A Sufficient Introduction to R

Derek L. Sonderegger

2016-08-24

Contents

Chapter 1

Introduction

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

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

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

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

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

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

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

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

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

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

1.1 R as a simple calculator

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

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

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

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

```
6*8
```

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

```
4^3
```

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

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

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

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

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

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

```
sin(0)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2 Assignment

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

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

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

```
my.test.var
```

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

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

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

As your analysis gets more complicated, you'll want to save the results to a variable so that you can access the results later[1]. *If you don't assign the result to a variable, you have no way of accessing the result.* [2]

1.3 Scripts and RMarkdown

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

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

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

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

1.3.1 R Scripts (.R files)

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

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

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

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

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

1.3.2 R Markdown (.Rmd files)

When I was a graduate student, I had to tediously copy and past tables of output from the R console and figures I had made into my Microsoft Word document. Far too often I would realize I had made a small mistake in part (b) of a problem and would have to go back, correct my mistake, and then redo all the laborious copying. I often wished that I could write both the code for my statistical analysis and the long discussion about the interpretation all in the same document so that I could just re-run the analysis with a click of a button and all the tables and figures would be updated by magic. Fortunately that magic[3] now exists.

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

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

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

The R code in my document is nicely separated from my regular text using the three backticks and an instruction that it is R code that needs to be evaluated. The output of this document looks good as a HTML, PDF, or MS Word document. I have actually created this entire document using RMarkdown.

1.4 Exercises

Create an RMarkdown file that solves the following exercises.

1. Calculate $\log(6.2)$ first using base e and second using base 10. To figure out how to do different bases, it might be helpful to look at the help page for the `log` function.
2. Calculate the square root of 2 and save the result as the variable named sqrt2. Have R display the decimal value of sqrt2. *Hint: use Google to find the square root function. Perhaps search on the keywords "R square root function".*

Chapter 2

Vectors

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[1] The "c" stands for collection.

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

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

2.1 Accessing Vector Elements

Suppose I have defined a vector

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2 Scalar Functions Applied to Vectors

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

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

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

```
abs(x)
```

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

```
exp(x)
```

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

2.3 Vector Algebra

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

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

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

```
x * y
```

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

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

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

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

2.4 Commonly Used Vector Functions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.5 Exercises

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

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

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

Chapter 3

Statistical Tables

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

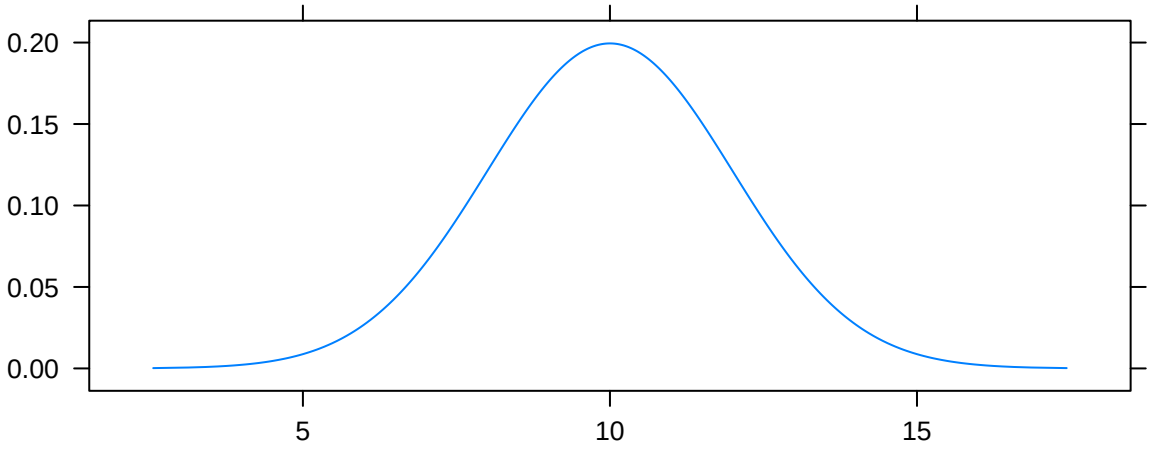
3.1 `mosaic::plotDist()` function

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

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

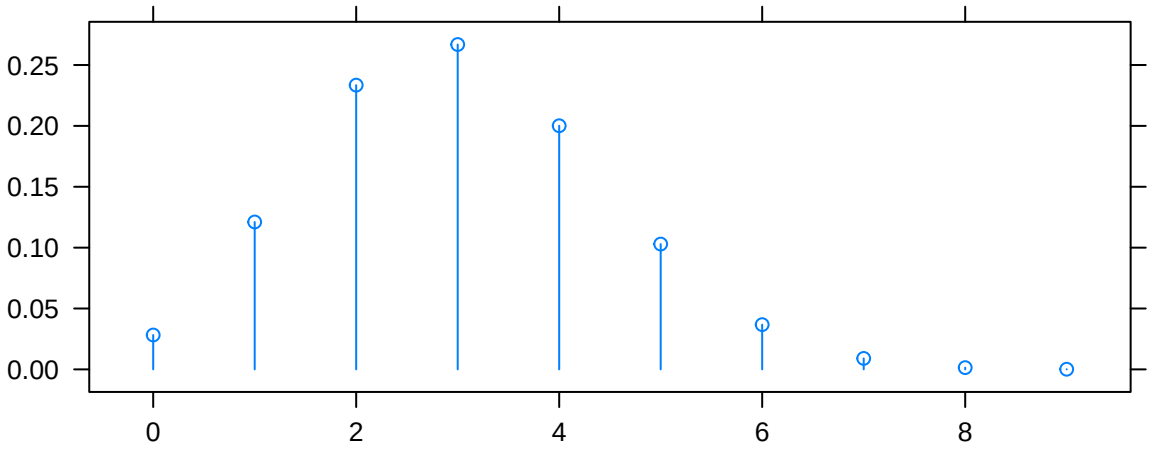
For example, to see the normal distribution with mean $mu = 10$ and standard deviation $\sigma = 2$, we use

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



This function works for discrete distributions as well.

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



3.2 Base R functions

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

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

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

3.2.1 d-function

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

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

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

thus

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

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

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

```
?dbinom
```

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

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

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

3.2.2 p-function

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

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

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

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

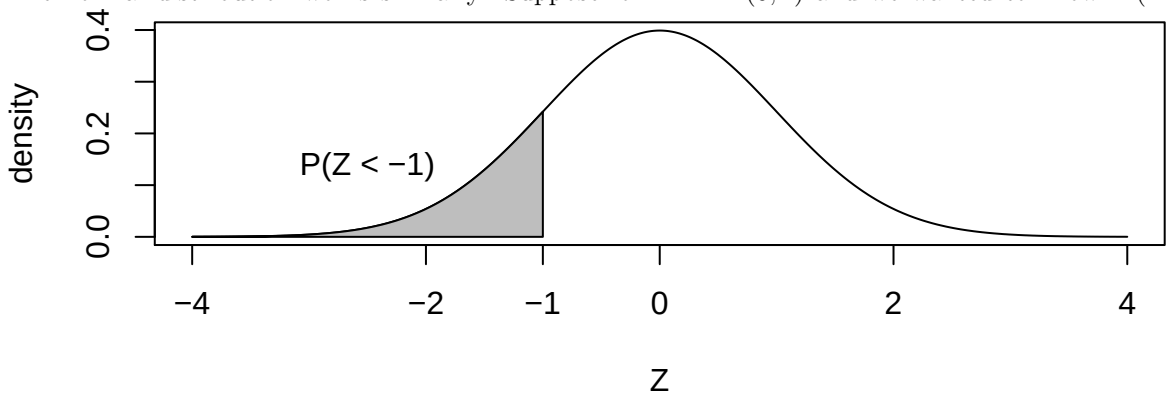
but this would get tedious for binomial distributions with a large number of trials. The shortcut is to use the `pbinom()` function.

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

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

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

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



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

