Lab #1, Part #2 – Introduction to R-Studio

Last time, we built a data set in Excel and saved it as a CSV file. This week, we'll begin to move away from Excel, and into the statistical programming package **R-Studio** (or R for short). We will use this throughout the rest of the semester. I will note that you are likely to get frustrated and find some difficulty in learning R, but that being hands on with it and problem solve is one of the best ways to learn. I will provide substantial support here in lab and any time you might run into an issue, so please use me as a resource as you go through labs.

Activity #1: Installing R-Studio. Begin by going to this link and downloading R and R-Studio. I suggest downloading R Studio Desktop (the free version!) so that it's always on your computer. This avoids having to access it via web browser, and there should be downloads for various operating systems (i.e. Windows or Mac). It should download in a straight forward way, and at this point I suggest just doing the default download procedure (without customization). Once everything is installed, go ahead and open up R Studio.

Activity #2: R As a Calculator. You'll notice there are multiple windows opened up. We can start by playing around in R together and treating it as a calculator. On the left-hand side, there should be a cursor where you can type commands. Click over there and type:

```
2 + 2
```

Then press **Enter**. You should see that it works pretty well at adding things together by showing you the correct answer, 4, below. Now on your own, try **subtracting**, **multiplying**, and **dividing** any numbers you'd like. Note that for multiplication you'll use the **asterisk** (*), and dividing you use the **slash** (/).

Keep going with this with the following below:

Exponents: `2^2`
Parentheticals: `(2+2)*(2-2)` and then try `2+2*2-2` to see that they provide meaning to orders of operations. Note that R Studio helps you complete the parentheses, so be careful here.

Throughout, notice that R will always provide the answer to you right there below your equation. It also keeps track of all the different things you've typed in since you started the R instance.

Activity #3 R Objects. One problem with all your answers is that although R keeps them shown, you can't recover them in some way. However, R allows you to assign names to anything you create in it. Let's return to 2 + 2, but this time, we're going to name it. R uses the **<-** symbol to **assign** something to a name. We'll name our result **A**, for addition (note that R is case-sensitive!). To do so, use the following:

```
A <- 2+2
```

Then type `A`. It should return or **print** the result of `4`. Now, let's assume we want to know the answer to:

```
7*(2+2)
```

We could of course type that directly into R. Alternatively, we can use our new object, A, instead of the parenthetical. Try the following, then see if the equation above ends up with the same answer:

```
7*A
```

Hopefully both returned `28`. Now let's reassign `A`. Instead let's do:

```
A <- 7+1
A
```

Now try `7*A` again. Hopefully this time you can see that `A` is equal to `8`, and your new result is `56`. This is important to remember: if you want to assign multiple objects, you'll need to choose different names for them. Let's now make a second object, `B`, which we will make equal to `3+5`:

```
B <- 3+5
B
```

And then, since `A` and `B` are objects that are numeric, we should be able to perform any mathematical operations we want on them alone such as:

```
A+B
A-B
A*B
A/B
```

Try these out on your own and make sure they seem to be working right. Then, assign a new object that is equal to `A^B`. Let's call this one `Z`. What is the value of `Z` in R now?

Activity #4: Logical Operators. R can not only calculate things, but it can check whether or not something is true. For example, we've now defined A, B, and Z. It's easy to see that A and B are both 8 in the Global Environment box (or by typing them in the console). But let's use these to do some quick checks. In the console, type:

```
A == B
```

You should get an answer that says `TRUE`. Now type:

```
A == Z
```

Here, we should get the answer `FALSE`, since `Z` is a very large number not equal to `8`. (Side note: these are also called **Booleans).** Next, try and see that `Z` is equal to the equation we used to calculate it:

```
Z == A^B
```

Excellent, it worked! Note that here we are using a double equals sign, rather than a single one. `A` single = actually works similarly to the object assignment code, `<-`. So always be careful here. See what happens when you type:

```
Z = A
```

There is nothing returned. Go ahead and see what the value of `Z` is now (it should be `8`, because we've assigned `Z` the value of `A`).

