its place at the forefront of data analysis, `R` has nearly unlimited resources for handling regression problems. The plethora of material/resources makes running a regression in `R` annoying (how does one sort through all of the functions to pick the most relevant one) but convenient (most of the time, the needed function will be "out there" somewhere).

12.1 Pearson Correlations

12.1.1 Descriptive Statistics and Graphing

Most datasets are of the appropriate form for correlational analyses. For example, using the following commands, load the BDI (Beck's Depression Inventory), BFI (Big Five Inventory), Anx (Anxiety) dataset into R.

```
> link <- "http://personality-project.org/r/datasets/maps.mixx.epi.bfi.data"
> dat <- read.table(url(link), header = TRUE)
> head(dat)
  epiE epiS epiImp epilie epiNeur bfagree bfcon bfext
1   18   10      7      3       9     138    96   141
2   16    8      5      1      12     101    99   107
3    6    1      3      2       5     143   118    38
4   12    6      4      3      15     104   106    64
5   14    6      5      3       2     115   102   103
6    6    4      2      5      15     110   113    61
  bfneur bfopen bdi traitanx stateanx
1     51    138   1       24       22
2    116    132   7       41       40
3     68     90   4       37       44
4    114    101   8       54       40
5     86    118   8       39       67
6     54    149   5       51       38
```

Using the BDI-BFI-Anx dataset, we can assess the relationship between many pairs of variables. For our purposes, we will choose `bfneur` as our predictor (x) variable and `traitanx` as our criterion (y) variable. Even though neuroticism as predicting anxiety is not a particularly interesting question, we do have an intuitive idea of how our results should appear.

```
> dat2 <- data.frame(bfneur = dat$bfneur, traitanx = dat$traitanx)
> x    <- dat2$bfneur     # our x-variable
> y    <- dat2$traitanx   # our y-variable
```

As shown in an earlier chapter, calculating a Pearson correlation on two vectors (or even on a `data.frame`) is straightforward in R by using the `cor` function. In general, the `cor` function either takes a matrix/data frame and then calculates a "correlation matrix" of all pairs of variables or takes two vectors and calculates a single correlation.

```
> cor(dat2)  # correlation of a matrix    --> returns a matrix
            bfneur  traitanx
bfneur   1.0000000 0.5930101
traitanx 0.5930101 1.0000000
> cor(x, y)  # correlation of two vectors --> returns a scalar
[1] 0.5930101
```

We can also calculate the correlation directly with simple R functions. The `cov` function works similarly to the `cor` function but returns individual covariances rather than correlations.

```
> cov.xy <- cov(x, y) # covariance of two vectors
> sd.x   <- sd(x)     # standard deviation of one variable
> sd.y   <- sd(y)     # standard deviation of the other variable
> cov.xy/(sd.x*sd.y)  # correlation between x and y
[1] 0.5930101
```

Or we can find the correlations directly by using the correlation formula.

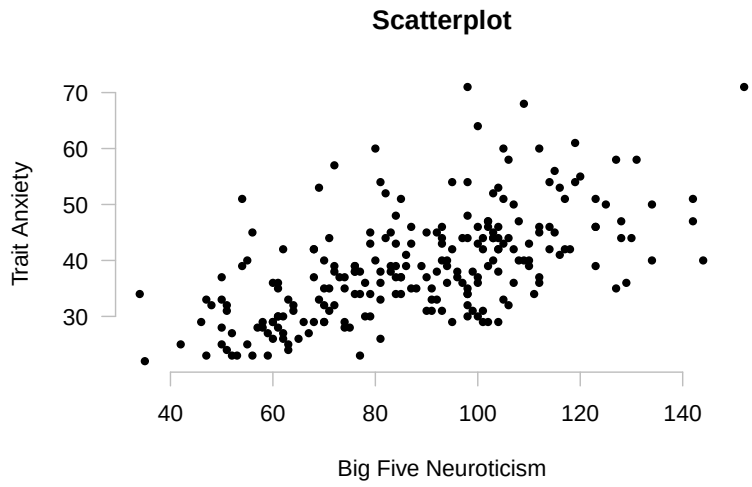
$$r_{xy} = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sqrt{\sum(x - \bar{x})^2 \sum(y - \bar{y})^2}}$$

```
> SS.xy <- sum( (x - mean(x)) * (y - mean(y)) ) # sum of cross-products
> SS.x  <- sum( (x - mean(x))^2 )               # sum of squares 1
> SS.y  <- sum( (y - mean(y))^2 )               # sum of squares 2
> SS.xy/(sqrt(SS.x*SS.y))                       # correlation again
[1] 0.5930101
```

In any case, the correlation indicates that there is a moderately strong, positive, linear relationship between neuroticism and trait anxiety as measured by these questionnaires.

We should next assess whether the correlation is a reasonable descriptive tool for these data by constructing a bivariate scatterplot. Constructing a scatterplot in `R` is a straightforward (and relatively intuitive) extension of the `plot` function.

```
> plot(x = x, y = y,
       xlab = "Big Five Neuroticism",
       ylab = "Trait Anxiety",
       main = "Scatterplot",
       pch = 20, axes = FALSE)
> axis(1, col = "grey")
> axis(2, col = "grey", las = 1)
```



Based on the scatterplot, the relationship does appear to be linear, but you might notice both some range restriction and the low end of the trait anxiety scale and some minor heteroscedasticity.

12.1.2 Hypothesis Testing on Correlations

Many researchers frequently desire to test whether a correlation is significantly different from zero. Testing $H_0 : \rho = 0$ is often not advisable due to the test statistic being solely a function of sample size, but the process of simple hypothesis testing is still rather straightforward by using the `cor.test` function. The input (and output) of `cor.test` is nearly identical to that from the `t.test` function.

```
> ( mod.cor <- cor.test(x, y,
                        alternative = "two.sided",
                        conf.level = .95) )
        Pearson's product-moment correlation

data:  x and y
t = 11.145, df = 229, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.5023874 0.6707460
sample estimates:
      cor
0.5930101
```

Not surprisingly, the p-value associated with the correlation is very small, and we would reject H_0 at any conventional α level.

We could also determine whether a correlation is significant by plugging our data into

$$t_{N-2} = \frac{r}{\sqrt{\frac{1-r^2}{N-2}}} = r\sqrt{\frac{N-2}{1-r^2}}$$

where N is the number of pairs of observations.

```
> r <- cor(x, y)      # the correlation between x and y
> N <- length(x)      # the number of pairs of observations
> ( t <- r * sqrt( (N - 2)/(1 - r^2) ) )
[1] 11.14497
> ( p <- 2*pt(abs(t), df = N - 2, lower.tail = FALSE) )
[1] 2.491328e-23
> # Is the t-statistic identical to before?
> mod.cor$statistic
       t
11.14497
```

12.1.3 Confidence Intervals

Constructing confidence intervals for population correlations is (perhaps) more useful but a bit tricker. The sampling distribution of r is only approximately normally distributed when $\rho = 0$. For all other population correlations, the sampling distribution of r is skewed toward 0. The $r \rightarrow z'$ transformation was proposed by Fisher as a variance stabilizing transformation to approximate normality, so that (approximate) normal theory would apply to the transformed value. Therefore, constructing a confidence interval for ρ requires the following three steps.

1. Calculate $r \rightarrow z'$ using: $z' = \tanh^{-1}(r) = .5 \ln\left(\frac{1+r}{1-r}\right)$.
2. Form CIs on z' using normal theory: $CI_{z'} = z' \pm z_{(1+\gamma)/2}\sqrt{\frac{1}{N-3}}$.
3. Calculate $z' \rightarrow r$ for CI bounds on ρ: $CI_r = \tanh(CI_{z'}) = \frac{\exp(2CI_{z'})-1}{\exp(2CI_{z'})+1}$

Useful functions in calculating the correlation confidence interval include `tanh` and `atanh`. Note that we do not even need to know the exact formula for the $r \rightarrow z' \rightarrow r$ transformation, as long as we know that the `tanh` and `atanh` functions are doing the right thing. Go trig!

```
> # Pick the confidence level:
> gamma <- .95
> ## 1. ## Go from r to z:
> z     <- atanh(r)
> ## 2. ## Build a confidence interval around z:
> CI.z  <- z + c(-1, 1)*qnorm( (1 + gamma)/2 )*sqrt(1/(N - 3))
> ## 3. ## Back transform the endpoints of the CI to r:
> CI.r  <- tanh(CI.z)
```

And the formula-based confidence interval should be identical to the one constructed using the `cor.test` function.

```
> CI.r             # formula based
[1] 0.5023874 0.6707460
> mod.cor$conf.int # cor.test construction
[1] 0.5023874 0.6707460
attr(,"conf.level")
[1] 0.95
```

Other uses of the $r \rightarrow z'$ transformation are to test non-zero null hypotheses for ρ, and build confidence intervals/test the difference between pairs of correlations.

12.2 Alternative Correlations

Unfortunately, Pearson correlations only adequately describe the underlying relationship between two variables if those variables are linearly related. What should one do if they have variable that (by design or due to data collection) have non-linear relationships? One solution would be to add non-linear terms to a prediction equation and

calculate the correlation between y_i and predicted y_i. This method is for a different book at a different time. Another solution is to develop theory appropriate for measuring non-linear relationships. Common non-oval-shaped variables include those based on ranks and those based on discretizing continuous variables.

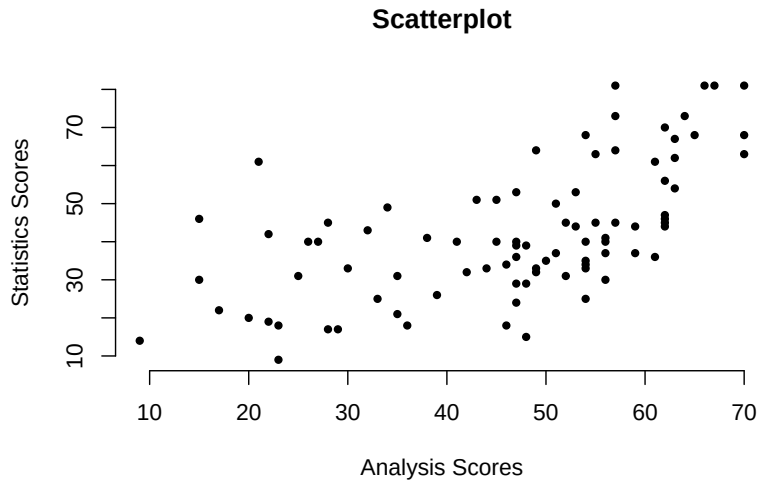
12.2.1 Rank-Order Correlations

Common rank-order correlations include Spearman's ρ and Kendall's τ. Both correlations (in different ways) circumvent problems in interpreting Pearson correlations on ordinal level data or data with outliers. For this section, we will use the `marks` dataset in the `ggm` package.

```
> data(marks, package = "ggm")
> head(marks)
  mechanics vectors algebra analysis statistics
1        77      82      67       67         81
2        63      78      80       70         81
3        75      73      71       66         81
4        55      72      63       70         68
5        63      63      65       70         63
6        53      61      72       64         73
```

The `marks` dataset includes $N = 88$ students' grades on each of $J = 5$ academic exams. When plotting `analysis` against `statistics` using the `plot` function of the previous section, we find fairly skewed variables with additional, problematic outliers.

```
> plot(x = marks$analysis, y = marks$statistics,
       xlab = "Analysis Scores",
       ylab = "Statistics Scores",
       main = "Scatterplot",
       pch = 20, axes = FALSE)
> axis(1)
> axis(2)
```



One can alleviate problems with skew and outliers by calculating either Spearman's ρ or Kendall's τ. Spearman's ρ is simply a correlation on the ranks of the variables. Therefore, you can calculate Spearman's ρ simply by following these steps:

1. Rank order the data, making sure to assign ties the average of the corresponding ranks.
2. Compute the Pearson correlation on the ranks.

The end. That's it! And because multiplying both variables by -1 will not change the correlation, you can rank from highest = 1 to lowest = N **or** from highest = N to lowest = 1. It does not matter! So, we must first rank-order both variables. The easiest function that rank-orders our data is the `rank` function. For the standard Spearman's ρ, we require the rank of ties to be the average of the corresponding ranks, so we should set `ties.method` to "`average`".