```
pnorm(-1)
```

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

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

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

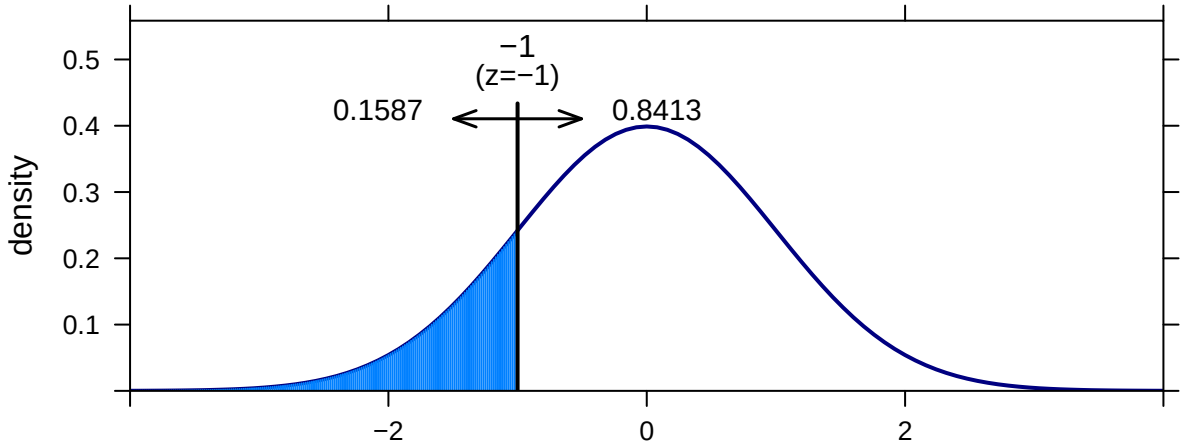
$$P(Z \geq -1) = P(Z \leq 1) = 1 - P(Z < -1)$$

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

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

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

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




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

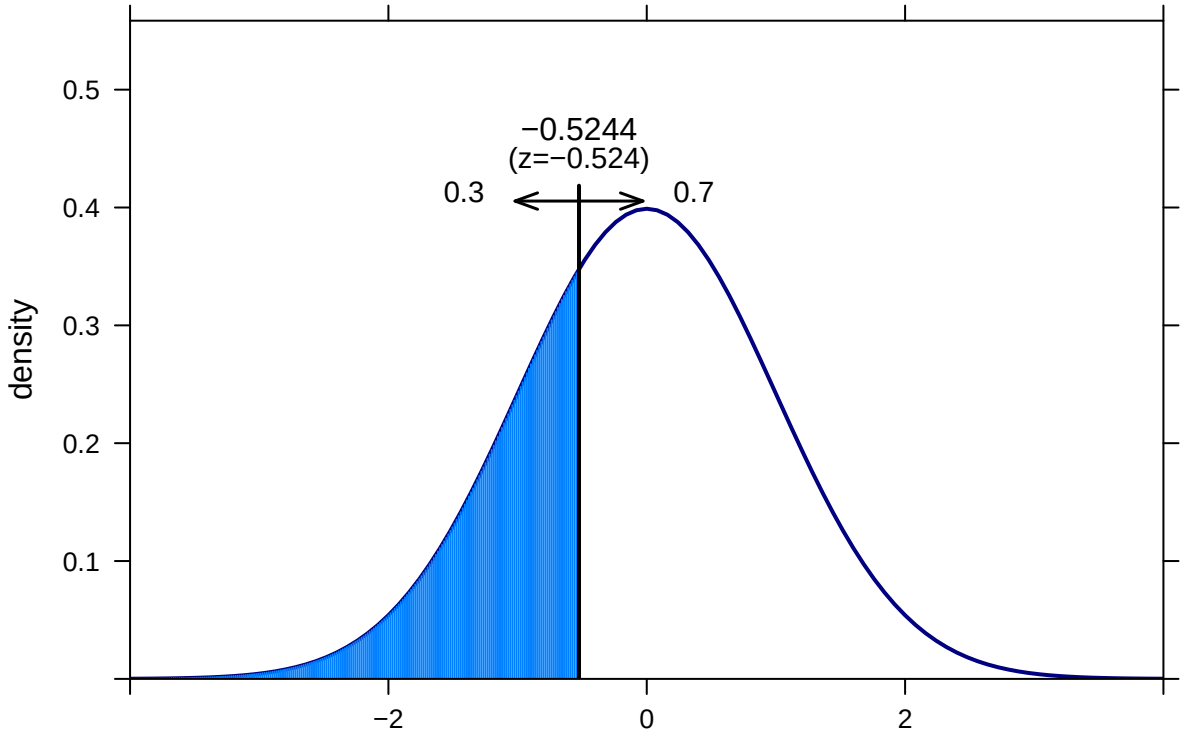
3.2.3 q-function

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

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

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

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




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

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

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

3.2.4 r-function

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

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

```
## [1] 19.37532 20.69179 21.99315 19.73156 20.24364
```

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

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

3.3 Exercises

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

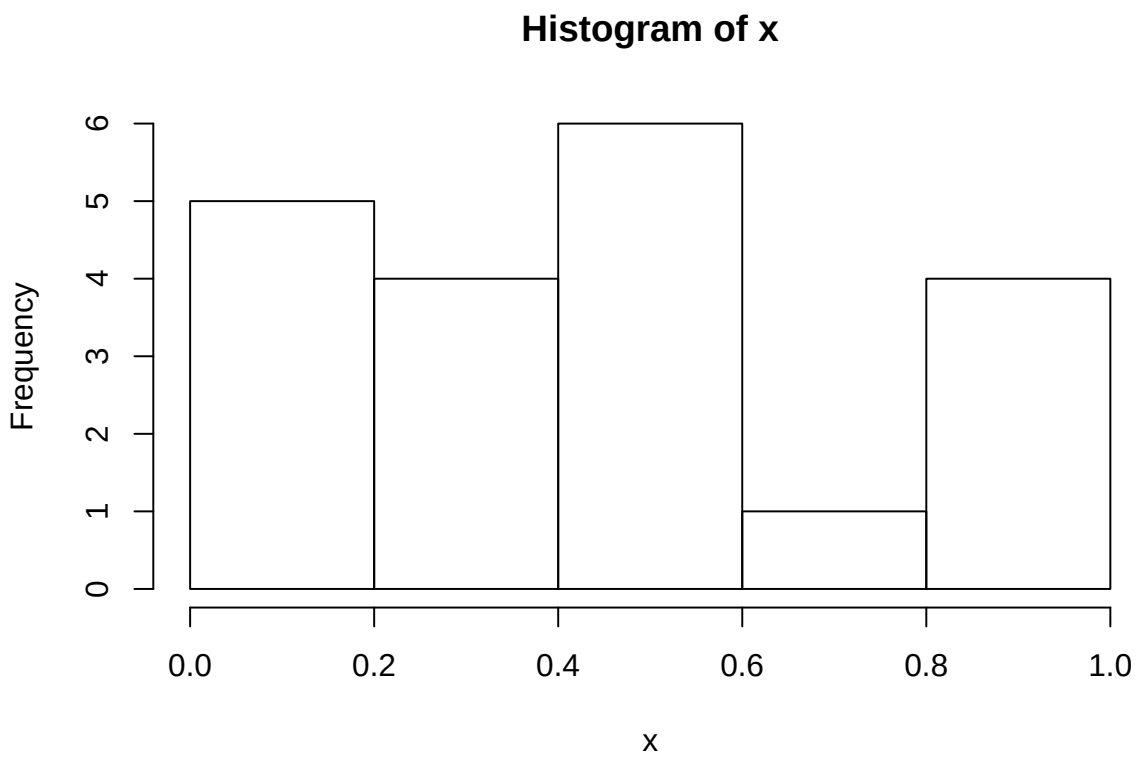
a) Generate a value of 1 from a uniform distribution with minimum 0, and maximum 1 using the `runif()` function. Repeat this step several times and confirm you are getting different values each time.

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

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

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

