STA234: Homework 1

Derin Gezgin

2025-02-04

Problem 1

30/30

Report the output of the following R codes and explain what each part did.

```
c(1, FALSE)

## [1] 1 0
```

This code snippet uses the concatenation function `c` in order to combine multiple values into a vector. In this case, it combines values 1 and `FALSE` to create a vector.

Throughout the examples in this section that uses the concatenate (c) function, we can see that when there are type conversions in cases where the concatenated values are different types.

According to the R documentation [help(c)], this type conversion is done in the hierarchy of (lowest to highest):

NULL < raw < logical < integer < double < complex < character < list < expression

In this example as we are trying to concatenate two different types -double with logic- and double is higher in the hierarchy, FALSE would be converted into 0.

```
c("a", 1)

## [1] "a" "1"
```

This code snippet uses the concatenate function in order to merge a **character** "a" and a **double** 1 into a vector. As we can see in the hierarchy I added above, character has a higher priority than double which would make 1 convert to "1".

In the output we can see this as well.

```
c(list(1), "a")

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

```
## [[2]]
## [1] "a"
```

This code snippet creates a list with only one double 1, and merges this list and the character “a” in a list.

While the previous two examples concatenated values into vectors, this examples concatenates values in a list.

If we check the R documentation, we can see that the *recursive* argument in the **c** function is FALSE by default. If this was set to TRUE, the function would recursively descend through the lists and combine all elements in a vector.

As the recursive argument is FALSE, **c** function does not recursively converts the list into a vector and in order to keep a list and a double in the same structure, this function returns a new list rather than a vector.

```
c(TRUE, 1L)

## [1] 1 1
```

This code snippet merges a **logical** values (TRUE) and an **integer** 1L. Following the hierarchy I presented in the first example, we can say that TRUE is converted to 1 with a type of integer.

```
seq(4,10)

## [1]  4  5  6  7  8  9 10
```

This sequence generation function creates a sequence starting from 4 ending at 10 (both inclusive). The first argument is our starting value, and the second argument is our ending value.

```
seq(4,10,2)

## [1]  4  6  8 10
```

Similar to the previous example, this time we start from 4, end at 10 and generate this sequence with increments of 2. The added third argument gives us the increments we are creating the vector with.

```
oops = c(7,9,13)
rep(oops,3)
```

```
## [1]  7  9 13  7  9 13  7  9 13
```

This code snippet starts by creating a vector of numbers (type of double) using the concatenate function and saves these values into a variable called oops.

Following this, it uses the **rep** function in order to replicate the **oops** vector for 3 times.

```
rep(1:2,c(10,15))

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

In this case, the **rep** function takes a vector of two element -1 2- as an input. It replicates each element for a specific amount of times.

In the documentation of this function, we can see that the third element (called times) can have a single integer value (like the previous example) to repeat the whole vector for that many times or it can have another vector (equal length of the input vector) which would determine how many times each element would be repeated.

In this case, 1 will be repeated for 10 times, while 2 will be repeated for 15 times.

```
x = matrix(1:12, nrow=3, byrow=T)
t(x)

##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

In the first part of this code snippet, the code creates a matrix (**x**) with a vector of doubles from 1 to 12.

If we look at the argument values of the matrix function, we can see that it is set to have 3 rows and being filled up by row (byrow argument is True). This means that the values will fill up a row first and then go to the next row.

In the second part of this code-snippet, the transpose function is called on the matrix x in order to return the transpose of x which is interchanging rows and columns of a matrix.

```
weight_kg <- c(60,72,57,90,95,72)
height_m <- c(1.75,1.80,1.65,1.90,1.74,1.91)
bmi <- weight_kg/(height_m^2)
bmi>25
```

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

This last code snippet creates a vector of weight values in kilograms. After this, it creates another vector with the same length, which stores the height values in meters.

To calculate the BMI for each value in weight and height vectors, the program writes the BMI formula (weight / (height * height)) directly using the vectors. R would automatically create a new vector by running this formula for each pair of weight and height coming from the respective vectors thanks to the element-wise operation.

Lastly, this vector of BMI values is converted into an array of logical values using the element-wise comparison. The program would check for each element in the vector on whether they are larger than 25 or not. If so it will return TRUE for that element, otherwise FALSE.

From the output, we can see that only one of the values is larger than 25.

Problem 2 5/5

Create a vector with three different names (characters).

