Lab: Week 1

36-350 – Statistical Computing

Week 1 – Fall 2020

Name: Kimberly Zhang

Andrew ID: kyz

You must submit **your own** lab as a PDF file on Gradescope.

After each question, you will see the following:

```
# FILL ME IN
```

This, in `R Markdown` parlance, is a "code chunk." To answer the question, replace this line with your answer. Note that anything following a "#" symbol is a comment (or is code that is "commented out"). Also note that you do not need to remove the question or make other edits. Just fill in the code chunks.

To run the chunk to see if it works, simply put your cursor *inside* the chunk and, e.g., select "Run Current Chunk" from the "Run" pulldown tab. Alternately, you can click on the green arrow at the upper right-most part of the chunk, or use "<cntl>-<return>" as a keyboard shortcut.

Here is an example (that only makes sense if you are looking at the Rmd file):

Question 0

Print "Hello, world."

```
print("Hello, world.")
```

```
## [1] "Hello, world."
```

For some questions, you will also be prompted for, e.g., written explanations. For these, in addition to a code chunk, you will also see the following:

```
FILL ME IN
```

Note how there is no "{r linewidth=80}" following the tick marks in the first line. This is a verbatim block; any text you write in this block appears verbatim when you knit your file.

When you have finished answering the questions, click on the "Knit" button. This will output an HTML file; if you cannot find that file, go to the console and type `getwd()` (i.e., "get working directory")... you may find that your working directory and the directory in which you've placed the Rmd file are not the same. The HTML file should be in the working directory.

`R Markdown` may prompt you to install packages to get the knitting to work; do install these.

Vector Basics

Question 1

(3 points)

Notes 1B (3)

Initialize a vector y with one logical value, one numeric value, and one character value, and determine the type of y.

```
y= c(TRUE, 0, "K")
y
```

```
## [1] "TRUE" "0"    "K"
```

```
typeof(y)
```

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

y is a character vector.

Question 2

(3 points)

Notes 1B (5)

Sort the vector y into ascending order. Comment on the order: what type of ordering is it?

```
sort(y)
```

```
## [1] "0"    "K"    "TRUE"
```

It is sorted alphabetically; stringified numbers before letters.

Question 3

(3 points)

Notes 1B (3)

Initialize a vector y of integers, with first value 4 and last value -4, stepping down by 1. Do this *two* different ways. After each initialization, print the vector.

```
y= seq(4, -4, by=-1)
print(y)
```

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

```
y= 4:-4
print(y)
```

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

Mathematical operations between vectors was not covered directly in class. Standard operations include

Operation	Description
+	addition
-	subtraction
	multiplication

Operation	Description
/	division
^	exponentiation
%%	modulus (i.e., remainder)
%/%	division with (floored) integer round-off

Note the concept of vectorization: if x is an n-element vector, and y is an n-element vector, then, e.g., $x + y$ is an n-element vector that contains the sums of the first elements and of the second elements, etc. In other words, one does not have to loop over vector indices to apply operations to each element.

Question 4

(3 points)

Notes 1B (4)

What variable type is 1? Divide 1 by 2. Note to yourself whether you get zero or 0.5.

```
typeof(1)
```

```
## [1] "double"
```

```
1/2
```

```
## [1] 0.5
```

Question 5

(3 points)

Use R Help Pane or Google

Apply the `append()`, `cbind()`, and `rbind()` functions to the two vectors defined below. Note that "cbind" means "column bind" (glue two columns together) and "rbind" means "row bind" (glue two rows together). What is the class of the output from `cbind()` and `rbind()` functions?

```
x = 7:9
y = 4:6
class(cbind(x, y))
```

```
## [1] "matrix"
```

```
class(rbind(x, y))
```

```
## [1] "matrix"
```

```
The class of the output from cbind and rbind functions is matrix.
```

Question 6

(3 points)

Notes 1B (5)