```
hist(x)
```



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

Chapter 4

Data Types

There are some basic data types that are commonly used.

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

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

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

4.1 Integers and Numerics

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

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

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

4.2 Character Strings

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

```
x <- "Goodnight Moon"

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

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

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

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

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

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

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

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

```
x
```

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

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

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

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

```
x+1
```

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

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

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

4.3 Factors

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

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

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

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

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

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

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

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

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

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

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

```
levels(z)
```

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

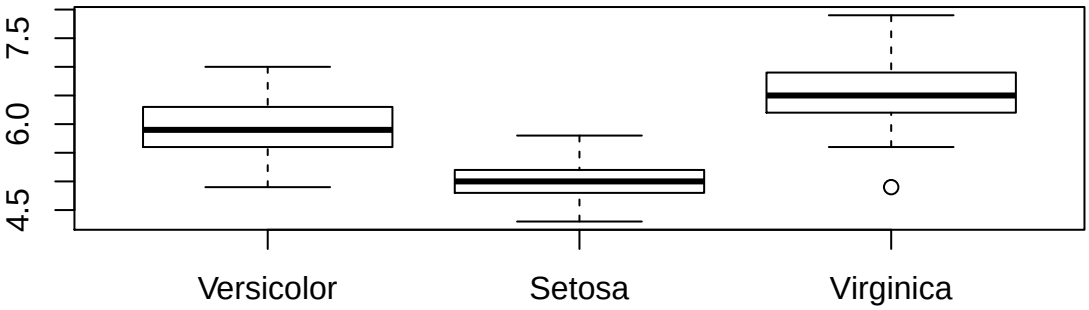
Notice that the order of the group names was done alphabetically, which we did not chose. This ordering of the levels has implications when we do an analysis or make a plot and R will always display information about the factor levels using this order. It would be nice to be able to change the order.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

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

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

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

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



Often we wish to take a continuous numerical vector and transform it into a factor. The function cut() takes a vector of numerical data and creates a factor based on your give cut-points.

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

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

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

```
# divide x into four groups, where I specify all 5 break points
# Notice that the the outside breakpoints must include all the data points.
# That is, the smallest break must be smaller than all the data, and the largest
# must be larger (or equal) to all the data.
cut(x, breaks = c(0, 2.5, 5.0, 7.5, 10))
```

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

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

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

4.4 Logicals

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.5 Exercises

1. Create a vector of character strings with six elements

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

and then

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

2. Given the vector of ages,

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

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

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

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

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

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

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

Chapter 5

Basic Graphing

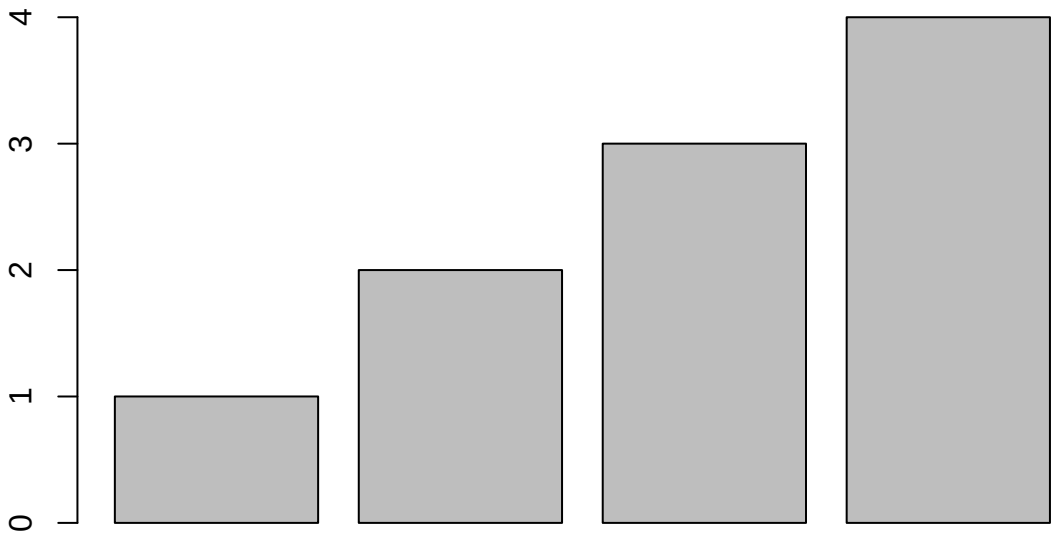
One of the most important things we can do in exploratory statistics is to just look at the data in graphical form so that we can more easily see trends. Most of the graph functions have similar options for setting the main title, x-label, y-labels, etc.

5.1 Univariate Graphs

6.1.1 Barcharts

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

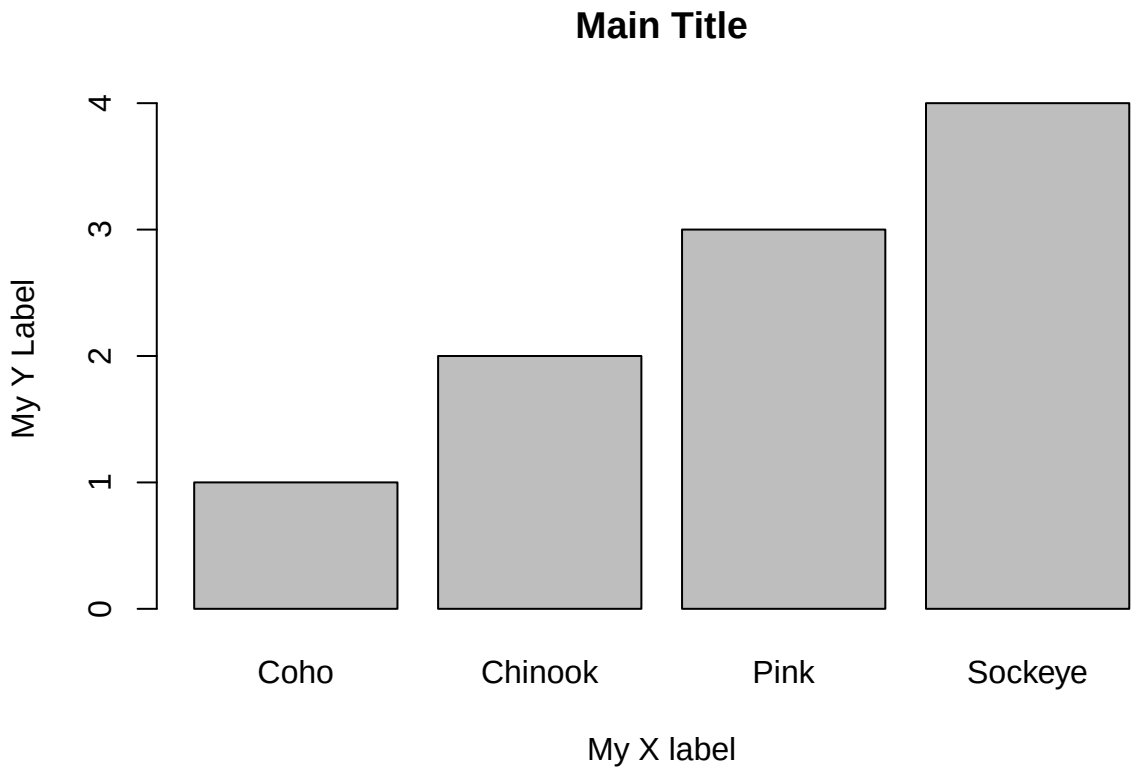
```
x <- 1:4      # The vector 1,2,3,4
barplot(x)    # First bar has height 1, next is 2, etc
```