```
name_vector = c("derin", "johnny", "sababa")
name_vector

## [1] "derin"  "johnny" "sababa"
```

Problem 3 15/15

Report the outputs for f1 before and after modifying the levels of a factor.

```
f1 = factor(letters)
f1

##  [1] a b c d e f g h i j k l m n o p q r s t u v w x y z
## Levels: a b c d e f g h i j k l m n o p q r s t u v w x y z

str(f1)

##  Factor w/ 26 levels "a","b","c","d",..: 1 2 3 4 5 6 7 8 9 10 ...
```

In this code snippet we encode the vector of letters (a built-in R vector for lower-case letters) into a factor. When we view f1, we can see the values are lower-case letters and the correspondent levels (their labels) are also lower-case letters.

```
levels(f1) = rev(levels(f1))
f1

##  [1] z y x w v u t s r q p o n m l k j i h g f e d c b a
## Levels: z y x w v u t s r q p o n m l k j i h g f e d c b a
```

```
str(f1)

##  Factor w/ 26 levels "z","y","x","w",..: 1 2 3 4 5 6 7 8 9 10 ...
```

In this code snippet, we reverse the order of the level labels.

As the internal structure factor does not change, only the text labels would be reversed and first element in the un-reversed factor would be labeled as “z”.

If we check the structure of the new factor and compare it to the old structure, we can see that 1 corresponds to “z” now rather than “a”.

```
f2 <- rev(factor(letters))
f2

##  [1] z y x w v u t s r q p o n m l k j i h g f e d c b a
## Levels: a b c d e f g h i j k l m n o p q r s t u v w x y z

str(f2)

##  Factor w/ 26 levels "a","b","c","d",..: 26 25 24 23 22 21 20 19 18 17 ...
```

In this case, we create a new factor `f2` using the same `letters` vector. The reverse function is called on this factor itself rather than only the levels of it like the previous example. It directly reverses the letters.

This would reverse the factor itself as we can see that a is now correspond to 26 rather than 1.

```
f3 <- factor(letters, levels = rev(letters))
f3

##  [1] a b c d e f g h i j k l m n o p q r s t u v w x y z
## Levels: z y x w v u t s r q p o n m l k j i h g f e d c b a

str(f3)

##  Factor w/ 26 levels "z","y","x","w",..: 26 25 24 23 22 21 20 19 18 17 ...
```

In this last example we create a factor from the built-in letters vector but the levels of this vector is the reverse letters vector.

This means that the letter “a” belongs to levels “z” (encoded as 26), etc.

Problem 4 5/5

```
weekday_factor <- factor(c('M', 'T', 'W', 'Th', 'F', 'M', 'W'),
                         levels = c('M', 'T', 'W', 'Th', 'F'),
                         labels = c('Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday'))
weekday_factor
```

```
## [1] Monday    Tuesday   Wednesday Thursday  Friday    Monday    Wednesday
## Levels: Monday Tuesday Wednesday Thursday Friday
```

This code snippet creates a new factor from a vector (‘M’, ‘T’, ‘W’, ‘Th’, ‘F’, ‘M’, ‘W’).

The levels of this factor is set to the weekday abbreviations (‘M’, ‘T’, ‘W’, ‘Th’, ‘F’) and the labels of these levels are set to the full names of these weekdays.

When we view the weekday_factor, we can see that we see the full names of the weekdays as they are the labels of this factor.

```
str(weekday_factor)

##  Factor w/ 5 levels "Monday","Tuesday",..: 1 2 3 4 5 1 3
```

In this part the structure function is called on the weekday which can be used to inspect on the weekday_factor object.

The first part of the output shows that the factor has 5 levels and returns the first two of these level labels “Monday” and “Tuesday”.

The second part of the output shows the specific integer values representing the level of each element in the factor.

```
summary(weekday_factor)

##    Monday   Tuesday Wednesday  Thursday    Friday
##         2         1         2         1         1
```

The summary function shows each level of the factor (using their labels) in the first row, and in the second row, it shows the frequency count of each level.

Problem 5

10/10

```
event_indicator <- c(1, 0, 0, 1, 0, 0)
event_fct <- factor(event_indicator,
                    levels = c(0, 1),
                    labels = c('No', 'Yes'))
summary(event_fct)

##  No Yes
##   4   2
```

In this code-snippet, R converts a vector of 1 and 0s into a factor.

The levels are set to 0 (label - No) and 1 (label - Yes).

If we check the summary of this factor, we can see that No comes before Yes as it is set before Yes in the generation of the factor.

