Homework 2

Stats 20 Lec 1

Winter 2023

General Guidelines

Please use R Markdown for your submission. Include the following files:

- Your .Rmd file.
- The compiled/knitted HTML document.

Name your .Rmd file with the convention `123456789_stats20_hw0.Rmd`, where `123456789` is replaced with your UID and `hw0` is updated to the actual homework number. Include your first and last name and UID in your exam as well. When you knit to HTML, the HTML file will inherit the same naming convention.

The knitted document should be clear, well-formatted, and contain all relevant R code, output, and explanations. R code style should follow the Tidyverse style guide: https://style.tidyverse.org/.

Any and all course material, including these homework questions, may not be posted online or shared with anyone at any time without explicit written permission by the instructor (Michael Tsiang), even after the quarter is over. Failure to comply is a breach of academic integrity.

Note: All questions on this homework should be done using only functions or syntax discussed in Chapters 1–2 of the lecture notes or Homeworks 1–2. No credit will be given for use of outside functions.

Basic Questions

Collaboration on basic questions must adhere to Level 0 collaboration described in the Stats 20 Collaboration Policy.

The following code is used in Questions 1 and 2.

Consider the following code:

```
mixed1 <- c(TRUE, FALSE, FALSE, TRUE, 4, 0, 3)
mixed1
```

```
[1] 1 0 0 1 4 0 3
```

```
mixed2 <- c(TRUE, FALSE, FALSE, TRUE, 4, 0, "3")
mixed2
```

```
[1] "TRUE"  "FALSE" "FALSE" "TRUE"  "4"     "0"     "3"
```

```
mixed3 <- c(c(TRUE, FALSE, FALSE, TRUE, 4, 0), "3")
mixed3
```

```
[1] "1" "0" "0" "1" "4" "0" "3"
```

Question 1

The objective of this question is to help further your understanding of the mode hierarchy and the order in which commands are evaluated.

(a)

Explain why `mixed2` and `mixed3` produce different results.

(b)

Use the same input values of `TRUE, FALSE, FALSE, TRUE, 4, 0, "3"` with the `c()` function to create the following vector:

```
[1] "TRUE"  "FALSE" "0"     "1"     "4"     "0"     "3"
```

Question 2

The objective of this question is to introduce the concept of type casting.

Type Casting: The `as.logical()`, `as.numeric()`, and `as.character()` functions allow us to coerce (or **cast**) a vector into one of a different mode.

For example:

```r
as.logical(c("TRUE", "FALSE", "TRUE"))
```

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

```r
as.numeric(c("4", "0", "3"))
```

```
[1] 4 0 3
```

```r
as.character(c(TRUE, FALSE, TRUE))
```

```
[1] "TRUE"  "FALSE" "TRUE"
```

(a)

Run the commands `as.numeric(mixed2)` and `as.numeric(mixed3)` and explain why they produce different results.

(b)

Run the commands `as.logical(mixed2)` and `as.logical(mixed3)` and explain why they produce different results.

(c)

Use type casting functions to coerce `mixed3` into a meaningful logical vector (i.e., with no `NA` values).

Question 3

The objective of this question is to give practice with the `seq()` and `rep()` functions and understand their use and limitations.

(a)

Generate the following sequence in *four* ways: using (i) the `seq()` function, (ii) the colon `:` operator, (iii) the `seq_len()` function, and (iv) the `seq_along()` function.

```
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

(b)

Using `seq()` and `rep()` as needed, create the following vectors in R.

Note: Sequences should be written in such a way that they would be generalizable if the desired pattern was longer. For example, `c(1, 2, 3, 4, 5)` should be written as `seq(1, 5)`, and `c(1, 1, 1, 1, 1)` should be written as `rep(1, 5)`. Extending these patterns to 1000 values would be prohibitively cumbersome without using `seq(1, 1000)` and `rep(1, 1000)`.

(i)

```
[1] 8.0 7.5 7.0 6.5 6.0 5.5 5.0 4.5 4.0 3.5 3.0 2.5 2.0
```

(ii)

```
[1] -1  0  3  5 -1  0  3  5 -1  0  3  5 -1  0  3  5 -1  0  3  5
```

(iii)

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

(iv)

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

(v)

```
[1] 10  9  9  8  8  8  7  7  7  7  6  6  6  6  6
```

(c)

Explain why the following sequence cannot be generated using *only* the `seq()` and `rep()` functions in the same way as the sequences in (b). What makes this pattern different from the patterns in (b)? How can you generate this sequence?

```
[1] 10  9  8  7  6  9  8  7  6  5  8  7  6  5  4  7  6  5  4  3  6  5  4  3  2
```

Hint: Since this sequence cannot be generated using only `seq()` and `rep()`, consider another operation in addition to `seq()` and `rep()` to combine simpler sequences into this more complicated pattern.

Question 4

R has built-in trigonometric functions, such as `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, and `atan()`. The `sin()` function computes the sine of any numeric input.

Using at most two lines of code, compute $\sin(k\pi)$, for $k = 0, 1, 2, \ldots, 100$. Explain your result. Does the output match what you would expect mathematically?

Question 5

The objective of this question is to give practice with vectorized operations and writing functions using vectors.

The **coefficient of skewness** measures how much the distribution of a sample differs from symmetry. A perfectly symmetric distribution will have a skewness of 0. If the skewness coefficient is significantly greater than 0, then the distribution is right-skewed. If the skewness coefficient is significantly less than 0, then the distribution is left-skewed.

The skewness coefficient for a sample of values $x_1, x_2, \ldots, x_n$ is defined as

$$\text{skew} = \frac{\sqrt{n}\sum_{i=1}^{n}(x_i - \bar{x})^3}{\left[\sum_{i=1}^{n}(x_i - \bar{x})^2\right]^{3/2}},$$

where $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ is the sample mean.

(a)

Write a function called `my_skew()` that computes the skewness for any numeric input vector `x`.

(b)

Use your `my_skew()` function to compute the skewness of Chris Traeger's running times from the lecture notes. Interpret your answer.

