Chapter 1
An Introduction to R

As I mentioned in the start of this course, we will be using a statistical software to analyze our data. The software we will be using is **R**. This software is free and available for download. It has a fairly low learning curve, is engineered for statistical analysis and offers features that make it easy to extend the functionality for personal needs. Many analysts have even written their own extensions and are available online for free to download. In our course, we will only be using the basic features of R as well as some common extension packages that handle specific types of problems.

The first step is to download R. This can be done by going to https://cran.r-project.org. The website should automatically point you to the download site. If you are using a Mac, after clicking "Download R for Max OS X", you should click on the most up to date version. If install fails, then you will need to point your browser to an earlier version of the software. At the time of this writing, the most recent version is R-3.5.1.pkg.

For Windows Users, the install is fairly straightforward. Click on "Download R for Windows". This will bring you to a subdirectory page that has many options. The only package you need to download is the package named "base". After clicking on that, you will see a link that is labeled as "Download R 3.5.1 for Windows" on a new page, click this link and it will download the software. For our needs, both for mac and windows, we only need the default options. All you need to do when you install the software is to keep clicking "Next". After it is installed, you are ready to go!

I will be going back and forth in this course. Another alternative to the raw R software is to use RStudio. RStudio offers a much more user-friendly interface to R. You can download it and install it from https://www.rstudio.com/. A third option is a bit more advanced, and so I will not go into too much detail. However, this option will allow you to access R from any device that has a browser, and will save your work if you happen to close out the browser (assuming your virtual machine is still running). You could install RStudio Server on an Amazon EC2 Instance. Amazon EC2 Instances are *virtual machines* that you can quickly get up and running on Amazon's hardware. Typically, when you sign up for an Amazon AWS (Amazon Web Services) account, the first year is free. Their options of virtual machines are limited. Despite this, this is still a very low-cost option, and should not cost you more than $10/month (ASSUMING YOU STOP YOUR MACHINE). I only recommend this option to those of whom are more advanced with Comp Sci and IT, as the install process can be grusome for those of whom have little IT or Comp Sci background. You can go to https://www.r-bloggers.com/instructions-for-installing-using-r-on-amazon-ec2/ for a good tutorial on how to get it set up. Once it's done, your work throughout the semester should be a walk in the park!

Once R is installed, you open it up and you will be brought to a very simple interface with a box that has a blinking cursor. Welcome to R! It is as simple as that. It has no fancy

bells or whistles visually, but I can assure you, underneath it is one of the most powerful statistical softwares you will ever use in your life. In this section we will give a brief overview of R. We will review through some of the fundamental basics in working within R as well as introduce the concept of a variable and a function. Both of these tools are important to be able to conduct any task within R. Without the knowledge of these two ideas, you will be lost the remainder of the course. So ensure that you fully understand them!

1.1 The Basics

We can think of R as an over-glorified calculator. Similar to many other types of more advanced calculators, R has as it's interface a simple console to input commands, press the return key, and output a corresponding response. This may seem like an outdated type of software, however, R's capability of performing advanced statistical analysis is quite incredible. In many regards, data analysis is much easier to conduct with software similar to R than using a spreadsheet-based software.

When you first open R, you are brought a screen with a blinking cursor. This is the console and it should like this:

```
R version 3.2.3 (2015-12-10) -- "Wooden Christmas-Tree"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

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

  Natural language support but running in an English locale

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

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

>
```

In here and next to the ">", you type commands and R will give you a result. To start, we can type simple mathematical inquiries such as multiplication, addition, subtraction and division. You first type your inquiry in and press the return key. R then gives you a result next to the symbol [1]. We will discuss the meaning of this [1] at a later time. Here are some examples of inputs and subsequent outputs provided into and out of R:

```
> 3+3
[1] 6
> 4-1
```

```
[1] 3
> 4*3
[1] 12
> 1/5
[1] 0.2
```

As we can see, R is able to handle basic mathematical inquiries. In addition to the four basic arithmetic operators, R can also handle exponents and remainders of division:

```
> 5^3
[1] 125
> 6%%4
[1] 2
```

Basic mathematics is not all R can handle. It can perform a wide variety of tasks. We will see this in the subsequent sections in this set of notes as well as future sets of notes. The key idea to take away from this section, however, is that R operates as an input-return-output type of system. In other words, it is an interactive system. Let us now discuss some other features of R that we will find useful in our future analysis.

1.2 Variables

As we had seen at the start of this section, R can handle a variety of basic mathematical operations. So far, however, R just simply gives us an output as a result of an inquiry. We currently have no way of keeping track of this output other than to run the same query again. In R, as we will see as we progress in this course, we need some way of "storing" information. We need a mechanism to store the results of questions we ask R so that we can either use the outputs in subsequent calculations or be able to refer back to it again. The mechanism in R that allows us to do just that is a *variable*.

Separate from our earlier discussions of research variables, a variable in R is nothing more than a placeholder. Think back to your wonderful days in Algebra where you learned the concept of a variable. Such a concept is at the heart of Algebra. You represented numbers with letters and then subsequently used those letters in more advanced equations to represent "something". You then had the ability to replace every occurrence of the variable with a number and calculate an answer. The same idea in Algebra is applied to the idea of a variable in R.

Unlike Algebra, however, R allows you to store whatever you wish in a variable. Think of a variable as a box. You can put not only numbers in this box but anything else. You then can move the box from one place to another and finally unpack the box when whatever is stored in it is needed for calculation or an inquiry.

We will introduce variables using numbers only. First, to use a variable we must have something assigned to the variable. This is a process known as *declaring and assigning* the variable. In order to declare and assign, you must first choose a *variable name*. Just as in Algebra where you represented "something" with a unique and subjectively chosen letter, the same works in R. The difference however is that variable names need not be just single letters. Other than a few exceptions and rules for naming your variables, you can choose any name you please. Once you decide on a variable name to use, you then must store the value in the variable. This is done by first typing the variable name into R, followed by an

"=" sign, followed by the value to be stored in the variable. When a value is assigned to a variable, we can then query R as to what value is stored in there by typing the name of the variable into the console and pressing the return key. Here is a small example of a variable being assigned to a value and the variable being queried:

```
> x=4
> x
[1] 4
```

As you can see, the value 4 is stored in the variable x. We then ask R to show us the value stored in the variable x by typing in the variable name and pressing return. Once a value is stored in a variable, we can use that variable in a variety of mathematical equations. We can conduct all of the same type of arithmetic as we had done with numbers, only this time involving the variable:

```
> 2*x
[1] 8
> x*7
[1] 28
> x-1
[1] 3
> x+9
[1] 13
> x^2
[1] 16
> x%%8
[1] 4
> 4*(x-1)^x
[1] 324
```