```
event_fct_2 <- factor(event_indicator,
                      levels = c(1, 0),
                      labels = c('Yes', 'No'))
summary(event_fct_2)

## Yes  No
##   2   4
```

In this second example, we can see that the same factor is created with a slight difference in the order of labels and levels.

We can see this difference of ordering in the summary of the factor as well.

```
levels(event_fct)

## [1] "No"  "Yes"

levels(event_fct_2)

## [1] "Yes" "No"
```

In this code-snippet, we individually view the factor levels and can see that they differ in the ordering of the factor levels.

```
fct_from_chr <- factor(c('Yes', 'No', 'No', 'Yes'))
str(fct_from_chr)

##  Factor w/ 2 levels "No","Yes": 2 1 1 2

fct_from_num <- factor(c(1, 1, 1, 4, 5))
str(fct_from_num)

##  Factor w/ 3 levels "1","4","5": 1 1 1 2 3
```

In this final code-snippet, we can see that two different factors are created.

In the first factor, there are two levels (Yes and No).

In the second part, we can see that another factor with numerical variables is created. In this factor, there are three levels: 1, 4, and 5.

It is important to note that, if the label and levels are not inputted as an argument, R would automatically assign the levels by numerical / alphabetical order.

Problem 6 15/15

The survey vector below represents survey responses where people indicated their level of comfort with data analysis.

```
survey <- c(1, 3, 3, 2, 2, 1, 1, 1, 1)
```

The numeric values have the following meaning: 1 represents ‘not comfortable’ 2 represents ‘moderately comfortable’ 3 represents ‘very comfortable’

Use the factor() function to label this vector.

In the previous examples, we used the `factor` function to create a factor and specify label & levels.

In this case, we can see that the levels are 1,2, and 3. On the other side, the respective labels where these levels are ‘not comfortable’, ‘moderately comfortable’ and ‘very comfortable’.

```
survey_factor = factor(survey,
                       levels = c(1, 2, 3),
                       labels = c("Not Comfortable", "Moderately
Comfortable", "Very Comfortable"))

str(survey_factor)

##  Factor w/ 3 levels "Not Comfortable",..: 1 3 3 2 2 1 1 1 1

summary(survey_factor)

##        Not Comfortable Moderately Comfortable       Very Comfortable
##                      5                      2                      2
```

Using the `factor` function, and the level/label information we got from the question, we can create a factor and name it survey_factor.

When we check the structure of the factor, we can see that there are three levels, first label in the factor, and the factor itself.

The summary function shows us the frequency of each label in the factor.

Problem 7 15/20

Suppose you track your commute times for 10 days and you find the following times in minutes: 17, 16, 20, 24, 22, 15, 21, 15, 17, 22.

Enter this data into R and find the longest commute time (use function max), minimum commute time (use function min) and the average commute time (use function mean).

Oops you realize that 24 was a mistake and it should have been 18. Correct one value in the data. Find the new maximum, minimum and average commute time after fixing this error.

```r
commute_times = c(17, 16, 20, 24, 22, 15, 21, 15, 17, 22)
commute_times

##  [1] 17 16 20 24 22 15 21 15 17 22
```

In this part of the code, we first create a vector of commute times to work with while tackling the rest of the task.

```r
max_commute_time = max(commute_times)
max_commute_time

## [1] 24
```

Using the `max` function, we can see that the maximum commute time is 24.

```r
min_commute_time = min(commute_times)
min_commute_time

## [1] 15
```

Using the `min` function, we can see that the minimum commute time is 15.

```r
mean_commute_time = mean(commute_times)
mean_commute_time

## [1] 18.9
```

Using the `mean` function, we can see that the mean commute time is 18.9.

To replace 24 by 18, we can have two different approaches

```r
commute_times[4] = 18
commute_times

##  [1] 17 16 20 18 22 15 21 15 17 22
```

In this methodology, we can replace 24 in the 4th index with 18.

```r
commute_times = c(17, 16, 20, 24, 22, 15, 21, 15, 17, 22)
commute_times[commute_times == 24] = 18
commute_times

##  [1] 17 16 20 18 22 15 21 15 17 22
```