R Introduction

Jared DiDomenico

Spring 2020

R as a Calculator

To begin, it's nice to get to know the R environment by doing some simple calculations.

```
1+1
```

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

```
2*2
```

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

```
50+2*4
```

```
## [1] 58
```

```
(50+2)*4
```

```
## [1] 208
```

```
10^2
```

```
## [1] 100
```

```
exp(1)
```

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

Vectors

We'll want to work with more than a few values at a time, so we can create a vector to work with many values at once.

```
x <- c(1,2,3) # the c is short for combine
x
```

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

```r
y <- 1:3
y
```

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

```r
z <- 1:50
z
```

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

Note that the [1] in the code output is the index of the first value within the line. Also, notice that in the code above I made a comment. The "#" makes comments in the code which allows the user to make notes that are ignored by R.

Most of the time we will have the data ready in files to load into R. We will visit this later. However, we may want to quickly generate some data to mess around with. One function is "seq()" which generates a sequence of values. The inputs for this *function* ask for a value to start from, end at, and the length of the sequence. You can also specify it to go from the starting value to the ending value by a particular number

```r
seq(from = 1, to = 10)
```

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

```r
1:10 #another way to do it
```

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

```r
seq(from = 1, to = 10, length = 10) #length tells us how many values are generated in this sequence
```

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

```r
seq(from = 1,to = 100, by = 20) #by generates a sequence that starts at your desired value and increase
```

```
## [1]  1 21 41 61 81
```

```r
seq(1, 100, 20) #by is the default 3rd input, so length needs to be specified
```

```
## [1]  1 21 41 61 81
```

```r
seq(1,100, length = 20)
```

```
##  [1]   1.000000   6.210526  11.421053  16.631579  21.842105  27.052632
##  [7]  32.263158  37.473684  42.684211  47.894737  53.105263  58.315789
## [13]  63.526316  68.736842  73.947368  79.157895  84.368421  89.578947
## [19]  94.789474 100.000000
```

Another useful function is "rep()". This will replicate a value or vector a specified number of times.

```r
rep(7,3)
```

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

```r
rep(c(1,2,3),3) # replicating vectors can be very useful for generating data
```

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

Another useful way to generate data is by creating random numbers. R has several built in distributions, and to generate values from these distributions we will use "r" followed by the specified distributions. For instance, let's get a random sample of 10 values from a standard normal population.

```r
rnorm(n=10, mean = 0, sd = 1)
```

```
##  [1] -0.3114455  0.6791989 -0.5244563 -0.8534663 -0.2323497 -2.1483668
##  [7] -1.7002920 -0.5682020  0.7056588 -1.2080134
```

This can be done with tons of different distributions such as the exponential, gamma, uniform, binomial, etc. Each distribution will have a different function in R. You can search for these by using the help menu in the bottom right corner section in R studio.

Vectors do not have to be numerical values, they can also be words, names, or any non-numerical character. To create these, wrap your text in quotations and R will read this as a character vector.

```r
x <- "Jared"
x
```

```
## [1] "Jared"
```

```r
y <- Jared # creates an error without the quotations
```

```
## Error in eval(expr, envir, enclos): object 'Jared' not found
```

```r
y
```

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

```r
The_Office <- c("Michael","Jim","Dwight", "Pam", "Angela") #Note that you cannot have spaces in the vec
The_office #Note that they are also case sensitive
```

```
## Error in eval(expr, envir, enclos): object 'The_office' not found
```

```r
The_Office
```

```
## [1] "Michael" "Jim"     "Dwight"  "Pam"     "Angela"
```

Arithmetic can be applied to the vector in total, or using vector algebra

```r
x <- 1:5
y <- rep(2,5)
z <- 1:4
a <- c(0,0,0,0,0)
x+1
```

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

```r
x+y
```

```
## [1] 3 4 5 6 7
```

```r
x*y
```

```
## [1]  2  4  6  8 10
```

```r
x%*%y #this performs matrix multiplication
```

```
##      [,1]
## [1,]   30
```

```r
a+z #because z is not the same length as a it will repeat itself over when adding through
```

```
## Warning in a + z: longer object length is not a multiple of shorter object
## length
```

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

You can also combine vectors together into a longer vector or into a matrix (If they are the appropriate dimensions).

```r
x <- 1:5
y <- 6:10
z <- c(x,y)
z
```

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

```r
z <- cbind(x,y) #creates matrix using the vectors as its columns
z
```

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

```r
z <- rbind(x,y) # creates matrix using the vectors as its rows
z
```

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

Functions

There are TONS of functions that you'll end up using in R. Some are intuitive, some you should commit to memory, and some you'll rarely use. Here are a few that I think are good to know for this class. If you don't know how to do something in R or you want to learn a particular function, then the help menu and google are your best friends.

```r
data <- 1:10
mean(data) # returns the arithmetic average of a vector
```

```
## [1] 5.5
```

```r
var(data) # returns the variance of a vector
```

```
## [1] 9.166667
```

```r
sd(data) #returns the standard deviation of a vector
```

```
## [1] 3.02765
```

```r
sum(data) #retuns the sum of the vector
```

```
## [1] 55
```

```r
prod(data) #retuns the product of the vector
```

```
## [1] 3628800
```

```r
summary(data) #gives the five number summary
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    3.25    5.50    5.50    7.75   10.00
```