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

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

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

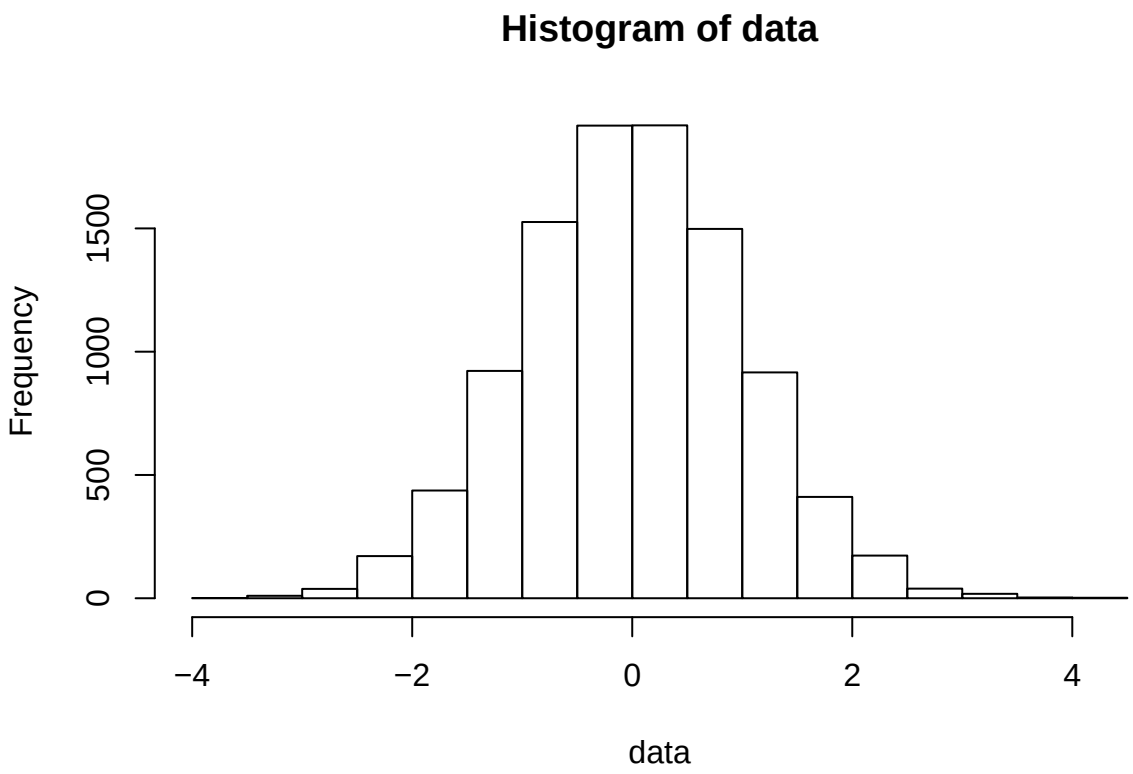


5.1.1 Histograms

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

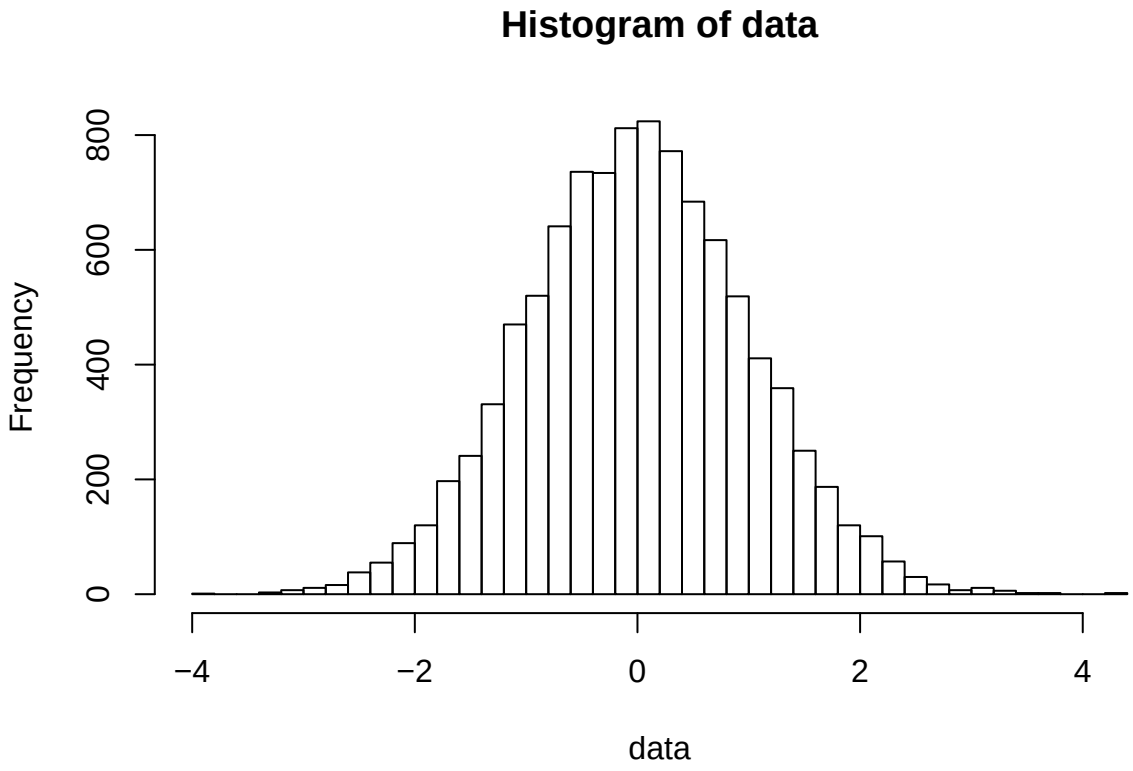
To generate data for the histogram, we'll use the function `rnorm(n, mean=0, sd=1)` which generates n random variables from a normal distribution with mean `mean` and standard deviation `sd`. Notice that rnorm will default to giving random variables from a standard normal distribution.

