Lab 1, Fall 2019

Your Friendly GEs!

10/8/2019

Introduction:

Hi everybody, I'm your friendly neighborhood GE and I will be helping you get through 421!

We're going to start with a walkthrough of R.

Lesson 0: Basics

In general:

- We will be working with what is called a *script*. This is similar to a do file in Stata. It's basically your workspace.
- To execute a script, hit cmd+return (control+enter if on PC.) To save the script, cmd+s. There are other shortcuts as well. If you want to run a specific line, then you can move your cursor to that line and hit cmd+return and the R script will only run that one line.
- R uses *functions*, which we apply to *objects*. More on this shortly, but if you aren't sure what a function does, or how it works, you can use ? before the function to get the documentation. Ex:

```
?mean
```

There are a ton of different types of objects (numeric (numbers), character (letters) and logical (true false statements) are the most common types), and not all functions will work on all objects. Let's talk a bit about objects.

Lesson 1: All things are objects

An object is an assignment between a name and a value. You assign values to names using `<-` or `=`. The first assignment symbol consists of a `<` next to a dash `-` to make it look like an arrow.

If we want to make an object name 'a' refer to the number '2', we can do that by:

```
#assign the value '2' to the name 'a'
a <- 2
```

We can do the same thing using the 'equals' sign

```
a = 2
a
```

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

When I run this code, we see an output that looks like `[1] 2`. The `[1]` refers to the output line. There is only one line here, as we only called one object. The `2` is the value associated with our object. In this case, `a`, which we set equal to 2.

You can combine objects together as well which lets us do some basic math operations

If you want to print intermediate steps of your code (ie, see them show up in your terminal) you can put parentheses around your code to get them to display

Let's find the value of `2*3` (equal to two times three), which should be equal to 6. Since a is already equal to 2, we can use that and add it to another variable. Of course, we could simply type `2*3`, but this way let's me show things off.

```
#assign the value of 3 to the name b
b <- 3
#assign the value of b (3) times the value of a (2), to a new name, c (now 6). Remember, parentheses wi
(c <- a * b)
```

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

```
#display c
(c)
```

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

objects however can hold more than one thing inside of them. For instance, we can make vectors. R also has a cool method to create a vector of integers with the colon operator.

```
#let's create a vector with the integers from 1 through 10.
tmp <- 1:10
```

We can also create vectors with the following syntax:

```
avect <- c(1,4,6,-10,-5.25)

(avect)
```

```
## [1]   1.00   4.00   6.00 -10.00  -5.25
```

we can even apply mathematical operators to vectors. Let's square this vector (this will square all values)

```
(avect^2)
```

```
## [1]   1.0000  16.0000  36.0000 100.0000  27.5625
```

one caveat though... R has two tendencies/rules to keep in mind. One, vectors have to all be the same 'type' of object, and two, R tries to help a little too much.

Let's make a 'bad' vector: we'll add to our temporary vector a few 'character' values, and one 'Null' value. We will work with this guy later.

```
#make a vector that doesn't behave itself
bad <- c(avect, 11, "red", "dinosaurs", NULL)
```

before we look into what goes wrong in 'bad' let's learn a bit about functions. We've actually worked with a function already! The 'c' is actually a function. Let's do a quick overview of how functions work in R.

Lesson 2: Functions

Functions are operations that can transform your created **object** in a ton of different ways. Let's look at some convenient ways to get a snapshot of the data, and summary statistics

Examples: `head`, `tail`, `mean`, `median`, `sd`, `summary`, `min`, `max`

These functions are good at summarizing data in a variety of ways. Let's see how they work

```
#print the first few objects in 'tmp'
head(tmp)
```

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

```
#print the first 3 objects in 'tmp'
head(tmp, 3)
```

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

```
#print the last few objects in 'tmp'
tail(tmp)
```

```
## [1]  5  6  7  8  9 10
```

Now let's look at what happened to our 'bad' vector.

```
#let's print the last 6 objects in 'bad'
tail(bad,6)
```

```
## [1] "6"         "-10"       "-5.25"     "11"        "red"       "dinosaurs"
```

Interesting. Let's circle back to this.