Indexing and Logic

Most of the time you're going to have large vectors and datasets. It's particularly useful to find specific values within the vector, and their locations (called an index). You can easily find the value in a specified index of the vector by using the closed brackets "[]".

```r
x <- seq(1,10, by = 2)
x
```

```
## [1] 1 3 5 7 9
```

```r
x[2] # finds the value in the 2nd position
```

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

```r
x[c(2,4)] #finds the value in the 2nd and 4th position
```

```
## [1] 3 7
```

```r
x[-c(2,4)] #finds the values that are not in 2nd and 4th position
```

```
## [1] 1 5 9
```

Indexing matrices works in a very similar way, all you need to do is specify both the row (the first value) and the column (the second value) that you want to see.

```r
z <- matrix(data = 1:16,nrow = 4, ncol = 4) #notice it sorts the vector into columns
z
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

```r
t(z) #transposes a matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
```

```r
z[1,1]
```

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

```r
z[1,2]
```

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

```
z[2,1]
```

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

```
z[,1] #leaving the row value blank returns all the values in column 1
```

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

```
z[1,] #leaving the column value blank returns all the values in column 2
```

```
## [1]  1  5  9 13
```

Well what if I wanted to know which values in the vector are smaller than a value or larger than some value? R allows us to do this by returning True and False for each index and whether in meets the desired condition.

```
x <- seq(1,10,by = 2)
x
```

```
## [1] 1 3 5 7 9
```

```
x[x > 3]
```

```
## [1] 5 7 9
```

```
x[x < 0]
```

```
## numeric(0)
```

```
x[x == 3]
```

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

```
x[x <= 3]
```

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

```
x[x >= 3]
```

```
## [1] 3 5 7 9
```

```
x[x != 3] # the ! is used to mean "not"
```

```
## [1] 1 5 7 9
```

Boolean operators are also within R, and can help find more specified conditions.

```r
x <- seq(1,10)
x[x > 5 & x <= 7] #The "&" works as an "and".
```

```
## [1] 6 7
```

```r
x[x > 5 | x <= 7] # The "|" works as an "or"
```

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

Sometimes the vectors are too large and generating a string of True and False is just too cumbersome. Luckily, we can locate the indexes that meet conditions of interest using the "which()" function and indexing.

```r
set.seed(123) # this sets a seed from which a random sample is generated. VERY USEFUL for when you're g
x <- rnorm(100,0,1)
x[1:10] #See the first 10 values of the vector
```

```
##  [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774  1.71506499
##  [7]  0.46091621 -1.26506123 -0.68685285 -0.44566197
```

```r
x > 2 # not helpful!
```

```
##   [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
##  [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
##  [73] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [85] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [97]  TRUE FALSE FALSE FALSE
```

```r
which(x > 2) # this output is a vector so we can use it as such
```

```
## [1] 44 70 97
```

```r
x[which(x>2)]
```

```
## [1] 2.168956 2.050085 2.187333
```

```r
x[x>2] # also works for searching for specific values
```

```
## [1] 2.168956 2.050085 2.187333
```

NA's

Missing data will occur in datasets and is just a consequence of data input. R usually returns NA whenever a calculation including an NA occurs. Typically, functions have an option to ignore the NA's during calculations "na.rm = T". Note, that here T in R means True.

```r
x <- c(1,2,3,4,NA)
sum(x) # the calculation cannot be performed
```

```
## [1] NA
```

```r
sum(x, na.rm = T)
```

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

Datasets

R will most often be used to analyze data sets in this class, and to make plots to examine these data. There are several ways to read in a data set to the R environment. The easiest way is the "read.table()" function. From here there are many places where the data is located. If its online then all you need is the exact web address.

Otherwise, you'll need to tell R the path in your computer where the data file is, and its file name. To easily access a particular folder where you will be working from an saving in, use "setwd()", the set working directory. Put in the path in the function and R will access from it.

Finally, some base packages in R already have data ready to be used in the Global Environment. To install packages into R use the "install.packages()" function in R. Within the parentheses of this function, be sure to wrap the name of the package in "". e.g. install.packages("ggplot2") to install ggplots, a very useful plotting package.

For now, lets just use the cars package to get some data; all you need to do is input "cars" and the dataset will be available.

```r
cars[1:10,] # examines the first 10 values in the dataset
```

```
##    speed dist
## 1      4    2
## 2      4   10
## 3      7    4
## 4      7   22
## 5      8   16
## 6      9   10
## 7     10   18
## 8     10   26
## 9     10   34
## 10    11   17
```

```r
head(cars) # works as well
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

```
cars$speed [1:10] # the "$" extracts vectors out of the dataset
```

```
##  [1]  4  4  7  7  8  9 10 10 10 11
```

```
cars$dist [1:10]
```

```
##  [1]  2 10  4 22 16 10 18 26 34 17
```