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



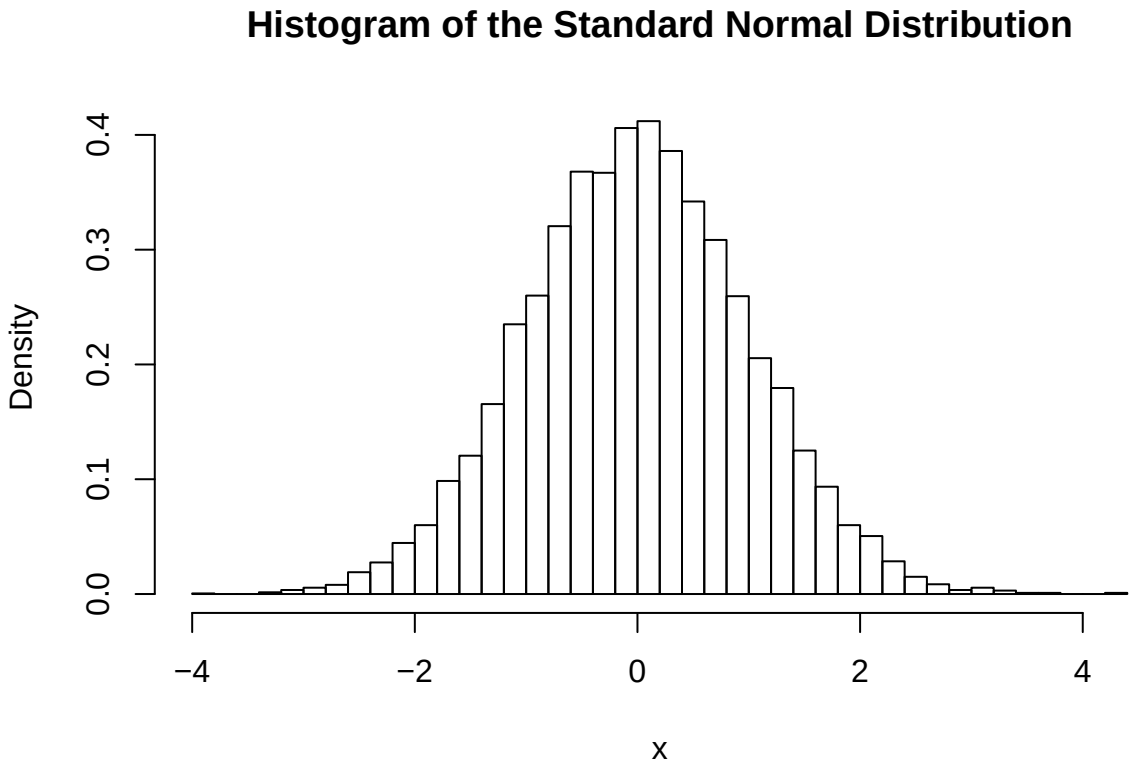
R does a decent job of picking reasonable defaults for the number of bins and breakpoints, but with 10,000 observations, I think we should have more bins. To do this, I'll use the optional breaks argument to specify where to split the bins.

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



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

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

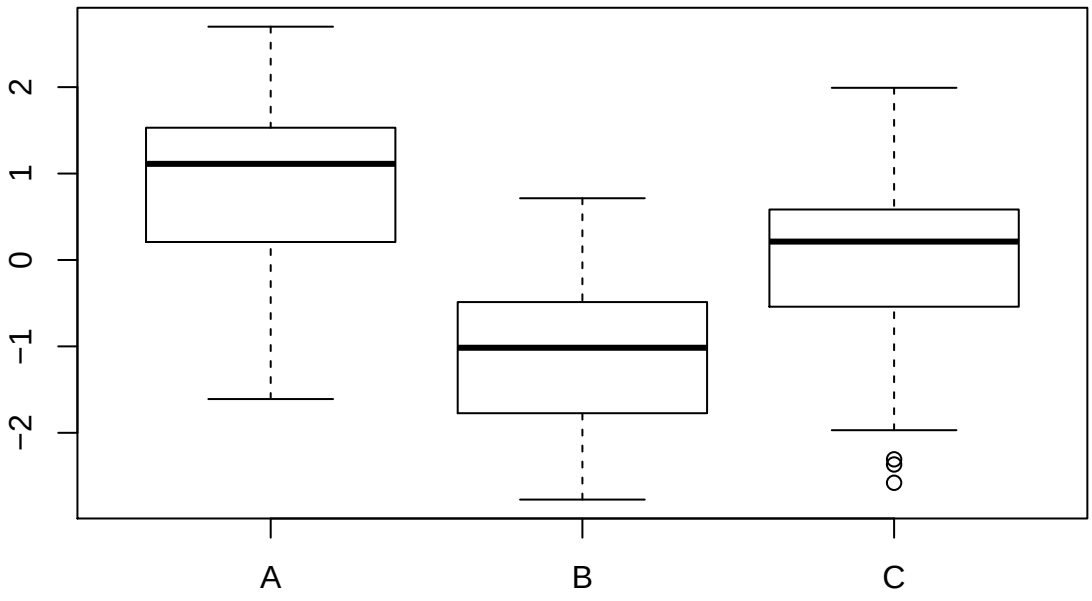


5.2 Bivariate Plots

5.2.1 Boxplots

Boxplots are designed to compare distributions among multiple groups.

