Homework Set 1

Dr. Christopher Brown

November 9, 2018

Introduction

Welcome to the first homework set in the Data Analysis class. In the exercise sets in this class, you should expect to be learning as you work, so if you come to an exercise and you don't know what to do off the top of your head, well... that's what is supposed to happen sometimes!

My expectations for you in all homework assignments are:

- You'll use available resources to work through the exercises. Resources include the internet and each other.
- You'll try to learn techniques, concepts, and R commands as you progress through the exercises. Struggle with these exercises a bit, practice your "figuring it out" skills, and it will pay off in the long run.
- You'll experiment a bit beyond the required exercises. Messing around with the ideas you encounter in this exercise set will get you some more practice. You may encounter situations you can't handle exactly as you did in these exercises, and you'll have to adapt... and this will help you immeasurably as we progress.

I've provided some R code examples in many exercises. Just to be clear, this is not **the only** R code to carry out the tasks, it is **some workable** R code to carry out these tasks.

Exercises to Get Started

Exercise 1 Install R. Link

Exercise 2 Install RStudio. Link

Exercise 3 Open RStudio, and find an "Introduction to RStudio" tutorial on YouTube. Orient yourself to the interface.

Exercise 4 Install the tidyverse package set in R. Get connected to the internet, open RStudio, visit the console, issue the command *install.packages("tidyverse")*, and sit back and wait for a few minutes.

Note: At this point, between this install and the datasets in the course repository on Github, you should have everything you need for the exercises in this course. For projects, you may have to install other packages or get other data sets depending on your needs.

Exercise 4.1 (Optional, useful) Install the GGally package with *install.packages("GGally")*. This has a graphics function, *ggpairs*, that is highly useful in the initial exploration of a data set.

Exercise 4.2 (Optional, highly recommended for CSC majors, aspiring data scientists) Get a Github account at https://github.com/ . Also download the Github Desktop software and install it. Find a tutorial on using the commit, clone, push, and pull operations on repositories, and get an understanding of branching.

Exercises with Data

You'll be using the *R for Beginners* document as a reference throughout these exercises. However, you should also use Google! You may learn some new ways of doing things in R!

Exercise 5 Create a vector of numbers from 0 to 1 with jumps of 0.01 and store it in x with the command

```
x <- seq(from=0,to=1,by=0.01)
```

What is the length of x? Use the *length* command. What is the first element in x? Try *x[1]* and *head(x,1)*. What is the third element in x? Try *x[3]*. What is the last element in x? Try *x[length(x)]* and *tail(x,1)*.

Exercise 6 Create a new vector y from x in the previous exercise using

```
y <- x^2+2
```

What did this command do? If you're not using the word "function" in your explanation, go back and re-explain.

Exercise 7 We are now finished with x and y. Remove them from your R environment with the *rm* command.

Exercise 8 Create a vector of numbers from -50 to 50 by using the command *-50:50* and store it in x. Execute the command *x[3:10]*; what does this return?

Exercise 9 Using the same x as in the previous exercise, execute the command *x>10*; what did this command return? Now try the commands *x[x>10]* and *which(x>10)* and *x[which(x>10)]*; what do they return?

Exercise 10 We may want to select data based on more than one logical condition. For example, to find all values in x that are greater than 10 and whose squares are less than 400, we can enter *x[x>10 & x^2<400]*. Try it! Then, find all values in x whose squares are greater than 20 and less than 600.

Exercise 11 We may want to connect logicals with an or statement. For example, we may want to select all values in x which are less than -20 or whose square is greater than 600. We can do so with the command *x[x<-20 | x^2>600]*; try it!

Exercise 12 There are many uses of selecting data with logical statements. We'll see lots when we look at data frames, data structures with more than one variable recorded. A simpler use of logical data selection is data transformation. For example, suppose I happen to know that all of the data coming from a certain sensor should be nonnegative values, but I also know that the sensor is faulty. If the sensor returned a vector like x, I would (a) know that the sensor had been throwing faults at a high rate, and (b) I would want to mark those faulty data points so they don't "look like" data points. One way to do this is with an *ifelse* command. *ifelse* takes three arguments: a logical vector like *x<0*, a filler when the condition is true, and a filler when the condition is false. For example, to mark all the negative values in x with *NA*, leave all other values in x unchanged, and store the result in y I can use *y <- ifelse(x<0,NA,x)*. Try it! Now try replacing all values in x less than 10 by 0, all other values by 1, and then storing the result in the variable z.