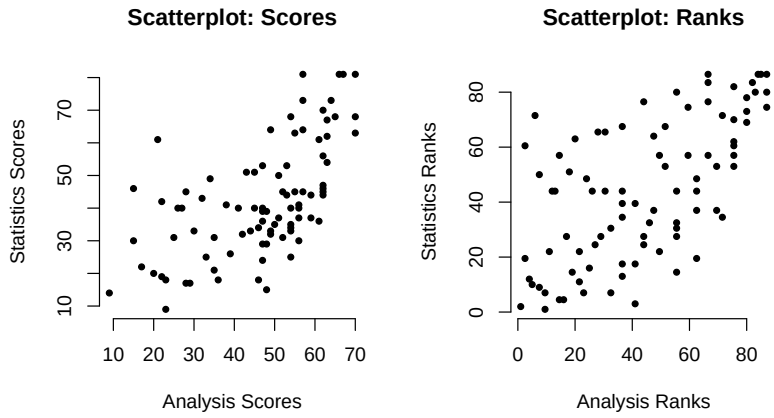
```
> head(anal.r <- rank(marks$analysis, ties.method = "average"))
[1] 85 87 84 87 87 82
> head(stat.r <- rank(marks$statistics, ties.method = "average"))
[1] 86.5 86.5 86.5 80.0 74.5 83.5
```

Notice that `rank` sets the minimum score equal to 1 (unless there are ties) and the maximum score equal to N (unless there are ties).

```
> (anal.m <- which.min(marks$analysis))  # where is the minimum score?
[1] 88
> anal.r[anal.m]                         # rank of minimum score?
[1] 1
```

Also notice that ranking data results in a quadrilateral shape (as opposed to the elliptical shape typical of interval data).

```
> par(mfrow = c(1, 2))
> plot(x = marks$analysis, y = marks$statistics,
       xlab = "Analysis Scores",
       ylab = "Statistics Scores",
       main = "Scatterplot: Scores",
       pch = 20, axes = FALSE)
> axis(1)
> axis(2)
> plot(x = anal.r, y = stat.r,
       xlab = "Analysis Ranks",
       ylab = "Statistics Ranks",
       main = "Scatterplot: Ranks",
       pch = 20, axes = FALSE)
> axis(1)
> axis(2)
> par(mfrow = c(1, 1))
```



After calculating the ranks on the data, you only need to correlate those ranks to obtain Spearman's ρ.

```
> (cor.rho1 <- cor(anal.r, stat.r))
[1] 0.6280038
```

An additional method of finding Spearman's ρ is by use the following formula:

$$\rho = 1 - \frac{6 \sum d^2}{N(N^2 - 1)}$$

Because ρ is simply a Pearson correlation on the ranks, this formula is obtained via simplification. The d stands for the difference between the ranks of each observation. Unfortunately, this computation is not exact with ties in the data. So without ties:

```
> v1 <- c(1, 2, 5, 3, 7, 8)
> v2 <- c(1, 3, 4, 9, 2, 6)
> # Method 1: Correlation on Ranks
> cor(rank(v1), rank(v2))
```

```
[1] 0.4285714
> # Method 2: Using Formula
> d <- rank(v1) - rank(v2) # differences in ranks
> N <- length(d)           # length of differences
> 1 - 6*sum(d^2)/(N*(N^2 - 1))
[1] 0.4285714
```

But in our case, if we try to take `anal.r`, subtract `stat.r`, square the resulting values, and add up all of those squared terms, we get something slightly different from our original correlation.

```
> # Method 1: Correlation on Ranks
> cor.rho1
[1] 0.6280038
> # Method 2: Using Formula
> head(d <- anal.r - stat.r) # difference in ranks
[1] -1.5  0.5 -2.5  7.0 12.5 -1.5
> (N <- length(d))           # length of differences
[1] 88
> (cor.rho2 <- 1 - (6*sum(d^2))/(N*(N^2 - 1)))
[1] 0.6284738
>
> # The formulas are not the same?  Weird!
```

And if you are in doubt as to whether the first or second method is more appropriate, you can set `method` equal to "`spearman`" in the `cor` function.

```
> (cor.rho3 <- cor(marks$analysis, marks$statistics,
                   method = "spearman"))
[1] 0.6280038
```

Notice that the correlation computed by the `cor` function is equivalent to the Pearson correlation on the ranks and not the computational formula.

An alternative to Spearman's ρ in the case of ordinal data is Kendall's τ. Unlike Spearman's ρ, Kendall's τ has a defined standard error with a known sampling distribution. Therefore, hypotheses tests can be performed on Kendall's τ without simulations. With all unique scores, then Kendall's τ can be written

$$\tau = \frac{\#\{\text{concordant pairs}\} - \#\{\text{discordant pairs}\}}{N(N-1)/2},$$

where a concordant pair means that if $x_i > x_j$ then $y_i > y_j$ *or if* $x_i < x_j$ then $y_i < y_j$; a discordant pair means that if $x_i > x_j$ then $y_i < y_j$ *or if* $x_i < x_j$ then $y_i > y_j$; and N is the number of pairs. Note that all possible $\binom{N}{2} = \frac{N(N-1)}{2}$ pairs of (x_i, y_i) and (x_j, y_j) are compared, which is why the denominator is $\frac{N(N-1)}{2}$. An equivalent (and slightly simpler formula) for Kendall's τ (without pairs) is

$$\tau = \frac{\sum_{j>i} \sum_{i=1}^{N-1} \text{sgn}(x_i - x_j)\text{sgn}(y_i - y_j)}{N(N-1)/2},$$

where sgn is the sign function and represents 1 if $x_i - x_j > 0$, -1 if $x_i - x_j < 0$, and 0 if $x_i - x_j = 0$. For a dataset with no ties, this τ formula is correct and fairly straightforward (although annoying) to compute in R (using the `sign` function to calculate the differences).

```
> tau.n <- 0
> x  <- v1
> y  <- v2
> N  <- length(x)
> for(i in 1:(N - 1)){
  for(j in i:N){

    tau.n <- tau.n + sign(x[i] - x[j])*sign(y[i] - y[j])

  }} # END ij LOOPS
```

The sum of sign products is 5 and equivalent to the difference between the number of concordant pairs minus the number of discordant pairs. We can use `tau.n` to easily calculate Kendall's τ.

```
> tau.n/(N*(N - 1)/2)
[1] 0.3333333
```

Not surprisingly, you can also calculate Kendall's τ using the `cor` function by changing `method` to `"kendall"`. And if you were to check the hand calculations, you would find that they match up pretty well.

```
> tau.n/(N*(N - 1)/2)           # calculated by hand
[1] 0.3333333
> cor(x, y, method = "kendall") # calculated in R
[1] 0.3333333
```

Unfortunately, if you were to repeat the Kendall's τ calculation using the `marks` dataset (which has ties), then the two results would be different.

```
> # Method 1: By Hand
> tau.n <- 0
> x  <- marks$analysis
> y  <- marks$statistics
> N  <- length(x)
> for(i in 1:(N - 1)){
  for(j in i:N){

    tau.n <- tau.n + sign(x[i] - x[j])*sign(y[i] - y[j])

  }} # END ij LOOPS
> (tau1 <- tau.n/(N*(N - 1)/2))
[1] 0.4568966
> # Method 2: Using R Formula
> (tau2 <- cor(x, y, method = "kendall"))
```

```
[1] 0.4664624
>
> # Close ... but different!
```

The `cor` function uses a much more complicated method to calculate τ with ties in the data.

12.2.2 Approximate/Categorical Correlations

When trying to compute correlations on categorical variables, the appropriate formula depends on the existence of hypothetical, underlying, normally distributed variables. To help see what some of these correlations are doing, I will first generate a bivariate, normally distributed variable, then I will discretize it, and finally, I will compute correlations on the discretized variable. The function that I will use to generate multivariate normally distributed variable is `mvrnorm` (multivariate random normal) in the `MASS` package.

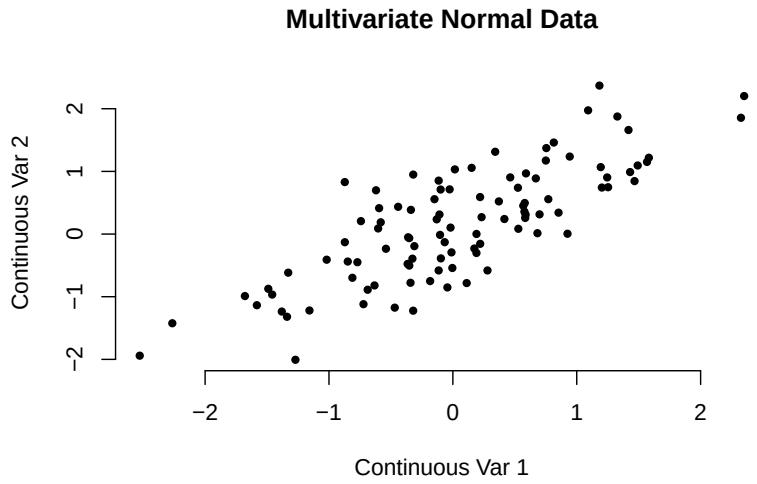
```
> set.seed(8912)
> library(MASS)
> X <- mvrnorm(n = 100, mu = c(0, 0),
               Sigma = matrix(c(1, .8, .8, 1), nrow = 2))
> cont.v1 <- X[ , 1]
> cont.v2 <- X[ , 2]
```

`cont.v1` and `cont.v2` are correlated approximately .8 as per the off-diagonals of the `Sigma` matrix.

```
> cor(cont.v1, cont.v2)
[1] 0.7965685
```

And if you plot `cont.v1` against `cont.v2`, the variables are pretty elliptical-like.

```
> plot(x = cont.v1, y = cont.v2,
       xlab = "Continuous Var 1",
       ylab = "Continuous Var 2",
       main = "Multivariate Normal Data",
       pch = 20, axes = FALSE)
> axis(1)
> axis(2)
```



Next, I will discretize the first variable into three parts.

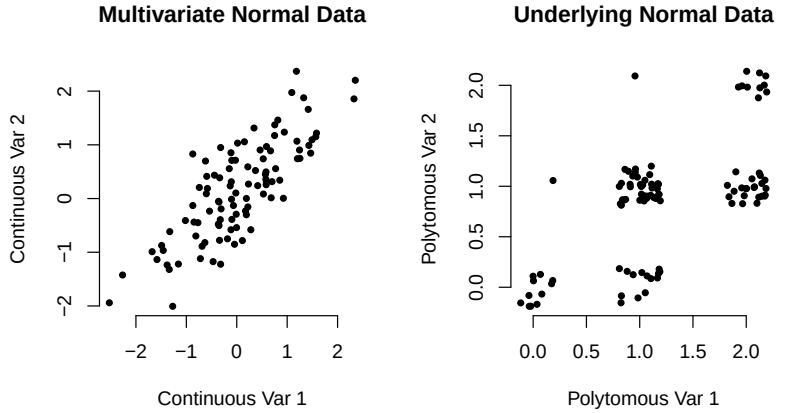
```
> # If V1 < -1 --> POLY = 0
> # If V1 > .5 --> POLY = 2
> # If V1 Between -1 and .5 --> POLY = 1
> poly.v1 <- rep(0, length(cont.v1))
> poly.v1[(cont.v1 >= .5)] <- 2
> poly.v1[(cont.v1 >= -1) & (cont.v1 < .5)] <- 1
> # What does poly.v1 look like?
> table(poly.v1)
poly.v1
 0  1  2
12 55 33
```

And I will discretize the second variable into three parts.

```
> # If V1 < -.5 --> POLY = 0
> # If V1 > 1.5 --> POLY = 2
> # If V1 Between -.5 and 1.5 --> POLY = 1
> poly.v2 <- rep(0, length(cont.v2))
> poly.v2[(cont.v2 >= 1.2)] <- 2
> poly.v2[(cont.v2 >= -.5) & (cont.v2 < 1.2)] <- 1
> # What does poly.v2 look like?
> table(poly.v2)
poly.v2
 0  1  2
25 64 11
>
> # How comparable are the cutpoints for poly.v1/poly.v2?
```

And when plotting `poly.v1` against `poly.v2`, things are a little chunkier. Note that the `jitter` function is useful in plotting scores with lots of repetitions, as it randomly shakes the scores a little so that you can see how many scores are at a point.

```
> par(mfrow = c(1, 2))
> plot(x = cont.v1, y = cont.v2,
       xlab = "Continuous Var 1",
       ylab = "Continuous Var 2",
       main = "Multivariate Normal Data",
       pch = 20, axes = FALSE)
> axis(1)
> axis(2)
> plot(x = jitter(poly.v1), y = jitter(poly.v2),
       xlab = "Polytomous Var 1",
       ylab = "Polytomous Var 2",
       main = "Underlying Normal Data",
       pch = 20, axes = FALSE)
> axis(1)
> axis(2)
> par(mfrow = c(1, 1))
```



As evident by the scatterplot, discretizing continuous variables results in decreased correlations.

```
> cor(cont.v1, cont.v2) # original variables
[1] 0.7965685
> cor(poly.v1, poly.v2) # polytomous cuts
[1] 0.6440504
```

I had already discretized `cont.v1` and `cont.v2` into polytomous variable with three categories. Next, I will discretize the first continuous variables into two, dichotomous parts.

