Homework Assignment #5

Matt Kline

March 6, 2017

Part 1: Factors and Tables

1.) The built-in data set state.region shows the regions (North, South, Central, West, or Northeast) for each of the 50 United States. Type

```
state.region

##  [1] South         West          West          South         West
##  [6] West          Northeast     South         South         South
## [11] West          West          North Central North Central North Central
## [16] North Central South         South         Northeast     South
## [21] Northeast     North Central North Central South         North Central
## [26] West          North Central West          Northeast     Northeast
## [31] West          Northeast     South         North Central North Central
## [36] South         West          Northeast     Northeast     South
## [41] North Central South         South         West          Northeast
## [46] South         West          South         North Central West
## Levels: Northeast South North Central West
```

to view the dataset and type

```
#?state.region
```

to view the help file. We want to summarize the 5 regions according to the sizes of the states within them. The following command splits the 50 states into 5 separate size-categories:

```
area <- cut(rank(state.x77[ , "Area"]), 5, c("Tiny", "Small", "Medium", "Large", "Huge"))
```

a) What class of objects do state.region and area belong to? Use class().

```
class(state.region)

## [1] "factor"

class(area)

## [1] "factor"
```

b) Use table() to create the table we're after:

```
table(state.region, area)
```

```
##                area
## state.region    Tiny Small Medium Large Huge
##   Northeast        6     2      1     0    0
##   South            3     6      5     1    1
##   North Central    0     2      4     6    0
##   West             1     0      0     3    9
```

c) Which state in the West region is Tiny? Which one in the Northeast is Medium? Hint: The built-in data set state.name lists the names of the states.

```
state.name[(state.region == "West" && area == "Tiny")]

## character(0)

state.name[(state.region == "Northeast" && area == "Medium")]

## character(0)
```

Part 2: R Programming Structures

2.) Print the message "Hello World" to the R console 7 times using three different methods:

a) Using a for() loop

```
for(i in 1:7){
  print("Hello World")
}

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

b) Using a while() loop

```
i <- 0
while(i < 7){
  print("Hello World")
  i <- i + 1
}

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

c) Using a repeat command with a break statement.

```
i <- 0
repeat {
 print("Hello World")
 i <- i + 1
 if (i > 6) break
}

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

3.) In each of the following, determine the final value of answer without the use of R. Explain your answer and then check it R.

a) The answer should be 15.

```
answer <- 0
for (j in 1:5) {
answer <- answer + j
}
answer

## [1] 15
```

b) The answer should be a list of 1 to 5

```
answer <- NULL
for (j in 1:5) {
answer <- c(answer, j)
}
answer

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

c) The answer should be a list of 0 to 5

```
answer <- 0
for (j in 1:5) {
answer <- c(answer, j)
}
answer

## [1] 0 1 2 3 4 5
```

d) The answer should be 120

```
answer <- 1
for (j in 1:5) {
answer <- answer * j
}
answer
```

```
## [1] 120
```

4.) The file wayans.txt has a command to create a list containing two generations of the celebrity Wayans family. Loop over the list of the first generation of Wayanses, keeping trackof how many children each one has in a vector.

```
source("/Users/mkline6/Desktop/wayans.txt")

## Warning in file(filename, "r", encoding = encoding):  cannot open file '/Users/mkline6/Deskto
No such file or directory

## Error in file(filename, "r", encoding = encoding):  cannot open the connection

children <- NULL
for(i in wayans){
  children <- c(children, length(i))
}

## Error in eval(expr, envir, enclos):  object 'wayans' not found

children

## NULL
```