We can do all sorts of analyses with the datasets. For instance, let's see what is the correlation is between speed and distance, and generate a 95% confidence interval for the car's speed.

```
cor(cars$speed,cars$dist, method = "pearson")
```

```
## [1] 0.8068949
```

```
confint(cars$speed, level = 0.95)
```

```
## Error: $ operator is invalid for atomic vectors
```

Now, most of these functions will be memorized with time and practice. If you're wanting to find particular functions for certain analyses, use the help menu. (Again, Google is also your best friend)

Let's create our own data set. From a sample of 50 students in an introductory statistics class, it was found that 14 people liked rock music, 19 people liked rap music, 5 people liked country music, and 12 people liked pop music. Let's take this information and create out our own data set. Here, we will use the "data.frame()" function. There's two ways to summarize these data here, let look at them in turn.

```
rock <- 14
rap <- 19
country <- 5
pop <- 12
musicdataset1 <- data.frame(rock,rap,country,pop)
musicdataset1
```

```
##   rock rap country pop
## 1   14  19       5  12
```

```
## OR ##

rock <- rep("rock",14)
rap <- rep("rap",19)
country <- rep("country", 5)
pop <- rep("pop", 12)
musicdataset2 <- data.frame("music_type" = c(rock,rap,country,pop))
musicdataset2[c(1:3,20:22,34:37,39:41),]
```

```
##  [1] rock    rock    rock    rap     rap     rap     country country country
## [10] country pop     pop     pop
## Levels: country pop rap rock
```

R typically can work with either of these. There are some functions in different packages that are not pre-built or pre-installed in R that can change the one of the data set structures I have above to the other and vice versa. It likely won't come up until Advanced Data Analysis but for those interested it is the melt function in the "Reshape" package.

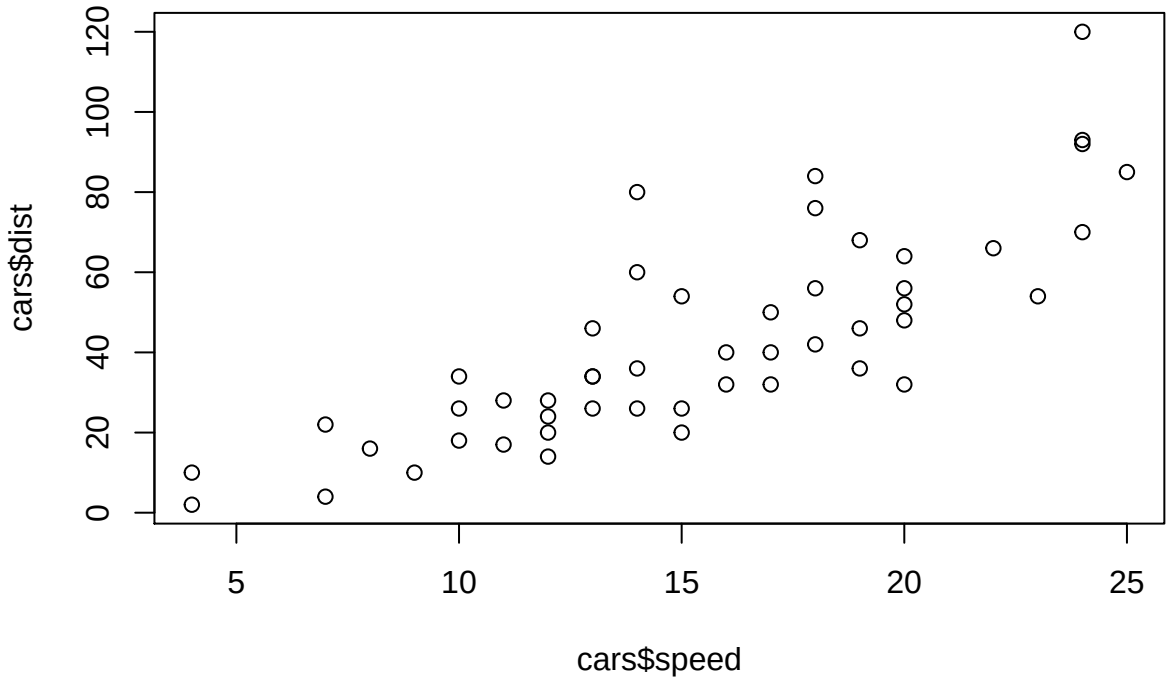
Plotting

Examining the data is also very important in statistics, and R can make some very nice plots. You will likely see HW problems that ask you to plot some functions in R, so here are some examples to make the basic plots you learned in intro stats.

