Functions Lab

Marcio Woitek

8/19/2022

Question 1

Review the roll functions from Section 2 in *Hands-On Programming in R*. Using these functions as an example, create a function that produces a histogram of 50,000 rolls of three 8-sided dice. Each die is loaded so that the number 7 has a higher probability of being rolled than the other numbers. Assume all other sides of the die have a 1/10 probability of being rolled.

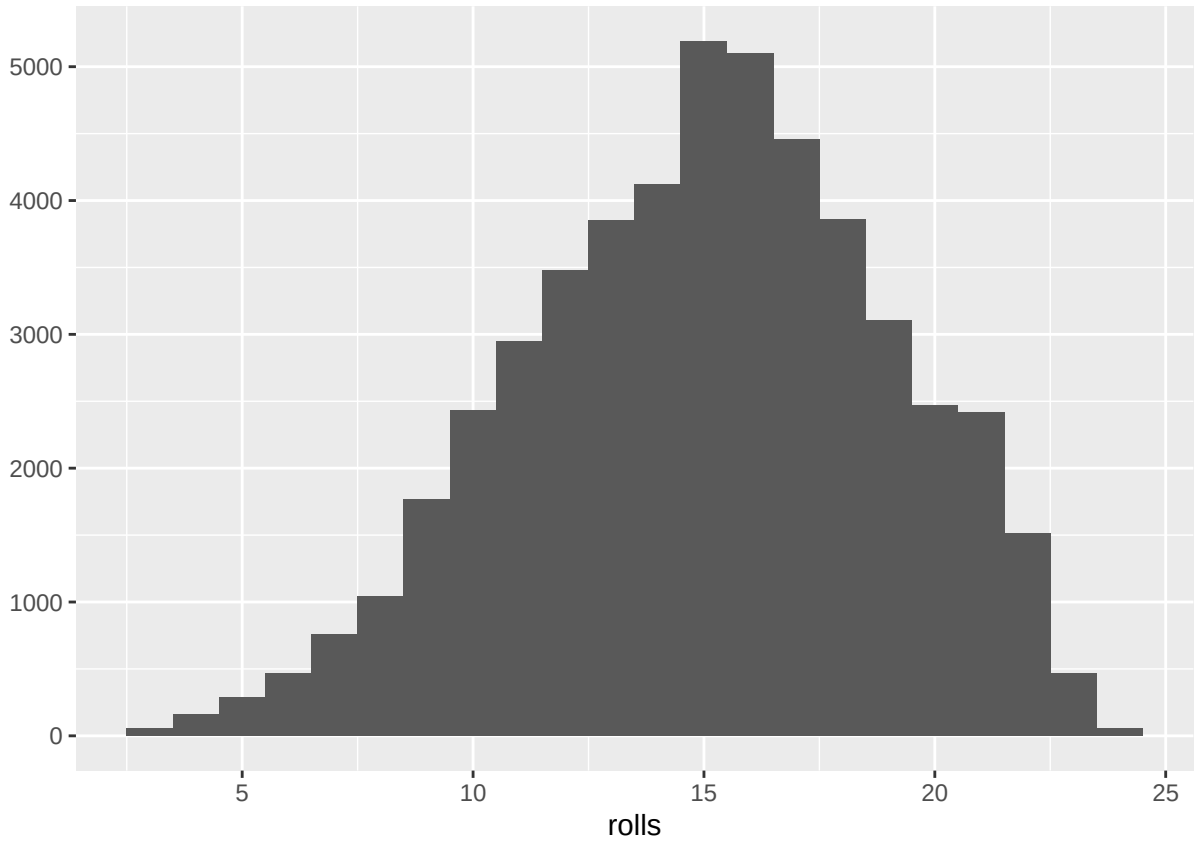
Your function should contain the arguments `max_rolls`, `sides`, and `num_of_dice`. You may wish to set some of the arguments to default values.

```
library(ggplot2)
library(magrittr)
```

```
create_histogram_rolls <- function(max_rolls, sides = 6, num_of_dice = 1) {
  # Create a die with the specified number of sides.
  die <- 1:sides

  # Create the vector of probabilities associated with the faces of the die. If
  # the number of sides is less than 7, then every number is equally likely.
  # Otherwise, the number 7 has probability 0.3, and all the other numbers are
  # assigned the same probability.
  if (sides < 7) {
    prob <- rep(1 / sides, times = sides)
  } else {
    prob_7 <- 3 / 10
    prob <- rep((1 - prob_7) / (sides - 1), times = sides)
    prob[7] <- prob_7
  }

  # Simulate the specified number of rolls.
  rolls <- replicate(
    max_rolls,
    sample(die, size = num_of_dice, replace = TRUE, prob = prob) %>% sum()
  )

  # Plot the histogram.
  qplot(rolls, binwidth = 1)
}
```