Finally, note that here are other logical operators of interest as follows:

Less than: <
Less than or equal to: <=
Greater than: >
Greater than or equal to: >=
Not equal to: `!=`

Reassign `Z <- A^B` back to its original value, and use the operators above to see what the results are for each one with each pair `A` and `Z` and `A` and `B`. These logical operators will become important later down the road, in addition to **and** (`&`) and **or** (`|`) operators.

Activity #5: Saving R Script. Notice that while you have all your code shown in the R window, and you can assign objects that R remembers (note that they're all shown in a list in the top right window called **Global Environment**), you don't have that code saved anywhere when you close R. You may want to return to the code, edit it, or run it again at some point in the future. Remembering all the code, or typing it all out again by hand, is a very poor way to do that. Instead, R allows you to open a new window called an R Script.

At the top left of the program, just under **File**, there is a little white page with a green plus sign. Click that, and then click **R Script**. A new window should pop up with a blank area to type text.

In this new window, type `2 + 2` and press **ENTER** like you have before. Nothing will happen other than going to the next line. To run the code that you're now typing, you'll need to highlight the line and press **CTRL+ENTER**. Note that there is also a **Run** button at the top right of the script window. This is fine to use, too. Notice that it should now run down in the **Console** window.

Activity #6: Save Your Script. So that you can come back to this script at some point, you'll want to save it. Let's save it as **R Script - Lab 1-2** in the folder you created last week. Click the little save button at the top of the script window to do this. It will save automatically as an R file, but note that you can easily open it outside of R in Notepad or other text file reading programs. This is important so that it's not tied to a specific program, just like we wanted our data file to be in a general comma separated format.

Activity #7: Code Notation and Commenting Out. Once code starts getting longer and more detailed, it's going to be important to include notes in your R Script about what each piece of code is doing. It turns out that it's often really hard to remember what you were trying to do when you return to code later. Believe me: I've come back to a 2,000 line R Script with no clue why I was doing what I was doing.

Let's start by giving this a header title so you know what it is when you open it up again. We can just name it as we did with the file: **R Script – Lab 1-2**. Go ahead and do that above the `2+2` line. Highlight the entire script (both lines) and press **Run** (or **CTRL+ENTER**) again. It will come up with an error because it doesn't know what to do with random text. To avoid this, we use the **hash** or **pound sign** (`#`) to indicate to R, "Hey, this is just text for me, please don't do stuff with it." Just type a `#` at the left-hand side of your script title and run it again. It will show the text in the R Console, but it won't try to do anything with it.

Note that using the # can also help to "comment out" code you may want to work with later, but don't want to run now. Add another line below the script title that says `#2+7`. Now run the whole script. R will not run anything with the `#2+7` line because you asked it not to. Now remove the hash from that line and run the whole thing again. Hopefully it gives you answers for both `2+7` and `2+2`. Commenting will become more and more important as we go forward, and note that you can use as many hashes as you want (sometimes this is useful for noting very big breaks or changes in what you're doing within the R Script).

Side Note: Using the `#` symbol will not send your code to any aggregation of hashtags on Twitter or Instagram.

Activity #8: Making a Vector. R doesn't just allow us to assign a single object a single value. We can actually put a bunch of numbers or text into what is called a **vector**. A vector holds multiple things within it, but knows each one is separate. We can create a vector of numbers using the `c()` function. Make a new comment in your code below your `2+2` (skip a line to have some space) and create a vector as follows:

```
#goofing around with vectors
myVec <- c(1,2,3,4,5)
myVec
```

Your vector should return in the R console as a row of the numbers you included. Just below that, go ahead and make a vector of fruits. Be sure that you have the words in quotes, otherwise R will think they're objects.

```
fruit <- c("apple", "orange", "pear", "banana", "cherry")
fruit
```

Notice that R keeps the quotes for each fruit. We can check that R knows the type of information stored by using the `class()` function. (A side note here that a function is just a program that performs a specific task for us. We'll talk about functions throughout the course and use them extensively). Check for each one:

```
class(myVec)
class(fruit)
```

Hopefully it has **numeric** and **character**, respectively. Character variables are effectively nominal variables for our purposes today, but it really just means R is recognizing them as text.