Scatterplots

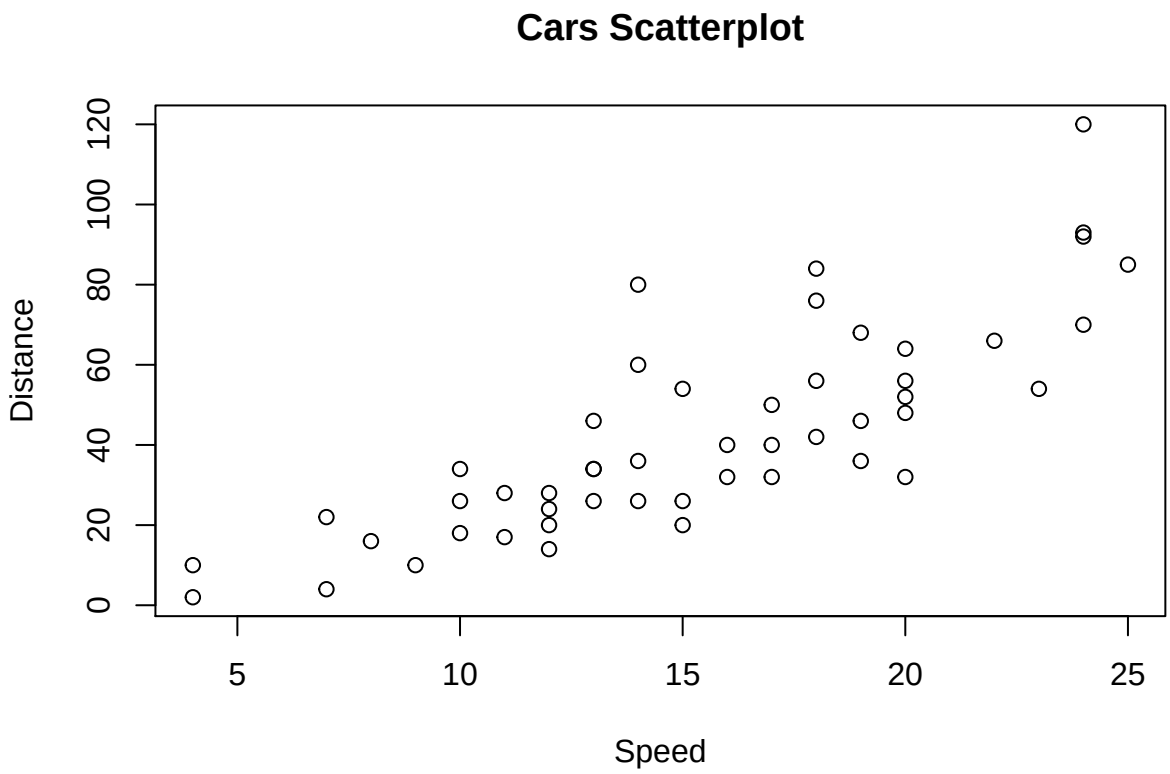
The "plot()" function in R makes scatterplots of two quantitative (numerical) variables. Let's see the base plot using the cars dataset.

```
plot(cars$speed,cars$dist)
```



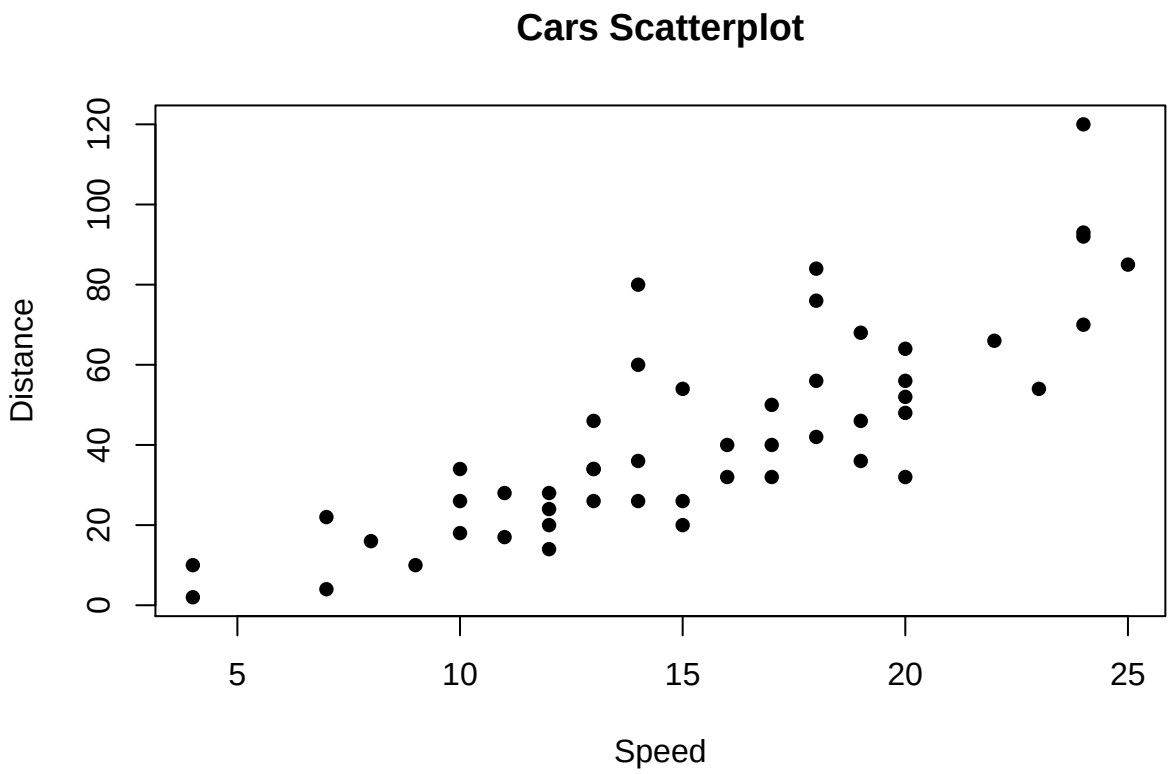
Doesn't look half bad. But there are TONS of inputs that can adjust various aspects of the plot within the plot function. For instance, we can use "xlab", "ylab", and "main", to adjust the axis labels and give a title to the plot.

```
plot(cars$speed,cars$dist, xlab = "Speed", ylab = "Distance", main = "Cars Scatterplot") #note the "" a
```