Exercise 13 You can also transform numerical data to text. Try replacing all the positive values in x with the word "Positive", all other values with "Not Positive", and store the result in the variable *coded*. This sort of transformation can improve readability of your data and graphics later.

Exercise 14 What happens when you want to encode three levels logically, rather than just two? *ifelse* only has two output possibilities! For example, we might want to code x with "low" for values below -10, "high" for values above 10, and "medium" for all other values. Well, there are (as always) several ways to handle this. You can nest your ifelse statements:

```
x <- -50:50
coded <- ifelse(x< -10,"low",ifelse(x>10,"high","medium"))
coded
```

```
##   [1] "low"    "low"    "low"    "low"    "low"    "low"    "low"
##   [8] "low"    "low"    "low"    "low"    "low"    "low"    "low"
##  [15] "low"    "low"    "low"    "low"    "low"    "low"    "low"
##  [22] "low"    "low"    "low"    "low"    "low"    "low"    "low"
##  [29] "low"    "low"    "low"    "low"    "low"    "low"    "low"
##  [36] "low"    "low"    "low"    "low"    "low"    "medium" "medium"
##  [43] "medium" "medium" "medium" "medium" "medium" "medium" "medium"
##  [50] "medium" "medium" "medium" "medium" "medium" "medium" "medium"
##  [57] "medium" "medium" "medium" "medium" "medium" "high"   "high"
##  [64] "high"   "high"   "high"   "high"   "high"   "high"   "high"
##  [71] "high"   "high"   "high"   "high"   "high"   "high"   "high"
##  [78] "high"   "high"   "high"   "high"   "high"   "high"   "high"
##  [85] "high"   "high"   "high"   "high"   "high"   "high"   "high"
##  [92] "high"   "high"   "high"   "high"   "high"   "high"   "high"
##  [99] "high"   "high"   "high"
```

Try this! For more than three codes, this gets pretty unwieldy. There's no reason you can't code one at a time:

```
coded <- ifelse(x< -10,"low",x)
coded <- ifelse(x>10,"high",coded)
coded <- ifelse(x >= -10 & x <= 10,"medium",coded)
coded
```

```
##   [1] "low"    "low"    "low"    "low"    "low"    "low"    "low"
##   [8] "low"    "low"    "low"    "low"    "low"    "low"    "low"
##  [15] "low"    "low"    "low"    "low"    "low"    "low"    "low"
##  [22] "low"    "low"    "low"    "low"    "low"    "low"    "low"
##  [29] "low"    "low"    "low"    "low"    "low"    "low"    "low"
##  [36] "low"    "low"    "low"    "low"    "low"    "medium" "medium"
##  [43] "medium" "medium" "medium" "medium" "medium" "medium" "medium"
##  [50] "medium" "medium" "medium" "medium" "medium" "medium" "medium"
##  [57] "medium" "medium" "medium" "medium" "medium" "high"   "high"
##  [64] "high"   "high"   "high"   "high"   "high"   "high"   "high"
##  [71] "high"   "high"   "high"   "high"   "high"   "high"   "high"
##  [78] "high"   "high"   "high"   "high"   "high"   "high"   "high"
##  [85] "high"   "high"   "high"   "high"   "high"   "high"   "high"
##  [92] "high"   "high"   "high"   "high"   "high"   "high"   "high"
##  [99] "high"   "high"   "high"
```

Try this! But again, this gets a little unwieldy. It is much easier and more readable to define a filter function and apply it across the data. This leads us into the notion of functions in R. At this point we are finished with x; remove it from memory using the *rm* command.

Exercises with Functions

Exercise 15 Let's look at an example of a function defined in R:

```
f <- function(x) {
  y <- x^2
  return(y)
}
```

Take a guess; what does this function do? The function declaration I've used here has three components. The **name** of the function is *f*, and is just another R variable that we store the function in. The **function declaration** is the statement *function(x)*, which tells R we are about to define a function and that function gets one input, which in the body of the function we'll call x. The **body** of the function falls between the curly braces *{* and *}*, and is the set of instructions the function will carry out to return an output. When we execute this code, nothing visibly happens in our terminal, but a lot goes on in the computer's memory, as the function is defined and stored.