Finally, let's show that we can have a vector of objects we've created to see what the double quotes are doing above. Type the following:

```
apple <- "apple"
orange <- "orange"
pear <- "pear"
banana <- "banana"
yummers <- "cherry"

newFruit <- c(apple, orange, pear, banana, yummers)
newFruit
fruit
```

Notice that for `newFruit`, you did not have to use quotes because you've already assigned each fruit to an object of its own name (except for `cherry`). By assigning `cherry` to `yummers`, where `yummers` is the object name, `yummers` substitutes for `"cherry"` in the `newFruit` vector. When the vector is returned, it looks just like the `fruit` vector.

Let's go back to our logical function from before. It turns out we can do this to see if vectors are the same, too. Type the following code:

```
newFruit == fruit
```

You should get a result that shows `TRUE` for all 5 of the values in the vector. You can also apply math to vectors to get a new vector:

```
myVec + myVec
```

Activity #9: Making a Data Frame. Objects and vectors are neat and all, but they're not on their own a data set. Remember that we like having a nicely structured data set with rows of observations and columns of variables. R understands how data matrices are organized, and has a class called a **data frame**. Let's start by thinking about binding our `fruit` and `myVec` vectors together into a matrix. We'll use the column bind function `cbind()`, the row bind function `rbind()` to make a **matrix**. Then we'll move to using the function that tells R we want it to be a data matrix, or data frame, with the function `data.frame()`.

Let's start by binding together the `myVec` and `fruit` vectors as rows in a matrix with `rbind()`. Be sure to put a new comment in your code to indicate we've started the next activity.

```
#Activity 9: matrices and data frames
rbind(myVec, fruit)
```

Notice that it just prints the matrix out with column numbers here. We can save a matrix as an object just like we did with individual values and vectors, so instead now type:

```
fruitRow <- rbind(myVec, fruit)
fruitRow
```

Hopefully the same matrix popped up. Notice that it will order the rows as you type them in: `myVec` is first (numbers) and `fruit` is second (fruits). We can check on the class of this object with:

```
class(fruitRow)
```

You can also check the class of each row or column of the matrix by using basic matrix notation. For example, in a 2x2 matrix, the top left cell can be represented by `[1,1]`, the top right by `[1,2]`, the bottom left as `[2,1]`, and the bottom right as `[2,2]`. Row number is first and column number is second. So you can check class of the first row by specifying just the row or column:

```
class(fruitRow[1, ])
class(fruitRow[ ,1])
```

And you can do the same with individual cells:

```
class(fruitRow[1,3])
```

Now: do you notice anything wrong with the `myVec` row? What is its class? Is this what we want it to be? We'll deal with this later, but it's something you'll have to constantly monitor in R.

Now let's make the matrix by column using `cbind()`:

```
fruitCol <- cbind(myVec, fruit)
fruitCol
```

You can see now that the **vectors** are listed as **columns**. In general, it's good for us to think about vectors as columns in the data: these vectors will represent our **variables**.