If a variable is used in an equation as input into R, that variable MUST be defined first. If it is not, then R will not know what you are referring to. For example, suppose we used the variable y in an equation, then we would obtain the following error:

```
> 4*y-1
Error: object 'y' not found
```

Only after assigning y to a value will R be able to compute the equation:

```
> 4*y-1
Error: object 'y' not found
> y=4
> 4*y-1
[1] 15
```

Not only can we use a single variable in equations, but we can also use multiple variables:

```
> x+y-4*2/(3*z-1)
[1] 9
```

So far we have only used single letters for variable names. Variable names need not be restricted to using only single letters however. For example, we can store and calculate numbers in (almost) any variable name we choose, say *thevalue*:

```
> thevalue=4
> thevalue*2-(1/thevalue)
[1] 7.75
```

Naming variables is a subjective process when working within R. The choice is up to you. A general guideline for giving variables names is to use a name that is clear, consistent and easily understood. There are some minor guidelines that R requires in a variable name however:

1. A variable name cannot contain any spaces.
2. A variable name cannot start with a number.
3. A variable name cannot use a reserved word.

The following is a list of names that cannot be used as a variable name:

```
if else repeat while function for in
next break TRUE FALSE NULL Inf NaN NA
NA_integer_ NA_real_ NA_complex_ NA_character_
```

If we attempt to break these rules, R will give an error message in response:

```
> the variable = 4
Error: unexpected symbol in "the variable"
> 5var=3
Error: unexpected symbol in "5var"
> else=2
Error: unexpected 'else' in "else"
```

As we can see, the first attempt contains spaces. The second attempt starts with a number. The third attempt used a reserved keyword. In all of these we are given an error message. In addition to this, we must be aware that variable names are *case sensitive*. For example, we can see below that the variable *var1* is different than the variable *Var1*:

```
> var1=4
> Var1=8
> var1
[1] 4
> Var1
[1] 8
```

Variables are frequently used when conducting work within R. Not only do that allow us to store numerical values, but we can also store strings or more complicated data structures, as well see momentarily. A common application for using variables is to store the output that R provides as a result of a calculation or (as we shall see in the following section) return values from function (procedure) calls. This is illustrated below:

```
> m="Male"
> f="Female"
> m
[1] "Male"
> f
[1] "Female"
> x=6
> y=3
> z=x+2*y-5
> z
[1] 7
```

1.3 Functions

While being able to compute basic mathematical equations in R is useful for some applications, we need additional ability to handle more complicated tasks. At times, we need to call a *procedure* to conduct some type of work on a given set of data or information. For example, a common task in data analysis on a numerical data set is to compute the mean, standard deviation and variance of the data. We could manually type in the data and use the equation from our standard statistics course, but this would take us too long for every data set we encounter. If there were a way to automatically compute these statistics (or parameters) automatically and all we must do is provide the data itself, we would be able to drastically shorten our time working on the data analysis. That is, we want to be able to *automate* certain tasks.

Automation of tasks comes in the form of a *procedure*, or within R known as a *function*. A function is a process that the computer conducts through a list of steps using inputs provided by the user for the goal of producing an output for the user. R has many of these functions built in to the software already. We do not need to know how each of these functions work, but rather we must know three things:

1. What does the function do?
2. What inputs must we provide the function?
3. What outputs does the function provide?

The approach of not knowing *how* a task is performed but rather of *what* task is performed is commonly referred to as the *black-box* approach. For example, in an automobile manufacturing plant, one step in the assembly line is the painting of the vehicle. Maybe, the paint workers need to partially paint the car, have tires put on the car, and have the car returned back to them to finish painting.

The action of "sending the car to the tire line" is analogous to us "calling a function". The "input" in our example would be the partially assembled car and the tires. The output would the be car with four tires on it. Now, the paint line does not care about *how* the tires are put on the car. The only thing they care about is (1) what should they send to the tire line, (2) what does the tire line do and (3) what output does the tire line provide? The paint guys do not care much about the substeps or procedure of how the tire guys

put on the tires. All they care about is the output, input and the purpose of the step in production.

We have a similar idea in R. At times, we would like R to conduct a series of steps to "do something" but we may not really care too much as to "how" this "something" is completed. The only thing we care about is the function that performs a given task, the inputs required for the function to manipulate and use to provide an output.

In R, a function is "called" by first specifying the *function name*. We can think of the function name as something similar to variable in R. It has a unique name that refers to a well defined list of steps it will conduct. Then, inputs are specified separated by commas. Last, the return key is pressed by the user and R will subsequently take the specified inputs and go through the steps defined in the function. Most often than not, we do not know what steps are carried out, nor do we really even care. We must know however what type of output we are expecting.

For example, suppose we would like to find the square root of the number 5. We can accomplish this by using the square root function, which is named *sqrt*. Now we don't care about *how* R will calculate the square root of 5, but all we care about is what will the function *sqrt* do to an input we provide it? By definition, it computes the square root of a number. There is only one input for this function: the number to find the square root of. Once we specify the name of the function, in our example *sqrt*, we need to use the symbol "(" to tell R that we are about to provide a list of inputs. In the case of the sqrt function, we only have a single input: the number to find the square root of. Once we list all of our inputs, we end our call with the symbol ")", to tell R that we are done listing our inputs. After pressing the return key, R will take your inputs and run through the list of steps (that we have no idea what each step does or what the steps are) to render an output that it provides on the next line of the console. Here is an example of how we place a function call:

```
> sqrt(5)
[1] 2.236068
```

Generally, a when we call a function, it has the following structure:

```
function_name(input_1,input_2,...,input_n)
```

In R there are thousands of functions that accomplish a lot of different tasks. Many of these are the basic mathematical functions, some of these demonstrated below:

```
> sqrt(5)
[1] 2.236068
> exp(4)
[1] 54.59815
> sin(3.14)
[1] 0.001592653
> cos(4)
[1] -0.6536436
> log(10)
[1] 2.302585
```

Mathematical functions are not the only types of functions that we can use in R. Other functions that perform a wide variety of tasks also require more than just a single input. For example, suppose we wanted to generate a sequence of numbers from 4 to 80 by skipping every 4 numbers. That is, we want R to generate a list of numbers such as 4 8 12 ... 80. We can accomplish this using the *seq* function:

```
> seq(4,80,4)
 [1]  4  8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72 76 80
```

In this function we indicated three inputs. The first input specified the number to start the sequence at, the second indicated the number to end the sequence, and the last number indicated the distance between consecutive numbers.

