Homework 1 - Drew Kearny - Due September 8

Name: Drew Kearny  
netID: 231005935  
Collaborated with:

Your homework **must be submitted in Word or PDF format, created by calling “Knit Word” or “Knit PDF” from RStudio on your R Markdown document. Submission in other formats may receive a grade of 0**. Your responses must be supported by both textual explanations and the code you generate to produce your result. Note that all R code used to produce your results must be shown in your knitted file.

## Syntax and class-typing.

1. For each line of the following code, either explain why they should be erroneous, or explain what tasks the non-erroneous ones perform.

vector1 <- c(5, 12, TRUE, 32)

This line above is not erroneous and it creates a vector with a numeric element 5, a numeric element 12, a logical element TRUE, and a numeric element 32, and all the numeric elements are of the ‘double’ type.

max(vector1)

## [1] 32

This line above is not erroneous and it calculates and returns the max value of vector1 which is 32.

sort(vector1)

## [1] 1 5 12 32

This line above is not erroneous because the sort function is meant to sort elements in ascending order and even though vector1 contains a boolean value “TRUE”, r will coerce this to a numeric value “1” when the function is applied.

sum(vector1)

## [1] 50

This line above is not erroneous because it sums each element of the vector which for vector1 is 50.

1. For each block of the following code, either explain why they should be erroneous, or explain what tasks the non-erroneous ones perform.

vector2 <- c(5,"7",12)  
 vector2[2] + vector2[3]

## Error in vector2[2] + vector2[3]: non-numeric argument to binary operator

The first line is not erroneous and it creates a vector with a numeric element 5, a character element “7”, and a numeric element 12. The second line is erroneous because attempts to perform addition between the character “7” and the numeric element 12, which causes an error.

dataframe3 <- data.frame(z1="5",z2=7,z3=12)  
 dataframe3[1,2] + dataframe3[1,3]

## [1] 19

The first line is not erroneous because it creates a data frame named dataframe3 with three columns. The first column z1 contains a character element “5”, the second column z2 contains a numeric element 7, and the thrid column z3 contains a numeric element 12. The second line is not erroneous either because it performs addition between the element in the first row and second column which is “z2”, with the element in the first row and third column which is “z3” of the data frame. The addition should be 7+12 = 19.

list4 <- list(z1="6", z2=42, z3="49", z4=126)  
 list4[[2]]+list4[[4]]

## [1] 168

list4[2]+list4[4]

## Error in list4[2] + list4[4]: non-numeric argument to binary operator

The first line is not erroneous as it creates a list named list4 with four named elements: “z1” contains the character “6”, “z2” contains the numeric element 42, “z3” contains the character element “49”, and “z4” contains the numeric element 126. The second line is not erroneous because it does addition between the numeric element at position 2 in the list and the numeric element at position 4 of the list which computes as 42 + 126 = 168. The third line is erroneous because it attempts to access elements using list indexing which returns a sub-list, and then adding the sub-lists is not a valid operation.

## Working with functions and operators.

1. The colon operator will create a sequence of integers in order. It is a special case of the function seq(). Using the help command ?seq to learn about the function, produce an expression that will give you the sequence of numbers from 1 to 10000 in increments of 369. Produce another that will give you a sequence between 1 and 10000 that is exactly 50 numbers in length (i.e., the first number is 1 and the last number is 10000; and the differences between a pair of consecutive numbers are the same).

sequence1 <- seq(from = 1, to = 10000, by = 369)  
sequence2 <- seq(from = 1, to = 10000, length.out = 50)

1. The function rep() repeats a vector some number of times. Explain the difference between `rep(1:3, times=3) and rep(1:3, each=3).

The difference is that the times=3 argument tells how many times the entire vector should be repeated while the each=3 argument tells how many times each element of the vector should be repeated before moving to the next element. So the first one would return the vector c(1, 2, 3, 1, 2, 3, 1, 2, 3) and the second one would return the vector c(1, 1, 1, 2, 2, 2, 3, 3, 3).

## The Binomial distribution.

The binomial distribution is defined by the number of successes in independent trials, each have probability of success. Think of flipping a coin times, where the coin is weighted to have probability of landing on heads.

The R function rbinom() generates random variables with a binomial distribution. E.g.,

rbinom(n=20, size=10, prob=0.5)

produces 20 observations from .