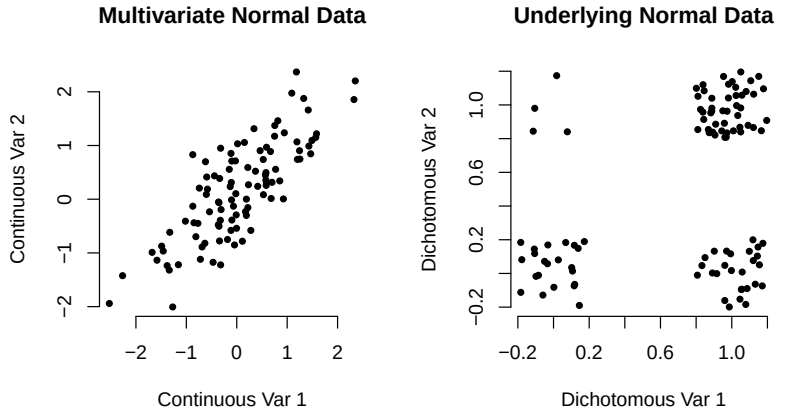
```
> # If V1 < -.5 --> DICH = 0
> # If V1 > -.5 --> DICH = 1
> dich.v1 <- rep(0, length(cont.v1))
> dich.v1[(cont.v1 >= -.5)] <- 1
> # What does dich.v1 look like?
> table(dich.v1)
dich.v1
 0  1
26 74
```

And I will discretize the second continuous variable into two parts.

```
> # If V1 < .2 --> DICH = 0
> # If V1 > .2 --> DICH = 1
> dich.v2 <- rep(0, length(cont.v2))
> dich.v2[(cont.v2 >= .2)] <- 1
> # What does dich.v2 look like?
> table(dich.v2)
dich.v2
 0  1
49 51
```

And when plotting `dich.v1` against `dich.v2`, the relationships are even chunkier.

```
> par(mfrow = c(1, 2))
> plot(x = cont.v1, y = cont.v2,
       xlab = "Continuous Var 1",
       ylab = "Continuous Var 2",
       main = "Multivariate Normal Data",
       pch = 20, axes = FALSE)
> axis(1)
> axis(2)
> plot(x = jitter(dich.v1), y = jitter(dich.v2),
       xlab = "Dichotomous Var 1",
       ylab = "Dichotomous Var 2",
       main = "Underlying Normal Data",
       pch = 20, axes = FALSE)
> axis(1)
> axis(2)
> par(mfrow = c(1, 1))
```



Notice how fewer cut-points results in lower correlations.

```
> cor(cont.v1, cont.v2) # original variables
[1] 0.7965685
> cor(poly.v1, poly.v2) # polytomous cuts
[1] 0.6440504
> cor(dich.v1, dich.v2) # dichotomous cuts
[1] 0.4223041
```

What do you do if you want to estimate the correlation among the underlying, normally distributed variables that have been butchered into discrete chunks? As we decrease the number of possible scores (i.e., decrease the number of cuts), the correlation drops. One proposed solution is to calculate tetrachoric (with a 2×2 table) or polychoric (with an any size table) correlations to approximate the original relationship. A tetrachoric correlation is an estimate of the correlation between two continuous variables underlying a 2×2 contingency table assuming that those continuous variables are bivariate normally distributed. The tetrachoric correlation is basically trying to go in the reverse direction of our discretizing step and recover the original relationship. You can calculate tetrachoric and polychoric correlations using the `polychor` function in the `polycor` package.

```
> library(polycor)
```

The `polychor` function takes four arguments:

1. `x`: Either a contingency table or an ordered, categorical variable.
2. `y`: An ordered categorical variable (if `x` is also an ordered categorical variable) or NULL.
3. `ML`: A logical letting `R` know whether to compute the maximum likelihood (best?!) estimate of the correlation or to quickly approximate that estimate.
4. `std.err`: A logical telling `R` whether or not to also include diagnostic information along with the correlation.

I always set `ML` to `TRUE` and `std.err` to `TRUE` so that I can perform follow-up tests on the significance of the correlations. We will also generally compute these correlation using a contingency table of counts based on the discretized variables.

```
> dich.tab <- table(dich.v1, dich.v2)
> poly.tab <- table(poly.v1, poly.v2)
> dich.r <- polychor(x = dich.tab, ML = TRUE, std.err = TRUE)
> poly.r <- polychor(x = poly.tab, ML = TRUE, std.err = TRUE)
```

The `polychor` object contains several sub-objects, including: `rho` (the actual correlation), `var` (the variance matrix between the correlation and the cuts), `row.cuts` (the estimated cutpoints on the row variable), and `col.cuts` (the estimated cutpoints on the column variable). Notice that both correlations are closer to the original correlation between the continuous variables than the correlations using dichotomized variables.

```
> dich.r$rho
0.6783541
> poly.r$rho
0.8530738
> cor(cont.v1, cont.v2)
[1] 0.7965685
```

And the cutpoints are also reasonably estimated.

```
> dich.r$row.cuts
         0
-0.6433454
> dich.r$col.cuts
          0
-0.02506891
> poly.r$row.cuts
         0          1
-1.1856482  0.4512085
> poly.r$col.cuts
         0          1
-0.6740325  1.2129506
```

In the final section of this chapter, I will discuss how to model linear relationships in `R`. The models discussed assume that the Pearson correlation adequately describes the linear relationship. Therefore, I will use the dataset from the previous section and not any of the datasets discussed in this section.

12.3 Simple Linear Regression

12.3.1 The lm Function

After determining that a linear relationship exists between two variables, one might want to build a linear prediction equation using OLS regression. The easiest method of regressing a criterion on one or more predictors is by using the `lm` function (which stands for "linear models.").

lm(formula, data)

In the above function, `formula` is of the form `y ~ x`, and `data` is a `data.frame` in which the `x` and `y` vectors are located. As in One-Way ANOVA, if you input into `lm` vectors outside of a data frame, you do not need to supply a value to the `data` argument. Because we have trait anxiety and big five neuroticism both inside and outside a data frame, the following function calls will result in identical output.

```
> ( mod.lm1 <- lm(traitanx ~ bfneur, data = dat2) )
Call:
lm(formula = traitanx ~ bfneur, data = dat2)

Coefficients:
(Intercept)       bfneur
     17.724        0.242
> ( mod.lm2 <- lm(dat2$traitanx ~ dat2$bfneur) )
Call:
lm(formula = dat2$traitanx ~ dat2$bfneur)

Coefficients:
(Intercept)  dat2$bfneur
     17.724        0.242
> x <- dat2$bfneur
> y <- dat2$traitanx
> ( mod.lm3 <- lm(y ~ x) )
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
     17.724        0.242
```

Alternatively, we could also calculate the intercept and slope by using standard `R` functions.

```
> b1 <- cov(x, y)/var(x)     # formula for slope
> b0 <- mean(y) - mean(x)*b1 # formula for intercept
```

Not surprisingly, the hand calculations result in the same output as when estimating the slope and intercept by using the `lm` function.

```
> coef(mod.lm1)  # the "coefficients" (intercept and slope)
(Intercept)      bfneur
 17.7235831   0.2419592
> b0             # our intercept (calculated)
[1] 17.72358
> b1             # our slope (calculated)
```

```
[1] 0.2419592
```

After saving an `"lm"` object (by using the `lm` function), the `summary` (or `summary.lm`) function meaningfully summarizes the linear model.

```
> summary(mod.lm1)
Call:
lm(formula = traitanx ~ bfneur, data = dat2)

Residuals:
    Min      1Q  Median      3Q     Max
-13.887  -4.959  -1.082   3.678  29.564

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 17.72358    1.97561   8.971   <2e-16 ***
bfneur       0.24196    0.02171  11.145   <2e-16 ***
---
Signif. codes:
0
```

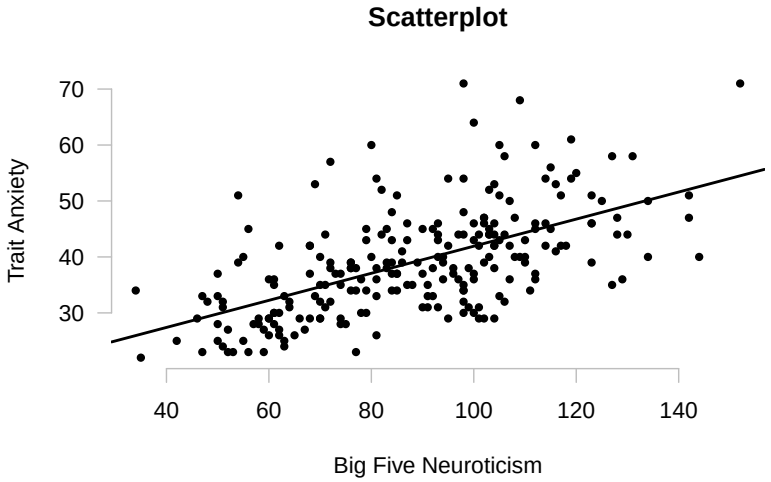
With `lm` objects, the `summary` function contains the model-building formula, the five number summary of the residuals (where $e_i = y_i - \hat{y}_i$), the coefficients, the standard errors of the coefficients, the t-statistics (and corresponding p-values) when testing significance of the coefficient, etc. To extract values from the summary object, we can save the summary into its own object (with a `"summary.lm"` class) and extract individual parts of the summary with the dollar sign operator.

```
> sum.lm1 <- summary(mod.lm1) # saving the summary
> names(sum.lm1)              # what is inside the summary?
 [1] "call"          "terms"         "residuals"
 [4] "coefficients"  "aliased"       "sigma"
 [7] "df"            "r.squared"     "adj.r.squared"
[10] "fstatistic"    "cov.unscaled"
> sum.lm1$coefficients        # our test statistics
             Estimate Std. Error   t value     Pr(>|t|)
(Intercept) 17.7235831 1.97561242  8.971184 1.076979e-16
bfneur       0.2419592 0.02171017 11.144967 2.491328e-23
> sum.lm1$sigma               # our residual standard error
[1] 7.683463
```

We can also plot the regression line on top of the (already constructed) scatterplot by using the `abline` function.

```
> # The original scatterplot:
> plot(x = x, y = y,
       xlab = "Big Five Neuroticism",
       ylab = "Trait Anxiety",
```

```
       main = "Scatterplot",
       pch = 20, axes = FALSE)
> axis(1, col = "grey")
> axis(2, col = "grey", las = 1)
> # Our regression line on top of the plot:
> abline(mod.lm1, lwd = 2)
```

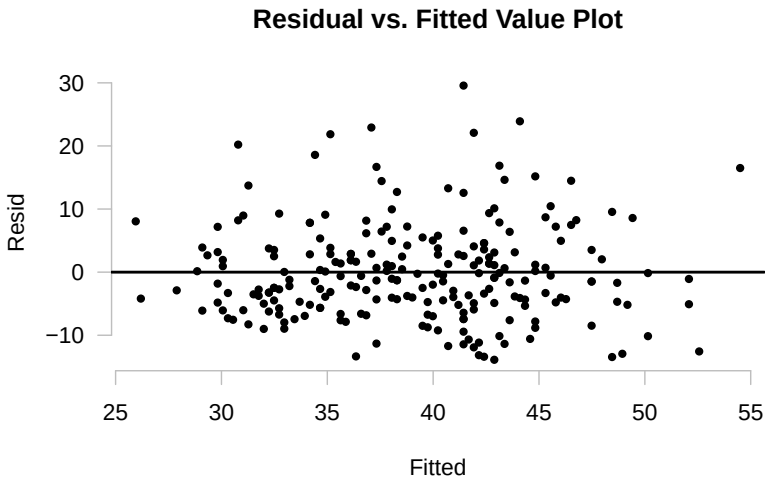


The `abline` function plots a vertical, horizontal, or slanted line across an entire (already existing) plotting surface.