Up to this point we have been using functions and we have observed the output that different functions have given us. But there are many times in which we seek to re-use the output of a function. There are even times when we seek to use the function right in a mathematical expression. If we seek to reuse outputs without manually typing those outputs into a variable, one way to store the output of a function is to assign a variable to it. This is illustrated below:

```
> x=sqrt(5)
> y=exp(6)
> z=seq(4,80,4)
> x
[1] 2.236068
> y
[1] 403.4288
> z
 [1]  4  8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72 76 80
```

As one can see from the example above, the outputs of each of these functions were assigned to a variable. When we asked R to determine what value is placed in the variable, it responds with the corresponding output that each function provided with the given inputs. This is a very useful tool to keep in mind as we progress in this course, as it is quite common to reuse the outputs from different functions for other purposes.

1.4 Data Structures, Vectors and Data Frames

Up to this point we have discussed the virtue of variables and functions. Both of these features in R help accelerate our progress in data analysis. In the last section we implicitly came across yet another feature R is capable of handling: data structures. A *data structure* is any structure that holds data. Variables are a type of data structure, yet a poor one. When faced with the daunting task of analyzing data, we typically have many individual responses to many research variables.

For example, if we had 100 individuals and 5 research variables, that is 500 values we would need to store and work with in our analysis. We could go through the daunting task of assigning to each value a variable, but this seems somewhat cumbersome and would cause more issues than need be. In R, we need some way to store all of these 500 values in a logical manner that is easily accessible for later analysis.

Index	Age
1	45
2	42
3	80
4	18
5	31

Table 1.1: Age Data that is Indexed

1.4.1 Vectors

Luckily, R has many different types of structures to easily store data. We will only discuss two commonly used and relevant forms: the vector and the data frame. In R, a *vector* is an *indexed* list of numbers. An index is a position in the list. For example, if we have age data from 5 individuals, we can list out this data and assign each age value a corresponding value that itself corresponds to the position in the list. This is illustrated in Table 1.1.

Vectors allow us to store large quantities of data while maintaining a logical way to refer to specific quantities. In our course, vectors in the scope of general research projects generally correspond to research variables. For example, suppose we sent a survey out to our consumers and we had three variables: age, income and product rating. Then we can store all the responses to age in a vector, all the responses to gender in a vector and all the responses to product rating in a vector, all in distinct vectors (more on this later).

In the previous section, we had encountered a basic vector in R when we used the *seq* function. While the inputs to the function were numbers, the output itself was a vector. We later stored this vector in the variable *z*. Once we have a vector to work with in R, we can perform a variety of tasks. The most fundamental task with working with vectors is to query a value in a specific position. For example, in the vector *z* from our earlier example, we may want to know what value is currently stored in the vector at position 5. Now since our list is small enough, we can just count over 5 places from the left and observe that the number is 20. Another approach is to ask R: "what value is in the 5th position?" We can do this as follows:

```
> z[5]
[1] 20
```

When asking R what is currently stored in a vector at a given position (henceforth referred to as "index"), we first must provide the variable name the vector is stored in, followed by a "[", then followed by the index (or a vector on indexes, more on this later), followed by a "]". Suppose we wanted to change a value in the vector at a specified index. That is, suppose we wanted to change the value in vector *z* at position 5 from 20 to 100. We can do this by simply treating the vector reference as it's own variable and conducting a standard reassignment. That is:

```
> z
 [1]  4  8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72 76 80
> z[5]=100
> z
 [1]   4   8  12  16 100  24  28  32  36  40  44  48  52  56  60  64  68  72  76
[20]  80
```

As we can see, by reassigning the value stored in index 5 of vector stored in variable *z*, we change the structure of the vector itself. Another point to note is the output given in the previous example. You may notice that on the first line there is a [1] and then on the second line there is a [20]. These numbers indicate the indexes of the printed numbers directly next to them in the printout. For example, following [1] is the number 4. This indicates that the number 4 is stored at index 1 in the vector. Number 80 follows the [20]. This tells us that the number 80 is stored in the 20th index. The number in brackets are there to allow for easy reading of the print out of the values in the entire vector.

Other than querying and reassigning values in the vector, we can perform a variety of other tasks. There are *vector-based functions* where some functions take as input a variable storing information in a vector and then conduct some type of manipulation on the values in the vector. For example, the function *mean* takes one input: a vector of numbers. The function it self finds the mean of the numbers and provides as an output a single number, which of course is the mean of the numbers in the vector provided. In our example, we can find the average of the numbers stored in the vector *z* by using the following function:

```
> z
 [1]   4   8  12  16 100  24  28  32  36  40  44  48  52  56  60  64  68  72  76
[20]  80
> mean(z)
[1] 46
```

There are many other functions that take in as it's input a vector of numbers. These functions all use the data stored in a vector and somehow combine them together (sound familiar?). Here are a few examples of some common types of functions that can be used on vectors:

```
> mean(z)
[1] 46
> sd(z)
[1] 26.35786
> var(z)
[1] 694.7368
> min(z)
[1] 4
> max(z)
[1] 100
> summary(z)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      4      27      46      46      65     100
```

Thus far we have explored how to manipulate vectors as well as use them in a variety of different functions. What we have yet to do is explain how one goes about creating a specific vector. A vector in R can be created by using the *c* function and providing as the inputs the data to be stored in the vector. When providing the inputs, one must keep in mind that R automatically assigns to the inputs to the corresponding indexes that themselves

correspond to the position in the input. That is, input number 1 gets assigned to index 1 in the vector, input number 2 gets assigned to index 2 in the vector, etc. Suppose we have the following data: 40,2,41,24,10,21,100, and suppose we would like to input this dataset into a vector in R. We can do this using the *c* function:

```
> data=c(40,2,41,24,10,21,100)
> data
[1]  40   2  41  24  10  21 100
```

As you can see, we put the data into a vector by providing the values as inputs into the *c* function. We also stored the resulting vector into a variable named *data*. After querying to R to provide the value stored in the variable *data*, we see the output was the vector of numbers we intended to input and use. We now can use the variable *data* together with a variety of vector-based functions to provide additional information about the data.

The nice feature about declaring vectors using the *c* function is that we can query in a quick manner the elements contained in a vector at a variety of indexes. For example, in our earlier example, we had the vector *z*. Suppose we wanted to know the values in this vector stored in indexes 1,5 and 7. We can ask R to give us these values by using the vector-index notation with a new vector, containing the values of 1, 5 and 7, inside the brackets. This is done as follows:

```
> z
 [1]   4   8  12  16 100  24  28  32  36  40  44  48  52  56  60  64  68  72  76
[20]  80
> z[c(1,5,7)]
[1]   4 100  28
```

Vectors are no doubt useful to our analysis. However, we must recognize that they are limited in what they can store. Vectors, in our context, can only store data from all individuals on a variable by variable basis. What if we have multiple variables? Then the analysis may become quite cumbersome. In addition, manually creating vectors by hand using the *c* function can become quite tiresome. There is another way, however, to store all of our data in a table format similar to the table format that spreadsheet software uses. Not only does the table format help keep all of our data structured and organized, but we can also quickly sift through and filter data if it is self-contained within a single table.