The following generates 300 binomials composed of 15 trials each with varying probability of success: 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, and 0.8, storing the results in vectors called bin.draws.0.2, bin.draws.0.3, bin.draws.0.4., bin.draws.0.5., bin.draws.0.6, bin.draws.0.7 and bin.draws.0.8. The means ares stored in the vector bin.draws.means.

set.seed(01202023) #for randomization; do not change  
  
bins.draws.0.2 <- rbinom(300, size = 15, prob = 0.2)  
bins.draws.0.3 <- rbinom(300, size = 15, prob = 0.3)  
bins.draws.0.4 <- rbinom(300, size = 15, prob = 0.4)  
bins.draws.0.5 <- rbinom(300, size = 15, prob = 0.5)  
bins.draws.0.6 <- rbinom(300, size = 15, prob = 0.6)  
bins.draws.0.7 <- rbinom(300, size = 15, prob = 0.7)  
bins.draws.0.8 <- rbinom(300, size = 15, prob = 0.8)  
  
bin.draws.means <- c(  
 mean(bins.draws.0.2),  
 mean(bins.draws.0.3),  
 mean(bins.draws.0.4),  
 mean(bins.draws.0.5),  
 mean(bins.draws.0.6),  
 mean(bins.draws.0.7),  
 mean(bins.draws.0.8)  
)

1. Create a matrix of dimension 300 x 7, called bin.matrix, whose columns contain the 7 vectors we’ve created, in order of the success probabilities of their underlying binomial distributions (0.2 through 0.8). Hint: use cbind().

bin.matrix <- cbind(  
 bins.draws.0.2,  
 bins.draws.0.3,  
 bins.draws.0.4,  
 bins.draws.0.5,  
 bins.draws.0.6,  
 bins.draws.0.7,  
 bins.draws.0.8  
)

b.Print the first five rows of bin.matrix. Print the element in the 66th row and 5th column. Compute the largest element in first column. Compute the largest element in all but the first column.

print(bin.matrix[1:5, ])

## bins.draws.0.2 bins.draws.0.3 bins.draws.0.4 bins.draws.0.5 bins.draws.0.6  
## [1,] 3 6 8 5 9  
## [2,] 5 4 8 7 10  
## [3,] 1 4 4 9 6  
## [4,] 2 6 6 6 9  
## [5,] 2 7 4 10 10  
## bins.draws.0.7 bins.draws.0.8  
## [1,] 13 12  
## [2,] 9 14  
## [3,] 10 11  
## [4,] 13 10  
## [5,] 13 11

print(bin.matrix[66,5])

## bins.draws.0.6   
## 6

print(max(bin.matrix[, 1]))

## [1] 8

print(apply(bin.matrix[, -1], 2, max))

## bins.draws.0.3 bins.draws.0.4 bins.draws.0.5 bins.draws.0.6 bins.draws.0.7   
## 10 11 14 13 15   
## bins.draws.0.8   
## 15

1. Calculate the column means of bin.matrix by using just a single function call.

column\_means <- colMeans(bin.matrix)  
print(column\_means)

## bins.draws.0.2 bins.draws.0.3 bins.draws.0.4 bins.draws.0.5 bins.draws.0.6   
## 2.903333 4.413333 6.106667 7.593333 8.946667   
## bins.draws.0.7 bins.draws.0.8   
## 10.420000 12.140000

1. Compare the means you computed in the last question to those in bin.draws.means, in two ways. First, using ==, and second, using identical(). What do the two ways report? Are the results compatible? Explain.

Using ‘==’ is a logical comparison that reports whether the corresponding mean are equal (or extemely close to equal) which they all are so it returns TRUE for all of them. Using ‘identical()’ reports if the two vectors are identical meaning they have the same values in the same order and the entire structure is identical which returns FALSE in this case. Therefore the results are not compatible which is likely indicates that the means are super close and essentially equal but not exactly identical due to the floating-point arithmetic.

print(column\_means == bin.draws.means)

## bins.draws.0.2 bins.draws.0.3 bins.draws.0.4 bins.draws.0.5 bins.draws.0.6   
## TRUE TRUE TRUE TRUE TRUE   
## bins.draws.0.7 bins.draws.0.8   
## TRUE TRUE

print(identical(column\_means, bin.draws.means))

## [1] FALSE

1. Take the transpose of bin.matrix and then take row means. Are these the same as what you just computed? Should they be?

