STS 112 - Notebook 1

Roella Louise Pineda

Week 1: Getting Started With R and RMarkdown in RStudio

Before you go through this notebook, make sure you have installed R and RStudio, or that you are working on a computer with R and RStudio already installed.

Introduction

This is an RMarkdown notebook. It consists of chunks or blocks of text and code. The text is formatted in markdown, which is a very simple markup language. For example, you can make text italic by *putting an asterisk before and after*, and you can make text bold by **putting two asterisks before and after**. You can see the formatting by saving and clicking the `Preview` button above. Be sure to save the file before you click `Preview`, and make sure your web browser allows popups from RStudio Cloud. Previewing the notebook creates or updates a file with an `.nb.html` extension that you can also open in your web browser. In this notebook view, you can insert your cursor and edit markdown text at any point. To insert a **code block**, simply place your cursor on a new line and type three backticks (upper left of your keyboard) followed by `{r}`. To end the code block, type another three backticks. Any text after that will automatically be in markdown. This is a code block:

```
3 + 5
```

```
## [1] 8
```

This code will be explained in greater detail below. For now, simply note how the block is formatted. Also, notice the green arrow in the upper right corner of the code block. Clicking that will run all of the code in the block. Try it now. Notice that the output appears right below the code chunk. Often, when you are running a code block, the results will depend on other code blocks in a document. If you have already run the earlier code blocks, you are all set. If not, you can click the icon to the left of the green arrow to run this code block and all previous ones.

Getting Started With R

R is a high-level programming language typically used for statistics and data science. The phrase **high-level** means that it is highly abstracted from the machine code your computer understands, which makes it easier to work with for beginning programmers. R is an interpreted language, which means that it is executed one line at a time. A code block that you write in an RMarkdown notebook may be a single line or a series of lines. When you run a code block, all lines in the block will be sent to the interpreter, where they will be executed one at a time.

At its simplest, R is like a large calculator. You saw this in the example above. Try it yourself in the empty code block below. When you run it, the R interpreter will execute your code and print the results (in this case the result of evaluating the mathematical expression) to the **console** (here the space under the code block).

```
7 + 9 * 2 - 5
```

```
## [1] 20
```

What happens if you enter an incomplete mathematical expression? Run this block:

```
#2 +
```

You may write text in a code block by prefacing it with #. The remainder of the line will not be interpreted. This is called a **comment**. What happens when you run the code block below?

```
#This is a comment.
```

What happens when you run the next block?

```
"This is not a comment."
```

```
## [1] "This is not a comment."
```

As you saw, nothing happened to the text that was **commented out** with the `#`. Any text encased in quotation marks will get printed verbatim to the console. If you try to run text that is not either commented out or encased in quotation marks, R will throw an error, unless you have previously assigned a value to the text (we will discuss this below).

Hello World

Traditionally, the first thing one does when learning a new programming language is to write a program that prints the phrase "hello world" to the console. In R, that is exceptionally easy. If you have read to this point, you know how to do it. Write a line of code in the block below that prints "hello world" to the console.

```
"Hello World"
```

```
## [1] "Hello World"
```

Working With Variables

The real power in programming comes from creating and manipulating **variables**. A variable is simply an object you create that takes the value you assign to it. Creating a variable is R is very easy: You simply type the name of the variable, the **assignment operator** (`<-`), which is made by typing the less than sign followed by the minus sign, and the value you want the variable to take. Run the code below.

```
a <- 5
```

Notice that nothing was printed to the console. Nonetheless, R created a variable called `a` that is equal to 5. How do we know this? We can use the `ls()` function to list all of the objects in our working environment. Run the code below.

```
ls()
```

```
## [1] "a"
```

This tells us that we currently have one object in our working environment, and that it is called `a`. We can see the value of `a` by simply entering its name, as below. Run the following code block.

```
a
```

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

We can also create a variable from a mathematical expression. Run the code below to make a new variable.

```
b <- 5 + 3
```

What is the value of b?