1.4.2 Data Frames

A *data frame* in R is a table of data, where individuals are assigned unique IDs and variables are indicated by column names. You can think of a data frame as an over glorified version of a spreadsheet. The advantages of data frames include a complete storage of all the data in a data set, easy filtering and querying, and analysis can be simultaneously performed on all variables.

In a few moments we will explain how to import a data set from a Microsoft Excel File. First, we will demonstrate the different features of a data frame. We will do so using the built in data sets that are in R. R contains a variety of example data sets to become familiar with the functionality of R. You can obtain a list of these datasets by using the

data() function. After calling the function, it provides a list of names that contain example data-sets. We will use one of these to illustrate some of the uses of a data frame. Let us use the data set *mtcars*. This dataset contains data about cars gathered from a magazine in the 1980's. Typing the variable *mtcars* in the console will bring up the dataset. First, let us get an understanding of this dataset by using the command *?mtcars*. This will bring us a help screen that will give a description of the dataset:

```
mtcars                package:datasets                R Documentation

Motor Trend Car Road Tests

Description:

     The data was extracted from the 1974 _Motor Trend_ US magazine,
     and comprises fuel consumption and 10 aspects of automobile design
     and performance for 32 automobiles (1973-74 models).

Usage:

     mtcars

Format:

     A data frame with 32 observations on 11 variables.

       [, 1]  mpg   Miles/(US) gallon
       [, 2]  cyl   Number of cylinders
       [, 3]  disp  Displacement (cu.in.)
       [, 4]  hp    Gross horsepower
       [, 5]  drat  Rear axle ratio
       [, 6]  wt    Weight (1000 lbs)
       [, 7]  qsec  1/4 mile time
       [, 8]  vs    V/S
       [, 9]  am    Transmission (0 = automatic, 1 = manual)
       [,10]  gear  Number of forward gears
       [,11]  carb  Number of carburetors

Source:

     Henderson and Velleman (1981), Building multiple regression models
     interactively.  _Biometrics_, *37*, 391-411.

Examples:

     require(graphics)
     pairs(mtcars, main = "mtcars data")
     coplot(mpg ~ disp | as.factor(cyl), data = mtcars,
            panel = panel.smooth, rows = 1)
```

We are told that this data set has 11 variables. Most are numerical, with only 1 categorical. We could have a debate as to the *cyl* and *carb* being numerical or categorical, but for now we will assume they are numerical. This data set is stored in a data frame under the variable *mtcars*. While you did not manually assign this variable to anything, R does by default, and hence we can use the data with no need to do additional work. Type into the console the variable name *mtcars* to see the data:

```
> mtcars
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE        16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128          32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic       30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla    33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona     21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger  15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin       15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28        13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird  19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9         27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2     26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa      30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L    15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino      19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora     15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E        21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

We notice a few things about this data set. First, id's are not by number but rather by name. Second, the variable (column) names are shortened. This is to help aid the readability of the data. Now, there are a few functions we can perform on a data frame

(which is what this is). The first basic function to use with a data frame is to know all the variable (column) names. We can do this by using the function *names()*:

```
> names(mtcars)
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"
```

As you can see, the result is a vector of column names in the data frame. Another useful function is to be able to pull out the column of indexes. Usually these are just ordered integers. However, in other data sets, like mtcars, the data may be indexed by a name. We can get a vector of the index names by using the function *rownames()*:

```
> rownames(mtcars)
 [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
 [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
 [7] "Duster 360"          "Merc 240D"           "Merc 230"
[10] "Merc 280"            "Merc 280C"           "Merc 450SE"
[13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
[16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
[19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
[22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
[25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
[28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
[31] "Maserati Bora"       "Volvo 142E"
```

Just like we did with vectors, we can also pull data out of a data frame. First, the most basic operation involving a dataframe is being able to pull out an entire column of data. Since columns correspond to research variables, summarizing the data in a single column amounts to summarizing the data for a particular research variable. We can pull entire vectors of data out of a data frame by using the following notation:

```
variable$column_name
```

We first specify the variable name the data frame is stored in and subsequently use a $ next to it. After, we indicate the column name of whose corresponding data we would like to pull out of the data frame. We usually assign the resulting vector to a variable so we can use and manipulate the data in a vector as we had done so earlier. For example, suppose we wanted to pull all the miles per gallon data. Then, we would use:

```
> mtcars$mpg
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
> mpg_data = mtcars$mpg
> mpg_data
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
```

The data in the mpg column in the mtcars data frame has now been extracted and stored as a vector in the variable *mpg_data*. We now can analyze this data as we had done so before with vectors by using a variety of functions to summarize the data. There are even some functions that allow us to summarize the entire data frame without resorting to extracting the data column by column. For example, suppose we wanted to calculate the five number summary and a frequency distribution (for categorical data) for every variable (column). We can do so with the following function:

```
> summary(mtcars)
      mpg              cyl             disp             hp
 Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0
 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5
 Median :19.20   Median :6.000   Median :196.3   Median :123.0
 Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7
 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0
 Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0
      drat             wt             qsec             vs
 Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000
 1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000
 Median :3.695   Median :3.325   Median :17.71   Median :0.0000
 Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375
 3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
 Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
       am              gear            carb
 Min.   :0.0000   Min.   :3.000   Min.   :1.000
 1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
 Median :0.0000   Median :4.000   Median :2.000
 Mean   :0.4062   Mean   :3.688   Mean   :2.812
 3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
 Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

As we can see, R goes through the work of calculating all of the statistics associated with a five number summary for each and every variable in the data set (saving an enormous amount of time!). There is one problem, however, which is that the variable *am* is treated as a numerical variable. Recall that even though data may be coded, it is not necessarily numerical data. In our case, the variable *am* indicates if a car is manual or automatic. This is clearly a categorical variable, yet, is stored using numerical values (which means it is coded). We must tell R that this variable is categorical. If the values were not coded, R would easily pick this up. But, since they hold numerical variables, we must force R to treat it as categorical. We can do so by using the function *as.factor()*. The input to this function is a vector. R will then convert the vector of numerical values into cateogorical. It will automatically identify all the categories (only numerically expressed). We then can replace the entire column *am* in *mtcars* data frame with the new vector. This is done using the following commands:

```
> new_am=as.factor(mtcars$am)
> mtcars$am=new_am
```

```
> summary(mtcars)
      mpg             cyl             disp             hp
 Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0
 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5
 Median :19.20   Median :6.000   Median :196.3   Median :123.0
 Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7
 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0
 Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0
      drat             wt             qsec             vs         am
 Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000   0:19
 1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000   1:13
 Median :3.695   Median :3.325   Median :17.71   Median :0.0000
 Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375
 3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
 Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
      gear            carb
 Min.   :3.000   Min.   :1.000
 1st Qu.:3.000   1st Qu.:2.000
 Median :4.000   Median :2.000
 Mean   :3.688   Mean   :2.812
 3rd Qu.:4.000   3rd Qu.:4.000
 Max.   :5.000   Max.   :8.000
```

The first command (*mtcars$am*)pulls out the *am* data from the data frame. It uses this vector as an input to the *as.factor* function. We then store the output of this function into a new variable (*new_am*). The next command tells R to go into the data frame stored in the variable *mtcars* and replace the entire column (indicated by the $ sign), with the data in the variable *new_am*. Notice what happens as a result after making a call to summary. It no longer calculates a 5-number summary on the *am* data. Instead, it counts the number of 0's that were in the data (19) and then counts the number of 1's that were in the data (13).