Once you have typed in the function *f* and executed that code, follow up by computing a few outputs. For example,

```
f(2)
```

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

and

```
f(-3)
```

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

produce the outputs we expect. However, what about a bad input?

```
f("red")
```

The function produces an error that indicates that plugging in the input - the argument - to the power operation ˆ has used a non-numeric value, and that causes a problem. Notice that plugging a non-numeric into the function did not immediately cause an error; the error was only thrown when the power operation was reached and could not be computed. (For those of you with experience programming in C, Java, or something similar: other languages provide structures for functions that filter out bad inputs based on type, and that filtration is baked into the function declaration. We'll see later why this is not necessarily desirable in a high-level language, especially one designed to deal with data.)

Plug in a few more inputs to *f*, both good and bad, and see what the result is.

Exercise 16 For the function *f* defined above, define a vector of inputs *a <- c(2,5,8,-10)* and then try computing *f(a)*. What happens?

Exercise 17 Let's define a function that returns the input value if the input value is positive, and 0 otherwise.

```
g <- function(x) {
  if (x>0) {
    y <- x
  } else {
    y <- 0
  }
  return(y)
}
```

Compute the outputs for a few inputs and make sure this function is doing what you think it should.

Exercise 18 Using the function *g* from the previous exercise and the input *a <- c(2,5,8,-10)*, execute *g(a)* and see what happens. Is this what you expected?

What we can see from this example is that the application of a function over a vector of inputs is not always automatic. In fact, our example with the function *f* earlier was a happy accident; most functions do not automatically apply over a vector if inputs. Fortunately, there is an easy way to fix this! The *Vectorize* command in R gives the function the ability to apply over a vector of inputs. Try executing

```
g <- Vectorize(g)
```

and then try to compute *g(a)* again.

Exercise 19 Now let's put this all together to create a data filter. Create a vector of inputs and store it in *x*

```
x <- -10:1000
```

Let's transform the data so that all negative values are replaced by *NA*, all nonnegative values up to 100 are replaced by "blue", and all other values are replaced by "red". Here is a function that accomplishes this:

```r
h <- function(x) {
  if (x<0) {
    y <- NA
  }
  if (x>=0 & x<100) {
    y <- "blue"
  }
  if (x>=100) {
    y <- "red"
  }
  return(y)
}
h <- Vectorize(h)
```

Apply this to the input vector x and see whether you get what you expect!

Exercise 20 Repeat the previous exercise, using a transformation with the following properties:

- If x is negative, replace x by its square.
- If x is nonnegative and has a square less than 20000, replace x by its square root.
- If x has a square greater than 20000, replace x by its natural logarithm.

Exercise 21 Wrap up this section by removing the variables you created from memory. (This is not critical for these exercises. However, when working with large datasets, removing some temporary variables may easily free up Gb's of memory, which can be helpful.)

Exercises with Dataframes

Before you work on this section you should watch a tutorial video on dataframes, like this one and you should read sections 1, 2, and 3 in Hadley Wickham's Tidy Data article.

Exercise 22 Load the iris data set, using

```
data("iris")
```

After you do, find the iris data set in the help system in R and read about it. You can look at the "top" of the data set using *head(iris)* and the structure of the data set using *str(iris)*.

Exercise 23 You can extract information about the dataframe using several R commands. This is helpful in larger dataframes; for example, if your dataframe has 125,000 observations (rows) and 85 variables (columns), it may not be possible to inspect it visually. So instead, we can use *nrow* to count the number of rows

```
nrow(iris)
```

```
## [1] 150
```

and we can use *ncol* to count the number of variables (columns).

```
ncol(iris)
```

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

If we want to list the names of the variables in a dataset, we can do so using the *names* command.

```
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
## [5] "Species"
```

Load the *mtcars* dataset as you did for the *iris* data set, and count the rows and columns, and list the names of the variables measured.