You can ask R to print the value of a variable as you create it by wrapping the whole expression in parentheses.

```
(c <- 7)
```

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

You can give a variable any name you want as long as it doesn't start with a number or include spaces or special characters. Let's clear out our working environment and create some new variables. You can clear the working environment with `rm(list = ls())`. Run the following code block.

```
rm(list = ls())
a <- 3
b4 <- 5
cat <- a + b4 - 1
```

Use the `ls()` function to see what variables are currently in your environment, then check their values.

```
ls()
```

```
## [1] "a"   "b4"  "cat"
```

Now that we have created these variables, we can use them in mathematical expressions or, as you have seen, to create new variables. You can change the value of a variable any time you want, simply by reassigning it. In the code block below, create a new variable called `dog` that is equal to 12.

```
dog <- 12
```

As well as addition and subtraction, we can use variables in multiplication, division, and exponentiation:

```
cat*dog
```

```
## [1] 84
```

```
cat/dog
```

```
## [1] 0.5833333
```

```
cat**2
```

```
## [1] 49
```

```
dog/cat*cat
```

```
## [1] 12
```

You can also create variables whose value is a text string. The string must be wrapped in quotation marks, even if it is only a single word. Run the code below.

```
g <- "coding is"
hi <- "fun"
```

In the code block below, make some of your own string variables.

```
hello <- "My name is Roella."
pet <- "I have a pet bunny."
```

Data Types

So far, we have made two different types of variables: **numeric** and **character**. There is one more data type that we will cover at this point, called **logical**. A logical variable can take one of two values: `TRUE` or `FALSE`. Run the code below to see this in action.

```
newvariable <- 2 == 5 - 3
newvariable
```

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

```
anothernewvariable <- 2 != 5 - 3
anothernewvariable
```

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

The logical data type is created with **Boolean operators**. These are equal to `==`, not equal to `!=`, less than `<`, less than or equal to `<=`, greater than `>`, and greater than or equal to `>=`.

Vectors and Lists

By default, variables in R are treated as **vectors**. A vector is a one-dimensional array of numbers, strings, or logical values. A vector can have as many or as few elements as you want. The variables we have created so far are vectors of one element. You can make a vector with more than one element using the `c()` function. The code below makes vectors called k, l, and m.

```r
k <- c(1, 5, 12, 7, 378)
l <- c("it's late", "I'm tired", "good night", "see you in the morning")
m <- c(T, F, T, T, F)
```

Run the following code block to see how R displays these vectors.

```r
k
```

```
## [1]   1   5  12   7 378
```

```r
l
```

```
## [1] "it's late"              "I'm tired"              "good night"
## [4] "see you in the morning"
```

```r
m
```

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

Note that all elements of a vector must be the same data type, either numeric, string, or logical. You can do mathematical operations on numeric vectors. This will be important later. Run the code block below to see some examples.

```r
k * 2
```

```
## [1]   2  10  24  14 756
```

```r
k - 1
```

```
## [1]   0   4  11   6 377
```

```r
sum(k)
```

```
## [1] 403
```

You can reference any element of a vector using the name of the vector variable followed by the element number in square brackets.

```r
l[2]
```

```
## [1] "I'm tired"
```

```r
m[-1]
```

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

Data Frames

The data type we will be working with most in this class is the **data frame**. A data frame is a vertical array of vectors where each element of each vector refers to the same thing. The vectors that make up the data frame can be numeric, character, or some of each. A data frame is very similar to a spreadsheet.

To understand data frames, imagine you are running a day care center for pets. You take dogs, cats, and rabbits. The following code makes a series of vectors: `days` lists the days of the week; `dogs` lists the number of dogs you cared for on each day of the week (`dogs[1]` is the number of dogs you had on Monday, `dogs[2]` is the number of dogs you cared for on Tuesday, etc.); `cats` lists the number of cats you cared for each day; `rabbits` lists the number of rabbits you cared for each day.

```
days <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
dogs <- c(4, 3, 4, 2, 4)
cats <- c(1, 2, 1, 1, 2)
rabbits <- c(0, 1, 1, 0, 1)
```

We can now use the `data.frame()` function to convert these vectors into a data frame called `pets`.

```
pets <- data.frame(days, dogs, cats, rabbits)
```

Since this is a small data frame, we can view it directly, the same way we would view any other variable.

```
pets
```

```
##        days dogs cats rabbits
## 1    Monday    4    1       0
## 2   Tuesday    3    2       1
## 3 Wednesday    4    1       1
## 4  Thursday    2    1       0
## 5    Friday    4    2       1
```