So far, up to this point, we have discussed how to access columns of a data frame, but not how to query the data frame itself for specific data within it. One way to accomplish this querying is through the use of the bracket-notation on data frames. It has the following syntax: `data_frame_name[rows,columns]`. For example, suppose we only wanted the first two rows of the data frame `mtcars`. We could then specify:

```
> mtcars[1:2,]
              mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4
```

In this example, we notice that the data frame name is `mtcars`. Next, we employ the bracket notation. The first entry into this notation is a vector of row numbers that we would like. In this example, we only want rows 1 and 2. Next, we put a comma with nothing after. This tells R to give us back all columns in the data frame `mtcars`. Last, we end it with a bracket. Here is another example where we only want the 1st and 3rd rows, but this time, we only want the mpg and hp columns. We can query the data frame as follows:

```
> mtcars[c(1,3),c("mpg","hp")]
            mpg  hp
Mazda RX4  21.0 110
Datsun 710 22.8  93
```

Notice in this example that we first create a vector containing the elements 1 and 3 (since we want the first and third rows). Next, we create a vector with the names "mpg" and "hp", since we only want the data from these specific columns for the 1st and 3rd row. Hence, we can see how the data frames can be queried using the bracket notation.

There are many other benefits to storing and working with data in data frames. We will come across this additional functionality in the practice exercises as well as in future chapters. You will be surprised at how powerful they truly are in summarizing and filtering data. While in this section we discussed how to work with data frames, we did not discuss how to get our own data into R to work with. We will now look at how to go about doing this.

1.4.3 Applying Functions to Data Structures and Aggregating Data

Once we have a data frame or a vector (and, as we will see later in the course, other types of data structures), many times we would like to use the data is specific ways. This could be in the form of manipulating the data, appending the data, summarizing it, or aggregating it. Such tasks can be burdensome in more traditional software such as Excel. However, in R, we are given access to a category of functions that allow us to easily conduct these aforementioned tasks in an easy manner.

This category of functions in R is known as the `apply` functions. At it's core, the `apply` function takes a data frame (and as we will see later in the course, also a matrix) and applies a given function to all the rows or columns, either column-wise or row-wise. The `apply` function takes three inputs: the first input is the name of the data frame to apply the function to, the second input is either the number 1 (indicating that you want to apply a function row-wise) or the number 2 (indicating that you want to apply the function column-wise), and the third input is the name of the function you would like to apply to the data frame.

To illustrate the use of this function, suppose we have the following closing stock prices for some of the Dow stocks stored in the data frame stored in the variable `stockData`:

```
> stockData
                HD   AAPL    JPM   MSFT   XOM
2018-08-20 195.84 218.10 114.59 107.51 78.26
2018-08-21 198.83 216.80 115.37 106.92 79.05
2018-08-22 200.80 214.10 115.31 105.85 79.11
2018-08-23 199.39 214.65 114.96 107.15 79.56
2018-08-24 200.35 216.60 114.98 107.67 79.52
2018-08-27 202.83 217.15 115.22 109.27 79.91
2018-08-28 202.30 219.01 117.00 109.94 80.58
2018-08-29 201.11 220.15 116.35 110.45 80.44
2018-08-30 201.00 223.25 115.59 111.67 80.45
2018-08-31 199.38 226.51 114.83 111.69 80.24
2018-09-04 200.69 228.41 114.34 110.85 80.41
```

```
2018-09-05 204.83 228.99 115.00 111.01 79.90
2018-09-06 204.59 226.23 114.50 108.25 81.11
2018-09-07 205.45 221.85 114.50 108.23 80.15
```

Suppose we would like to calculate the standard deviation and the mean of each stock for it's price over this time frame. Ordinarily, we could take the longer approach and go column by column to do this:

```
> mean(stockData$HD)
[1] 201.2421
> mean(stockData$AAPL)
[1] 220.8429
> mean(stockData$JPM)
[1] 115.1814
> mean(stockData$MSFT)
[1] 109.0329
> mean(stockData$XOM)
[1] 79.90643
> sd(stockData$HD)
[1] 2.61209
> sd(stockData$AAPL)
[1] 5.074431
> sd(stockData$JPM)
[1] 0.7403624
> sd(stockData$MSFT)
[1] 1.915268
> sd(stockData$XOM)
[1] 0.7470207
```

In some cases, however, this may take too long, especially if we have many columns in our data set. A faster way to calculate the standard deviation of each stock is to employ the `apply` function. We can do this by first specifying the name of the data frame (stockData), then tell R we would like to do this column-wise, since we want to find the standard deviation of each stock, which of course in our data is stored in each column, and last we tell R to use the `sd` function to apply to each column. Doing this, we have:

```
> apply(stockData,2,sd)
       HD      AAPL       JPM      MSFT       XOM
2.6120904 5.0744314 0.7403624 1.9152684 0.7470207
> apply(stockData,2,mean)
       HD      AAPL       JPM      MSFT       XOM
201.24214 220.84286 115.18143 109.03286  79.90643
```

We notice here that by leveraging the apply function, we have shortened our time in repeating the same calculation on each column in the data frame down to only two commands (one for standard deviation, the other for mean).