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



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

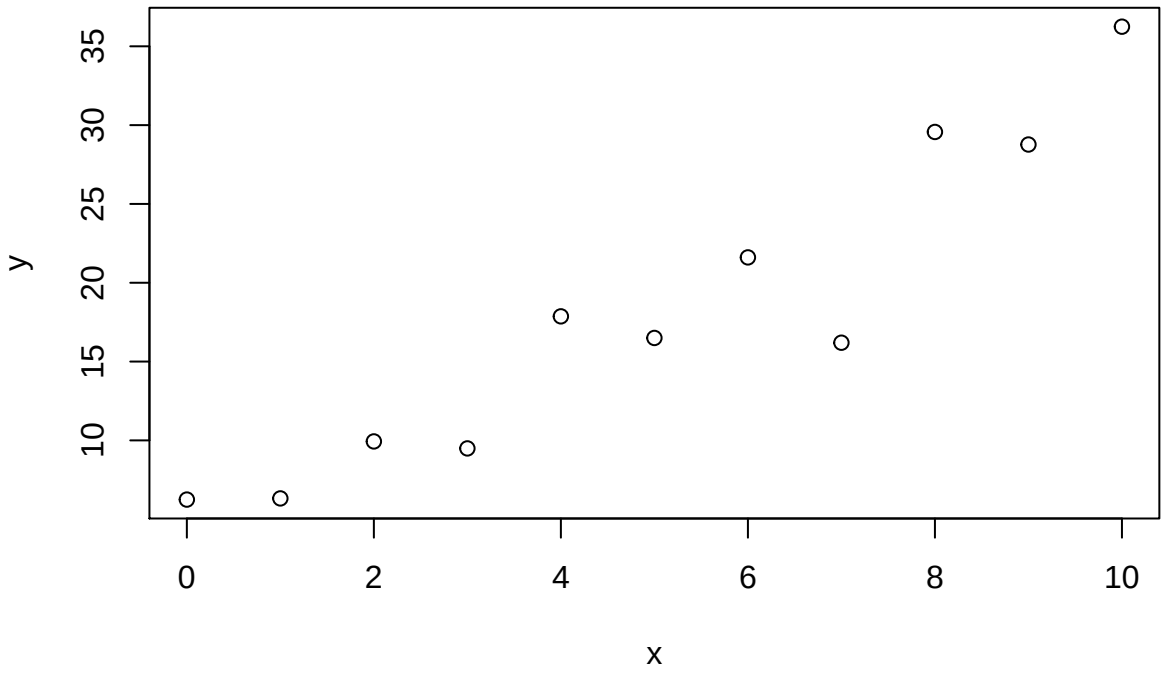
5.2.2 Scatterplots

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

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

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

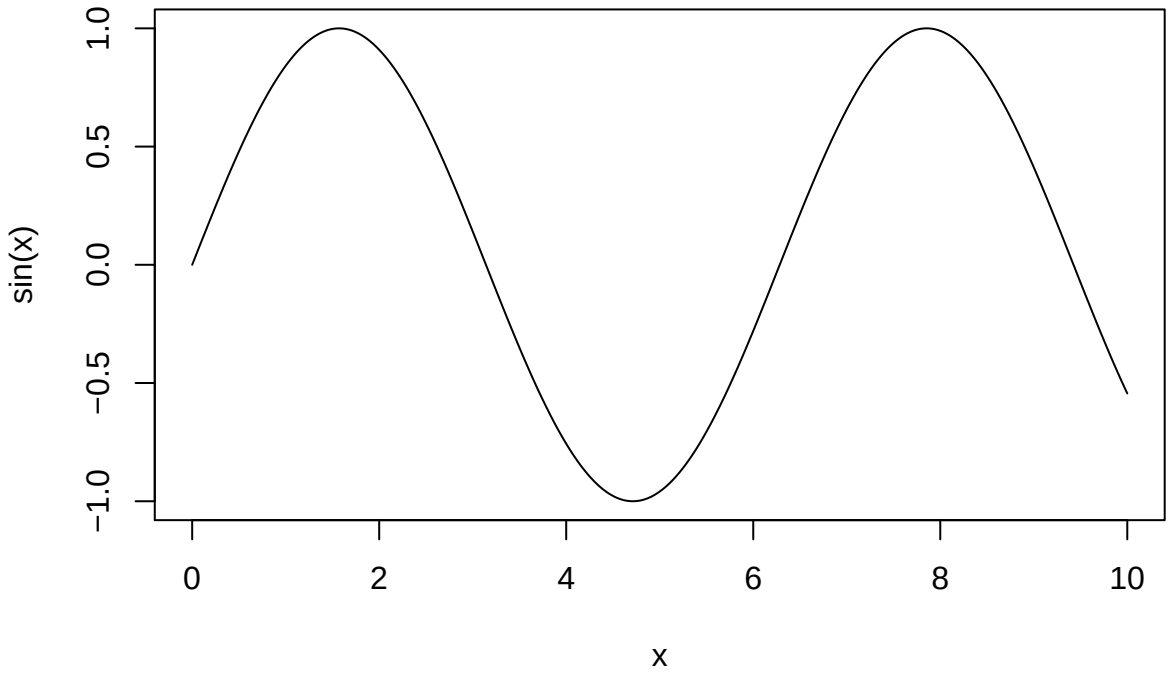
```
plot(x, y)
```



5.2.3 Arbitrary Curves

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

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



5.3 Advanced tricks

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

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

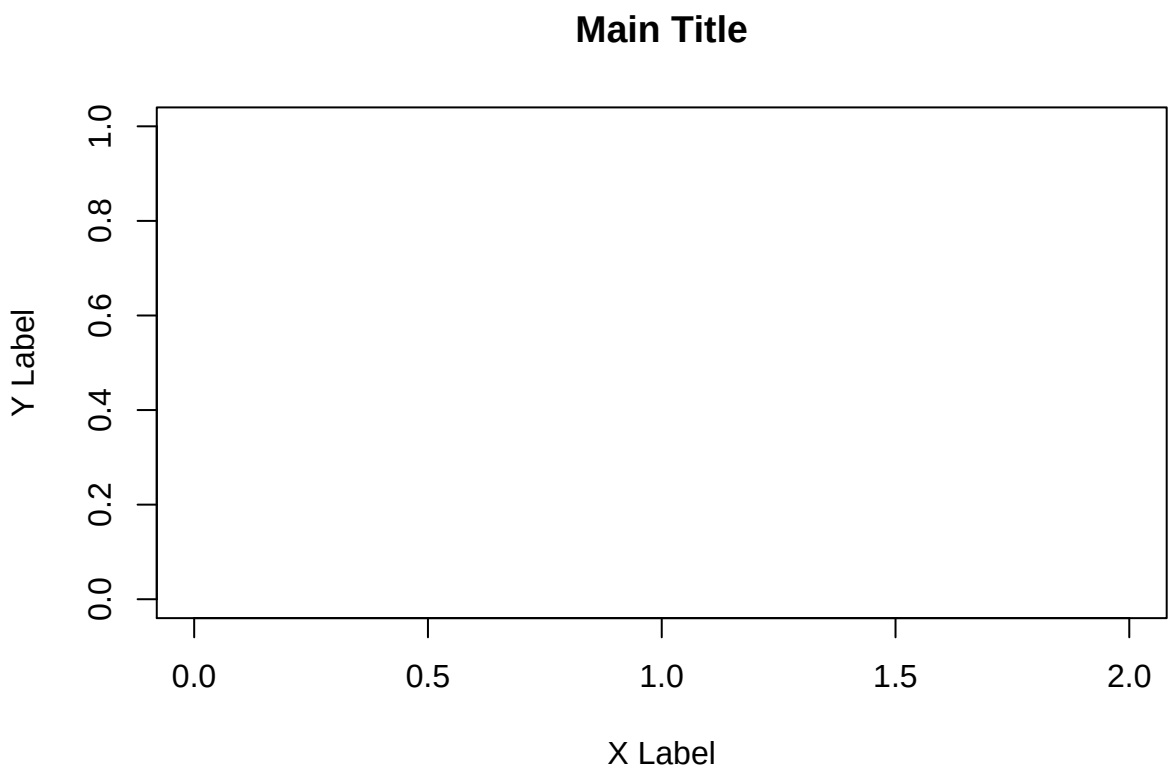
5.3.1 Blank plots

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

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

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

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