How about we adjust those points using "pch" within the plot function. (Use the help menu and search pch to see all the options available to change to)

```
plot(cars$speed,cars$dist, xlab = "Speed", ylab = "Distance", main = "Cars Scatterplot",
     pch = 16)
```

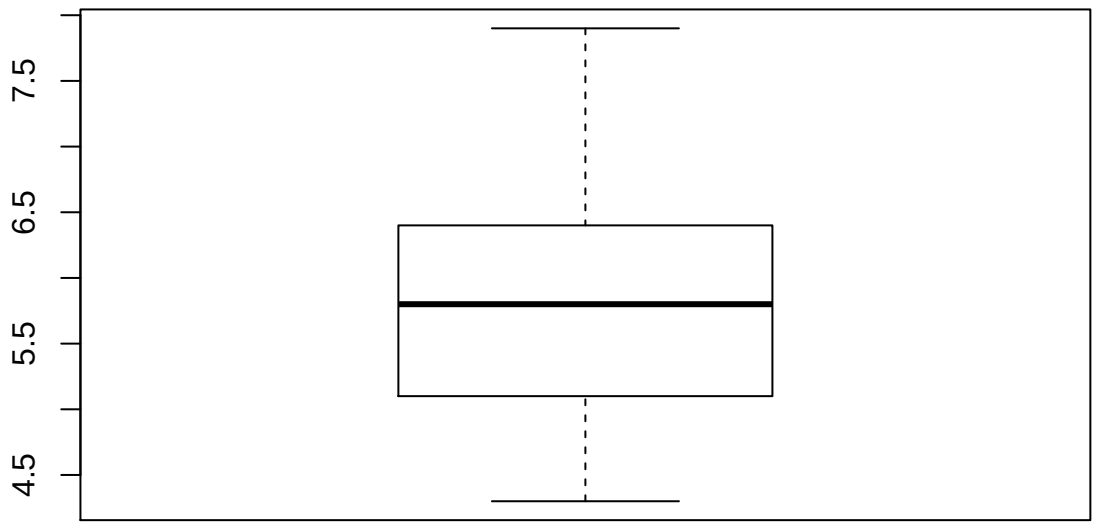


Boxplots Boxplots are very useful for examining the distribution of a single predictor or variable of interest. Here, we will use the "boxplot()" function. For this, let's use the "iris" data set about flowers. It's already in R.