Suppose we would like to do the same but row wise. That is, instead of finding the standard deviation of the stock data for each stock, we would like to find the standard

deviation of all the stock prices for a particular trading day. That is, we would like to find the standard deviation of all value for every row. Doing this by hand would require us to repeat the same calculation 14 times:

```
> sd(stockData[1,])
[1] 60.60098
> sd(stockData[2,])
[1] 60.65773
> sd(stockData[3,])
[1] 60.46659
> sd(stockData[4,])
[1] 60.01989
> sd(stockData[5,])
[1] 60.75695
> sd(stockData[6,])
[1] 61.13518
> sd(stockData[7,])
[1] 61.07572
> sd(stockData[8,])
[1] 61.18808
> sd(stockData[9,])
[1] 62.03208
> sd(stockData[10,])
[1] 62.855
> sd(stockData[11,])
[1] 63.87079
> sd(stockData[12,])
[1] 64.969
> sd(stockData[13,])
[1] 64.21309
> sd(stockData[14,])
[1] 63.32962
```

Obviously we can see that this took us too much time to input! Instead, we can leverage the apply function but in a different way. Instead of using the number 2 in the second input to indicate column-wise application of the function, we instead indicate the number 1 so R will apply the function to each and every row. Hence, we have:

```
> apply(stockData,1,sd)
2018-08-20 2018-08-21 2018-08-22 2018-08-23 2018-08-24 2018-08-27
  60.60098   60.65773   60.46659   60.01989   60.75695   61.13518
2018-08-28 2018-08-29 2018-08-30 2018-08-31 2018-09-04 2018-09-05
  61.07572   61.18808   62.03208   62.85500   63.87079   64.96900
2018-09-06 2018-09-07
  64.21309   63.32962
```

We notice that our computation was shorted from 14 lines of code down to just 1, with the same exact results. Hence, the apply function is a very powerful tool in aggregating data either row-wise or column-wise very quickly, without resorting to repeating the same calculation on all rows, and we avoid the use of programmatic looping (more on this later).

Not only can we apply functions to data frames but we can do the same with vectors. The `sapply` function takes two inputs: the vector that we would like to apply a function to each element and the function of which we would like to apply. Suppose we have the vector 1 2 3 4 5 6 7 8 9 10, we would like to find the square root of each value. While, technically speaking, we can just provide the vector name in the function `sqrt` itself, an alternative way is to use `sapply` (which becomes more useful for when applying custom functions to vectors!). To illustrate the use, we see that:

```
> sapply(1:10,sqrt)
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
 [8] 2.828427 3.000000 3.162278
```

1.5 Loading Data into R

The truth of the matter is, since R is so flexible, there are many different ways to get your data into R. The most intuitive, but time consuming, way is to manually input all of your data. For example, suppose you have the data that is show in Table 1.2. One way of importing this data into R is to manually create 3 vectors (we do not need to input the ID variable since R automatically generates this when we create the data frame). We then can combine the three vectors using the function *data.frame()* to create a data frame. Here is the list of commands to do this (note, due to the wrapping of the text, some of the values may not appear)

```
>region=c("West","East","West","South","West","West","East","North","South",
"East","East","East","East","North","East","West","West","North","East","West",
"West","East","West","West","East","South","East","South","North","North")

> gender=c("Male", "Female", "Male", "Female", "Male", "Female", "Female",
"Female", "Female", "Male", "Female", "Female", "Male", "Female", "Male",
"Female", "Male", "Male", "Male", "Female", "Female", "Female", "Male",
"Male", "Male", "Male", "Female", "Female", "Female", "Female")

>income = c(119000, 29000, 49000, 145000, 82000, 88000,
78000, 58000, 99000, 111000, 41000, 136000, 37000, 84000,
95000, 62000, 99000, 31000, 65000, 69000, 51000, 68000,
84000, 43000, 147000, 98000, 110000, 20000, 38000, 86000)

> the_data = data.frame(region,gender,income)
```

The input into the function *data.frame* is a list of variable names that store the vectors containing the data. The length of each vector (the number of elements or "mailboxes") must all be the same. If one is missing, R will either try to guess the values or throw you an error. Taking a peek inside the data frame for our custom data, we see that we have the following:

```
> the_data
   region gender income
```

```
1    West   Male 119000
2    East Female  29000
3    West   Male  49000
4   South Female 145000
5    West   Male  82000
6    West Female  88000
7    East Female  78000
8   North Female  58000
9   South Female  99000
10   East   Male 111000
11   East Female  41000
12   East Female 136000
13   East   Male  37000
14  North Female  84000
15   East   Male  95000
16   West Female  62000
17   West   Male  99000
18  North   Male  31000
19   East   Male  65000
20   West Female  69000
21   West Female  51000
22   East Female  68000
23   West   Male  84000
24   West   Male  43000
25   East   Male 147000
26  South   Male  98000
27   East Female 110000
28  South Female  20000
29  North Female  38000
30  North Female  86000
```

We now have a fully functioning data frame that we can use as before. This process is pretty straight-forward. However, it can become quite tedious to manually input all the data into vectors. This opens the door a bunch of issues, mostly including the possible introduction of human error. That is, there is too much typing involved. A faster way to get your own data into R is to have it already in a spread sheet file. If this is a case, then open up your spreadsheet software (Excel, Open Office, Google Sheets, etc). Open the file containing your data. Last, click on "Save As...". Down under the "File Type" drop down box, you want to click "CSV Text", or anything that has "CSV" in it.

Save it and write down THE FULL FILE DIRECTORY PATH. This path can be found on windows by right clicking the file you just created and clicking "properties". Once this screen is open, you should be able to to find the full path. If you are in windows, you must replace all the back slashes with forward slashes (it sounds strange but this is how R reads in a file.) On Mac, you can find the full directory path by clicking "Properties" or "Get Info...", depending on the version you have. Once you have the full directory path and file name, you are ready to import it into R.

We can import the data in this file by using the function *read.csv()*. This function has many input options, but if you saved your file using the default options in Excel, no

ID	Region	Gender	Income
1	West	Male	119000
2	East	Female	29000
3	West	Male	49000
4	South	Female	145000
5	West	Male	82000
6	West	Female	88000
7	East	Female	78000
8	North	Female	58000
9	South	Female	99000
10	East	Male	111000
11	East	Female	41000
12	East	Female	136000
13	East	Male	37000
14	North	Female	84000
15	East	Male	95000
16	West	Female	62000
17	West	Male	99000
18	North	Male	31000
19	East	Male	65000
20	West	Female	69000
21	West	Female	51000
22	East	Female	68000
23	West	Male	84000
24	West	Male	43000
25	East	Male	147000
26	South	Male	98000
27	East	Female	110000
28	South	Female	20000
29	North	Female	38000
30	North	Female	86000

Table 1.2: Example Data Set

additional inputs are required other than the file name and directory path. Unlike your scenario, my R works a bit different, and hence I have the ability to just plug in the file name with no full path. You on the other hand MUST plug this entire path into the function (just for now; we will discuss how to avoid this in a later chapter). For my example, I have this data stored in a file called "x.csv". The following commands help get the data into R:

```
> the_data=read.csv("x.csv")
> the_data
    X     r      g      i
1   1  West   Male 119000
2   2  East Female  29000
3   3  West   Male  49000
4   4 South Female 145000
5   5  West   Male  82000
6   6  West Female  88000
7   7  East Female  78000
8   8 North Female  58000
9   9 South Female  99000
10 10  East   Male 111000
11 11  East Female  41000
12 12  East Female 136000
13 13  East   Male  37000
14 14 North Female  84000
15 15  East   Male  95000
16 16  West Female  62000
17 17  West   Male  99000
18 18 North   Male  31000
19 19  East   Male  65000
20 20  West Female  69000
21 21  West Female  51000
22 22  East Female  68000
23 23  West   Male  84000
24 24  West   Male  43000
25 25  East   Male 147000
26 26 South   Male  98000
27 27  East Female 110000
28 28 South Female  20000
29 29 North Female  38000
30 30 North Female  86000
```