12.3.2 Checking Regression Assumptions

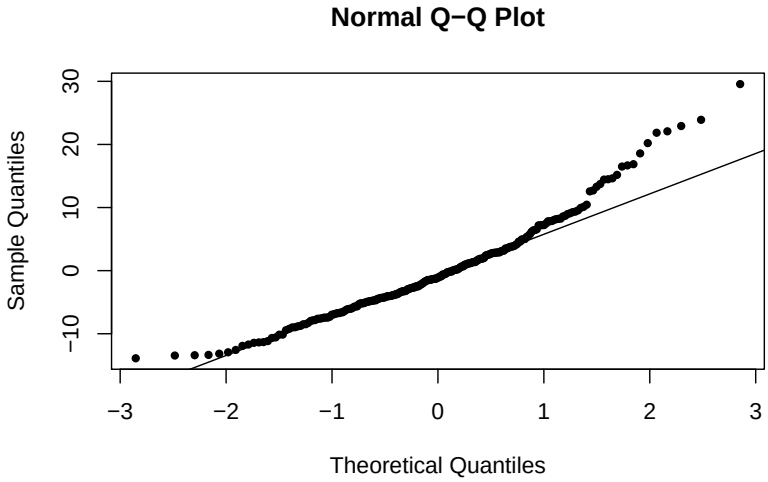
After fitting and plotting a regression function, we should check to make sure the simple linear regression assumptions are upheld. We can check for heteroscedasticity by pulling the residuals out of the `"lm"` object and plotting those residuals agains the fitted values.

```
> # Pulling out the residuals and fitted values:
> resid.lm  <- residuals(mod.lm1)  # the residuals
> fitted.lm <- fitted(mod.lm1)     # the fitted values
> # Plotting the fitted values/residuals on the x/y axes:
> plot(x = fitted.lm, y = resid.lm,
       xlab = "Fitted", ylab = "Resid",
       main = "Residual vs. Fitted Value Plot",
       pch = 20, axes = FALSE)
> axis(1, col = "grey")
> axis(2, col = "grey", las = 1)
> abline(h = 0, lwd = 2)
```



Based on the plot, the variability around the regression line appears constant for all values of $\hat{y}$. However, there might be a bit of non-normality (skew) due to the spread of the residuals above the regression line being larger than the spread of the residuals below the regression line. A better check for skew (and other non-normalities of the residuals) is via constructing a `qqnorm` plot of the residuals.

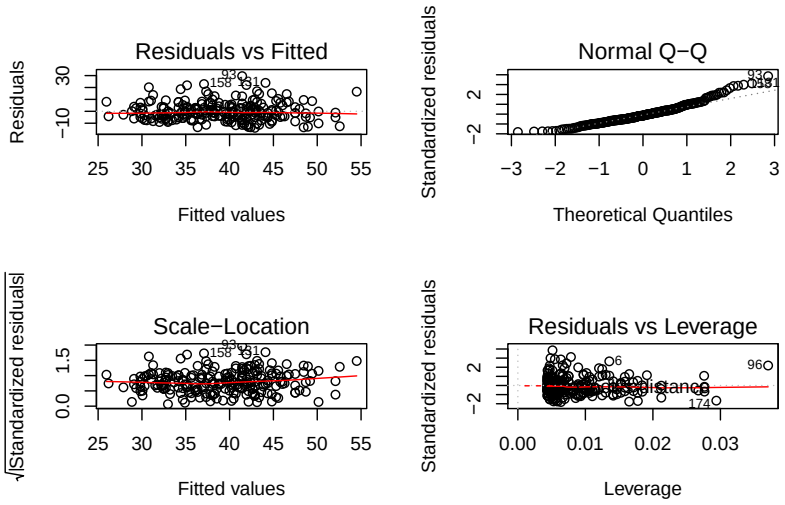
```
> qqnorm(resid.lm, pch = 20)  # qqnorm plot of the residuals
> qqline(resid.lm)            # qqline for the qqnorm plot
>
> # Curvature above the line?  Positive skew!
```



Given the evidence of positive skew, the assumption of residual normality does not appear to hold. However, regression analysis is usually robust to mild violations of normality.

The residual versus fitted plot and `qqnorm` plot can also be constructed automatically from the `"lm"` object.

```
> par(mfrow = c(2, 2))  # four plotting surfaces
> plot(mod.lm1)
> par(mfrow = c(1, 1))
```



Because the `plot` function is class specific, plugging different things into `plot` will result in different plotted output: (1) two vectors will yield a scatterplot; (2) a regression object yields diagnostic plots. And you will find many variations of the `plot` function, most of which will only appear when inserting a specific `R` object.

12.3.3 Hypothesis Testing on Regression Coefficients

As in correlation analyses, we can test the significance of the slope and intercept from a regression model. As a reminder, the coefficients of our linear model (predicting trait anxiety from neuroticism) are as follows.

```
> coef(mod.lm1)
(Intercept)      bfneur
 17.7235831   0.2419592
```

The model coefficients can be used to (of course) build a prediction equation.

$$E(Y|X) = \hat{y}_i = 17.724 + 0.242x_i$$

Therefore, a score of 0 on neuroticism is predicted to result in a score of 17.724 on the trait anxiety scale, and each additional point of neuroticism results in a predicted increase of 0.242 trait anxiety units. The `coefficients` sub-object inside of `mod.lm1` indicates that both the slope and intercept are significantly different from 0 at $\alpha = .05$.

```
> sum.lm1$coefficients  # find the slope/intercept p-value!
              Estimate Std. Error   t value     Pr(>|t|)
(Intercept) 17.7235831 1.97561242  8.971184 1.076979e-16
bfneur       0.2419592 0.02171017 11.144967 2.491328e-23
```

But we could also test the significance of a regression slope by using the appropriate formula.

$$t_{N-2} = \frac{\hat{\beta}_1}{\sqrt{\text{Var}(\hat{\beta}_1)}}$$

where

$$\widehat{\text{Var}}(\hat{\beta}_1) = \frac{\hat{\sigma}_\epsilon^2}{s_x^2(N-1)}$$

is the estimated standard error of the simple linear regression slope. In R, the procedure for testing the significance of a regression slope is as follows.

```
> N       <- length(resid.lm)
> ( var.eps <- sum( resid.lm^2 )/(N - 2) )
[1] 59.0356
> ( var.b1  <- var.eps/(var(x)*(N - 1)) )    # from above
[1] 0.0004713317
> ( t.b1    <- b1/sqrt(var.b1) )             # a typical t-test
[1] 11.14497
> ( p.b1   <- 2*pt(abs(t.b1), df = N - 2, lower.tail = FALSE) )
[1] 2.491328e-23
```

Notice that the t-statistic for testing the significance of a slope is identical to the t-statistic when testing the significance of a correlation coefficient. For simple linear regression, the slope/correlation are isomorphic assuming fixed variances.

We can also use the (estimated) standard error (above) to calculate confidence intervals for the true population slope. And not surprisingly, R contains a function to compute those confidence intervals automatically.

```
> gamma  <- .95
> ## 1 ## The easy method (using a build-in function):
> confint(mod.lm1, level = gamma)
                 2.5 %     97.5 %
(Intercept) 13.8308813 21.6162850
bfneur       0.1991819  0.2847364
> ## 2 ## The much more difficult method (by hand):
> ( CI.slp <- { b1 +
                c(-1, 1)*qt( (1 + gamma)/2, df = N - 2 )*sqrt(var.b1) } )
[1] 0.1991819 0.2847364
```

12.3.4 Fitted and Predicted Intervals

One of the benefits to regression analysis is the ability to form a confidence interval for a future observation. The function `predict` takes a regression object, predicts new values given a data frame (or list) of those new values by name, and calculates confidence intervals for predicted or fitted values.

```
predict(mod.lm,
        newdata = list( ... ),
        interval = c("none", "confidence", "prediction"),
        level = .95)
```

For example, to predict trait anxiety from a neuroticism score of 120, we can use the following code.

```
> ( fit.int  <- predict(mod.lm1, newdata = list(bfneur = 120),
                        interval = "confidence") )
       fit     lwr      upr
1 46.75868 45.0647 48.45266
> ( pred.int <- predict(mod.lm1, newdata = list(bfneur = 120),
                        interval = "prediction") )
       fit      lwr      upr
1 46.75868 31.52489 61.99248
```

Note: When using `predict` in `R`, you must specify your new data as a list of vectors. Each vector in that list corresponds to an independent variable (of which we only have one), and the *name* of the vectors *MUST* be identical to the names of the predictor variables in your regression model. If the vector names are not the same, `R` will not know which variables you want to predict. For example, in `mod.lm2`, the name of the x-variable is `bfneur`; in `mod.lm2`, the name of the x-variable is `dat2$bfneur`; and in `mod.lm3`, the name of the x-variable is `x`. Therefore, the contents of `newdata` depends on which model one uses to find predictions.

One might wonder how to predict future observations and find confidence intervals for predictions without the use of the (convenient) `predict` function. Not surprisingly, finding predicted values is a simple application of the "multiply" and "divide" operators.

```
> x.i <- 120
> ( yhat.i <- b0 + b1*x.i )
[1] 46.75868
```

And finding confidence intervals requires knowing the standard error of a predicted value. To find the standard error of the predicted value, one must first know the standard error of fit,

$$\hat{\sigma}_{\text{fitted}} = \hat{\sigma}_\epsilon \sqrt{\frac{1}{N} + \frac{(x_i - \bar{x})^2}{s_x^2 (N-1)}}$$

where x_i is the point used to predict $\hat{y}_i$.

```
> # Variance of the fitted value:
> var.fit <- var.eps*(1/N + (x.i - mean(x))^2/(var(x)*(N - 1)))
> # Standard error of the fitted value is the square root of the variance:
> ( se.fit <- sqrt(var.fit) )
[1] 0.8597229
```

The standard error of fit is a modification of the standard error of the intercept by forcing x_i to be equal to 0 on the x-axis. The standard error of prediction is equal to the standard error of fit plus extra variability of residuals around the fitted value,

$$\hat{\sigma}_{\text{predict}} = \hat{\sigma}_\epsilon \sqrt{1 + \frac{1}{N} + \frac{(x_i - \bar{x})^2}{s_x^2(N-1)}},$$

where the 1 under the radical indicates the extra residual variability.

```
> # Variance of the predicted value:
> var.pred <- var.eps*(1 + 1/N + (x.i - mean(x))^2/(var(x)*(N - 1)))
> # Standard error of the predicted value:
> ( se.pred <- sqrt(var.pred) )
[1] 7.731412
```

After finding predicted values and standard errors, the process to construct confidence intervals is the same as always.

```
> gamma      <- .95
> ( fit.int2  <- { yhat.i +
                   c(-1, 1)*qt( (1 + gamma)/2, df = N - 2 )*se.fit } )
[1] 45.06470 48.45266
> ( pred.int2 <- { yhat.i +
                   c(-1, 1)*qt( (1 + gamma)/2, df = N - 2 )*se.pred } )
[1] 31.52489 61.99248
```

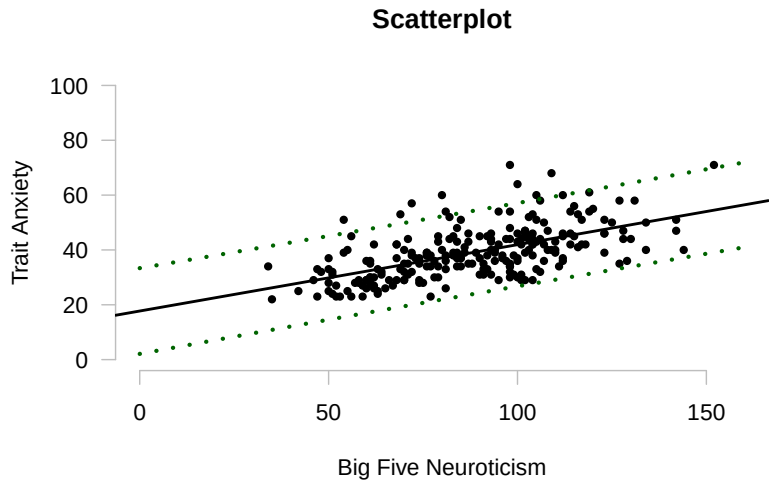
And the confidence intervals line up exactly with those calculated using the `predict` function.