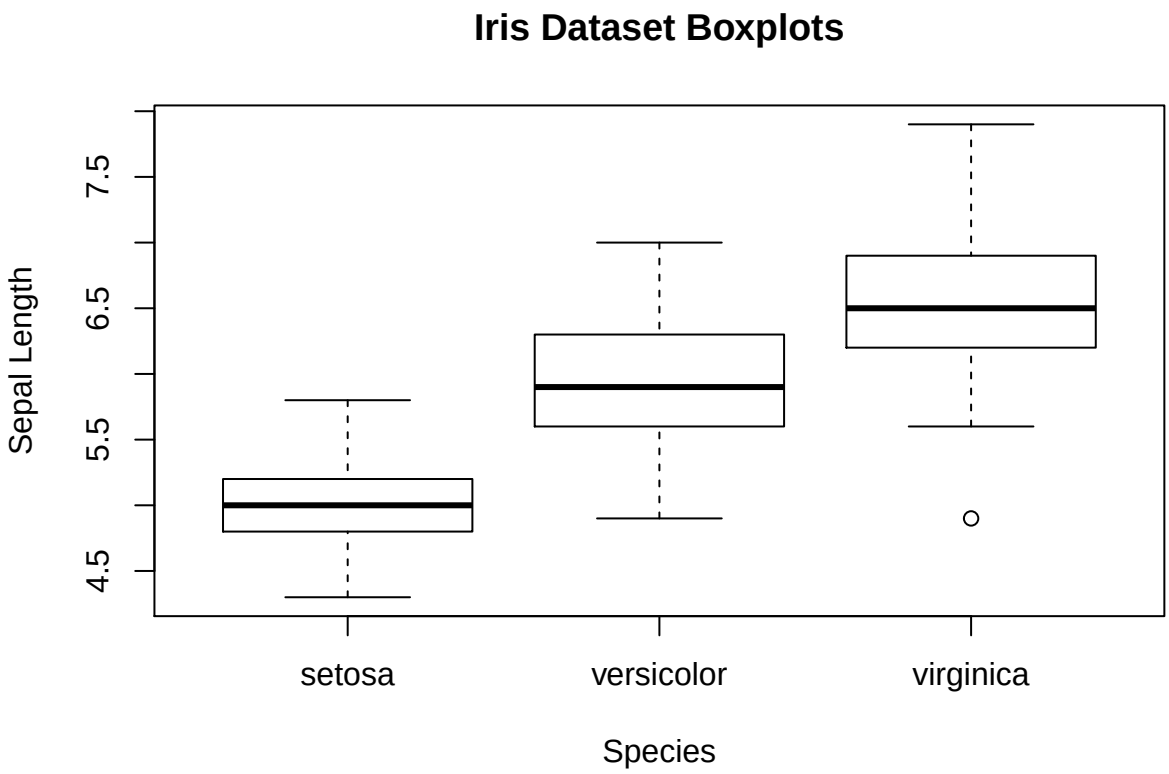
```
head(iris)
```

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

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

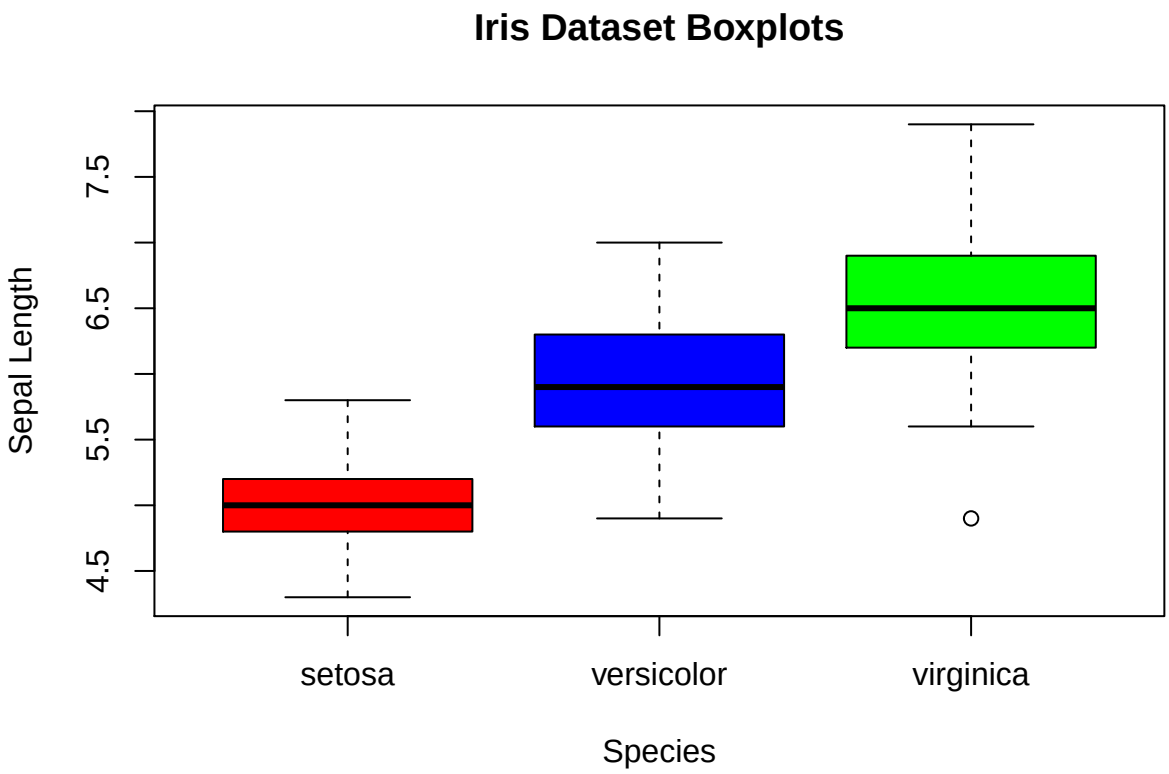



```
boxplot(iris$Sepal.Length~iris$Species, xlab = "Species", ylab = "Sepal Length", main = "Iris Dataset B
```



Not too bad, lets add some color to the plots. We can use the "col" input to adjust the colors. Because there are three colors, we want to specify three different colors for the plot. Note, R has A LOT of base colors.