While the column names are different (they were labeled r, g and i, respectively in my CSV file), the data is the same. The only difference between this and the prior set of data is how it was imported. Here, R was able to take the spreadsheet data and automatically import it as a data-frame. The other method is longer and more prone to human error. There are other ways to import data into R. A common approach is to use a data base and have R automatically parse and pull data directly and automatically from a large database. This is outside the score of this course, however. For now, all of our data analysis will be conducted on data that we import using this method described here.

1.6 Logical Control

When working in R, we often do so by writing all of our commands in a text file first and subsequently "running" all the commands during a single phase of execution. Our resulting text file of these commands is called a *script*, and it helps us organize, maintain, and execute our analysis without having to rely so much on the interactive console. While the writing of these commands in a simple linear sequence is tempting as well as common sense, it can at times get tedious, and our resulting script will explode. Furthermore, we often want to run certain commands automatically based on the outcome of other commands we have previously executed, without us having to monitor our progress so closely.

There are also situations where we would like to repeat a command over and over again, but with differing values. In situations like these, we want R to more logically control the "flow" of our commands being executed. That is, based on the outcome of some of our commands, we want R to skip some commands, but not skip others. In situations like these, we need to use what are called *logical control statements*. In our case, we will discuss two types of statements below.

1.6.1 Logical Statements

In R, we can ask a question and get a "TRUE" or "FALSE" response back based on whether or not the response to our question is TRUE or FALSE. For example, suppose we store the value of 5 in the variable `x` and the value of 10 in the variable `y`:

```
> x<-5
> y<-10
> x
[1] 5
> y
[1] 10
```

We can compare these values by using *logical operators*, which help us determine if one quantity is equal (==), not equal (!=), less than (¡), or larger (¿) than another quantity. These are basically asking R Yes/No questions. If we receive a "TRUE" value, then our question has an answer of "Yes", while if we observe a "FALSE" value, then our question has an answer of "No". For example, we have for the quantities defined above

```
> x==y
[1] FALSE
> x==10
[1] FALSE
> x==5
[1] TRUE
> x>5
[1] FALSE
> x<5
[1] FALSE
> y<11
[1] TRUE
```

```
> y>10
[1] FALSE
> x!=y
[1] TRUE
```

1.6.2 If-Else Statements

The first way in which we can control the "flow" of our commands, and how such a sequence of commands will execute, is by leveraging the `if` and `else` commands. To motivate their usage, let us take a simple example. Suppose that we wanted to generate a random number. If the result of the number is an odd number, then we would like to add 1 and divide by 2. On the other hand, if the number is an even number, then we would like to just simply divide by 2. This is very straightforward if we were to do this by hand:

```
> x<-sample(1:10,1)
> x
[1] 7
> y<-(x+1)/2
> x<-sample(1:10,1)
> x
[1] 2
> y<-x/2
```

Notice in the first case when we randomly sample a number between 1 and 10, we get 7, which is an odd number. Hence, to find the value of `y`, we need to add 1 to `x` and then divide by 2. But notice int he second case that the value of `x` is 2, and so all we nee to do in this instance is simply divide by 2. This is easy if we are doing this manually. That is, if we are executing one line at a time, physically examining the output, and then having us manually type in the next command. But what if we wanted to automate this? That is, what if we wanted R to determine whether it should compute `y` by first adding 1 and then dividing by 2, or if it should just simply divide it by 2? We can use an `if` statement to do just that.

In order to use an `if` statement, we need to use the keyword if, followed by a left parentheses, followed by a *condition*, followed by a right parentheses. We then indicate the *block* of commands we seek to run in the event that the condition we specified is true. This is sometimes followed by an *else* statement, which allows us to specify a different block of commands to run in the event that the condition is false. Generally, we can control the flow of code using an if-else statement by using the following structure:

```
if(logical condition is true){
    Command 1
    Command 2
    etc...
}
else{
    Command 1
    Command 2
    etc...
}
```

In our example, we can specify which equation to use by providing the condition. The condition in our instance is whether or not the number stored in `x` is an even or odd number. We can check if a number is odd or even by simply determining if it is divisible by 2. If it is divisible by 2, then by definition it must be an even number. Otherwise, it must be odd. In order to check if a number is divisible by 2, we can determine if the remainder after division by 2 is 0. We can do this in R by using the "%%" operation. Putting this all together, we can rewrite the previous block of code as:

```
x <- sample(1:10,1)
y <- 0
if(x%%2 == 0){
    y<-x/2
}
else{
    y<-(x+1)/2
}
```

Note the use of two "=" in the condition. When we want to compare if two things in R are equal to each other, we use two "=". Note that this is different than using a single "=", which would assign a value on the right hand side to a variable which is indicated on the left hand side. Note that the block of code to execute is indicated in the braces. This is how we can control our logical flow of the code based on conditions. We equally could have written the following code to accomplish the same exact output:

```
x <- sample(1:10,1)
y <- 0
if(x%%2 != 0){
    y<-(x+1)/2
}
else{
    y<-x/2
}
```

1.6.3 Loop Statements

Another way as to how we can control the flow of execution of commands is by repeating a common type of command over and over again. There are two ways in R to repeat the same command over and over again. The first way is based on whether or not a logical condition is true. If a logical condition is true, then R will re-execute the code in the block which follows the statement. Such control is called a *while loop*. The general structure is as follows:

```
while(logical condition is true){
    Command 1
    Command 2
    etc...
}
```

Note that within our block of code to repeat, we need a way to update a value that is used in the logical condition, otherwise we have what is called an *infinite loop*, and these can be very problematic when you run your code. Therefore, the trick to writing a good while loop is to ensure that you have some way to stop the loop based on a logical condition, and furthermore, that the condition will be reached at some point in time. Let us take an example. Suppose we want to print out the words stored in a vector, however, we only want to print the words in the vector until it gets to the word "THEWORD". We can do this by hand:

```
>words<-c("ONE Word", "Two Word", "A Word", "THEWORD", "Third Word")
> print(words[1])
[1] "ONE Word"
> print(words[2])
[1] "Two Word"
> print(words[3])
[1] "A Word"
```