Note that the names of the vectors that compose the data frame are now the names of the columns in the data frame. There are numerous ways of accessing the data in a data frame. You can reference a single value: `pets[2,3]` references the second row, third column, which is the number of cats on Tuesday (2). You can reference a whole column in two different ways: `pets$cats` and `pets[,3]` will both give you the cats vector. You can also reference whole rows: `pets[2,]` will give you the row for Tuesday. You can reference a subset of rows or columns: `pets[,2:4]` will give you columns 2-4 (dogs, cats, rabbits); `pets[2:4,]` will give you rows 2-4 (Tuesday, Wednesday, Thursday). You can see the column names with the `names()` function. Create a new code block below and try all of the examples listed in this text block.

```
pets[2,3]
```

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

```
pets$cats
```

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

```
pets[,3]
```

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

```
pets[2,]
```

```
##      days dogs cats rabbits
## 2 Tuesday    3    2       1
```

```
pets[,2:4]
```

```
##   dogs cats rabbits
## 1    4    1       0
## 2    3    2       1
## 3    4    1       1
## 4    2    1       0
## 5    4    2       1
```

```
pets[2:4,]
```

```
##        days dogs cats rabbits
## 2   Tuesday    3    2       1
## 3 Wednesday    4    1       1
## 4  Thursday    2    1       0
```

```
pets[2:3,2:3]
```

```
##   dogs cats
## 2    3    2
## 3    4    1
```

We can also create new columns in the data frame. For example, say we wanted a new column indicating the total number of animals on each day. We do it as if we were making a new variable, but using the `dataframe$column` notation.

```
pets$total <- pets$dogs + pets$cats + pets$rabbits
pets
```

```
##        days dogs cats rabbits total
## 1    Monday    4    1       0     5
## 2   Tuesday    3    2       1     6
## 3 Wednesday    4    1       1     6
## 4  Thursday    2    1       0     3
## 5    Friday    4    2       1     7
```

Now let's make another data frame. This one will have one row per customer for each day the customer brought in animals. Columns will list customer's name, day of the week, number of dogs, number of cats, and number of rabbits. The following code block contains the data for each column of the data frame.

```
name <- c("Al", "Bob", "Al", "Carmen", "Dana", "Al", "Bob", "Dana", "Al", "Al", "Bob", "Dana", "Evelyn")
day <- c("Monday", "Monday", "Tuesday", "Tuesday", "Tuesday", "Wednesday", "Wednesday", "Wednesday",
         "Thursday", "Friday", "Friday", "Friday", "Friday")
dogs <- c(2, 2, 2, 1, 0, 2, 2, 0, 2, 2, 2, 0, 0)
cats <- c(1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0)
rabbits <- c(0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1)
```

In the code block below, transform the vectors `name`, `day`, `dogs`, `cats`, and `rabbits` into a data frame called `customers` and display the data frame.

```
customers <- data.frame(name, day, dogs, cats, rabbits)
customers
```

```
##      name       day dogs cats rabbits
## 1      Al    Monday    2    1       0
## 2     Bob    Monday    2    0       0
## 3      Al   Tuesday    2    1       0
## 4  Carmen   Tuesday    1    0       0
## 5    Dana   Tuesday    0    1       1
## 6      Al Wednesday    2    1       0
## 7     Bob Wednesday    2    0       0
## 8    Dana Wednesday    0    0       1
## 9      Al  Thursday    2    1       0
## 10     Al    Friday    2    1       0
## 11    Bob    Friday    2    0       0
## 12   Dana    Friday    0    1       0
## 13 Evelyn    Friday    0    0       1
```

Now let's calculate how much we are going to bill each customer on each day. We charge $20 per day for dogs, $15 for cats, and $5 for rabbits. Create a new column in the data frame called `bill` that indicates how much we will charge each customer each day.

```
customers$bill <- customers$dogs *20 + customers$cats *15 + customers$rabbits *5
customers
```

```
##      name       day dogs cats rabbits bill
## 1      Al    Monday    2    1       0   55
## 2     Bob    Monday    2    0       0   40
## 3      Al   Tuesday    2    1       0   55
## 4  Carmen   Tuesday    1    0       0   20
## 5    Dana   Tuesday    0    1       1   20
## 6      Al Wednesday    2    1       0   55
## 7     Bob Wednesday    2    0       0   40
## 8    Dana Wednesday    0    0       1    5
## 9      Al  Thursday    2    1       0   55
## 10     Al    Friday    2    1       0   55
## 11    Bob    Friday    2    0       0   40
## 12   Dana    Friday    0    1       0   15
## 13 Evelyn    Friday    0    0       1    5
```