We can also use these to perform some basic or commonly used statistics, without the hassle of typing in the formula explicitly.

```
#mean of our vector tmp
mean(tmp)
```

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

```
#median of our vector
median(tmp)
```

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

```
#standard deviation of our vector
sd(tmp)
```

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

IMPORTANT We can also print a `summary` of our object.

```
#This can work on many object types and is useful to get an overview of the object in general
summary(tmp)
```

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

Lesson 3: types of Objects

So far, we've mostly worked with numeric objects, which can be integer or numeric. you can detect what sort of object you're working with using the `class()` function.

```
(class(a))
```

```
## [1] "numeric"
```

```
(class(tmp))
```

```
## [1] "integer"
```

```
(class(avect))
```

```
## [1] "numeric"
```

another common class is called `character`

Let's create a character object. These are surrounded by either "" or ''. This distinguishes them from *object names*.

```
(some_text <- "calvin go do something you hate, it builds character")
```

```
## [1] "calvin go do something you hate, it builds character"
```

```
class(some_text)
```

```
## [1] "character"
```

with that in mind, let's look at 'bad' again. Remember, it has the numbers 1-10 at first, so we'd expect those values to be stored as integers.

```
#we can use the 'head' function to look at the first value of our 'bad' vector. Remember, it should be
class(head(bad,1))
```

```
## [1] "character"
```

huh, it seems like R tried to be helpful and make all of your objects into character values. This is important to keep in mind as you start to work with data.

The last type of object you'll work with frequently is a dataframe. However, to help you get going on the homework, I'm going to do this primarily with your own homework dataset, which requires us to start loading packages.

Lesson 4: Load Packages & HW help

Base R (what comes installed on your computer) is an incredibly powerful programming language, but one of the best features of R are its packages, which are remotely stored functions written by anybody. You could even write a package if you wanted! This open source nature allows R to be extremely flexible. For now, we will load the `pacman` package management package, and then the `broom` and `tidyverse` packages which gives us access to the `tidy()` command (useful for summarizing regression objects) and a number of useful data manipulation tools.

Let's start by loading packages. Uncomment the install.packages function to get the pacman package to install. If you already have some of these packages, feel free to delete lines. The install.packages function can take a vector of package names, as characters, to install all of the above.

For the most part, we will use the super helpful package 'pacman' to load new packages into our workspace

```
install.packages(c("pacman"), dependencies=T, repos = "http://cran.us.r-project.org")
```

Congrats! The pacman package (package management... get it?) will allow us to install, update and load packages in such a way that you won't have to worry about conflicts. Load pacman with the `library()` function.

```
library(pacman)
#p_load is pacman's 'library' in that it lets you load packages. and features a number of improvements.
p_load(ISLR, tidyverse, broom)
```

We're also going to use a new function that lets us read CSV data into our workspace. read_csv, which comes from a package we just loaded called 'tidyverse'.

In order to use this function effectively, we should create a name for our dataset. To keep things easy for you, let's call this your_df, but really you could call this anything. In order to load a CSV, you need to find the **filepath** for your dataset. Go to Canvas to download the file, and then find it in your downloads folder.

If you're on **Mac**, you can find your filepath by right clicking on the file, and then holding control and selecting the 'copy as path' option.

If you're on **PC**, you can find your filepath by right clicking the file, going to 'properties' and copying the filepath from there. Unfortunately, the slashes on PC are also what are known as 'escape characters' in R. What this means for you is either you must replace every "\" with "\\" or with "/".

Turn this filepath into a string by placing quotes around it, and then pass it to the `read_csv()` command.

```
#remember, your filepath needs to be passed to read_csv() as a string, meaning "/Users/You/..." not /Us
your_df <-read_csv(filepath)
```

```
## Parsed with column specification:
## cols(
##   i_female = col_double(),
##   i_male = col_double(),
##   race = col_character(),
##   i_black = col_double(),
##   i_white = col_double(),
##   i_callback = col_double(),
##   n_jobs = col_double(),
##   n_expr = col_double(),
##   i_military = col_double(),
##   i_computer = col_double(),
##   first_name = col_character(),
##   sex = col_character()
## )
```