Exercise 24 We can manipulate the names of the dataframe if they are not informative enough. For example, we might want to include units in our column names in the *iris* dataset. Here's one way to go about that:

```
df <- iris  #make a new copy of the dataframe and change the copy
new.names <- c("Sepal.Length.cm","Sepal.Width.cm","Petal.Length.cm","Petal.Width.cm","Species")
names(df) <- new.names
```

At this point we can inspect the names of the dataframe df.

```
names(df)
```

```
## [1] "Sepal.Length.cm" "Sepal.Width.cm"  "Petal.Length.cm" "Petal.Width.cm"
## [5] "Species"
```

Use this same idea to make the names of the *mtcars* dataset more readable. You will have to make some judgement calls on what "more readable" means. You will not want to include spaces in your variable names; common practice is to separate words by periods or start new words with capital letters, or both.

Exercise 25 We can select parts of the data frame using indexing in many ways. It's important to remember that dataframes are rectangular representations of data. Rows are observation instances, and columns are variables. So

```
iris[2,3]
```

```
## [1] 1.4
```

selects the second observation (row) and the third variable (column). In R code it is usually clearer to access the variable by name rather than by column number. You can do this with

```r
iris[2,"Petal.Length"]
```

```
## [1] 1.4
```

Alternately, you can do this by selecting only the variable you wish to deal with and then selecting the observation. To select only one variable, use

```r
iris[,"Petal.Length"]
```

```
##   [1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.4 1.1 1.2 1.5 1.3
##  [18] 1.4 1.7 1.5 1.7 1.5 1.0 1.7 1.9 1.6 1.6 1.5 1.4 1.6 1.6 1.5 1.5 1.4
##  [35] 1.5 1.2 1.3 1.4 1.3 1.5 1.3 1.3 1.3 1.6 1.9 1.4 1.6 1.4 1.5 1.4 4.7
##  [52] 4.5 4.9 4.0 4.6 4.5 4.7 3.3 4.6 3.9 3.5 4.2 4.0 4.7 3.6 4.4 4.5 4.1
##  [69] 4.5 3.9 4.8 4.0 4.9 4.7 4.3 4.4 4.8 5.0 4.5 3.5 3.8 3.7 3.9 5.1 4.5
##  [86] 4.5 4.7 4.4 4.1 4.0 4.4 4.6 4.0 3.3 4.2 4.2 4.2 4.3 3.0 4.1 6.0 5.1
## [103] 5.9 5.6 5.8 6.6 4.5 6.3 5.8 6.1 5.1 5.3 5.5 5.0 5.1 5.3 5.5 6.7 6.9
## [120] 5.0 5.7 4.9 6.7 4.9 5.7 6.0 4.8 4.9 5.6 5.8 6.1 6.4 5.6 5.1 5.6 6.1
## [137] 5.6 5.5 4.8 5.4 5.6 5.1 5.1 5.9 5.7 5.2 5.0 5.2 5.4 5.1
```

or the useful shorthand

```r
iris$Petal.Length
```

```
##   [1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.4 1.1 1.2 1.5 1.3
##  [18] 1.4 1.7 1.5 1.7 1.5 1.0 1.7 1.9 1.6 1.6 1.5 1.4 1.6 1.6 1.5 1.5 1.4
##  [35] 1.5 1.2 1.3 1.4 1.3 1.5 1.3 1.3 1.3 1.6 1.9 1.4 1.6 1.4 1.5 1.4 4.7
##  [52] 4.5 4.9 4.0 4.6 4.5 4.7 3.3 4.6 3.9 3.5 4.2 4.0 4.7 3.6 4.4 4.5 4.1
##  [69] 4.5 3.9 4.8 4.0 4.9 4.7 4.3 4.4 4.8 5.0 4.5 3.5 3.8 3.7 3.9 5.1 4.5
##  [86] 4.5 4.7 4.4 4.1 4.0 4.4 4.6 4.0 3.3 4.2 4.2 4.2 4.3 3.0 4.1 6.0 5.1
## [103] 5.9 5.6 5.8 6.6 4.5 6.3 5.8 6.1 5.1 5.3 5.5 5.0 5.1 5.3 5.5 6.7 6.9
## [120] 5.0 5.7 4.9 6.7 4.9 5.7 6.0 4.8 4.9 5.6 5.8 6.1 6.4 5.6 5.1 5.6 6.1
## [137] 5.6 5.5 4.8 5.4 5.6 5.1 5.1 5.9 5.7 5.2 5.0 5.2 5.4 5.1
```

You can then select the observation you want by using the index, as you would for a vector.

```r
iris$Petal.Length[2]
```

```
## [1] 1.4
```

Select only the *hp* variable from the *mtcars* dataset and then select the 23rd entry from it.