Use the `append()` and `rev()` functions to merge the vectors x and y such that the output is 9, 4, 5, 6, 8, 7.

```
append(rev(x), y, after = 1)
```

```
## [1] 9 4 5 6 8 7
```

Logical Filtering

Question 7

(3 points)

Notes 1B (6-7)

Take the vector x defined below and display the elements that are less than -1 or greater than 1. Do this using the logical or symbol, and again via the use of the `abs()` function (for absolute value).

```
set.seed(199)
x = rnorm(20)
x[x< -1 | x > 1]
```

```
## [1] -1.909143 -2.216337 -1.132455 -1.763385  1.291574
```

```
x[abs(x) > 1]
```

```
## [1] -1.909143 -2.216337 -1.132455 -1.763385  1.291574
```

Question 8

(3 points)

Notes 1B (4,8)

What proportion of values in the vector x are less than 0.5? Use `sum()` and `length()` in your answer.

```
sum(x < 0.5)/length(x)
```

```
## [1] 0.5
```

Question 9

(3 points)

Notes 1B (8) and R Help Pane/Google

Use `any()` to determine whether any element of the vector x is less than -1. If the returned value is `TRUE`, determine which elements of x are less than -1.

```
any(x< -1)
```

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

```
x[x < -1]
```

```
## [1] -1.909143 -2.216337 -1.132455 -1.763385
```

Question 10

(3 points)

Notes 1B (5)

Sort all the values of x in *decreasing* order. Do this two different ways.

```
sort(x, decreasing= TRUE)
```

```
##  [1]  1.29157391  0.98448947  0.95177053  0.80563530  0.75756594  0.73688424
##  [7]  0.58058560  0.56505908  0.55516667  0.54160719  0.49414548  0.27542518
## [13] -0.04973667 -0.06946347 -0.29888471 -0.58057099 -1.13245520 -1.76338525
## [19] -1.90914272 -2.21633653
```

```
rev(sort(x))
```

```
##  [1]  1.29157391  0.98448947  0.95177053  0.80563530  0.75756594  0.73688424
##  [7]  0.58058560  0.56505908  0.55516667  0.54160719  0.49414548  0.27542518
## [13] -0.04973667 -0.06946347 -0.29888471 -0.58057099 -1.13245520 -1.76338525
## [19] -1.90914272 -2.21633653
```

Question 11

(3 points)

Notes 1B (8) and R Help Pane/Google

Replace all positive values in the vector x with zero, using `which()`. Confirm that all values in the new vector are ≤ 0 using `all()`.

```
# FILL ME IN
x= replace(x, which(x > 0), 0)
all(x <= 0)
```

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

Lists

Question 12

(3 points)

Notes 1C (2-3)

Create an empty list x. Then define its *second* entry as the vector 2:4. Then print the list. What value does the first entry default to?

```
x= list()
x[[2]]= 2:4
x
```

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

```
The first entry defaults to NULL.
```

Question 13

(3 points)

Use R Help Pane or Google

Use the `names()` function to rename the list entries to `x` and `y`. Print x to ensure your changes took hold.

```
names(x)= c("x", "y")
x
```

```
## $x
## NULL
##
## $y
```

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

Question 14

(3 points)

Use R Help Pane or Google

Change the name of the first entry of the list x to `a`. Do this by setting something equal to "a", i.e., *not* by simply repeating your answer to Q13. Hint: `names()` returns a vector, and you know how to change the values associated with individual entries in a vector.

```
names(x)[1]= "a"
x
```

```
## $a
## NULL
##
## $y
## [1] 2 3 4
```

Data Frames

Question 15

(3 points)

Notes 1C (5) and R Help Pane/Google

Create a data frame `df` that has columns `x` and `y` and has three rows. Use the `nrow()`, `ncol()`, and `dim()` functions to display the number of rows, the number of columns, and the dimensions of `df`. Let the first column contain numbers, and the second column contain logical values.

```
df= data.frame(x=1:3, y=c(TRUE, FALSE, TRUE))
nrow(df)
```