you'll see that R helpfully tells you what each class is for the variables in your dataset. However, the first thing to do is always to get a good summary- so let's use the summary command

```
summary(your_df)
```

```
##     i_female          i_male           race              i_black
##  Min.   :0.0000   Min.   :0.0000   Length:4499        Min.   :0.0000
##  1st Qu.:1.0000   1st Qu.:0.0000   Class :character   1st Qu.:0.0000
##  Median :1.0000   Median :0.0000   Mode  :character   Median :1.0000
##  Mean   :0.7708   Mean   :0.2292                      Mean   :0.5001
##  3rd Qu.:1.0000   3rd Qu.:0.0000                      3rd Qu.:1.0000
##  Max.   :1.0000   Max.   :1.0000                      Max.   :1.0000
##     i_white         i_callback          n_jobs          n_expr
##  Min.   :0.0000   Min.   :0.00000   Min.   :1.000   Min.   : 1.000
##  1st Qu.:0.0000   1st Qu.:0.00000   1st Qu.:3.000   1st Qu.: 5.000
##  Median :0.0000   Median :0.00000   Median :4.000   Median : 6.000
##  Mean   :0.4999   Mean   :0.08091   Mean   :3.658   Mean   : 7.848
##  3rd Qu.:1.0000   3rd Qu.:0.00000   3rd Qu.:4.000   3rd Qu.: 9.000
##  Max.   :1.0000   Max.   :1.00000   Max.   :7.000   Max.   :44.000
##    i_military        i_computer      first_name            sex
##  Min.   :0.00000   Min.   :0.0000   Length:4499        Length:4499
##  1st Qu.:0.00000   1st Qu.:1.0000   Class :character   Class :character
##  Median :0.00000   Median :1.0000   Mode  :character   Mode  :character
##  Mean   :0.09558   Mean   :0.8206
##  3rd Qu.:0.00000   3rd Qu.:1.0000
##  Max.   :1.00000   Max.   :1.0000
```

you'll see that for the numeric columns, you'll be looking at summary statistics, but for character (word) columns you only get the length and the Class/Mode.

we can use our snapshot tools here in the same way we did with vectors to take a peek at this df

```
head(your_df,14)
```

```
## # A tibble: 14 x 12
##    i_female i_male race  i_black i_white i_callback n_jobs n_expr
##       <dbl>  <dbl> <chr>   <dbl>   <dbl>      <dbl>  <dbl>  <dbl>
##  1        1      0 b           1       0          0      4      6
##  2        1      0 w           0       1          0      3     22
##  3        1      0 w           0       1          0      2      6
##  4        1      0 w           0       1          0      3      6
##  5        1      0 b           1       0          0      1      6
##  6        0      1 w           0       1          0      2      6
##  7        1      0 w           0       1          0      2      5
##  8        1      0 b           1       0          0      4     21
##  9        1      0 b           1       0          0      3      3
## 10        0      1 b           1       0          0      2      6
## 11        1      0 b           1       0          0      4      8
## 12        1      0 w           0       1          0      4      8
## 13        1      0 b           1       0          0      4      4
## 14        1      0 w           0       1          0      2      4
## # ... with 4 more variables: i_military <dbl>, i_computer <dbl>,
## #   first_name <chr>, sex <chr>
```

```
tail(your_df, 18)
```

```
## # A tibble: 18 x 12
##    i_female i_male race  i_black i_white i_callback n_jobs n_expr
##       <dbl>  <dbl> <chr>   <dbl>   <dbl>      <dbl>  <dbl>  <dbl>
##  1        1      0 w           0       1          0      3      4
##  2        1      0 b           1       0          0      2      4
##  3        1      0 w           0       1          0      2      5
##  4        1      0 b           1       0          0      5      9
##  5        1      0 b           1       0          0      4      5
##  6        1      0 b           1       0          0      4      2
##  7        1      0 w           0       1          0      6      4
##  8        1      0 w           0       1          0      2      3
##  9        0      1 w           0       1          0      3      4
## 10        1      0 b           1       0          0      3      1
## 11        1      0 b           1       0          0      4      7
## 12        1      0 b           1       0          0      5      5
## 13        1      0 w           0       1          0      4      5
## 14        1      0 b           1       0          0      2      4
## 15        1      0 w           0       1          0      4      4
## 16        1      0 b           1       0          0      3      3
## 17        1      0 w           0       1          0      4      6
## 18        1      0 b           1       0          0      6      7
## # ... with 4 more variables: i_military <dbl>, i_computer <dbl>,
## #   first_name <chr>, sex <chr>
```