Exercise 26 We can select parts of the data frame using indexing in many ways. We may want to look at only one observation, or row. We can do this by specifying the row with its numeric index.

```r
iris[2,]
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 2          4.9           3          1.4         0.2  setosa
```

The result is a **data frame**, not a vector, with only one row. So while this “looks like” a vector, it has data of possibly different types and it has headings for the variables (columns).

We may want to select some subset of rows. We can do this by specifying a vector of row indices. Try

```r
iris[10:15,]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 10          4.9         3.1          1.5         0.1  setosa
## 11          5.4         3.7          1.5         0.2  setosa
## 12          4.8         3.4          1.6         0.2  setosa
## 13          4.8         3.0          1.4         0.1  setosa
## 14          4.3         3.0          1.1         0.1  setosa
## 15          5.8         4.0          1.2         0.2  setosa
```

to select the 10th through 15th rows, or

```
iris[c(10,15,20,25),]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 10          4.9         3.1          1.5         0.1  setosa
## 15          5.8         4.0          1.2         0.2  setosa
## 20          5.1         3.8          1.5         0.3  setosa
## 25          4.8         3.4          1.9         0.2  setosa
```

to select the 10th, 15th, 20th, and 25th rows. Try selecting rows 10 through 25 of the *mtcars* dataframe.

Exercise 27 We can select parts of the data frame using indexing in many ways. One powerful tool we can use is logical selection of observations. For example, suppose that we want to study only those observations made on the setosa species. We can define a logical vector that selects the setosa species

```
my_selection <- iris$Species == "setosa"
```

and then we can select only those rows for which this logical is true:

```
iris[my_selection,]
```

```
##    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
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
## 11          5.4         3.7          1.5         0.2  setosa
## 12          4.8         3.4          1.6         0.2  setosa
## 13          4.8         3.0          1.4         0.1  setosa
## 14          4.3         3.0          1.1         0.1  setosa
## 15          5.8         4.0          1.2         0.2  setosa
## 16          5.7         4.4          1.5         0.4  setosa
## 17          5.4         3.9          1.3         0.4  setosa
## 18          5.1         3.5          1.4         0.3  setosa
## 19          5.7         3.8          1.7         0.3  setosa
## 20          5.1         3.8          1.5         0.3  setosa
## 21          5.4         3.4          1.7         0.2  setosa
## 22          5.1         3.7          1.5         0.4  setosa
## 23          4.6         3.6          1.0         0.2  setosa
## 24          5.1         3.3          1.7         0.5  setosa
## 25          4.8         3.4          1.9         0.2  setosa
## 26          5.0         3.0          1.6         0.2  setosa
## 27          5.0         3.4          1.6         0.4  setosa
## 28          5.2         3.5          1.5         0.2  setosa
## 29          5.2         3.4          1.4         0.2  setosa
## 30          4.7         3.2          1.6         0.2  setosa
## 31          4.8         3.1          1.6         0.2  setosa
## 32          5.4         3.4          1.5         0.4  setosa
## 33          5.2         4.1          1.5         0.1  setosa
## 34          5.5         4.2          1.4         0.2  setosa
## 35          4.9         3.1          1.5         0.2  setosa
## 36          5.0         3.2          1.2         0.2  setosa
## 37          5.5         3.5          1.3         0.2  setosa
## 38          4.9         3.6          1.4         0.1  setosa
## 39          4.4         3.0          1.3         0.2  setosa
## 40          5.1         3.4          1.5         0.2  setosa
## 41          5.0         3.5          1.3         0.3  setosa
## 42          4.5         2.3          1.3         0.3  setosa
## 43          4.4         3.2          1.3         0.2  setosa
## 44          5.0         3.5          1.6         0.6  setosa
## 45          5.1         3.8          1.9         0.4  setosa
## 46          4.8         3.0          1.4         0.3  setosa
## 47          5.1         3.8          1.6         0.2  setosa
## 48          4.6         3.2          1.4         0.2  setosa
## 49          5.3         3.7          1.5         0.2  setosa
## 50          5.0         3.3          1.4         0.2  setosa
```