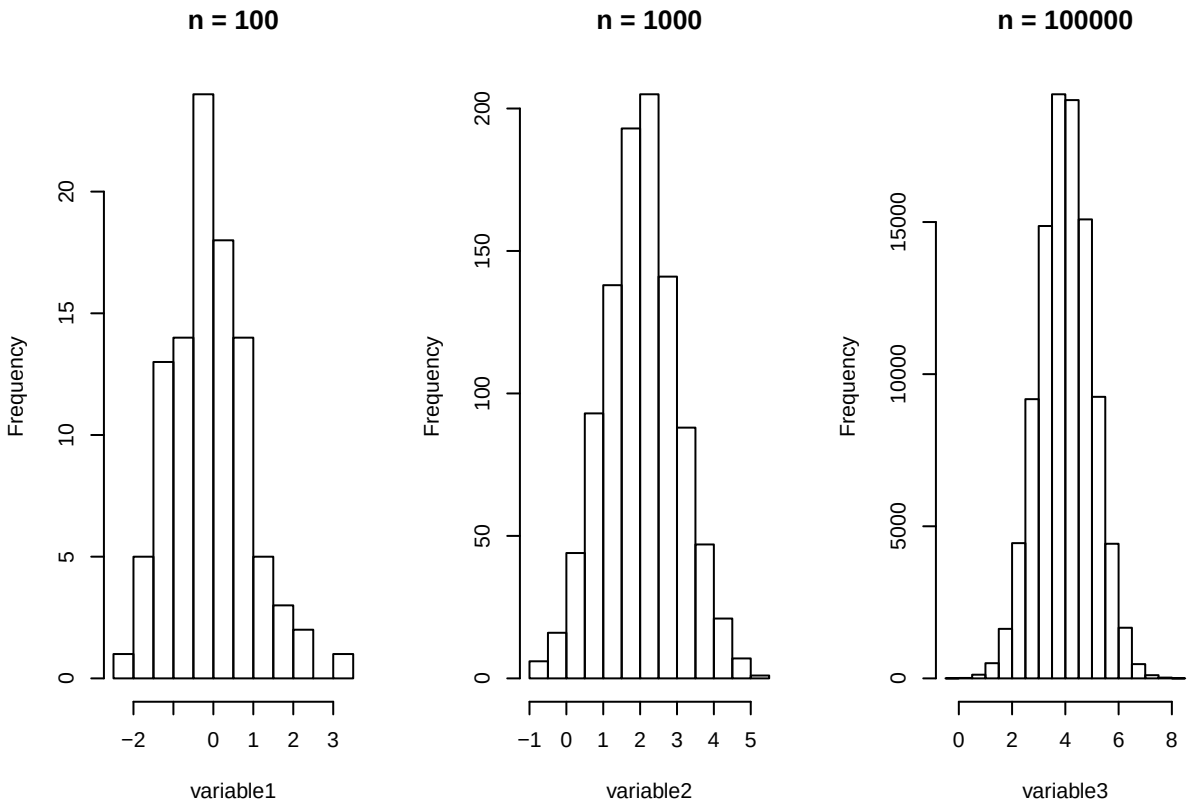
```
boxplot(iris$Sepal.Length~iris$Species, xlab = "Species", ylab = "Sepal Length", main = "Iris Dataset Bo
        col = c("red", "blue", "green"))
```



Histograms

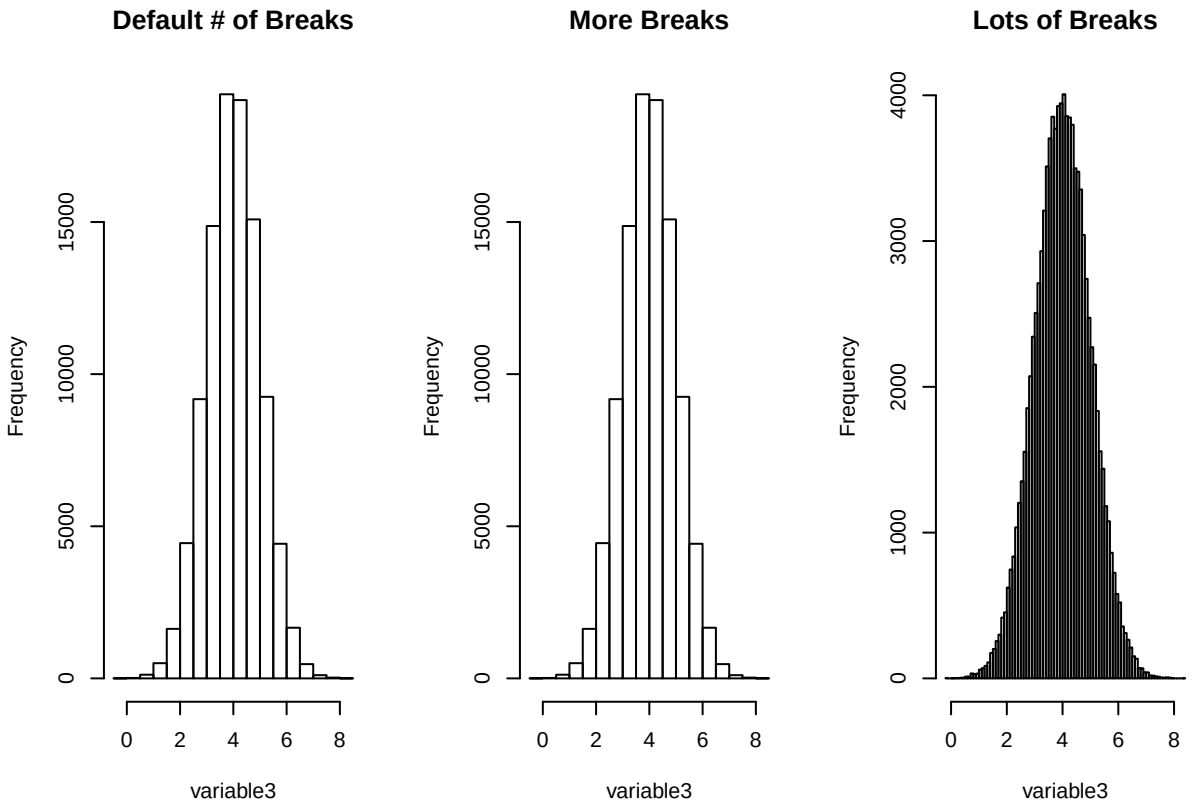
Let's make up some data to build some histograms (the "hist()" function). The same *graphical parameters* are used in these plots too (e.g.,"col", "xlab", "pch", etc.). I want to demonstrate how graphs change as the sample size increases, so when I generate the following data, I will make increasing sample sizes. I want to see all the plots at the same time, so I will use "par()" to adjust how many plots are displayed in R at once. The input that needs to be adjusted is "mfrow" is a matrix, with the default of 1 by 1. I want to see 3 at once so I will adjust it to 1 (row) by 3 (columns).

```
variable1 <- rnorm(100, mean = 0, sd = 1)
variable2 <-rnorm(1000, mean = 2, sd = 1)
variable3 <-rnorm(100000, mean = 4, sd = 1)
par(mfrow = c(1,3)) #1 row by 3 columns
hist(variable1, main = "n = 100")
hist(variable2,main = "n = 1000")
hist(variable3, main = "n = 100000")
```