We can also check the size of the dataframe out by using the `dim()` command

```
dim(your_df)
```

```
## [1] 4499   12
```

If we're interested in looking at a specific subset of the data, we can do this in one of two ways. The first is

by referencing the column name. Let's look at column 'race'

```
head(your_df$race,4)
```

```
## [1] "b" "w" "w" "w"
```

if you're interested in many columns, you can find them by using what's called "indexing". Indexing is passed to a set of square brackets, and is labeled row, column

```
#let's look at columns 2 & 3

head(your_df[,c(2,3)])
```

```
## # A tibble: 6 x 2
##   i_male race
##    <dbl> <chr>
## 1      0 b
## 2      0 w
## 3      0 w
## 4      0 w
## 5      0 b
## 6      1 w
```

```
#this creates a mini-dataframe with what you selected. You can also do this for a single column like be

head(your_df[,3])
```

```
## # A tibble: 6 x 1
##   race
##   <chr>
## 1 b
## 2 w
## 3 w
## 4 w
## 5 b
## 6 w
```

If you want to see the names of the columns you were looking at, you can use

```
names(your_df[,2:6])
```

```
## [1] "i_male"     "race"       "i_black"    "i_white"    "i_callback"
```

To print them out.

you can also select rows with indexing. Let's look at rows 5-10 of columns 2-6

```
(your_df[5:10,2:6])
```

```
## # A tibble: 6 x 5
##   i_male race  i_black i_white i_callback
##    <dbl> <chr>   <dbl>   <dbl>      <dbl>
## 1      0 b           1       0          0
## 2      1 w           0       1          0
## 3      0 w           0       1          0
## 4      0 b           1       0          0
## 5      0 b           1       0          0
## 6      1 b           1       0          0
```

or rows 2,7,10 and 12 of those same columns

```
your_df[c(2,7,10,12),2:6]
```

```
## # A tibble: 4 x 5
##   i_male race  i_black i_white i_callback
##    <dbl> <chr>   <dbl>   <dbl>      <dbl>
## 1      0 w           0       1          0
## 2      0 w           0       1          0
## 3      1 b           1       0          0
## 4      0 w           0       1          0
```

We can also use other features of the tidyverse to look at specific portions of our data. Let's use the filter command. The way this works is:

```
#let's filter on whether or not race is equal to white. In this dataset, that is represented by 'w'
filter(your_df, race == 'w')
```

```
## # A tibble: 2,249 x 12
##    i_female i_male race  i_black i_white i_callback n_jobs n_expr
##       <dbl>  <dbl> <chr>   <dbl>   <dbl>      <dbl>  <dbl>  <dbl>
##  1        1      0 w           0       1          0      3     22
##  2        1      0 w           0       1          0      2      6
##  3        1      0 w           0       1          0      3      6
##  4        0      1 w           0       1          0      2      6
##  5        1      0 w           0       1          0      2      5
##  6        1      0 w           0       1          0      4      8
##  7        1      0 w           0       1          0      2      4
##  8        0      1 w           0       1          0      3      4
##  9        0      1 w           0       1          0      3      5
## 10        1      0 w           0       1          0      2      6
## # ... with 2,239 more rows, and 4 more variables: i_military <dbl>,
## #   i_computer <dbl>, first_name <chr>, sex <chr>
```

This lets us find things like 'what percentage of white respondents were women"

```
#the mean of a binary variable is the percentage of times that binary variable is equal to '1'
(mean(filter(your_df, race == 'w')$i_female))
```

```
## [1] 0.7630058
```

looks like it's about 76% of the white respondents are women.

we can also do this type of operation in steps using pipes (which come from the tidyverse package), using this funny carrot symbol **%>%**

let's look at the first 8 rows and find out the first names of our respondents using a tidyverse command called `select`

```
head(your_df) %>% select(first_name)
```

```
## # A tibble: 6 x 1
##   first_name
##   <chr>
## 1 Latonya
## 2 Carrie
## 3 Allison
## 4 Kristen
## 5 Lakisha
## 6 Jay
```

Now let's look and see if there is any gender disparity between our two groups by doing a **z-test**. What do we need to generate a z-score again? We need the n for each of our groups, a mean for each group, and a mean for females in the sample overall.