> **Note**: A confidence interval for a fitted value is semantically different from a confidence interval for a predicted value. The CI for a fitted value is basically the confidence interval for the *regression line* at that point. The further away that the point is from the regression line, the less sure we are that the regression line describes the conditional mean. The CI for a predicted value is a confidence interval for the *point itself*. When we form a confidence interval for prediction, we are saying something about the actual y value and *not just* the regression line. Due to the variability of points around y, the prediction confidence interval is usually much larger than the fitted confidence interval.

For predictor scores far away from the mean of the distribution, the fitted/prediction intervals are wider than for fitted/predictor scores close to the mean of the distribution.

```
> # The original scatterplot:
> plot(x = x, y = y,
```

```
       xlim = c(0, 160), ylim = c(0, 100),
       xlab = "Big Five Neuroticism",
       ylab = "Trait Anxiety",
       main = "Scatterplot",
       pch = 20, axes = FALSE)
> axis(1, col = "grey")
> axis(2, col = "grey", las = 1)
> abline(mod.lm1, lwd = 2)
> # The prediction interval bands:
> x.t        <- seq(0, 160, by = .01)
> y.t        <- predict(mod.lm1, newdata = list(bfneur = x.t))
> var.pred2  <- var.eps*(1 + 1/N + (x.t - mean(x))^2/(var(x)*(N - 1)))
> se.pred2   <- sqrt(var.pred2)
> lines(x.t, y = y.t - qt((1 + gamma)/2, df = N - 2)*se.pred2,
        lty = 3, lwd = 3, col = "darkgreen")
> lines(x.t, y = y.t + qt((1 + gamma)/2, df = N - 2)*se.pred2,
        lty = 3, lwd = 3, col = "darkgreen")
```



12.4 Appendix: Functions Used

Here is a list of important functions from this chapter:

```
# Pearson Correlations in R
cor(x, y)  # correlation on two vectors
cor(X)     # correlation on a matrix or data.frame
cov(x, y)  # correlation on two vectors
cov(X)     # covariance on a matrix
cor.test(x, y) # is this correlation sig dif from 0?
atanh(r)   # r --> z (arc (inverse) hyperbolic tangent)
tanh(z)    # z --> r (hyperbolic tangent)

# Alternative Correlations in R
library(polycor)   # for the polychor function
cor(x, y, method)  # general correlation on two vectors
polychor(x, y, ML, # tetrachoric/polychoric cors
         std.err)
rank(x)            # ranks of the data
sign(x)            # sign (+/-) of each score


# Regression in R
lm(formula, data) # perform a linear regression
summary(mod.lm)   # summary statistics for our model
plot(mod.lm)      # interesting regression plots :)
coef(mod.lm)      # the slope and intercept
fitted(mod.lm)    # the fitted values (on the reg line)
resid(mod.lm)     # the residuals (off of the reg line)
confint(mod.lm,   # a confidence interval for reg parameters
        level)
predict(mod.lm, newdata,  # predict or fit new values
        interval,
        level)

# Scatterplots
plot(x, y, ... )  # a scatterplot
jitter(x)         # add some noise to a variable
abline(mod.lm)    # the linear fit (on top of the plot)
```

Chapter 13

Tests on Count Data

> Interviewer: *Was there anything unusual about Dinsdale?*
> Lady Friend: *I should say not! Dinsdale was a perfectly normal person in every way. Except inasmuch as he was convinced that he was being watched by a giant hedgehog he referred to as Spiny Norman.*
> —Monty Python's Flying Circus - Episode 14

One question that students fret over in introduction to statistics classes is: "how do we conduct inferential statistics with count or proportion data?" All of the inferential tests discussed in earlier chapters draw conclusions about the sample mean of one or several groups. Yet often (generally in made up problems for the emotional benefit of statistics teachers), one encounters questions about the relationship between categorical variables. These questions were actually addressed by the earliest statisticians. And many of the resulting test names: "Pearson's χ^2" or "Fisher's Exact Test" or "Yates' Continuity Correction" reference those early statisticians.

13.1 χ^2 Tests in R

There are two classic χ^2 tests on count data developed by Pearson (and then Fisher), the χ^2 goodness of fit test, and the χ^2 test of independence. The former test determines (inferentially) whether a set of sample counts are unlikely given proportions from a hypothetical population. Although the traditional χ^2 goodness of fit test is rarely used, the χ^2 distribution is often used to compare the fit of models. This comparison usually takes the form of a likelihood ratio, which will be discussed later in the chapter. The other χ^2 test (referred to as the χ^2 independence because, you know, the χ^2 test decided to escape from all of the other sillier tests described in earlier chapters) tries to determine whether several categorical variables are related. Both of these tests parallel the proportions, probability, Bayes' theorem stuff described in Chapter 4. Before getting into the nitty gritty of performing each test, I should review how to form tables appropriate for the χ^2.

13.1.1 Setting up Data

One of the initial problems in performing a χ^2 test is setting up the data in such a form as to be able to do calculations. An example dataset appropriate for χ^2 tests is the `survey` dataset in the `MASS` package, which we can load via the `data` function.

```
> data(survey, package = "MASS")
> head(survey)
     Sex Wr.Hnd NW.Hnd W.Hnd    Fold Pulse    Clap Exer
1 Female   18.5   18.0 Right  R on L    92    Left Some
2   Male   19.5   20.5  Left  R on L   104    Left None
3   Male   18.0   13.3 Right  L on R    87 Neither None
4   Male   18.8   18.9 Right  R on L    NA Neither None
5   Male   20.0   20.0 Right Neither    35   Right Some
6 Female   18.0   17.7 Right  L on R    64   Right Some
  Smoke Height      M.I    Age
1 Never 173.00   Metric 18.250
2 Regul 177.80 Imperial 17.583
3 Occas     NA     <NA> 16.917
4 Never 160.00   Metric 20.333
5 Never 165.00   Metric 23.667
6 Never 172.72 Imperial 21.000
```

The `survey` dataset describes attributes of students in an introduction to statistics class, including: gender, handedness, pulse, smoke, etc. To use the `survey` data (or data in a similar form) for a χ^2 test, you must first organize the data. The `table` function (if you don't remember) creates a contingency table based on the number of scores in each *combination* of levels. We could use the `table` function to count the number of males and females,

```
> (O.Sex <- table(survey$Sex))
Female   Male
   118    118
```

left-handed and right-handed students,

```
> (O.Hnd <- table(survey$W.Hnd))
 Left Right
   18   218
```

or the level of smoking for students.

```
> (O.Smk <- table(survey$Smoke))
Heavy Never Occas Regul
   11   189    19    17
```

And we could create a 2-way contingency table by listing pairs of (usually categorical) variables in order.

```
> (O.SH  <- table(survey$Sex,   survey$W.Hnd))
```

```
        Left Right
Female    7   110
Male     10   108
> (O.SSm <- table(survey$Sex,   survey$Smoke))
        Heavy Never Occas Regul
Female      5    99     9     5
Male        6    89    10    12
> (O.HSm <- table(survey$W.Hnd, survey$Smoke))
       Heavy Never Occas Regul
Left       1    13     3     1
Right     10   175    16    16
```

And we could even create a 3-way (or more) contingency table by listing triples of variables in order.

```
> (O.SHSm <- table(survey$Sex, survey$W.Hnd, survey$Smoke))
, ,  = Heavy


        Left Right
Female    0     5
Male      1     5

, ,  = Never


        Left Right
Female    6    92
Male      6    83

, ,  = Occas


        Left Right
Female    1     8
Male      2     8

, ,  = Regul


        Left Right
Female    0     5
Male      1    11
```

Datasets might also be given to you in those weird n-way contingency tables. For example, the `HairEyeColor` dataset describes the hair, eye color, and sex of statistics' students in a pretty-easy-to-interpret-but-difficult-to-manipulate way.

```
> data(HairEyeColor)  # yay - exciting data!
> HairEyeColor        # wtf do we do with this?
, , Sex = Male

       Eye
Hair    Brown Blue Hazel Green
  Black    32   11    10     3
  Brown    53   50    25    15
  Red      10   10     7     7
  Blond     3   30     5     8

, , Sex = Female

       Eye
Hair    Brown Blue Hazel Green
  Black    36    9     5     2
  Brown    66   34    29    14
  Red      16    7     7     7
  Blond     4   64     5     8
```

Although these really weird tables can be manipulated mathematically,

```
> HairEyeColor + 1
, , Sex = Male

       Eye
Hair    Brown Blue Hazel Green
  Black    33   12    11     4
  Brown    54   51    26    16
  Red      11   11     8     8
  Blond     4   31     6     9

, , Sex = Female

       Eye
Hair    Brown Blue Hazel Green
  Black    37   10     6     3
  Brown    67   35    30    15
  Red      17    8     8     8
  Blond     5   65     6     9
> HairEyeColor * 428
, , Sex = Male

       Eye
Hair    Brown  Blue Hazel Green
  Black 13696  4708  4280  1284
  Brown 22684 21400 10700  6420
  Red    4280  4280  2996  2996
```

```
  Blond  1284 12840  2140  3424

, , Sex = Female

       Eye
Hair    Brown  Blue Hazel Green
  Black 15408  3852  2140   856
  Brown 28248 14552 12412  5992
  Red    6848  2996  2996  2996
  Blond  1712 27392  2140  3424
```

you generally want to break down contingency tables into constituent parts. The (obviously existing) function for this job is `margin.table`, also discussed in Chapter 4:

```
margin.table(table, margin)
```

The first argument of `margin.table` is the table that you want to simplify (here, that table is `HairEyeColor`). The second argument of `margin.table` is the particular "dimensions" of the table that you want to keep in the resulting table, in order, inside of a vector, where: row is 1, column is 2, third dimension (after double comma) is 3, fourth dimension (after triple comma) is 4, etc. For example, the variable on the rows of `HairEyeColor` is `Hair`, so if we let `margin = 1`, we will end up with a table of the number of people with each hair color.

```
> (O.Hair <- margin.table(HairEyeColor, margin = 1))  # hair color
Hair
Black Brown   Red Blond
  108   286    71   127
```

We can also construct a margin table of the column variable (`Eye`) or the third-dimension variable (`Sex`):

```
> (O.Eye <- margin.table(HairEyeColor, margin = 2)) # eye color
Eye
Brown  Blue Hazel Green
  220   215    93    64
> (O.Sex <- margin.table(HairEyeColor, margin = 3)) # gender
Sex
  Male Female
   279    313
```

However, we can also use the `margin` argument to take this 3-way contingency table and force out of it a 2-way contingency table by listing the two desired dimensions to keep as elements of a numeric vector. For instance, we could form a `Hair` by `Eye` color contingency by letting `margin` equal the vector 1 (to stand for `Hair`) and 2 (to stand for `Eye`)

```
> # c(1, 2) --> 1 (Hair) and 2 (Eye)
> (O.HE <- margin.table(HairEyeColor, margin = c(1, 2)))
```

```
       Eye
Hair    Brown Blue Hazel Green
  Black    68   20    15     5
  Brown   119   84    54    29
  Red      26   17    14    14
  Blond     7   94    10    16
```

And using our vector rules, we could form every other combination of counts.

```
> # c(1, 3) --> 1 (Hair) and 3 (Sex)
> (O.HS <- margin.table(HairEyeColor, margin = c(1, 3)))
       Sex
Hair    Male Female
  Black   56     52
  Brown  143    143
  Red     34     37
  Blond   46     81
> # c(2, 3) --> 2 (Eye) and 3 (Sex)
> (O.ES <- margin.table(HairEyeColor, margin = c(2, 3)))
       Sex
Eye     Male Female
  Brown   98    122
  Blue   101    114
  Hazel   47     46
  Green   33     31
```

We could also take 4-way or 5-way or even 6-way contingency tables and let `margin` equal a length 3 or 4 vector to break down our bigger tables into slightly smaller parts. However, the annoying repetitiveness of the `margin.table` function is not needed anymore. Once we have small (1-way or 2-way) contingency tables, we are ready to use MATH to turn those contingency tables into beautiful χ^2 statistics.