At the end of the week, Al and Bob complain that they are spending too much on day care for their pets. You want to keep their business and you don't want their animals to be home alone all day, so you consider

implementing a discount: 10% off each day a person brings in at least two two pets. To calculate their new bill, we will need to use the `ifelse()` function. This function takes three variables: the condition you are testing, what will be done if the test is true, and what will be done if the test is false. In this case, the condition we are testing is whether or not a customer brought in at least two pets on a given day (`customers$cats + customers$dogs + customers$rabbits > 1`). If it is true, we want to reduce the bill by 10% (`customers$bill * 0.9`). If it is false, we want to keep the amount of the bill the same. In the code block below, we use the `ifelse()` function to create a column called `newbill()` that indicates the amount of each customer's discounted bill.

```
#ifelse(condition, doiftrue, doiffalse)
customers$newbill <- ifelse(customers$cats + customers$dogs + customers$rabbits > 1,
                            customers$bill * 0.9, customers$bill)
customers
```

```
##      name       day dogs cats rabbits bill newbill
## 1      Al    Monday    2    1       0   55    49.5
## 2     Bob    Monday    2    0       0   40    36.0
## 3      Al   Tuesday    2    1       0   55    49.5
## 4  Carmen   Tuesday    1    0       0   20    20.0
## 5    Dana   Tuesday    0    1       1   20    18.0
## 6      Al Wednesday    2    1       0   55    49.5
## 7     Bob Wednesday    2    0       0   40    36.0
## 8    Dana Wednesday    0    0       1    5     5.0
## 9      Al  Thursday    2    1       0   55    49.5
## 10     Al    Friday    2    1       0   55    49.5
## 11    Bob    Friday    2    0       0   40    36.0
## 12   Dana    Friday    0    1       0   15    15.0
## 13 Evelyn    Friday    0    0       1    5     5.0
```

We can also nest `ifelse()` functions. In the code block below, we use two nested `ifelse()` functions to give customers 10% off if they bring in two pets and 15% off if they bring in three or more pets.

```
customers$newbill <- ifelse(customers$cats + customers$dogs + customers$rabbits > 2,
                            customers$bill * 0.85,
                            ifelse(customers$cats + customers$dogs + customers$rabbits > 1,
                                   customers$bill *0.9,
                                   customers$bill))
customers
```

```
##      name       day dogs cats rabbits bill newbill
## 1      Al    Monday    2    1       0   55   46.75
## 2     Bob    Monday    2    0       0   40   36.00
## 3      Al   Tuesday    2    1       0   55   46.75
## 4  Carmen   Tuesday    1    0       0   20   20.00
## 5    Dana   Tuesday    0    1       1   20   18.00
## 6      Al Wednesday    2    1       0   55   46.75
## 7     Bob Wednesday    2    0       0   40   36.00
## 8    Dana Wednesday    0    0       1    5    5.00
## 9      Al  Thursday    2    1       0   55   46.75
## 10     Al    Friday    2    1       0   55   46.75
## 11    Bob    Friday    2    0       0   40   36.00
## 12   Dana    Friday    0    1       0   15   15.00
## 13 Evelyn    Friday    0    0       1    5    5.00
```

Dplyr

That's a lot of typing! Fortunately, there is a **package** called `dplyr` that makes manipulating data frames much easier (think of it as a pair of pliers for working with a data frame). A package is a set of functions that does not come standard with R. In order to use external packages, we must install and load them. Installation has already happened in our working environment, so we just have to load `dplyr` with the command `library(dplyr)`.

```
#install.packages("dplyr")
library(dplyr)
```

```
## Warning: package 'dplyr' was built under R version 4.0.5
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

Dplyr contains several functions that make it very easy to manipulate data frames. The first one we will use is `mutate()`, which creates a new column. There are two ways to use `dplyr` functions. The first is with the name of the data frame as the first argument to the function. The second and more straightforward is with the **pipe** operator (%>%). Let's go back to our original customers data frame. We will use the first method to calculate the bill and the second to calculate the total number of animals per customer per day.