```
running_times <- c(51, 40, 57, 34, 47, 50, 50, 56, 41, 38)
```

(c)

Through algebraic manipulation, a one-pass formula for the coefficient of skewness is

$$\text{skew} = \frac{\sqrt{n}\left(\sum_{i=1}^{n} x_i^3 - 3\bar{x}\sum_{i=1}^{n} x_i^2 + 2n\bar{x}^3\right)}{\left(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2\right)^{3/2}}.$$

Write a function called `my_skew_one()` that computes the skewness for any numeric input vector `x` using the one-pass formula. Verify that your new function gives the same answer as your function from (a).

(d)

Multiply the running times by 10^{10} (`1e10`) and compute the skewness on the new vector using both of your skewness functions. Do your results differ from your answers in (b) and (c)? Explain intuitively or mathematically why or why not.

(e)

Even though the skewness of Chris Traeger's running times should appear to be the same value in (b) and (c), you likely will find the following (or similar) result when considering the difference in the values:

```
my_skew(running_times) - my_skew_one(running_times)
```

```
[1] -4.157785e-14
```

Explain why the difference is not identically `0`.

Intermediate Questions

Collaboration on intermediate questions must adhere to Level 1 collaboration described in the Stats 20 Collaboration Policy.

Several exercises throughout this course will ask you to code your own version of basic built-in functions from scratch. While you will likely not need to rewrite existing functions from scratch when using R in practice, the thought process of thinking through and writing these functions allows you to build your skills in breaking down complicated problems into simpler steps and more deeply understanding the fundamental toolkit you are building throughout the course.

A general strategy for how to think through writing functions from scratch:

1. Come up with small general examples and think about what your function ideally should do for those examples. Do you notice patterns or using similar logic for every example?
2. Formalize and generalize the logic you used in your small examples to the general case. How should your function work on the intended input argument(s) in general? Outline or describe the steps your function needs to do to output the desired result.
3. Consider any edge cases, i.e., valid inputs that your function is supposed to work on but may not use the same logic as the general case to work. See if the logic you used in general extends to the edge case. If not, think about whether the general case can be modified to accommodate the edge case.

   In Chapter 4, we will consider a flow control statement (an `if` or `if-else` statement) that allows you to use/write entirely separate logic for different cases.

Question 6

(a)

Write a function called `my_length()` that computes the number of elements of a vector without the `length()` function. The output of `my_length(x)` and `length(x)` should be identical for any vector `x`. Your function must work for vectors of any mode and vectors of length 0.

(b)

Test your `my_length()` function from (a) with the following inputs:

(i) `numeric(0)`

(ii) `seq(1, 99, by = 2)`

(iii) `c("waffles", "friends", "work")`

Question 7

(a)

Write a function called `my_rev()` that reverses the order of elements in a vector without the `rev()` function. The output of `my_rev(x)` and `rev(x)` should be identical for any vector `x`. Your function must work for vectors of length 0.

(b)

Test your `my_rev()` function from (a) with the following inputs:

(i) `numeric(0)`

(ii) `seq(1, 99, by = 2)`

(iii) `c("waffles", "friends", "work")`

Question 8

(a)

Write a function called `front()` that inputs a vector `x` and a number `n` and outputs the first `n` elements of `x`. When `n` is greater than `length(x)`, your function should just output `x`.

With `n = 6`, verify that your `front()` function works on the vectors `numeric(0)`, `seq(1, 99, by = 2)`, and `c("waffles", "friends", "work")`. Also verify with `n = 0` that your `front()` function works on one of the aforementioned vectors.

(b)

Write a function called `back()` that inputs a vector `x` and a number `n` and outputs the last `n` elements of `x`. When `n` is greater than `length(x)`, your function should just output `x`.

With `n = 6`, verify that your `back()` function works on the vectors `numeric(0)`, `seq(1, 99, by = 2)`, and `c("waffles", "friends", "work")`. Also verify with `n = 0` that your `back()` function works on one of the aforementioned vectors.

(c)

Read through the following pseudocode:

```
FUNCTION: insert_at()
INPUTS: x, the vector the values are to be added to.
        values, the values to be included in the modified vector.
        at, the starting index of the inserted values.
OUTPUT: a vector with values inserted into vector x at the specified position.

beginning <- subset of `x` values with indices less than `at`
ending    <- subset of `x` values with indices greater-than or equal to `at`
new_vec   <- combination of beginning, values, and ending
OUTPUT: new_vec
```

Use your `front()` and/or `back()` functions from (a) and (b) to write a function called `insert_at()` that inserts elements in a vector.

(d)

Verify that your `insert_at()` function works by executing the following commands:

```r
insert_at(1:5, c(0, 0, 0), at = 1)
```

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

```r
insert_at(1:5, c(0, 0, 0), at = 3)
```

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

```r
insert_at(1:5, c(0, 0, 0), at = 5)
```

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

```r
insert_at(1:5, c(0, 0, 0), at = 6)
```

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