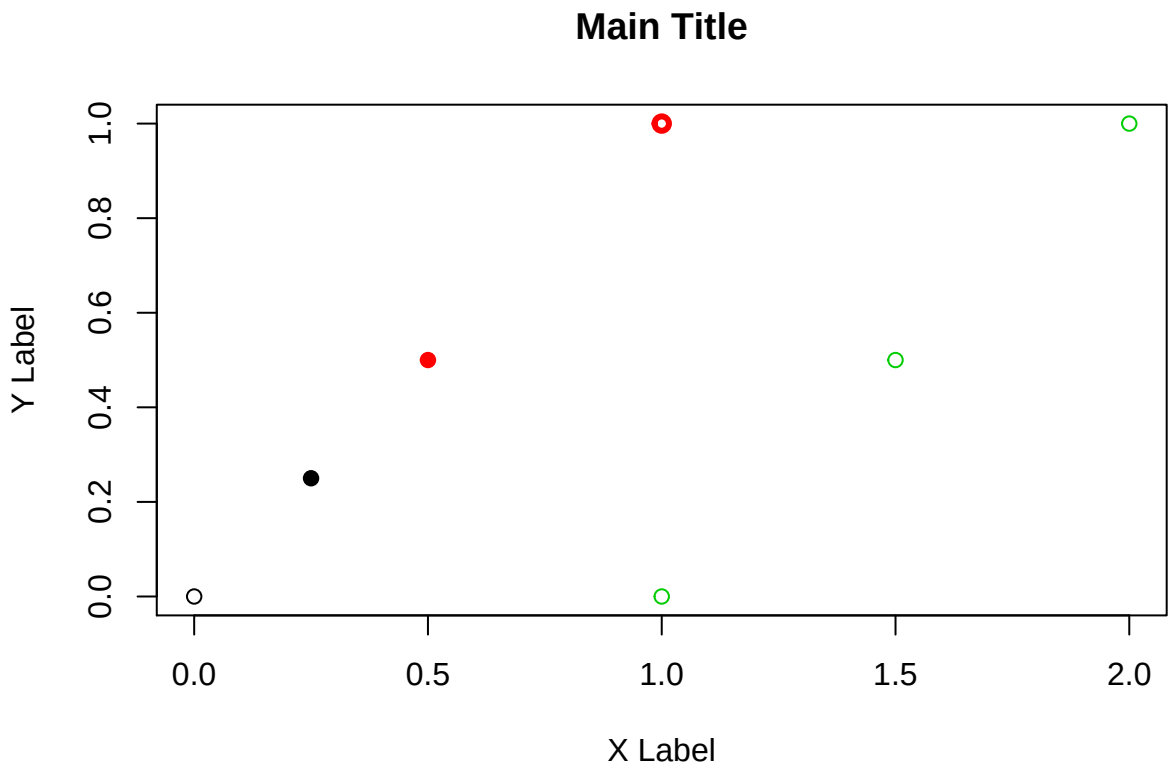


5.3.2 Points

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

- `pch` - point character. This can either be a single character or an integer code for one of a set of graphics symbols.
- `col` - Color of point. This can be a character string or an integer code. The default is black which is also #1. #2 is red and #3 is green.
- `lwd` - Line width for drawing the symbols

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



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

5.3.3 Lines

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

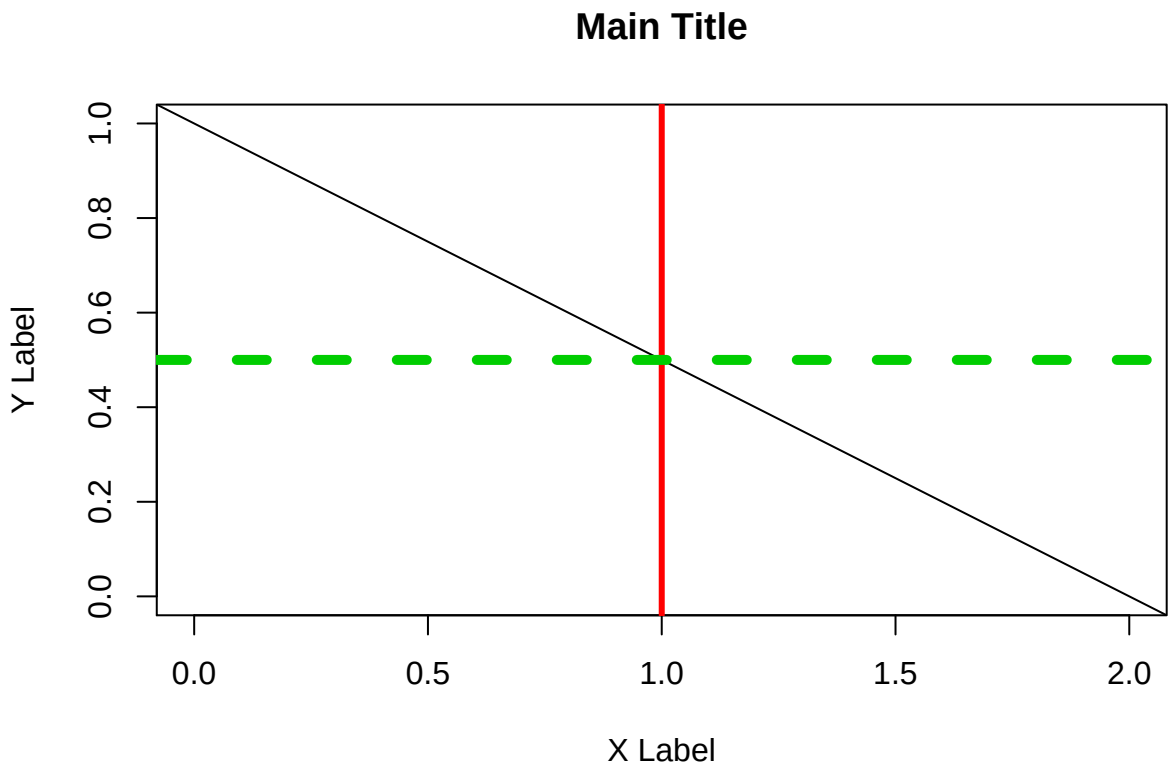
Arguments that affect the line characteristics are

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

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

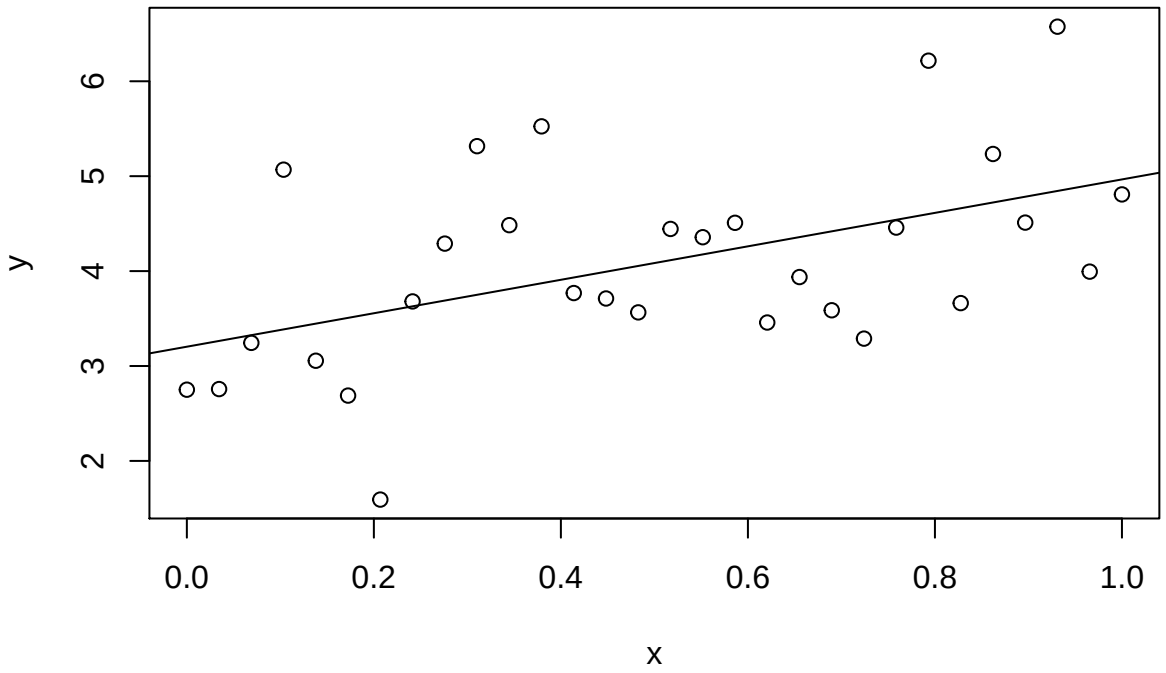
- `lty` - Line type. 1 is a solid line, 2 is dashed, 3 is dotted.