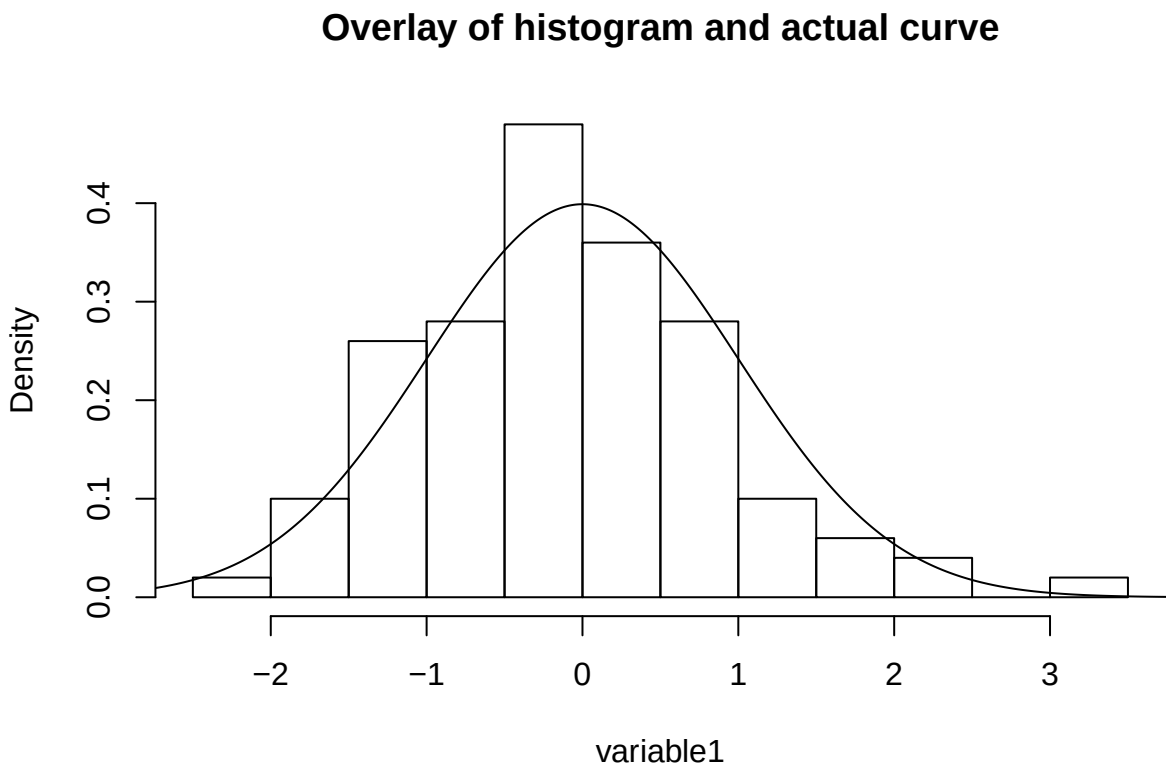
The difference is hard to see because we have the same number of bins for each plot. The larger your sample size, the more bins you can have with reasonable detail. To adjust the bins, use the "breaks" input within the "hist()" function.

```
par(mfrow = c(1,3))
hist(variable3, main = "Default # of Breaks")
hist(variable3, breaks = 20, main = "More Breaks")
hist(variable3, breaks = 100, main = "Lots of Breaks")
```



You can really start to see the shape better as the number of breaks increase. Well we know that variable #1 should be a normal distribution with mean = 0 and sd = 1. So let's use the "lines()" function to see how close our data is to the actual function.

```
par(mfrow = c(1,1))
hist(variable1, main = "Overlay of histogram and actual curve", freq = F) # freq = F gives the area und
x <- seq(-4,4, length = 10000) # gives the x values of the function I want to overlay
lines(x, dnorm(x, mean = 0, sd=1), col="black") # the "d" in "dnorm" means the density. So with small e
```



Most other plots and graphical parameters can be found using the help menu or a simple google search. Remember, R takes some time to learn and you will get error messages (typically when you forget a comma somewhere, the syntax is off, or vectors are the wrong size). But be patient and get lots of practice and most of this stuff will start to come to you more intuitively.