Let's finish up by telling R that this is data, not just a matrix. It turns out the `data.frame()` function does this for us, and assumes that vectors are columns with the following code:

```
fruitDat <- data.frame(myVec, fruit)
fruitDat

class(fruitDat)
```

This looks a bit tidier as well. Notice that it numbers the rows starting at 1, and it **names** the columns with our vector names. Again, these are our variables, conveniently organized like we did in Excel. You can call an entire vector or any cell or collection of cells from a data frame in two ways. First, you can treat it just like a matrix with:

```
fruitDat[, 1]
fruitDat[, 2]
fruitDat[2, 2]
```

You can also call individual variables (vectors) using the **dollar sign** operator, `$`.

```
fruitDat$myVec
fruitDat$fruit
```

You've probably noticed that for the `fruit` variable, R returns `levels` along with it. That's because R is now recognizing this as a factor, or **nominal** variable, with different **categories**. It has encoded those categories itself as the different fruit names. You can check that it's a factor, and that `myVec` remains numeric, again with the `class()` function:

```
class(fruitDat$myVec)
class(fruitDat$fruit)
```

The R console should return "`numeric`" and "`factor`" for each of the above.

Activity #10: Loading CSV Data. Obviously, as it did with Excel, the process of typing out individual data and binding it all together is cumbersome. However, R is able to read a number of data formats into it and recognize it as data. In this course, we'll use the function `read.csv()` to do this. Let's start with a new comment indicating you're now on Activity #10 in your code:

#Activity 10: Loading csv Data

Before we load any data into R, we're going to have to tell R where to look for the data. We do this with the **set working directory** function, or `setwd()`. This is why I suggested specific names for your folders. Go to your folder where your data from last week are and right click and then click **Properties**. In the window, you should be able to see a long **Location**. You'll need to type

this into R. Once you do this, it will be set for the rest of your session. You can do it as follows (adjusting based on the location of your data on your own computer):

```
setwd("c:/Users/…/Lab 1")
```

Notice that, while the backslash is used in the Location in Properties, you'll use a front slash in R. Now it's time to load in the data. For now, let's just call it `dat` (never use the name `data`, as this is a function in R, and you want to make sure you don't name objects with the same name as existing functions!). To load in the data, we'll use the `read.csv()` function, and assign it the name. To see how to use this function, first type in your R script:

```
help(read.csv)
```

You should see that a window pops up in the bottom right that includes a help file. It includes information about the options for each function. We will make use of the `header` option to tell R that, indeed, we have already entered headers (our variable names) in the CSV file. You should be able to load the data with the following code. If the data loaded appropriately, you should be able to type `dat`:

```
dat <- read.csv(file = "TouristDat.csv", header = TRUE)
dat
```

Note that instead of `TRUE`, you can also just use `T`. Notice that the equal sign is used inside functions. This is the main reason we'll stick with the `<-` code to assign objects: it avoids confusion inside functions and outside functions.

And print out the data set. It's a bit messier than looking at things in Excel, and it does not print out all rows of the data. There are a few things we can do to make sure it loaded right. First, we can look at just the first 6 rows of the data using the `head()` function or the last 6 rows with the `tail()` function This is helpful because it allows you to make sure all the variables made it into R. We can also check that there are the right number of rows (134) in the data by counting them with `nrow()`:

```
head(dat)
tail(dat)
nrow(dat)
```

If you really want to see the entire data set in the R Console, you can reassign the maximum printed rows for your R program using `max.print()` by resetting the number of total cells that will be printed (displayed) in the console (there are 9x134 = 1206 cells in our data). I don't recommend doing this, as `nrow()` and `head()` and `tail()` often give us enough to know that the data is correctly loaded.

```
options(max.print = 1500)
```

NOTE: When you close R Studio, it will ask you if you want to save your workspace. To simplify things for our purposes, just say Don't Save, as this will save everything you've got in R. However, be sure you save your R Script! You can run it once you load it back up and start where you left off.

Lab 1-2 Practice Questions. These questions should be completed prior to turning in your final lab. Please include all code (commented with the question number) in the same R Script that you used in class for each question.