```
customers <- data.frame(name, day, dogs, cats, rabbits)
customers
```

```
##      name       day dogs cats rabbits
## 1      Al    Monday    2    1       0
## 2     Bob    Monday    2    0       0
## 3      Al   Tuesday    2    1       0
## 4  Carmen   Tuesday    1    0       0
## 5    Dana   Tuesday    0    1       1
## 6      Al Wednesday    2    1       0
## 7     Bob Wednesday    2    0       0
## 8    Dana Wednesday    0    0       1
## 9      Al  Thursday    2    1       0
## 10     Al    Friday    2    1       0
## 11    Bob    Friday    2    0       0
## 12   Dana    Friday    0    1       0
## 13 Evelyn    Friday    0    0       1
```

```
#First way (this creates a new column called "bill"):
customers <- mutate(customers, bill = dogs * 20 + cats * 15 + rabbits * 5)
#mutate(dataframe,new variable, = expression)
```

```
#Second way (this creates a new column called "animals"):
#dataframe %>% mutate(newvariable = expression)
customers <- customers %>% mutate(animals = dogs + cats + rabbits)
customers
```

```
##      name       day dogs cats rabbits bill animals
## 1      Al    Monday    2    1       0   55       3
## 2     Bob    Monday    2    0       0   40       2
## 3      Al   Tuesday    2    1       0   55       3
## 4  Carmen   Tuesday    1    0       0   20       1
## 5    Dana   Tuesday    0    1       1   20       2
## 6      Al Wednesday    2    1       0   55       3
## 7     Bob Wednesday    2    0       0   40       2
## 8    Dana Wednesday    0    0       1    5       1
## 9      Al  Thursday    2    1       0   55       3
## 10     Al    Friday    2    1       0   55       3
## 11    Bob    Friday    2    0       0   40       2
## 12   Dana    Friday    0    1       0   15       1
## 13 Evelyn    Friday    0    0       1    5       1
```

Do you see the difference between the two approaches above? Note that we can make as many new columns as we want with a single `mutate()` function. In the code block below, we will make both new columns (`bill` and `animals`) at the same time. Notice that we are making the data frame from its component vectors and then piping it into the `mutate()` function all in one command.

```
customers <- data.frame(name, day, dogs, cats, rabbits) %>%
               mutate(bill = dogs * 20 + cats * 15 + rabbits * 5, animals = dogs + cats + rabbits)
customers
```

```
##      name       day dogs cats rabbits bill animals
## 1      Al    Monday    2    1       0   55       3
## 2     Bob    Monday    2    0       0   40       2
## 3      Al   Tuesday    2    1       0   55       3
## 4  Carmen   Tuesday    1    0       0   20       1
## 5    Dana   Tuesday    0    1       1   20       2
## 6      Al Wednesday    2    1       0   55       3
## 7     Bob Wednesday    2    0       0   40       2
## 8    Dana Wednesday    0    0       1    5       1
## 9      Al  Thursday    2    1       0   55       3
## 10     Al    Friday    2    1       0   55       3
## 11    Bob    Friday    2    0       0   40       2
## 12   Dana    Friday    0    1       0   15       1
## 13 Evelyn    Friday    0    0       1    5       1
```

Now we will calculate the discounted bill using the `ifelse()` function together with the `mutate()` function. Note that this is exactly the same as what we did before, but we are now using the new column `animals` instead of adding the number of dogs, cats, and rabbits.

```
customers <- customers %>% mutate(newbill = ifelse(animals > 2, bill * 0.85,
                                                   ifelse(animals > 1, bill * 0.9, bill)))
customers
```

```
##      name       day dogs cats rabbits bill animals newbill
## 1      Al    Monday    2    1       0   55       3   46.75
## 2     Bob    Monday    2    0       0   40       2   36.00
## 3      Al   Tuesday    2    1       0   55       3   46.75
## 4  Carmen   Tuesday    1    0       0   20       1   20.00
## 5    Dana   Tuesday    0    1       1   20       2   18.00
## 6      Al Wednesday    2    1       0   55       3   46.75
## 7     Bob Wednesday    2    0       0   40       2   36.00
## 8    Dana Wednesday    0    0       1    5       1    5.00
## 9      Al  Thursday    2    1       0   55       3   46.75
## 10     Al    Friday    2    1       0   55       3   46.75
## 11    Bob    Friday    2    0       0   40       2   36.00
## 12   Dana    Friday    0    1       0   15       1   15.00
## 13 Evelyn    Friday    0    0       1    5       1    5.00
```

There are three other `dplyr` functions you will need to do this week's lab: `arrange()`, `select()`, and `filter()`. You can use `arrange()` to organize your data frame in ascending or descending order on the basis of one or more columns. For example, we can sort our data alphabetically by customer:

```
customers %>% arrange(name)
```

```
##      name       day dogs cats rabbits bill animals newbill
## 1      Al    Monday    2    1       0   55       3   46.75
## 2      Al   Tuesday    2    1       0   55       3   46.75
## 3      Al Wednesday    2    1       0   55       3   46.75
## 4      Al  Thursday    2    1       0   55       3   46.75
## 5      Al    Friday    2    1       0   55       3   46.75
## 6     Bob    Monday    2    0       0   40       2   36.00
## 7     Bob Wednesday    2    0       0   40       2   36.00
## 8     Bob    Friday    2    0       0   40       2   36.00
## 9  Carmen   Tuesday    1    0       0   20       1   20.00
## 10   Dana   Tuesday    0    1       1   20       2   18.00
## 11   Dana Wednesday    0    0       1    5       1    5.00
## 12   Dana    Friday    0    1       0   15       1   15.00
## 13 Evelyn    Friday    0    0       1    5       1    5.00
```

Within the record for each customer, we can also sort by number of animals brought in each day:

```
customers %>% arrange(name, animals)
```

```
##      name       day dogs cats rabbits bill animals newbill
## 1      Al    Monday    2    1       0   55       3   46.75
## 2      Al   Tuesday    2    1       0   55       3   46.75
## 3      Al Wednesday    2    1       0   55       3   46.75
## 4      Al  Thursday    2    1       0   55       3   46.75
## 5      Al    Friday    2    1       0   55       3   46.75
## 6     Bob    Monday    2    0       0   40       2   36.00
## 7     Bob Wednesday    2    0       0   40       2   36.00
## 8     Bob    Friday    2    0       0   40       2   36.00
## 9  Carmen   Tuesday    1    0       0   20       1   20.00
## 10   Dana Wednesday    0    0       1    5       1    5.00
## 11   Dana    Friday    0    1       0   15       1   15.00
## 12   Dana   Tuesday    0    1       1   20       2   18.00
## 13 Evelyn    Friday    0    0       1    5       1    5.00
```

We can do the same thing in descending order of number of animals (this really only makes a difference for Dana, who brings her rabbit on Wednesday and her cat on Friday and both on Tuesday):

```
customers %>% arrange(name, -animals)
```

```
##      name       day dogs cats rabbits bill animals newbill
## 1      Al    Monday    2    1       0   55       3   46.75
## 2      Al   Tuesday    2    1       0   55       3   46.75
## 3      Al Wednesday    2    1       0   55       3   46.75
## 4      Al  Thursday    2    1       0   55       3   46.75
## 5      Al    Friday    2    1       0   55       3   46.75
## 6     Bob    Monday    2    0       0   40       2   36.00
## 7     Bob Wednesday    2    0       0   40       2   36.00
## 8     Bob    Friday    2    0       0   40       2   36.00
## 9  Carmen   Tuesday    1    0       0   20       1   20.00
## 10   Dana   Tuesday    0    1       1   20       2   18.00
## 11   Dana Wednesday    0    0       1    5       1    5.00
## 12   Dana    Friday    0    1       0   15       1   15.00
## 13 Evelyn    Friday    0    0       1    5       1    5.00
```

The `filter()` function allows us to see only the rows where a certain condition is true. For example, say we only wanted to see the records for Bob:

```
customers %>% filter(name == "Bob")
```

```
##   name       day dogs cats rabbits bill animals newbill
## 1  Bob    Monday    2    0       0   40       2      36
## 2  Bob Wednesday    2    0       0   40       2      36
## 3  Bob    Friday    2    0       0   40       2      36
```

What if we wanted to see everyone except Bob? Do that in the code block below:

```
customers %>% filter(name !="Bob")
```

```
##      name       day dogs cats rabbits bill animals newbill
## 1      Al    Monday    2    1       0   55       3   46.75
## 2      Al   Tuesday    2    1       0   55       3   46.75
## 3  Carmen   Tuesday    1    0       0   20       1   20.00
## 4    Dana   Tuesday    0    1       1   20       2   18.00
## 5      Al Wednesday    2    1       0   55       3   46.75
## 6    Dana Wednesday    0    0       1    5       1    5.00
## 7      Al  Thursday    2    1       0   55       3   46.75
## 8      Al    Friday    2    1       0   55       3   46.75
## 9    Dana    Friday    0    1       0   15       1   15.00
## 10 Evelyn    Friday    0    0       1    5       1    5.00
```

The `select()` function allows us to see only a **subset** of the columns. For example, if we just wanted to see name, day of the week, and total number of animals for each customer, there are two ways to do it:

```
#First way:
customers %>% select(name, day, animals)
```

```
##      name       day animals
## 1      Al    Monday       3
## 2     Bob    Monday       2
## 3      Al   Tuesday       3
## 4  Carmen   Tuesday       1
## 5    Dana   Tuesday       2
## 6      Al Wednesday       3
## 7     Bob Wednesday       2
## 8    Dana Wednesday       1
## 9      Al  Thursday       3
## 10     Al    Friday       3
## 11    Bob    Friday       2
## 12   Dana    Friday       1
## 13 Evelyn    Friday       1
```

```
#Second way:
customers %>% select(-dogs, -cats, -rabbits, -bill, -newbill)
```

```
##      name       day animals
## 1      Al    Monday       3
## 2     Bob    Monday       2
## 3      Al   Tuesday       3
## 4  Carmen   Tuesday       1
## 5    Dana   Tuesday       2
## 6      Al Wednesday       3
## 7     Bob Wednesday       2
## 8    Dana Wednesday       1
## 9      Al  Thursday       3
## 10     Al    Friday       3
## 11    Bob    Friday       2
## 12   Dana    Friday       1
## 13 Evelyn    Friday       1
```

Note that when we were working with `arrange()`, `filter()`, and `select()` in the code blocks above, we did not change the `customers` data frame: we only viewed the results of our actions. As you can see, the `customers` data frame remains as it previously was:

```
customers
```

```
##      name       day dogs cats rabbits bill animals newbill
## 1      Al    Monday    2    1       0   55       3   46.75
## 2     Bob    Monday    2    0       0   40       2   36.00
## 3      Al   Tuesday    2    1       0   55       3   46.75
## 4  Carmen   Tuesday    1    0       0   20       1   20.00
## 5    Dana   Tuesday    0    1       1   20       2   18.00
## 6      Al Wednesday    2    1       0   55       3   46.75
## 7     Bob Wednesday    2    0       0   40       2   36.00
## 8    Dana Wednesday    0    0       1    5       1    5.00
## 9      Al  Thursday    2    1       0   55       3   46.75
## 10     Al    Friday    2    1       0   55       3   46.75
## 11    Bob    Friday    2    0       0   40       2   36.00
## 12   Dana    Friday    0    1       0   15       1   15.00
## 13 Evelyn    Friday    0    0       1    5       1    5.00
```

If we had wanted to overwrite the `customers` data frame with the results of any of these operations, we simply would have had to add customers <- at the beginning, just as we did when we were working with the mutate() function. Alternatively, we could have saved the results as a new data frame, by giving it a new name (`newname <- customers %>% arrange(...)`. Let's do one more example to show how you can use all of these functions together. Let's say you wanted to send a coupon to customers with dogs. The code block below makes a data frame that lists each person's name, the number of dogs they brought in on any given day, and the number of other animals they brought in with their dog. It includes only dog owners and is sorted alphabetically by name. Notice that this is one "line" of code, but it is split up over multiple lines to make it more readable. When you do that, R needs some indication to keep reading to the next line. In this case, that indication is the pipe operator.

```
dogowners <- customers %>% mutate(other = cats + rabbits) %>%
                           select(name, day, dogs, other) %>%
                           filter(dogs >=1) %>%
                           arrange(name)
dogowners
```

```
##     name       day dogs other
## 1     Al    Monday    2     1
## 2     Al   Tuesday    2     1
## 3     Al Wednesday    2     1
## 4     Al  Thursday    2     1
## 5     Al    Friday    2     1
## 6    Bob    Monday    2     0
## 7    Bob Wednesday    2     0
## 8    Bob    Friday    2     0
## 9 Carmen   Tuesday    1     0
```

You now have all the tools you need to complete this week's lab assignment! To turn in this notebook, save and preview this file. Then upload the `Notebook1.nb.html` file to Canvas under Notebook 1.