- number of black people in sample

we can use the nrow command to count the number of rows in the dataset after you filter by race = 'b'

```
n_b = filter(your_df, race == "b") %>% nrow()
```

- number of white people

```
n_w <- filter(your_df, race == "w") %>% nrow()
```

- the means

```
#percentage of females who are black
mean_b <- filter(your_df, race == "b")$i_female %>% mean()
#percentage of females who are white
mean_w <- filter(your_df, race == "w")$i_female %>% mean()
#percentage of females in the sample overall
mean_all <- your_df$i_female %>% mean()
```

and we need to put them all together. We're going to be using a few mathematical operators here, primarily, `sqrt()` (square root) and `pnorm()` (distribution function of normal).

pnorm is a weird one, and it takes a quantile (so a z-score), mean, standard deviation, and a few other things. We're interested in a two sided z-test so we'll use lower.tail equal to false.

```
#use the formula for a z-score to calculate the z-stat
z_stat <- (mean_b - mean_w) / sqrt(mean_all*(1-mean_all)*(1/n_b + 1/n_w))

#plug it into the probability of seeing a value as large as our z-stat
2*pnorm(abs(z_stat), lower.tail = FALSE)
```

```
## [1] 0.2114242
```

this doesn't look too bad. There's a difference here but not statistically significant. Let's look at this problem another way.

Lesson 5: Regression in R

we can also run regressions in R using the 'lm' command. Let's run a regression testing if female is significant by race.

```
#equations are a NEW object type and are denoted usually by the tilde (~) object
your_reg <-lm(i_female ~ i_black, data = your_df)
```

yYou can see here that lm is a function that takes an "equation"" object of the following form:

`y ~ x1 + x2 + x3 +` etc.

It also needs to be able to reference the dataframe in question, so we need to tell it the name our dataframe has. In this case, `your_df`.

we can pass this to 'tidy', which comes from the broom package, to display the results nicely, we also can use `summary()`

```
your_reg %>% tidy()
```

```
## # A tibble: 2 x 5
##   term        estimate std.error statistic p.value
##   <chr>          <dbl>     <dbl>     <dbl>   <dbl>
## 1 (Intercept)   0.763    0.00886     86.1    0
## 2 i_black       0.0157   0.0125       1.25   0.212
```

Now for summary:

```
summary(your_reg)
```

```
##
## Call:
## lm(formula = i_female ~ i_black, data = your_df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.7787  0.2213  0.2213  0.2370  0.2370
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.763006   0.008863   86.09   <2e-16 ***
## i_black     0.015661   0.012533    1.25    0.212
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4203 on 4497 degrees of freedom
## Multiple R-squared:  0.0003471,  Adjusted R-squared:  0.0001248
## F-statistic: 1.561 on 1 and 4497 DF,  p-value: 0.2115
```

let's look at a more complicated equation involving interaction terms. We can call out interaction terms in R by putting a colon between the two interacted variables

```
your_reg_comp <- lm(i_female ~ i_black + i_callback + i_black:i_military, data = your_df)

summary(your_reg_comp)
```

```
##
## Call:
## lm(formula = i_female ~ i_black + i_callback + i_black:i_military,
##     data = your_df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.8120  0.1880  0.2057  0.2387  0.3723
##
## Coefficients:
##                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)         0.761282   0.009111  83.557  < 2e-16 ***
## i_black             0.032974   0.012852   2.566   0.0103 *
## i_callback          0.017705   0.022950   0.771   0.4405
## i_black:i_military -0.166564   0.029387  -5.668 1.54e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4189 on 4495 degrees of freedom
## Multiple R-squared:  0.007624,   Adjusted R-squared:  0.006962
```

```
## F-statistic: 11.51 on 3 and 4495 DF,  p-value: 1.626e-07
```

PHEW. That's it. I will see you guys next week! Thank you!