5.) Use nested for() loops to create a matrix containing the multiplication table for the integers 1 through 12. The matrix should have 12 rows and 12 columns. The first row should contain the values 1 ? 1, 1 ? 2, . . . , 1 ? 12, the second row should contain 2 ? 1, 2 ? 2, . . . , 2 ? 12, and so on, as shown below.

```
X <- matrix(0, nrow = 12, ncol = 12)
for(i in 1:12){
  for(j in 1:12){
    X[i,j] <- (i*j)
  }
}
X

##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
##  [1,]    1    2    3    4    5    6    7    8    9    10    11    12
##  [2,]    2    4    6    8   10   12   14   16   18    20    22    24
##  [3,]    3    6    9   12   15   18   21   24   27    30    33    36
##  [4,]    4    8   12   16   20   24   28   32   36    40    44    48
##  [5,]    5   10   15   20   25   30   35   40   45    50    55    60
##  [6,]    6   12   18   24   30   36   42   48   54    60    66    72
##  [7,]    7   14   21   28   35   42   49   56   63    70    77    84
##  [8,]    8   16   24   32   40   48   56   64   72    80    88    96
##  [9,]    9   18   27   36   45   54   63   72   81    90    99   108
## [10,]   10   20   30   40   50   60   70   80   90   100   110   120
## [11,]   11   22   33   44   55   66   77   88   99   110   121   132
## [12,]   12   24   36   48   60   72   84   96  108   120   132   144
```

6.) For each of the following compound logical expressions, decide whether it will evaluate to TRUE or FALSE. Explain your answer and then check it in R:

a) The answer should be FALSE

```
(4 > 5 & 8 < 9) | 2 > 3

## [1] FALSE
```

b) The answer should be FALSE

```
4 > 5 & (8 < 9 | 2 > 3)

## [1] FALSE
```

c) The answer should be TRUE

```
(4 > 5 & 2 > 3) | 8 < 9

## [1] TRUE
```

d) The answer should be FALSE

```
(4 > 5 & (2 > 3 | 8 < 9)) | 7 < 6

## [1] FALSE
```

7.) An exception to the rule that NAs in operators necessarily produce NAs is given by the AND and OR operators.

a) What are the values of:

The first will give a TRUE because its nothing or true causing true.

The second will give FALSE because anything and false is false.

```
NA | TRUE

## [1] TRUE

FALSE & NA

## [1] FALSE
```

b) What are the values of:

The first will give NA because false is being compared with na which is nothing.

The second will give NA because NA is nothing and nothing cannot be true for the and case to be true.

```
NA | FALSE

## [1] NA

TRUE & NA

## [1] NA
```

8.) Write a function that checks the mode of its argument. If the mode is "numeric", "logical", or "character", return the sorted argument; otherwise, stop with an appropriate error message.

```r
checkMode <- function(argument){
  if (is.character(argument)){
    sort(argument)
    return
  } else if (is.numeric(argument)){
    sort(argument)
    return
  } else if (is.logicat(argument)){
    sort(argument)
    return
  } else stop("Not a recognized mode")
}
```

9.) Functions can be defined to take a variable number of arguments. The special argument name "..." in a function definition will match any number of arguments in the call. For example, a function that returns the mean of all the values in an arbitrary number of different vectors is:

```r
meanOfAll <- function(...) {
  mean(c(...))
}
```

so that

```r
#meanOfAll(usSales, europeSales, otherSales)
```

would combine the three objects and take the mean of all the data. The effect of c(...) is as if c() had been called with the same three arguments passed to meanOfAll(). The use of "...", passing it as an argument in a call to another function (e.g. c(...)), is its only legal use in the body of a function. This restriction doesn't get in the way. To loop over arguments, for example, we can create a list of the evaluated arguments, list(...), and then use a for loop to get at each in turn. The names, if any, attached to the actual arguments will also be passed down to the call to list(...). In the previous example, to compute the mean of the individual means, instead of the mean of the combined data, we could use:

```r
meanOfMeans <- function(...) {
  means <- NULL
  for (x in list(...)) {
    means <- c(means, mean(x))
  }
  return(mean(means))
}
```

Is there a difference between the values computed by meanOfAll() and meanOfMeans()? Try it on some simple data. Under what conditions on the vectors passed as arguments will the two functions return the same value?

```r
x <- c(4,5,1,6,89,35,21)
y <- c(86,35,1,1,5,76,7)
z <- c(2,16,4,1346,7,24)

meanOfAll(x,y)
```

```
## [1] 26.57143

meanOfMeans(x,y)

## [1] 26.57143

meanOfAll(x,y,z)

## [1] 88.55

meanOfMeans(x,y,z)

## [1] 95.43651
```

The vales are different when there are more than two vectors being passed. They are different because of the way they are calculated. The first calculates the total mean by computing all the individual means then the total mean with no rounding. The second does each mean seperately, rounds them then takes the mean of that rounded set of means.