13.1.2 The χ^2 Goodness of Fit Test

The χ^2 goodness of fit test starts with two "vectors": observed counts and null hypothesis probabilities. Once you have those two vectors, you can perform the χ^2 test by `R` or let the `chisq.test` function do all of the magic itself.

Pretend that you have collected quite a few statistics' students and you want to determine whether there's evidence to believe that the proportion of students with brown and blue eyes are the same, the proportion of students with hazel and green eyes are the same, and the proportion of students with brown/blue eyes is twice the proportion of students with hazel/green eyes. Then your null and alternative hypotheses can be written as follows:

$$\begin{aligned} &H_0 : \pi_{\text{brown}} = \pi_{\text{blue}} = .33;\ \pi_{\text{hazel}} = \pi_{\text{green}} = .17 \\ &H_0 : H_0 \text{ is not true.} \end{aligned}$$

Our observed frequencies are listed in `O.Eye` object.

```
> O.Eye
Eye
Brown  Blue Hazel Green
  220   215    93    64
```

And the observed proportions should be compared to the expected proportions *according to the null hypothesis*.

```
> Ep.Eye <- c(.33, .33, .17, .17)
```

The *order* of the probability vector must correspond to the order of the observed vector, or R will match the incorrect frequency with the incorrect probability. We can then create a vector of *expected* frequencies (under the null hypothesis) by multiplying the expected probabilities by the total number of students:

```
> N <- sum(O.Eye)
> (E.Eye <- N * Ep.Eye)
[1] 195.36 195.36 100.64 100.64
```

The χ^2 statistic is then calculated by translating the Old MacDonald formula:

> Old MacDonald had a statistical procedure: E_i, E_i, O (i)
> And using this statistical procedure he tested counts: E_i, E_i, O (i)
> What a useless test, what a silly test,
> Here a whew, there a rue,
> If the null hypothesis is never true,
> Why don't we just collect more people? I don't know! Statistics is silly!

$$\chi^2_{df} = \sum_{i=1}^{C} \left[\frac{(O_i - E_i)^2}{E_i} \right]$$

with $df = C - 1$. It's fairly easy to plug the numbers into the formula using what we already know about "math" in R.

```
> (chisq.obt <- sum( (O.Eye - E.Eye)^2/E.Eye ))
[1] 19.00171
> (df.obt <- length(O.Eye) - 1)
[1] 3
```

Because `O.Eye` is a vector of counts, `length(O.Eye)` tells us the number of eye colors. Once we have χ^2_{obt}, what do we do? Well, as always, there are two options. We could find the critical value using the `qchisq` function, noting that our one parameter is `df` and the χ^2-test is always an upper-tailed test.

```
> alpha <- .05
> (chisq.crit <- qchisq(alpha, df = df.obt, lower.tail = FALSE))
[1] 7.814728
> (chisq.crit < chisq.obt)  # do we reject the null hypothesis?
[1] TRUE
```

Or we could find the p-value directly in R by using the `pchisq` function, noting that the p-value is the area *greater than* (`lower.tail = FALSE`) our χ^2_{obt}.

```
> (p <- pchisq(chisq.obt, df = df.obt, lower.tail = FALSE))
[1] 0.0002731764
> (p < alpha) # do we reject the null hypothesis?
[1] TRUE
```

An alternative method of performing the χ^2 goodness of fit test is by using the (automatically awesome) `chisq.test` function in R.

```
chisq.test(x, y = NULL, correct = TRUE,
           p, rescale.p = FALSE,
           simulate.p.value = FALSE, B = 2000)
```

The `chisq.test` function takes the following arguments:

- `x`: Either a vector/matrix of counts (using the `table`, `margin.table`, `c`, or `matrix` functions) or a factor vector with *another* factor vector in the `y` argument. For a goodness of fit test, `x` *must* be a vector of counts. The "matrix of counts" and "factor vector" only work with the the test of independence.
- `y`: Either *nothing* (for a goodness of fit test), a numeric matrix of counts (if performing a test of independence and `x` is a numeric matrix of counts) or a factor vector (if performing a test of independence and `x` is a factor vector). Because we are only concerned about the goodness of fit test at the moment, `y` will not be needed.
- `correct`: A logical vector (TRUE/FALSE) indicating whether to do Yates' continuity correction (which *does not* apply to the goodness of fit test).
- `p`: A vector of null hypothesis probabilities for the goodness of fit test. ← We care about this argument. If `p` is left blank, then R assumes that the null hypothesis is that each category has *equal* probabilities in the population.
- `rescale.p`: A logical vector (TRUE/FALSE) indicating whether the `p` vector should be rescaled to sum to 1.
- `simulate.p.value`: A logical vector indicating whether the traditional/approximate χ^2 test should be performed or whether an "exact" p-value should be calculated by simulation.
- `B`: The number of replications of the simulation if `simulate.p.value` is TRUE.

For the χ^2 goodness of fit test, `x` should always be a vector of *counts* and *not* the original factor vector.

```
> # Test the chisq.test function with the Sex variable in survey:
> head(survey$Sex)
[1] Female Male   Male   Male   Male   Female
Levels: Female Male
```

```
> # Plug the original vector into chisq.test? Bad!
> try(chisq.test(x = survey$Sex), silent = TRUE)[1]
[1] "Error in chisq.test(x = survey$Sex) : \n  all entries of 'x' must be nonnegat
> # Find counts prior to plugging into chisq.test?  Good!
> chisq.test(x = table(survey$Sex))
        Chi-squared test for given probabilities

data:  table(survey$Sex)
X-squared = 0, df = 1, p-value = 1
```

But if we plug *just* a vector of counts into the `chisq.test` function without changing any of the other arguments, `R` will *assume* that our null hypothesis is "all probabilities are equal" and spit out a χ^2_{obt} value far away from the one that we already calculated.

```
> chisq.test(x = O.Eye)  # not correct :(
        Chi-squared test for given probabilities

data:  O.Eye
X-squared = 133.473, df = 3, p-value < 2.2e-16
```

We could correct the null hypothesis by explicitly plugging a vector of probabilities into the `p` argument.

```
> chisq.test(x = O.Eye,
             p = Ep.Eye, rescale.p = TRUE)
        Chi-squared test for given probabilities

data:  O.Eye
X-squared = 19.0017, df = 3, p-value = 0.0002732
> chisq.obt # does our chi^2 match? yup!
[1] 19.00171
> p         # does our p-value match? yup!
[1] 0.0002731764
```

> **Note**: You might want to set `rescale.p` equal to TRUE or `R` will give you an error if your `p` vector does not add up to 1. Because the entries in your `p` vector are just approximate probabilities, setting `rescale.p` equal to TRUE will prevent rounding error resulting in really annoying `R`-ounding error ☺.

Examining the output of the `chisq.test` function

```
> mod.chisq <- chisq.test(x = O.Eye,
                          p = Ep.Eye, rescale.p = TRUE)
> names(mod.chisq) # what is in the chi^2 object?
[1] "statistic" "parameter" "p.value"   "method"
[5] "data.name" "observed"  "expected"  "residuals"
[9] "stdres"
```

The `mod.chisq` object contains the χ^2 test statistic (in the `statistic` sub-object), degrees of freedom (in the `parameter` sub-object, for some reason), p-value (in the `p.value` sub-object), observed and expected counts (in the `observed` and `expected` sub-objects) and two other wacky things. We can pull each of the sub-objects out of `mod.chisq` with the `$` operator:

```
> mod.chisq$statistic  # chi^2 statistic (to plug into pchisq)
X-squared
 19.00171
> mod.chisq$parameter  # df (weird!?!??!)
df
 3
> mod.chisq$p.value    # p.value (output of pchisq)
[1] 0.0002731764
> mod.chisq$observed   # observed counts (what we plugged in)
Eye
Brown  Blue Hazel Green
  220   215    93    64
> mod.chisq$expected   # expected counts (p-vector * N)
 Brown   Blue  Hazel  Green
195.36 195.36 100.64 100.64
```

The second to last wacky thing is the `residuals`. If we square and sum `residuals`, we end up with our χ^2_{obt} test statistic:

```
> mod.chisq$residuals
Eye
     Brown       Blue      Hazel      Green
 1.7628805  1.4051531 -0.7615669 -3.6523312
> sum(mod.chisq$residuals^2)
[1] 19.00171
```

And the last wacky thing is the `stdres`. As it turns out, `stdres` are *like* the regular residuals (the things we square and sum), but the difference between O_i and E_i is standardized in a different manner. We can think about a regular residual as

$$e_{i,\,\text{reg}} = \frac{O_i - E_i}{\sqrt{E_i}} = \frac{O_i - E_i}{\sqrt{N \times Ep_i}}$$

the difference between O_i and E_i standardized by the square-root of the *expected count in each cell* over repeated sampling. The standardized residual is instead

$$e_{i,\,\text{std}} = \frac{O_i - E_i}{\sqrt{EV_i}} = \frac{O_i - E_i}{\sqrt{N \times Ep_i(1 - Ep_i)}}$$

the difference between O_i and E_i standardized by the square-root of the *expected variance in each cell* over repeated sampling.

```
> mod.chisq$stdres
```

```
Eye
     Brown       Blue      Hazel      Green
 2.1537013  1.7166678 -0.8359282 -4.0089543
> (O.Eye - E.Eye)/sqrt(N * Ep.Eye * (1 - Ep.Eye))
Eye
     Brown       Blue      Hazel      Green
 2.1537013  1.7166678 -0.8359282 -4.0089543
```

13.1.3 The χ^2 Test of Independence

Unlike the χ^2 goodness of fit test, the χ^2 test of independence starts with a matrix of observed counts. Because the test of independence uses a very specific null hypothesis, the matrix of expected counts can be formed directly from the matrix of observed counts without needing to know a strange, hypothesized `p` vector. And then the χ^2 test follows as always.

Pretend that we have collected data from statistics students in Canada (using a time machine and flying ability) and want to know (for some reason) whether exercise is dependent on sex. The null and alternative hypotheses can be written as follows:

$$\begin{aligned} H_0 &: \text{ Exercise is Independent of Sex} \\ H_1 &: \text{ Exercise is Dependent on Sex} \end{aligned}$$

The data are stored in the dataset `survey` in the `MASS` package, which can be accessed by using the `data` function and letting the `package` argument equal `"MASS"`.

```
> data(survey, package = "MASS")
> head(survey)
     Sex Wr.Hnd NW.Hnd W.Hnd    Fold Pulse    Clap Exer
1 Female   18.5   18.0 Right  R on L    92    Left Some
2   Male   19.5   20.5  Left  R on L   104    Left None
3   Male   18.0   13.3 Right  L on R    87 Neither None
4   Male   18.8   18.9 Right  R on L    NA Neither None
5   Male   20.0   20.0 Right Neither    35   Right Some
6 Female   18.0   17.7 Right  L on R    64   Right Some
  Smoke Height      M.I    Age
1 Never 173.00   Metric 18.250
2 Regul 177.80 Imperial 17.583
3 Occas     NA     <NA> 16.917
4 Never 160.00   Metric 20.333
5 Never 165.00   Metric 23.667
6 Never 172.72 Imperial 21.000
```

We want to construct a table of Sex $\times$ Exer, which we can easily do using the `table` function and specifying the appropriate variables.

```
> (O.SE <- table(Sex = survey$Sex, Exercise = survey$Exer))
```

```
        Exercise
Sex      Freq None Some
  Female   49   11   58
  Male     65   13   40
```

Unfortunately, we now need an *expected table* of frequencies, and obtaining the expected table for a χ^2 test of independence is not nearly as straightforward as for the goodness of fit test.