You can pull all this together using only one command.

```
iris[iris$Species == "setosa",]
```

```
##    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
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
## 11          5.4         3.7          1.5         0.2  setosa
## 12          4.8         3.4          1.6         0.2  setosa
## 13          4.8         3.0          1.4         0.1  setosa
## 14          4.3         3.0          1.1         0.1  setosa
## 15          5.8         4.0          1.2         0.2  setosa
## 16          5.7         4.4          1.5         0.4  setosa
## 17          5.4         3.9          1.3         0.4  setosa
## 18          5.1         3.5          1.4         0.3  setosa
## 19          5.7         3.8          1.7         0.3  setosa
## 20          5.1         3.8          1.5         0.3  setosa
## 21          5.4         3.4          1.7         0.2  setosa
## 22          5.1         3.7          1.5         0.4  setosa
## 23          4.6         3.6          1.0         0.2  setosa
## 24          5.1         3.3          1.7         0.5  setosa
## 25          4.8         3.4          1.9         0.2  setosa
## 26          5.0         3.0          1.6         0.2  setosa
## 27          5.0         3.4          1.6         0.4  setosa
## 28          5.2         3.5          1.5         0.2  setosa
## 29          5.2         3.4          1.4         0.2  setosa
## 30          4.7         3.2          1.6         0.2  setosa
## 31          4.8         3.1          1.6         0.2  setosa
## 32          5.4         3.4          1.5         0.4  setosa
## 33          5.2         4.1          1.5         0.1  setosa
## 34          5.5         4.2          1.4         0.2  setosa
## 35          4.9         3.1          1.5         0.2  setosa
## 36          5.0         3.2          1.2         0.2  setosa
## 37          5.5         3.5          1.3         0.2  setosa
## 38          4.9         3.6          1.4         0.1  setosa
## 39          4.4         3.0          1.3         0.2  setosa
## 40          5.1         3.4          1.5         0.2  setosa
## 41          5.0         3.5          1.3         0.3  setosa
## 42          4.5         2.3          1.3         0.3  setosa
## 43          4.4         3.2          1.3         0.2  setosa
## 44          5.0         3.5          1.6         0.6  setosa
## 45          5.1         3.8          1.9         0.4  setosa
## 46          4.8         3.0          1.4         0.3  setosa
## 47          5.1         3.8          1.6         0.2  setosa
## 48          4.6         3.2          1.4         0.2  setosa
## 49          5.3         3.7          1.5         0.2  setosa
## 50          5.0         3.3          1.4         0.2  setosa
```

Use a logical to extract those observations of the setosa species with sepal length greater than 4.8 and sepal width less than 3.5.

Exercise 28 Use a logical to extract observations from the *mtcars* dataset that satisfy:

- miles per gallon is greater than 18
- number of cylinders is 4
- horsepower is greater than 85

Exercise 29 Just as with vectors of data, we may want to perform transformations on our data, such as applying filters or functions to the data. For example, in the *iris* data set we may want to estimate petal area by computing the product of petal width and petal length.

```
iris$Petal.Width * iris$Petal.Length
```

```
##   [1]  0.28  0.28  0.26  0.30  0.28  0.68  0.42  0.30  0.28  0.15  0.30
##  [12]  0.32  0.14  0.11  0.24  0.60  0.52  0.42  0.51  0.45  0.34  0.60
##  [23]  0.20  0.85  0.38  0.32  0.64  0.30  0.28  0.32  0.32  0.60  0.15
##  [34]  0.28  0.30  0.24  0.26  0.14  0.26  0.30  0.39  0.39  0.26  0.96
##  [45]  0.76  0.42  0.32  0.28  0.30  0.28  6.58  6.75  7.35  5.20  6.90
##  [56]  5.85  7.52  3.30  5.98  5.46  3.50  6.30  4.00  6.58  4.68  6.16
##  [67]  6.75  4.10  6.75  4.29  8.64  5.20  7.35  5.64  5.59  6.16  6.72
##  [78]  8.50  6.75  3.50  4.18  3.70  4.68  8.16  6.75  7.20  7.05  5.72
##  [89]  5.33  5.20  5.28  6.44  4.80  3.30  5.46  5.04  5.46  5.59  3.30
## [100]  5.33 15.00  9.69 12.39 10.08 12.76 13.86  7.65 11.34 10.44 15.25
## [111] 10.20 10.07 11.55 10.00 12.24 12.19  9.90 14.74 15.87  7.50 13.11
## [122]  9.80 13.40  8.82 11.97 10.80  8.64  8.82 11.76  9.28 11.59 12.80
## [133] 12.32  7.65  7.84 14.03 13.44  9.90  8.64 11.34 13.44 11.73  9.69
## [144] 13.57 14.25 11.96  9.50 10.40 12.42  9.18
```