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



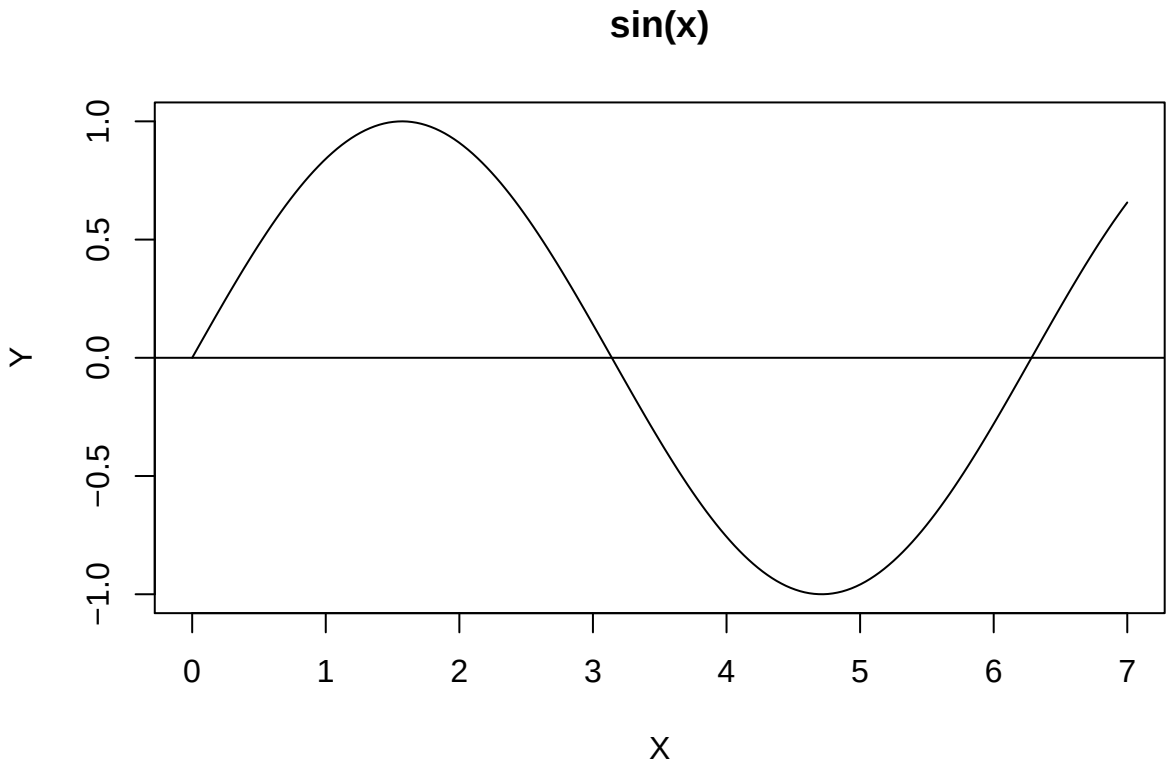
The way that `abline()` is defined, it is very convenient to add the least-squares regression line to a plot of data points.

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



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

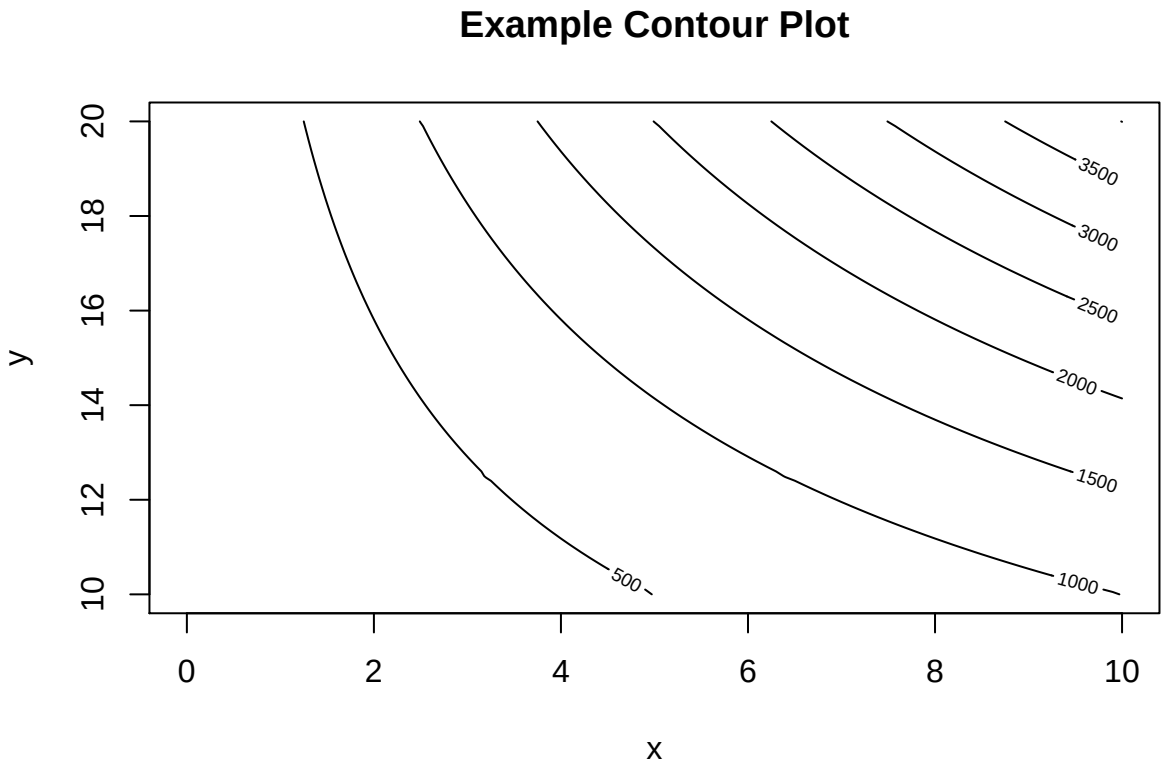
```
plot(NA, NA, xlim=c(0,7), ylim=c(-1,1), type='n',    # again a blank plot
    xlab='X', ylab='Y', main='sin(x)')
x <- seq(0,7, length=201)                            # 201 values from 0 to 7
lines(x, sin(x))                                     # sin(x)
abline(h=0)                                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



5.3.4 Contour Plots

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

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



Chapter 6

Matrices, Data Frames, and Lists

6.1 Matrices

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
M1
```

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

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

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

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

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

6.2 Data Frames

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

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

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

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

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

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

Because the columns have meaning and we have given them column names, it is desirable to want to access an element by the name of the column as opposed to the column number.In large Excel spreadsheets I often get annoyed trying to remember which column something was in and muttering "Was total biomass in column P or Q?" A system where I could just name the column Total.Biomass and be done with it is much nicer to work with and I make fewer dumb mistakes.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.3 Lists

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
results$p.value
```

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

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

```
results
```

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

6.4 Exercises

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

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

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

2. Create and manipulate a data frame.

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

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

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

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

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

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

```
summary(m)
```

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

Bibliography