However, we can simplify this down by simply using a while loop, where the conditional statement would be if the word is equal to "THEWORD". What we can do is remove the word from the vector, check if the word is not equal to "THEWORD", if it is not then print it, otherwise we can remove it:

```
> while(words[1]!="THEWORD"){
+     print(words[1])
+     words<-words[-1]
+ }
[1] "ONE Word"
[1] "Two Word"
[1] "A Word"
```

Another way in which we can repeat code in a block is by using a `for` loop. Unlike a while loop, which executes code so long as a condition is true, a for loop repeats code by iterating through a vector, list, or other data structure in R. We can actually execute the same code above by using a for loop, which will allow us to have the same exact output, but without altering the vector itself:

```
>words<-c("ONE Word", "Two Word", "A Word", "THEWORD", "Third Word")

> for(word in words){
+     if(word == "THEWORD")
+         break
+     print(word)
+ }
[1] "ONE Word"
[1] "Two Word"
[1] "A Word"
```

Notice that we put an if statement inside the for loop. Basically what is happening here is that R will sequentially take the variable `word` and assign it to the next element in the

vector `words`. We then check if the current word is equal to our intended word. If it is, we use the command `break` to tell R to prematurely stop the loop (otherwise, it will continue until R gets to the end of the vector). Loops are very helpful, and allow us to repeat code without having to monitor it or having to run it command by command. "Writing Code" essentially boils down to using the proper functions, mathematical operations, and logical control statements.

1.7 Exercises

1. Suppose we ran the following commands:

```
x=5
y=x
x=2
```

   What values should we expect to see in the variables x and y, respectively?

2. Create a sequences of numbers that start from 10 and ends at 100, where each number in the sequence is 5 more than the previous number in the sequence. After, replace all the even numbers in the sequence with the number 5.

3. Find a function that will create a vector of random numbers that is drawn from a normal distribution with mean 0 and standard deviation 1. Use this function to generate such a vector of length 10. Create a second such vector. Now, find a vector such that each element is the result of multiplying the two corresponding elements in the generated vector.

4. Randomly generate two types of data with 500 observations: one vector of numerical data, one vector of categorical data. For the numerical data-vector, generate the data by randomly sampling from the sequence of numbers starting at 4 and ending at 300, with an inner-number difference of 2. For the categorical data, randomly sample 500 observations from the values of "Male", "Female". After this is done, create a dataframe from these two vectors.

5. Find a shortcut (NOT using the `seq` function), that will produce the following numbers: 100 101 102 103 104 105 106 107 108 109 110

6. Suppose we are given the vector of time - based numerical data (ordered from left to right as start of time to end of time): `45 54 24 64 65 45 65 45 34 65 44 33`. Take this vector of data, lag it by one period, and create two vectors: the first vector will be the "after" data, which will comprise of data from element 2 to element 12, the second vector will be the "before data", which will comprise of data from element 1 to element 11. Put these two vectors into a data frame, and find a statistical summary of all the vectors using one command (recall that a statistical summary comprises of finding the median, mean, 1st and 3rd quartiles, the minimum and the maximum).

7. In R, a string text data that is assigned by a variable by using quotations. For example, we can store the word "Dog" into a variable called `pet` by using the command: `pet = "Dog"`. Create a second variable called `pet2` which will store the work "Cat". Next,

find a function that will concatenate these two strings into one string, and store it in a variable called `allPets`.

8. Consider the data set `mtcars` again. Suppose you have found a new car that you would like to add. It currently has 40mpg, 8 cyl, 500 disp, 300 hp, 4.25 drat, 6.24 wt, 20.41 qsec, 0 vs, 1 am, 3 gear and 4 carb. You would like to append this data to the data frame mtcars. Find a function that will do this, and demonstrate how it can be accomplished. (Challenge: Try and figure out how to add the car data along with a new row name of this car. We will call it "Custom Car").

9. Suppose we have the vector `y=c(4,2,6,4,4,2,5,2,4,2)`. Further suppose that we would like to append the following numbers, in order, to the end of this vector: 8,2,5,2. Find a way to append these numbers, and store the final vector in a variable called `vector2`.

10. Let the vector `z` be assigned to the vector
`c("Male","Male","Female","Female","Male","Female","Female","Male")`
Find a function that will return a vector of indexes of vector `z` such that the corresponding values in those positions are equal to "Male".

11. Suppose that we have the following data frame (you can copy and paste this code into your R for easy solving, just remove the $>$, and obviously the output at the end):

```
> region = c("North","South","North","West","East","South")
> age = c(34,20,51,32,54,35)
> data = data.frame(region,age)
> data
  region age
1  North  34
2  South  20
3  North  51
4   West  32
5   East  54
6  South  35
```

Find a way in R to extract the ages of only individuals living in the North, and store the resulting vector in a variable called `ages_north`. (Hint: This can be done with only one line of code).

12. A data frame stored in the variable `data` has the following column names:
`age,gender,region,income`. Suppose we would like to change these column names to `age,sex,area,money`. How would we go about doing this? (Hint: this can be done with only one line of code).

13. Below is a table of a small sample size of individuals.
(a) Take the data in this table and input it into R as a data frame. Store this data frame in a variable called `data`.

ID	scores	age	income	educationLevel
1	6.00	60	32334.00	6.00
2	6.00	19	67132.00	7.00
3	4.00	44	55485.00	5.00
4	5.00	39	57360.00	6.00
5	6.00	56	27190.00	6.00
6	4.00	36	43131.00	6.00
7	5.00	45	62421.00	4.00
8	5.00	56	44581.00	8.00
9	6.00	18	43376.00	3.00
10	5.00	51	48412.00	6.00

(b) Find a function that will calculate the standard deviation of all columns. You should be able to calculate the standard deviation of all columns using only one line of code.

14. Suppose we had a string stored in the variable `string = "Hello World!"`. Suppose we would like to replace all the characters "o" with the character "p". Find a function in R that will do this.

15. Take the data frame from problem 13 and sort it according to scores from least to greatest. This should be accomplished with only a single line of code.

1.8 Extra Help

Want some additional basic R problems to practice? Check out these websites:

1. http://rstatistics.net/r-lang-practice-exercises-level-1-beginners/
2. https://www.r-exercises.com/tag/vectors-and-sequences/
3. https://www.r-exercises.com/tag/lists-and-dataframes/

Want more tutorials? Check out these sites:

1. https://datascienceplus.com/learn-r-by-intensive-practice/
2. https://www.datacamp.com/courses/free-introduction-to-r
3. https://cran.r-project.org/web/packages/IPSUR/vignettes/IPSUR.pdf
4. https://www.listendata.com/p/r-programming-tutorials.html
5. https://en.wikibooks.org/wiki/R_Programming

References