We would also like to attach this to our data set as a new, computed column. We can do so by simply storing the result in a new column in the dataframe.

```
iris$Petal.Area <- iris$Petal.Width * iris$Petal.Length
```

You can now see that a new column has been added to the *iris* dataframe in memory (this does not affect the *iris* dataframe stored in the datasets library in R).

```r
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Petal.Area
## 1          5.1         3.5          1.4         0.2  setosa       0.28
## 2          4.9         3.0          1.4         0.2  setosa       0.28
## 3          4.7         3.2          1.3         0.2  setosa       0.26
## 4          4.6         3.1          1.5         0.2  setosa       0.30
## 5          5.0         3.6          1.4         0.2  setosa       0.28
## 6          5.4         3.9          1.7         0.4  setosa       0.68
```

Add another new column to the *iris* dataframe, corresponding to sepal area.

Exercise 30 We can also create new columns in dataframes by creating functions, applying them to existing columns, and then storing the result in a new column in the dataframe. For example, let's code petal length of the irises as long or short, where long is a petal length greater than 5.5 cm.

```r
code <- function(x) {
  if (x>5.5) {
    y <- "long"
  } else {
    y <- "short"
  }
  return(y)
}
code <- Vectorize(code)
df$LongPetals <- code(iris$Petal.Length)
```

You can now inspect the new *LongPetals* variable.

```r
df$LongPetals
```

```
##   [1] "short" "short" "short" "short" "short" "short" "short" "short"
##   [9] "short" "short" "short" "short" "short" "short" "short" "short"
##  [17] "short" "short" "short" "short" "short" "short" "short" "short"
##  [25] "short" "short" "short" "short" "short" "short" "short" "short"
##  [33] "short" "short" "short" "short" "short" "short" "short" "short"
##  [41] "short" "short" "short" "short" "short" "short" "short" "short"
##  [49] "short" "short" "short" "short" "short" "short" "short" "short"
##  [57] "short" "short" "short" "short" "short" "short" "short" "short"
##  [65] "short" "short" "short" "short" "short" "short" "short" "short"
##  [73] "short" "short" "short" "short" "short" "short" "short" "short"
##  [81] "short" "short" "short" "short" "short" "short" "short" "short"
##  [89] "short" "short" "short" "short" "short" "short" "short" "short"
##  [97] "short" "short" "short" "short" "long"  "short" "long"  "long"
## [105] "long"  "long"  "short" "long"  "long"  "long"  "short" "short"
## [113] "short" "short" "short" "short" "short" "long"  "long"  "short"
## [121] "long"  "short" "long"  "short" "long"  "long"  "short" "short"
## [129] "long"  "long"  "long"  "long"  "long"  "short" "long"  "long"
## [137] "long"  "short" "short" "short" "long"  "short" "short" "long"
## [145] "long"  "short" "short" "short" "short" "short"
```

Create a new variable *LongSepals* for *df* that codes sepal length as "long" if the sepal length is greater than 6.5 cm and "short" otherwise.

Applications

Exercise 31 Create a new variable for the mtcars dataframe that measures the ratio of horsepower to displacement. Then code this variable as “high” if it exceeds 0.8 and “low” otherwise.

Exercise 32 Use selection on the mtcars dataframe to study the following questions. **Do cars with 6 or 8 cylinders tend to have a high or low horsepower to displacement ratio? How about cars with 4 cylinders?** Write a short paragraph with clear statements about your conclusions.

Exercise 33 Load the *airquality* dataset in R. Inspect it, and write a short paragraph describing the dataset. In particular, you should note any *NA* values in the dataset by describing how many *NA*’s are in each variable.

Exercise 34 How many of the observations in the *airquality* dataset are complete, i.e. have no *NA* values? Use R’s *complete.cases* command, together with any others that may be useful.

Exercise 35 Create a new dataset, stored in *df*, made up of only the complete observations in the airquality dataset.