Lab 3: Data Frames and Apply

Name: Rufus Petrie

This week's agenda: getting familiar with data frames; practicing how to use the apply family of functions.

States data set

Below we construct a data frame, of 50 states x 10 variables. The first 8 variables are numeric and the last 2 are factors. The numeric variables here come from the built-in `state.x77` matrix, which records various demographic factors on 50 US states, measured in the 1970s. You can learn more about this state data set by typing `?state.x77` into your R console.

```
state.df = data.frame(state.x77, Region=state.region, Division=state.division)
```

Q1. Basic data frame manipulations

- **1a.** Add a column to `state.df`, containing the state abbreviations that are stored in the built-in vector `state.abb`. Name this column `Abbr`. You can do this in (at least) two ways: by using a call to `data.frame()`, or by directly defining `state.df$Abbr`. Display the first 3 rows and all 11 columns of the new `state.df`.

```
state.df[,"Abbr"] <- state.abb
head(state.df, 3)
```

```
##         Population Income Illiteracy Life.Exp Murder HS.Grad Frost   Area
## Alabama       3615   3624        2.1    69.05   15.1    41.3    20  50708
## Alaska         365   6315        1.5    69.31   11.3    66.7   152 566432
## Arizona       2212   4530        1.8    70.55    7.8    58.1    15 113417
##         Region           Division Abbr
## Alabama  South East South Central   AL
## Alaska    West            Pacific   AK
## Arizona   West           Mountain   AZ
```

- **1b.** Remove the `Region` column from `state.df`. You can do this in (at least) two ways: by using negative indexing, or by directly setting `state.df$Region` to be `NULL`. Display the first 3 rows and all 10 columns of `state.df`.

```
state.df$Region <- NULL
head(state.df, 3)
```

```
##         Population Income Illiteracy Life.Exp Murder HS.Grad Frost   Area
## Alabama       3615   3624        2.1    69.05   15.1    41.3    20  50708
## Alaska         365   6315        1.5    69.31   11.3    66.7   152 566432
## Arizona       2212   4530        1.8    70.55    7.8    58.1    15 113417
##                   Division Abbr
## Alabama East South Central   AL
## Alaska             Pacific   AK
## Arizona           Mountain   AZ
```

- **1c.** Add two columns to `state.df`, containing the x and y coordinates (longitude and latitude, respectively) of the center of the states, that are stored in the (existing) list `state.center`. Hint: take a look at this list in the console, to see what its elements are named. Name these two columns `Center.x` and `Center.y`. Display the first 3 rows and all 12 columns of `state.df`.

```
state.df$Center.x <- state.center$x
state.df$Center.y <- state.center$y
state.df[1:3,]
```

```
##         Population Income Illiteracy Life.Exp Murder HS.Grad Frost   Area
## Alabama       3615   3624        2.1    69.05   15.1    41.3    20  50708
## Alaska         365   6315        1.5    69.31   11.3    66.7   152 566432
## Arizona       2212   4530        1.8    70.55    7.8    58.1    15 113417
##                   Division Abbr  Center.x Center.y
## Alabama East South Central   AL  -86.7509  32.5901
## Alaska             Pacific   AK -127.2500  49.2500
## Arizona           Mountain   AZ -111.6250  34.2192
```

- **1d.** Make a new data frame which contains only those states whose longitude is less than -100. Do this in two different ways: using manual indexing, and `subset()`. Check that they are equal to each other, using an appropriate function call.

```
df1 <- state.df[state.df$Center.x < -100,]
df2 <- subset(state.df, Center.x < -100)
all.equal(df1, df2)
```

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

- **1e.** Make a new data frame which contains only the states whose longitude is less than -100, and whose murder rate is above 9%. Print this new data frame to the console. Among the states in this new data frame, which has the highest average life expectancy?

```
df1 <- state.df[state.df$Center.x < -100 & state.df$Murder > 9,]
df1
```

```
##            Population Income Illiteracy Life.Exp Murder HS.Grad Frost   Area
## Alaska            365   6315        1.5    69.31   11.3    66.7   152 566432
## California      21198   5114        1.1    71.71   10.3    62.6    20 156361
## Nevada            590   5149        0.5    69.03   11.5    65.2   188 109889
## New Mexico       1144   3601        2.2    70.32    9.7    55.2   120 121412
##            Division Abbr Center.x Center.y
## Alaska      Pacific   AK -127.250  49.2500
## California  Pacific   CA -119.773  36.5341
## Nevada     Mountain   NV -116.851  39.1063
## New Mexico Mountain   NM -105.942  34.4764
```

Among the states in the new data frame, California has the highest life expectancy.