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

```
ncol(df)
```

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

```
dim(df)
```

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

Question 16

(3 points)

Notes 1C (3-4)

Add columns to `df` using the dollar sign operator, using the double bracket notation with number, and using the double bracket notation with character name.

```
df$z= 4:6
df[[4]]= 7:9
df[['v']]= rep(0, 3)
df
```

```
##   x     y z V4 v
## 1 1  TRUE 4  7 0
## 2 2 FALSE 5  8 0
## 3 3  TRUE 6  9 0
```

Question 17

(3 points)

Use R Help Pane or Google

Use `row.names()` to change the names of the rows of `df` to "1st", "2nd", and "3rd".

```
row.names(df)= c("1st", "2nd", "3rd")
df
```

```
##     x     y z V4 v
## 1st 1  TRUE 4  7 0
## 2nd 2 FALSE 5  8 0
## 3rd 3  TRUE 6  9 0
```

Question 18

(3 points)

Use Google

Display the contents of the first row of `df` using the row number and then using the row name. Note that you access the elements of a two-dimensional object using `[row number/name,column number/name]`.

```
df[1, ]
```

```
##     x    y z V4 v
## 1st 1 TRUE 4  7 0
```

```
df["1st", ]
```

```
##     x    y z V4 v
## 1st 1 TRUE 4  7 0
```

Matrices

Question 19

(3 points)

Notes 1C (6)

Initialize a 2 x 2 matrix where all the matrix elements are 1. Display the matrix.

```
matrix(rep(1, 4), nrow=2)
```

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

Question 20

(3 points)

Notes 1C (6)

Initialize another matrix that is 2 x 2, and fill the first column with your first and last name, and the second column with the first and last name of your favorite professor. (No pressure.) Display the matrix.

```
tmp= matrix(c("Kimberly", "Zhang", "Bill", "Hrusa"), nrow=2)
tmp
```

```
##      [,1]       [,2]
## [1,] "Kimberly" "Bill"
## [2,] "Zhang"    "Hrusa"
```

Question 21

(3 points)

Notes 1B (5)

Flip the order of entries in the second column of the matrix in the last question, in just one line of code. Display the matrix.

```
rev(tmp[, 2])
```

```
## [1] "Hrusa" "Bill"
```

Question 22

(3 points)

Notes 1C (6) and R Help Pane/Google

Define a 2 x 2 matrix with elements 1, 2, 3, 4, and another with elements 4, 3, 2, 1. Multiply the two using the `%*%` operator. Then take the transpose of the second matrix and multiply the two matrices. (See `t()`.) Then, last, compute the inverse of the first matrix. (See `solve()`.) Verify that the matrix inverse multiplies with the original matrix to yield the identity matrix.

```
u= matrix(1:4, nrow=2)
v= matrix(4:1, nrow=2)
u%*%v
```

```
##      [,1] [,2]
## [1,]   13    5
## [2,]   20    8
```

```
u%*%t(v)
```

```
##      [,1] [,2]
## [1,]   10    6
## [2,]   16   10
```

```
inv= solve(u)
inv
```

```
##      [,1] [,2]
## [1,]   -2  1.5
## [2,]    1 -0.5
```

```
u%*%inv
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Question 23

(3 points)

Use Google

When you define a (non-sparse) matrix, you set aside memory to hold the contents of that matrix. Assuming that your matrix holds double-precision floating-point numbers, and that your laptop's memory is 8 GB, what is the largest square matrix ("square" = same number of rows and columns) that you can define? (An approximate answer is fine.) This is an important consideration if, e.g., you have a set of n data points and you wish to construct a matrix that contains all the pairwise distances between points. If n gets too large, you will run out of memory.