The expected table of frequencies assumes that: (1) The marginal frequencies are the same, and (2) The variables are independent. Independence $\implies$ (implies) Product Rule, which after a few lines of "math" reduces to the following formula

$$E_{ij} = \frac{n_{i\bullet} n_{\bullet j}}{N},$$

where $n_{i\bullet}$ is the number of people in row i, $n_{\bullet j}$ is the number of people in column j, and N is the total number of people in the table. Two `R` commands will find the total frequencies in the rows and columns of the table: `rowSums` and `colSums`, which, as described in Chapter 8 take matrices, add up all of the entries on the columns or rows of those matrices, and return a vector of sums.

```
> (O.S <- rowSums(O.SE)) # the row sums
Female   Male
   118    118
> (O.E <- colSums(O.SE)) # the column sums
Freq None Some
 114   24   98
> (N   <- sum(O.SE))     # the sum of EVERYONE!
[1] 236
```

But what now!??! How can we take these vectors and form a new expected count matrix? Well the trick is to use specialized **binary** operators. You already know of several binary operators: `+`, `-`, `*`, `\`. Binary operators take two things -- one before the operator and one following the operator -- and combine those things in some specialized way. Most of the binary operators in `R` use the following formulation: `%thing%`, where `thing` is the unique part of the operation. Some examples of binary operators are: `%%` (modulus or remainder, e.g., `13%%5 = 3`), `%/%` (quotient or integer part of division, e.g., `13%/%5 = 2`), `%*%` (matrix multiplication), `%x%` (Kronecker product), etc. A useful binary operator for our purposes is the outer product operator: `%o%`. Let's say we have two vectors, `x` and `y`, where `x` is length 3 and `y` is length 2. Then `%o%` will result in the following matrix:

$$x \;\%\text{o}\%\; y = \begin{bmatrix} x[1] * y[1] & x[1] * y[2] \\ x[2] * y[1] & x[2] * y[2] \\ x[3] * y[1] & x[3] * y[2] \end{bmatrix}$$

So using `%o%` results in the first column in the new matrix being `x` multiplied by the first entry of `y`, the second column in the new matrix being `x` multiplied by the second entry of `y`, etc. Using this logic, we can efficiently form an expected frequency matrix:

```
> (E.SE <- (O.S %o% O.E)/N)
```

```
        Freq None Some
Female    57   12   49
Male      57   12   49
```

And then we can form our typical, obtained χ^2 statistic as always:

$$\chi^2_{df} = \sum_{i=1}^{C}\sum_{j=1}^{R}\left[\frac{(O_{ij} - E_{ij})^2}{E_{ij}}\right]$$

```
> (chisq.obt <- sum( (O.SE - E.SE)^2/E.SE ))
[1] 5.718403
> (df <- (length(O.S) - 1)*(length(O.E) - 1))
[1] 2
```

And once we have χ^2_{obt}, we can either find the critical value using the `qchisq` function or find the p-value directly in `R` via the `pchisq` function.

```
> alpha <- .05
> (chisq.crit <- qchisq(alpha, df = df, lower.tail = FALSE))
[1] 5.991465
> (chisq.crit < chisq.obt) # do we reject the null hypothesis?
[1] FALSE
> (p <- pchisq(chisq.obt, df = df, lower.tail = FALSE))
[1] 0.0573145
> (p < alpha)              # do we reject the null hypothesis?
[1] FALSE
```

Not surprisingly, the χ^2 test of independence can also be pretty easily performed using the `chisq.test` function. For the χ^2 test of independence, `x` can either be a matrix of counts (if `y` is null) or the original factor vector (if `y` is set to the other factor vector). And ... that's it! As long as your table of counts is larger than 2×2, `R` will automatically perform the test of independence.

```
> # First, using the matrix of observed counts:
> chisq.test(x = O.SE, correct = FALSE)
        Pearson's Chi-squared test

data:  O.SE
X-squared = 5.7184, df = 2, p-value = 0.05731
> # Second, using the appropriate factor vectors:
> chisq.test(x = survey$Sex, y = survey$Exer, correct = FALSE)
        Pearson's Chi-squared test

data:  survey$Sex and survey$Exer
X-squared = 5.7184, df = 2, p-value = 0.05731
>
> # Note: I set correct to FALSE to make sure the Yates correction
> #       was not performed.
```

But what if we had a 2×2 table? For instance, we could create a new observed data matrix by combining the `None` and `Sum` categories of the previous table.

```
> O.SE2 <- cbind(O.SE[ , 1], rowSums(O.SE[, 2:3]))
> colnames(O.SE2) <- c("Yeh", "Meh")
> O.SE2
       Yeh Meh
Female  49  69
Male    65  53
```

Using the new Sex $\times$ Exercise table, we could (of course) perform the typical χ^2 test of independence, as before.

```
> (chisq.toi <- chisq.test(O.SE2, correct = FALSE))
        Pearson's Chi-squared test

data:  O.SE2
X-squared = 4.344, df = 1, p-value = 0.03714
```

But with a 2×2 table, there are many, alternative options.

13.1.4 Alternatives to the Typical χ^2

The Yates' Continuity Correction

The simplest alternative to the standard χ^2 test of independence is called Yates' continuity correction. The basic idea behind Yates' continuity correction is simple. If X is binomially distributed, then the probability that X is less than or equal to x is identical to the probability that X is less than $x + 1$ (because X can only take on integer values), or:

$$\Pr(X \leq x) = \Pr(X < x + 1)$$

But if we want to approximate X with some continuous variable (say Y, which is normally distributed), then we should pretend that x and $x + 1$ are somehow connected. And we do this by violating nature! If $\Pr(X \leq x) = \Pr(X < x + 1)$, then it might be better to use the value between the two points, $x + 1/2$, as a continuous approximation halfway between the discrete parts of x and $x + 1$. Of course, X can never take the value $x + 1/2$ (as X is discrete). But we are approximating discrete with continuous, and Y can take *any* value. Therefore

$$\Pr(Y \leq x + 1/2) \approx \Pr(Y \leq x) \approx \Pr(Y < x + 1)$$

might be a continuous value that better approximates the area between the jumps of a discrete sum.

Rather than using a normal distribution to approximate a binomial distribution, the Yates' correction uses a χ^2 distribution to approximate a very specific multinomial distribution. But in the same manner as for the standard continuity correction, we shift the observed counts a bit toward the expected counts to overcome the discrete jumps in the multinomial distribution. The Yates' correction is defined as

$$\chi_{df;\,\text{Yates}} = \sum_{i=1}^{C}\sum_{j=1}^{R}\left[\frac{(|O_{ij} - E_{ij}| - .5)^2}{E_{ij}}\right]$$

if every O_{ij} is separated by more than .5 from every E_{ij}. If an O_{ij} is separated by less than .5 from a specific E_{ij}, the Yates' correction is not applied *only* for that part of the sum. Using the new Sex × Exercise table, the Yates' χ^2 value is

```
> # Our observed table from before:
> O.SE2
       Yeh Meh
Female  49  69
Male    65  53
> # Our new expected table:
> (E.SE2 <- (rowSums(O.SE2) %o% colSums(O.SE2))/N)
       Yeh Meh
Female  57  61
Male    57  61
> # Note: every E.SE2 is at least .5 away from the O.SE2 value.
>
> (yates.obt <- sum( (abs(O.SE2 - E.SE2) - .5)^2/E.SE2 ))
[1] 3.817947
> (df2 <- (nrow(O.SE2) - 1)*(ncol(O.SE2) - 1))
[1] 1
```

and the corresponding p-value is

```
> (p.yates <-  pchisq(yates.obt, df = df2, lower.tail = FALSE))
[1] 0.05070635
```

The Yates' correction can be done automatically using the `chisq.test` function by setting `correct` equal to TRUE.

```
> (chisq.yates <- chisq.test(O.SE2, correct = TRUE))
        Pearson's Chi-squared test with Yates' continuity
        correction

data:  O.SE2
X-squared = 3.8179, df = 1, p-value = 0.05071
```

As promising as Yates' might be, one finds three important comments on the Yates' correction. First, the Yates' correction is only appropriate with 2×2 contingency tables. Second, given a 2×2 table, the Yates' correction is the *default* in R. You must set `correct` equal to FALSE to *not* perform the Yates' correction. And finally, the Yates' p-value (of 0.051) is quite a bit larger/more conservative than the standard χ^2 test of independence p-value (of 0.037). Later in this chapter, I will describe exactly *how* conservative the Yates' correction is relative to the standard χ^2 test.

The Likelihood Ratio Test

Another alternative option to the standard χ^2 that still has an approximate χ^2 distribution is the likelihood ratio test,

$$\chi^2_{df;\,\text{Likelihood}} = 2\sum_{i=1}^{C}\sum_{j=1}^{R}\left[O_{ij}\log\left(\frac{O_{ij}}{E_{ij}}\right)\right]$$

where log is the natural logarithm. The likelihood ratio test is similar to the standard χ^2 test of independence in that both are only an approximations to the χ^2 distribution and both use the same degrees of freedom. The only difference between the two χ^2 statistics is the arithmetic form. Using the statistics obtained in the previous part,

```
> # The likelihood ratio (easy!)
> (lik.obt <- 2*sum(O.SE2*log(O.SE2/E.SE2)))
[1] 4.357463
> # The degrees of freedom (same as before!)
> df2 # (R - 1)(C - 1)
[1] 1
> # The p-value (same method as before!)
> (p.lik <- pchisq(lik.obt, df = df2, lower.tail = FALSE))
[1] 0.03684714
```

Notice that the likelihood ratio p-value is actually smaller than the original χ^2 statistic (and *much* smaller than the Yates' correction) as the likelihood ratio is (oddly) *more* asymptotic than the original approximation!

Fisher's Exact Test

An alternative option to test dependency in a 2×2 table is Fisher's exact test. Unlike the typical χ^2 test of independence, the χ^2 with Yates' continuity correct, and the likelihood ratio test, Fisher's exact test is not an approximation to the p-value under certain (unlikely) situations. The inspiration for Fisher's exact test came from the lady tasting tea episode, whereby a lady proclaimed that she could tell whether a tea bag was added before milk or whether a tea bag was added before milk when making tea. And like the lady tasting tea experiment, Fisher's exact test assumes *fixed* marginals (e.g., that the number of cups with tea first and the number of cups with milk first are chosen in advance, and the testing lady knows, and therefore will choose, a specific number of cups to have tea or milk first). Fixed marginal counts is indeed a difficult assumption to make for most data collection designs.