1. We showed that R has some easily usable mathematical capabilities today in lab. These can be used on data sets in addition to how we used them on individual numbers and objects. Let's try that here. I want you to use some new functions and apply them to find the mean, median of population, median household income, and tourist revenue. Let's start by using a new function, `sum()` together with `nrow()`. Note that `sum()` will sum a column you give it, while `nrow()` will be the number of observations (row). Remember that the mean is calculated as:

$$\frac{\sum_{i=1}^{N} x_i}{N}$$

   Here, the `sum()` function can serve as the numerator and the `nrow()` function as the denominator, and is just equal to N. Remember that you can call a variable (vector) using the dollar sign. So `dat$Population` would call the population vector, `dat$MedHHInc` calls the median household income vector, and `dat$TouristRev` calls the tourist tax revenue vector. I'll show the calculation for population, then you try the other two variables:

```
sum(dat$Population)/nrow(dat)
```

   Try this in your console, and be sure that you also get `295,749`. Now do the same for the other two variables. And report your answer. Remember to keep all your clean working code that you use in your R Script for submission with lab.

   Ok. Now that we've gone through that, let's instead use R's built-in functions instead. As it turns out, R has a function called `mean()`. It also has one for `median()`. Let's use these on each variable and make sure we get the same answer. Again, I'll start with the population, and you do the other two variables. Remember to record your answers in your lab response and save your code in the R Script.

```
mean(dat$Population)
median(dat$Population)
```

2. Now that we've got some measures of central tendency, it's worth noting that R can also calculate a standard deviation using the `sd()` function. Go ahead and calculate the standard deviation for each of these variables. Which one has the largest standard

deviation? Is there a problem comparing standard deviation across these variables? Why or why not?

3. We can actually go one step further and reduce the amount of code we use by implementing the `summary()` function. You can do this in different ways. One is to type, for example:

   ```
   summary(dat$Population)
   ```

   This will give you a summary of population, including each quartile, the minimum, the maximum, the mean, and the median. You can also try this on non-numeric variables like month:

   ```
   summary(dat$Month)
   ```

   Note that because this variable is nominal, R only returns the count of each category. You can also apply the `summary()` function to the entire data set at once with:

   ```
   summary(dat)
   ```

   Report the 1st quartile, 3rd quartile, minimum, and maximum for each of the numeric variables, now including `TouristRate` (the tourist tax rate).

4. We'll now introduce the `table()` function. This allows you to cross-tabulate categorical variables. For our purposes, we'll treat `TouristRate` as a categorical variable (it takes 6 discrete values). Using table, create a table of tax rates and the coast category of each county with the following code:

   ```
   table(dat$Coast, dat$TouristRate)
   ```

   How many `Atlantic` counties have a tax rate above 3%? How many `Landocked` counties have a tax rate above 3%? What percentage of `Gulf` counties have a tax rate below 5%?

5. For this final question, we'll use a new function called `aggregate()` to find group means. This function allows you to apply another function, such as `mean()`, for each set of groups identified by some other variable. For example, we might want to know the average tourist tax revenue separately for Coastal and Non-Coastal counties to make a comparison. We would tell the function to calculate this with:

   ```
   aggregate(TouristRev ~ Coastal, data = dat, FUN = "mean")
   ```

   Note that the `aggregate()` function results in a data frame. Name the one above `RevCoastAgg` and assign using the assignment operator "<-". Now type `RevCoastAgg` to see the result. We'll use this ability to make new data frames later in the course. The squiggle "~" tells R that we want to know the means of `TouristRev` as

a function of the `Coastal` category they're in. This symbol will come back in various forms throughout the course. Also notice that we separate each argument by a comma, and then end by typing the function we want to use. Use the above code to report the average collections for `Coastal` and non-`Coastal` counties. From there, do the same using the `Coast` variable (`Atlantic`, `Gulf`, and `Landlocked`). Is there much difference between them? Do the same group level means for population and median household income as well.

Make sure you record all code used for this assignment and submit it with the answers to all lab questions. This includes the lab portion that took place in class.