```
Each double element is 8 bytes. 8GB approx. equals 8*10^9 bytes or 10^9 double elements.
Taking the square root of 10^9 and rounding down, we get that the largest square matrix
is 31622 by 31622.
```

Handy Vector Functions

Here we define some vectors:

```
set.seed(1201)
u = sample(100,100,replace=TRUE)
v = sample(100,100,replace=TRUE)
l = list("x"=sample(1:10,5),"y"=sample(11:20,5))
df = data.frame("x"=sample(1:10,5),"y"=sample(11:20,5))
x = c(1,2,3,4)
y = c(-2,2,-3,3)
z = c(-5,1,2,-4,3,4,-3,6)
```

Question 24

(3 points)

Notes 1D (2)

Display the list `l` as a numerical vector, with names associated with each element.

```
unlist(l)
```

```
## x1 x2 x3 x4 x5 y1 y2 y3 y4 y5
##  6  8  7  3  9 15 16 17 18 12
```

Question 25

(3 points)

Notes 1D (2)

Display the list `l` as a numerical vector, while stripping away the names seen in Q24.

```
as.vector(unlist(l))
```

```
##  [1]  6  8  7  3  9 15 16 17 18 12
```

Question 26

(3 points)

Notes 1B (5) and Notes 1D (2)

Repeat Q25, but display the vector in *descending* order.

```
sort(as.vector(unlist(l)), decreasing= TRUE)
```

```
##  [1] 18 17 16 15 12  9  8  7  6  3
```

Question 27

(3 points)

Notes 1D (3)

Here are the contents of the data frame `df`:

```
df
```

```
##   x  y
## 1 8 12
## 2 4 16
## 3 3 19
## 4 6 17
## 5 9 15
```

Reorder the rows so that the entries of the `x` column are in numerical order and the association between the ith entry of `x` and the ith entry of `y` is not lost. Display the result.

```
o= order(df$x)
df$x= sort(df$x)
df$y= df$y[o]
df
```

```
##   x  y
## 1 3 19
## 2 4 16
## 3 6 17
## 4 8 12
## 5 9 15
```

Question 28

(3 points)

Notes 1B (4) and Notes 1D (4)

Display the proportion of the total number of unique values in `u` to the number of values in `u`.

```
length(unique(u))/length(u)
```

```
## [1] 0.62
```

Question 29

(3 points)

Notes 1D (5)

Display a table that shows how often each value of `v` appears.

```
table(v)
```

```
## v
## 2 4 5 7 11 12 13 14 15 16 17 22 23 26 27 28 29 30 33 35 36 37 40 41 42 43
```

```
## 5 1 1 1 2 1 3 1 3 1 2 1 1 3 5 1 1 2 3 1 1 1 1 1 5 1
## 45 46 47 49 50 52 54 57 58 61 62 64 65 66 67 73 74 77 78 79 80 81 84 87 88
92
## 1 2 1 3 2 2 1 3 1 1 1 2 2 1 3 2 2 1 1 2 2 1 3 1 1 2
## 93 94 95 96 98
## 1 1 2 1 2
```

Question 30

(3 points)

Notes 1D (7)

How many unique values do `u` and `v` have in common?

```
unique_u= unique(u)
unique_v= unique(v)
length(intersect(unique_u, unique_v))
```

```
## [1] 39
```

Question 31

(3 points)

Notes 1B (4) and Notes 1D (7)

Write down an expression that returns `TRUE` if the union of `u` and `v` has 100 elements and `FALSE` otherwise.

```
length(union(u, v)) == 100
```

```
## [1] FALSE
```

Question 32

(3 points)

Notes 1B (5) and Notes 1D (7)

Display the (sorted!) values of `u` that do not appear in `v`.

```
sort(setdiff(u, v))
```

```
## [1] 1 3 9 10 19 20 24 34 38 39 44 55 60 63 70 72 75 82 85
## [20] 89 97 99 100
```

Question 33

(4 points)

Notes 1D (5-7)

Display a table showing how many values that are in `v` but not in `u` fall into the bins [1,50] and [51,100].

```
table(findInterval(setdiff(v, u), 50))
```

```
##
## 0 1
## 9 9
```