They are the same and they should be the same because the values in the row are just in a different order now.

transposed\_bin.matrix <- t(bin.matrix)  
row\_means\_trans <- rowMeans(transposed\_bin.matrix)  
print(row\_means\_trans)

## bins.draws.0.2 bins.draws.0.3 bins.draws.0.4 bins.draws.0.5 bins.draws.0.6   
## 2.903333 4.413333 6.106667 7.593333 8.946667   
## bins.draws.0.7 bins.draws.0.8   
## 10.420000 12.140000

1. Lastly, let’s look at memory useage. The command object.size(x) returns the number of bytes used to store the object x in your current R session. Convert bin.matrix into a list using as.list() and save the result as bin.list. Find the number of bytes used to store bin.matrix and bin.list. How many megabytes (MB) is this, for each object? Which object requires more memory, and why do you think this is the case? Remind yourself: why are lists special compared to vectors, and is this property important for the current purpose (storing the binomial draws)? Hint: look at the help page for object.size to see how to change the units to MB.

It takes 9352 bytes to store bin.matrix and 134448 bytes to store bin.list. This is 0.009352 megabytes and 0.134448 megabytes respectively. The bin.list object takes up more memory I believe because of the fact that a list has the ability to store different data types and also more information for each element than a matrix. This property of being able to store different data types is not important and is unnecessary because we are only using numeric (double) data types.

bin.list <- as.list(bin.matrix)  
memory\_bin.matrix <- object.size(bin.matrix)  
memory\_bin.list <- object.size(bin.list)  
print(memory\_bin.matrix)

## 9352 bytes

print(memory\_bin.list)

## 134448 bytes

## Going big with lists

R’s capacity for data storage and computation is very large compared to what was available 10 years ago. The following code generate 5 million numbers from distribution and store them in a vector called big.bin.draws.

big.bin.draws <- rbinom(n = 5e6, size = 1e6, prob = 0.5)

1. Create a new vector, called big.bin.draws.standardized, which is given by taking big.bin.draws, subtracting off its mean, and then dividing by its standard deviation. Calculate the mean and standard deviation of big.bin.draws.standardized. (These should be 0 and 1, respectively, or very close to it; if not, you’ve made a mistake somewhere).

big.bin.draws.standardized <- (big.bin.draws - mean(big.bin.draws)) / sd(big.bin.draws)  
  
mean\_bin.standardized <- mean(big.bin.draws.standardized)  
sd\_bin.standardized <- sd(big.bin.draws.standardized)  
  
print(mean\_bin.standardized)

## [1] 5.229312e-14

print(sd\_bin.standardized)

## [1] 1

1. Convert big.bin.draws into a list using as.list() and save the result as big.bin.draws.list. Check that you indeed have a list by calling class() on the result. Check also that your list has the right length, and that its 1159th element is equal to that of big.bin.draws.

big.bin.draws.list <- as.list(big.bin.draws)  
  
print(class(big.bin.draws.list))

## [1] "list"

print(length(big.bin.draws.list))

## [1] 5000000

print(big.bin.draws.list[[1159]] == big.bin.draws[1159])

## [1] TRUE

1. Run the code below, to standardize the binomial draws in the list big.bin.draws.list. Note that lapply() applies the function supplied in the second argument to every element of the list supplied in the first argument, and then returns a list of the function outputs. (We’ll learn much more about the apply() family of functions later in the course.) Did this lapply() command take longer to evaluate than the code you wrote in part a? (It should have; otherwise your previous code could have been improved, so go back and improve it.) Why do you think this is the case?

Yes it took quite a bit longer to evaluate than in part a. I think this is because lapply() applies standardize() to each element in the list by sequence which takes more time.

big.bin.draws.mean = mean(big.bin.draws)  
big.bin.draws.sd = sd(big.bin.draws)  
standardize = function(x) {  
 return((x - big.bin.draws.mean) / big.bin.draws.sd)  
}  
big.bin.draws.list.standardized.slow = lapply(big.bin.draws.list, standardize)

1. Find the number of bytes used to store big.bin.draws and big.bin.draws.list. How many megabytes (MB) is this, for each object? Which object requires more memory, and why do you think this is the case? Discuss any additional observations compared to part f of the previous question.

It took 20000048 bytes to store ‘big.bin.draws’ and 320000048 bytes to store ‘big.bin.draws.list’. This is equal to 20.000048 and 320.000048 megabytes respectively. The big.bin.draws.list required more memory, so the list once again required more memory just like part f from before. Again this seems to just be because a list can hold different data types.

memory\_big.bin.draws <- object.size(big.bin.draws)  
  
memory\_big.bin.draws.list <- object.size(big.bin.draws.list)  
  
print(memory\_big.bin.draws)

## 20000048 bytes

print(memory\_big.bin.draws.list)

## 320000048 bytes