In the lady tasting tea experiment, Fisher assumes m cups with tea before milk, n cups with milk before tea, and the lady will say a priori that k cups have tea before milk (and, therefore, $m + n - k$ cups have milk before tea). If the lady cannot tell whether a cup has "tea before milk" or "milk before tea", then the number of TRUE "tea before milk cups" *that she will say* has "tea before milk" can be described with the hypergeometric distribution. And therefore, we can use the hypergeometric distribution when trying to find p-values. In our case, we have 118 females and 118 males. If exercising was held constant, and if gender was independent of exercise level, then the probability of

observing *as few* female exercisers as we did could be found by using the hypergeometric distribution.

```
> # - m is the numer of females
> # - n is the number of NOT-females
> # - k is the number of high exercisers (determined a priori!)
> phyper(q = O.SE2[1, 1], m = O.S[1], n = O.S[2], k = colSums(O.SE2)[1])
[1] 0.02523843
>
> # If Sex is independent of Exercise:
> # -- prob of O.SE[1, 1] or less female exercisers?
> # -- hypergeometric distribution!
```

The above calculation is an example of a *lower* tailed Fisher's exact test. Odd! For the typical χ^2, we must perform a two-tailed test (independent versus not independent), whereas for the Fisher's exact test, we can perform lower tailed (less than a particular number in the upper left quadrant of the table), upper tailed (greater than a particular number) or two-tailed tests (more extreme than a particular number). In `R`, the above calculation can be easily performed by using the `fisher.test` function.

```
> fisher.test(O.SE2, alternative = "less")

        Fisher's Exact Test for Count Data

data:  O.SE2
p-value = 0.02524
alternative hypothesis: true odds ratio is less than 1
95 percent confidence interval:
 0.0000000 0.9224437
sample estimates:
odds ratio
 0.5803901
```

Of course, one would usually assume a two-tailed alternative hypothesis when looking at any 2×2 table, which is the default in `R`.

```
> fisher.test(O.SE2)

        Fisher's Exact Test for Count Data

data:  O.SE2
p-value = 0.05048
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.334605 1.001085
sample estimates:
odds ratio
 0.5803901
```

The two-tailed p-value for the Fisher's exact test is more complicated than the typical p-value (and, therefore, more difficult to calculate), but it can usually be well approximated

by 2 times the appropriate one-tailed p-value. Interestingly, Fisher's exact test isn't really testing the same statistic as any of the χ^2 tests. Fisher's exact test takes as its statistic the odds ratio. And with the Fisher's exact test, you can change the null hypothesis odds ratio (by making the `or` argument different from 1) or even change the confidence interval around the odds ratio (by manipulating the `conf.level` argument). However, the odds ratio that `fisher.test` calculates is only an estimate of the odds ratio based on some maximization algorithm. How could one actually find the actual odds ratio in any given sample?

13.2 Odds and Risk

The natural descriptors of 2×2 contingency tables are the relative risk and the odds ratio. The risk is essentially the conditional probability of being in one cell of the table under the assumption of being in a particular category. For example, the risk of exercising (an odd phrasing to say the least, unless exercising causes death!) given that one is female is

$$\text{Risk Exercising} \mid \text{Female} = \frac{\#\text{ Exercising Females}}{\#\text{ Females}}.$$

In our case, that risk can be easily calculated by using the appropriate joint count and the appropriate marginal count.

```
> (risk.f <- O.SE2[1, 1]/rowSums(O.SE2)[1])
   Female
0.4152542
```

Using the same logic, one could also calculate the risk of exercising given that one is male:

```
> (risk.m <- O.SE2[2, 1]/rowSums(O.SE2)[2])
     Male
0.5508475
```

and then the relative risk of exercising given that one is female is the ratio of the two risks with the "female" part of the risk put in the numerator.

```
> (relrisk.f <- risk.f/risk.m)
   Female
0.7538462
```

If the counts of each part of one of the variables are chosen in advance of the experiment (e.g., the number of females equalling the number of males), then the risk only makes sense in one direction. The relative risk of being female for heavy exercisers, while possible to calculate,

```
> (risk.y <- O.SE2[1, 1]/colSums(O.SE2)[1])
      Yeh
0.4298246
> (risk.n <- O.SE2[1, 2]/colSums(O.SE2)[2])
```

```
      Meh
0.5655738
> (relrisk.y <- risk.y/risk.n)
      Yeh
0.7599797
```

does not make sense as being female is not a consequence of exercising.

An alternative to the risk/relative risk is the odds/odds ratio. The odds are also conditional on being in a particular category:

$$\text{Odds Exercising} \mid \text{Female} = \frac{\#\text{ Exercising Females}}{\#\text{ Not Exercising Females}},$$

and calculated directly in the 2×2 contingency table without having to find the marginal counts.

```
> (odds.f <- O.SE2[1, 1]/O.SE2[1, 2])
[1] 0.7101449
> (odds.m <- O.SE2[2, 1]/O.SE2[2, 2])
[1] 1.226415
```

And the odds ratio of females exercising to males exercising is just the odds of exercising for females divided by the odds of exercising for males, or:

```
> (oddsrat.f <- odds.f/odds.m)
[1] 0.5790412
```

But in a retrospective study, it also makes sense to look at the odds of being female given that one exercises and the odds of being female given that one does not exercise.

```
> (odds.y <- O.SE2[1, 1]/O.SE2[2, 1])
[1] 0.7538462
> (odds.n <- O.SE2[1, 2]/O.SE2[2, 2])
[1] 1.301887
```

And unlike the relative risk, the odds ratio looking at the table in one direction and the odds ratio looking at the table the other direction are identical.

```
> (oddsrat.y <- odds.y/odds.n) # odds ratio of y/f same as ...
[1] 0.5790412
> oddsrat.f                    # odds ratio of f/y.
[1] 0.5790412
```

13.3 Testing the χ^2 Test Statistic

You might wonder whether the χ^2 distribution adequately approximates the distribution of the χ^2 test statistic under the null hypothesis. One might also wonder if the other tests described in this chapter are better or worse approximations of reality. The

following function takes the total number of observations (N), true proportions in the rows of the table (R_p), true proportions in the columns of the table (C_p), whether or not the margins are fixed, the particular test, and the number of replications, and simulates p-values from the (hopefully) null distribution under the chosen test. If the test is accurate, approximately α p-values should be less than α.

```
> ######################
> # ChisqSim FUNCTION #
> ######################
>
> ChisqSim <- function(N = 10, Rp = c(.5, .5), Cp = c(.5, .5),
                       margin.fixed = FALSE,
                       Yates = FALSE, Fisher = FALSE,
                       reps = 10000){

 #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
 # Arguments:                                                                 #
 # - N      - the sum of all of the joint counts                              #
 # - Rp     - the marginal probabilities on the rows                          #
 # - Cp     - the marginal probabilities on the columns                       #
 # - margin.fixed - whether to assume fixed or varying marginal counts        #
 # - Yates  - whether (or not) to use Yates' correction                       #
 # - Fisher - whether (or not) to use Fisher's exact test                     #
 # - reps   - the number of samples to take                                   #
 #                                                                            #
 # Values:                                                                    #
 # - stat    - a vector of sample chi^2s or odds rats (only if Fisher)        #
 # - p.value - a vector of corresponding p-values                             #
 #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#

 ## 1. SCALE THE MARGINAL PROBABILITIES (IN CASE THEY DON'T ADD TO ONE) ##
   Rp <- Rp/sum(Rp)
   Cp <- Cp/sum(Cp)

 ## 2. VECTORS TO STORE THINGS ##
   stat    <- NULL
   p.value <- NULL

 ## 3. FINDING REPS STATISTICS/P-VALUES ##
   for(i in 1:reps){

 # "repeat" is to make sure that our statistic is a number:
     repeat{

 # If we assume fixed marginals, we:
 # a) Simulate a count in one cell of the table, and
 # b) Subtract to get counts in the other cells!
       if(margin.fixed == TRUE){
```

```
        O <- matrix(O, nrow = 2, ncol = 2)
        O[1, 1] <- rhyper(nn = 1, m = N*Rp[1], n = N*Rp[2], k = N*Cp[1])
        O[2, 1] <- N*Cp[1] - O[1, 1]
        O[1, 2] <- N*Rp[1] - O[1, 1]
        O[2, 2] <- N - (O[1, 1] + O[2, 1] + O[1, 2])

      } else{

# If we do not assume fixed marginals, we:
# a) Find the joint probabilities for all cells,
# b) Simulate counts based on these probabilities, and
# c) Combine these counts into an appropriate matrix!
        O <- rmultinom(1, size = N, prob = c(Rp %o% Cp))
        O <- matrix(O, nrow = 2)

      } # END ifelse STATEMENT

# If Fisher is TRUE --> Use fisher.test.
      if(Fisher == TRUE){

        samp    <- fisher.test(O, conf.int = FALSE)
        stat[i] <- samp$estimate

      } else{

# If Fisher is FALSE --> Use chisq.test:
# a) Yates correction only if Yates is TRUE,
# b) suppressWarnings prevents annoying output.
        samp    <- suppressWarnings( chisq.test(x = O, cor = Yates) )
        stat[i] <- samp$statistic

      } # END ifelse STATEMENT

      p.value[i] <- samp$p.value

# Checking to make sure our statistic is a number:
# --> Yes?  Leave the repeat and do another iteration of for
# --> No?   Ignore our stat and stay on the same iteration of for
      if( !is.nan(stat[i]) )
        break;

    } # END repeat LOOP

  } # END i LOOP

## 4. PUTTING INTO A LIST AND RETURNING ##
  return(list(stat = stat, p.value = p.value))
```

```
} # END chisq.sim FUNCTION
```

After loading the above function, we can test whether sample χ^2 statistics are actually χ^2 distributed. For example, if we assume a really small sample size ($N = 10$), assume that the marginal frequencies can vary, and assume that the marginal probabilities are identical, then we could simulate sample χ^2 statistics under the null hypothesis.

```
> set.seed(8923)
> chisq.samp <- ChisqSim(N = 10, Rp = c(.5, .5), Cp = c(.5, .5),
+                        margin.fixed = FALSE, reps = 10000)
```

Our new object has two sub-objects: `stat` and `p.value`,

```
> names(chisq.samp)
[1] "stat"    "p.value"
```

both of which we can access using the `$` operator.

```
> head(chisq.samp$stat)
[1] 0.4000000 0.2777778 1.6666667 4.4444444 0.6250000
[6] 2.8571429
> head(chisq.samp$p.value)
[1] 0.52708926 0.59816145 0.19670560 0.03501498 0.42919530
[6] 0.09096895
```

One can check the correctness of the χ^2 assumption by finding the proportion of p-values less than some set α rate (say $\alpha = .05$):

```
> alpha <- .05
> mean(chisq.samp$p.value < alpha)
[1] 0.053
```

and the theoretical α is pretty close to actual α. So the χ^2 test did a pretty good job. Both the Yates' correction and Fisher's exact test assume fixed marginals. And both are bound to be extremely conservative if that assumption can not be made. You should play around with the `ChisqSim` function on your own to determine the accuracy of your p-values under all of the methods with a particular total count and either assuming (or not) fixed marginal counts.

13.4 Appendix: Functions Used

Here is a list of important functions from this chapter:

```
# Binary operators:
%o% # outer product

# Categorical Variables:
table( ... )              # (complicated) contingency tables
margin.table(tab, margin) # sum across the "margin" of tab

# Chi-Squared Tests in R
chisq.test(x, y,                # goodness of fit test or ...
           correct,             # ... test of independence
           p, rescale.p,
           simulate.p.value, B)
fisher.test(x, alternative)     # Fisher's exact test
```

Bibliography

[1] Armstrong, J. (2007). Statistical significance tests are unnecessary: Reply to commentaries. *International Journal of Forecasting*, *23*, 335–377.

[2] Cohen, J. (1988). *Statistical power analysis for the behavioral sciences* (2nd ed.). Hillsdale, NJ: Lawrence Erlbaum.

[3] Cohen, J. (1994). The earth is round ($p < .05$). *American Psychologist*, *49*, 997–1003.

[4] Faraway, J. J. (2005) *Extending the linear model with R: Generalized linear, mixed effects and nonparametric regression models*. Boca Raton, FL: Chapman & Hall/CRC Press.

[5] Fox, J. Weisberg, S. (2010). *An R companion to applied regression*. Thousand Oaks, CA: Sage Publications.

[6] Howell, D. C. (2012). *Statistical methods for psychology* (12th Edition). Belmont, CA: Wadsworth.

[7] Maloff, N. (2011). *The art of R programming: A tour of statistical software design*. San Francisco, CA: No Starch Press.

[8] Mackowiak, P. A., Wasserman, S. S., & Levine, M. M. (1992). A critical appraisal of 98.6°F, the upper limit of normal body temperature, and other legacies of Carl Reinhold August Wunderlich. *The Journal of the American Medical Association*, *268*, 1578–1580.

[9] Pinheiro, J., & Bates, D. (2009). *Mixed effects models in S and S-plus*. New York, NY: Springer-Verlag.

[10] Satterthwaite, F. E. (1946). An approximate distribution of estimates of variance components. *Biometrics Bulletin*, *2*, 110–114.

[11] Venables, W. N., & Ripley, B. D. (2002). *Modern applied statistics with S*. New York, NY: Springer-Verlag.

[12] Verzani, J. (2005). *Using R for introductory statistics*. Boca Raton, FL: Chapman & Hall/CRC Press.