Prostate cancer data set

Below we read in the prostate cancer data set that we looked in the last lab. You can remind yourself about what's been measured by looking back at the lab.

```
pros.dat =
  read.table("http://www.stat.cmu.edu/~ryantibs/statcomp/data/pros.dat")
```

Q2. Practice with the apply family

- **2a.** Using `sapply()`, calculate the mean of each variable. Also, calculate the standard deviation of each variable. Each should require just one line of code. Display your results.

```
# YOUR CODE GOES HERE
```

- **2b.** Let's plot each variable against SVI. Using `lapply()`, plot each column, excluding SVI, on the y-axis with SVI on the x-axis. This should require just one line of code. **Challenge**: label the y-axes in your plots appropriately. Your solution should still consist of just one line of code and use an apply function. Hint: for this part, consider using `mapply()`.

```
# YOUR CODE GOES HERE
```

- **2c.** Now, use `lapply()` to perform t-tests for each variable in the data set, between SVI and non-SVI groups. To be precise, you will perform a t-test for each variable excluding the SVI variable itself. For convenience, we've defined a function `t.test.by.ind()` below, which takes a numeric variable `x`, and then an indicator variable `ind` (of 0s and 1s) that defines the groups. Run this function on the columns of `pros.dat`, excluding the SVI column itself, and save the result as `tests`. What kind of data structure is `tests`? Print it to the console.

```
t.test.by.ind = function(x, ind) {
  stopifnot(all(ind %in% c(0, 1)))
  return(t.test(x[ind == 0], x[ind == 1]))
}
```

```
# YOUR CODE GOES HERE
```

- **2d.** Using `lapply()` again, extract the p-values from the `tests` object you created in the last question, with just a single line of code. Hint: first, take a look at the first element of `tests`, what kind of object is it, and how is the p-value stored? Second, run the command `` `[[`(pros.dat, "lcavol") `` in your console—what does this do? Now use what you've learned to extract p-values from the `tests` object.

```
# YOUR CODE GOES HERE
```

Rio Olympics data set

Now we're going to examine data from the 2016 Summer Olympics in Rio de Janeiro, taken from https://github.com/flother/rio2016 (complete data on the 2020 Summer Olympics in Tokyo doesn't appear to be available yet). Below we read in the data and store it as `rio`.

```
rio = read.csv("http://www.stat.cmu.edu/~ryantibs/statcomp/data/rio.csv")
```

Q3. More practice with data frames and apply

- **3a.** What kind of object is rio? What are its dimensions and columns names of `rio`? What does each row represent? Is there any missing data?

```
# YOUR CODE GOES HERE
```

- **3b.** Use `rio` to answer the following questions. How many athletes competed in the 2016 Summer Olympics? How many countries were represented? What were these countries, and how many athletes competed for each one? Which country brought the most athletes, and how many was this? Hint: for a factor variable `f`, you can use `table(f)` see how many elements in `f` are in each level of the factor.

```
# YOUR CODE GOES HERE
```

- **3c.** How many medals of each type—gold, silver, bronze—were awarded at this Olympics? Are they equal? Is this result surprising, and can you explain what you are seeing?

```
# YOUR CODE GOES HERE
```

- **3d.** Create a column called `total` which adds the number of gold, silver, and bronze medals for each athlete, and add this column to `rio`. Which athlete had the most number of medals and how many was this? Gold medals? Silver medals? In the case of ties, here, display all the relevant athletes.

```
# YOUR CODE GOES HERE
```

- **3e.** Using `tapply()`, calculate the total medal count for each country. Save the result as `total.by.nat`, and print it to the console. Which country had the most number of medals, and how many was this? How many countries had zero medals?

```
# YOUR CODE GOES HERE
```

- **3f.** Among the countries that had zero medals, which had the most athletes, and how many athletes was this? (Ouch!)

```
# YOUR CODE GOES HERE
```

Q4. Young and old folks

- **4a.** The variable `date_of_birth` contains strings of the date of birth of each athlete. Use the `substr()` function to extract the year of birth for each athlete, and then create a new numeric variable called `age`, equal to 2016 - (the year of birth). (Here we're ignoring days and months for simplicity.) Hint: to extract the first 4 characters of a string `str`, you can use `substr(str, 1, 4)`. As always, you can also look at the help file for `substr()` for more details.

  Add the `age` variable to the `rio` data frame. variable Who is the oldest athlete, and how old is he/she? Youngest athlete, and how old is he/she? In the case of ties, here, display all the relevant athletes.

```
# YOUR CODE GOES HERE
```

- **4b.** Answer the same questions as in the last part, but now only among athletes who won a medal.

```
# YOUR CODE GOES HERE
```

- **4c.** Using a single call to `tapply()`, answer: how old are the youngest and oldest athletes, for each sport?

```
# YOUR CODE GOES HERE
```

- **4d.** You should see that your output from `tapply()` in the last part is a list, which is not particularly convenient. Convert this list into a matrix that has one row for each sport, and two columns that display the ages of the youngest and oldest athletes in that sport. The first 3 rows should look like this:

```
                  Youngest Oldest
athletics               14     41
archery                 17     44
athletics               16     47
```

You'll notice that we set the row names according to the sports, and we also set appropriate column names. Hint: `unlist()` will unravel all the values in a list; and `matrix()`, as you've seen before, can be used to create a matrix from a vector of values. After you've converted the results to a matrix, print it to the console (and make sure its first 3 rows match those displayed above).

```
# YOUR CODE GOES HERE
```

- **Challenge.** Determine the *names* of the youngest and oldest athletes in each sport, along with their ages (so your result should have 4 columns), without using any explicit iteration. In the case of ties, just return one relevant athlete name. (For this part, you can use another package, such as `plyr` or `dplyr` if you want to.)

```
# YOUR CODE GOES HERE
```

Q5. Sport by sport

- **5a.** Create a new data frame called `sports`, which we'll populate with information about each sporting event at the Summer Olympics. Initially, define `sports` to contain a single variable called `sport` which contains the names of the sporting events in alphabetical order. Then, add a column called `n_participants` which contains the number of participants in each sport. Use one of the apply functions to determine the number of gold medals given out for each sport, and add this as a column called `n_gold`. Using your newly created `sports` data frame, calculate the ratio of the number of gold medals to participants for each sport. Which sport has the highest ratio? Which has the lowest?

```
# YOUR CODE GOES HERE
```

- **5b.** Use one of the apply functions to compute the average weight of the participants in each sport, and add this as a column to `sports` called `ave_weight`. Important: there are missing weights in the data set coded as `NA`, but your column `ave_weight` should ignore these, i.e., it should be itself free of `NA` values. You will have to pass an additional argument to your apply call in order to achieve this. Hint: look at the help file for the `mean()` function; what argument can you set to ignore `NA` values? Once computed, display the average weights along with corresponding sport names, in decreasing order of average weight.

```
# YOUR CODE GOES HERE
```

- **5c.** As in the last part, compute the average weight of athletes in each sport, but now separately for men and women. You should therefore add two new columns, called `ave_weight_men` and `ave_weight_women`, to `sports`. Once computed, display the average weights along with corresponding sports, for men and women, each list sorted in decreasing order of average weight. Are the orderings roughly similar?

```
# YOUR CODE GOES HERE
```

- **Challenge.** Use one of the apply functions to compute the proportion of women among participating athletes in each sport. Use these proportions to recompute the average weight (over all athletes in each sport) from the `ave_weight_men` and `average_weight_women` columns, and define a new column `ave_weight2` accordingly. Does `ave_weight2` differ from `ave_weight`? It should. Explain why. Then show how to recompute the average weight from `ave_weight_men` and `average_weight_women` in a way that exactly recreates `average_weight`.

```
# YOUR CODE GOES HERE
```