Using the above function to create the desired histogram:

```
create_histogram_rolls(50000, sides = 8, num_of_dice = 3)
```



Question 2

Write a function, `rescale01`, that receives a vector as an input and checks that the inputs are all numeric. If the input vector is numeric, map any `-Inf` and `Inf` values to 0 and 1, respectively. If the input vector is non-numeric, stop the function and return the message "inputs must all be numeric".

Be sure to thoroughly provide test cases. Additionally, ensure to allow your response chunk to return error messages.

Function implementation

```
rescale01 <- function(x) {
  # Check if x contains only numbers. Stop if that is not the case.
  if (any(!is.numeric(x))) {
    stop("inputs must all be numeric")
  }

  # Map -Inf and Inf to the specified values:
  x[x == -Inf] <- 0
  x[x == Inf] <- 1

  # Return updated x
  x
}
```

Test cases

A couple of cases in which we should see the error message:

```
x <- c("a", "b", "c")
rescale01(x)
```

```
## Error in rescale01(x): inputs must all be numeric
```

```
x <- c(1:10, "a")
rescale01(x)
```

```
## Error in rescale01(x): inputs must all be numeric
```

Next, consider two cases in which an error should NOT occur. In the first case, no value should change. In the second case, the first and last values should be mapped to 0 and 1, respectively.

```
x <- 1:10
rescale01(x)
```

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

```
x <- c(-Inf, 0.5, Inf)
rescale01(x)
```

```
## [1] 0.0 0.5 1.0
```

Question 3

Write a function that takes two vectors of the same length and returns the number of positions that have an `NA` in both vectors. If the vectors are not the same length, stop the function and return the message "vectors must be the same length".

Function implementation

```
both_na <- function(x, y) {
  # Check if the length of x is equal to the length of y. Stop if that is not
  # the case.
  if (length(x) != length(y)) {
    stop("vectors must be the same length")
  }

  # Return the number of positions
  sum(is.na(x) & is.na(y))
}
```

Test cases

First, consider a case in which we should see the error message:

```
x <- c(0, 1, NA, NA)
y <- c(-1, NA, 0)
both_na(x, y)
```

```
## Error in both_na(x, y): vectors must be the same length
```

Next, consider a couple of cases where the number of positions should be zero:

```r
x <- 1:5
y <- 6:10
both_na(x, y)
```

```
## [1] 0
```

```r
x <- c(NA, 1, 3, NA)
y <- c(2, NA, NA, 4)
both_na(x, y)
```

```
## [1] 0
```

Finally, consider a few cases in which the number of positions should be greater than zero. Notice that, for the first test case, the expected result is 1. In the last case, the output should be 2.

```r
x <- c(NA, 1, NA, 10)
y <- c(2, NA, NA, NA)
both_na(x, y)
```

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

```r
x <- c(0, 1, NA, NA)
y <- c(2, 3, NA, NA)
both_na(x, y)
```

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

Question 4

Implement a `fizzbuzz` function. It takes a single number as input. If the number is divisible by three, it returns "fizz". If it's divisible by five, it returns "buzz". If it's divisible by three and five, it returns "fizzbuzz". Otherwise, it returns the number.

Function implementation

To avoid a lot of if-else statements, I am going to use the `ifelse` function.

```r
fizzbuzz <- function(num) {
  # Check if num is divisible by three.
  if (num %% 3 == 0) {
    # Check if num is also divisible by five. If it is, then return "fizzbuzz".
    # Otherwise, return "fizz".
    return(ifelse(num %% 5 == 0, "fizzbuzz", "fizz"))
  }

  # Check if num is divisible by five. If it is, then return "buzz". Otherwise,
  # return num.
  ifelse(num %% 5 == 0, "buzz", num)
}
```

Test cases

In the first case, the input is divisible by three:

```r
fizzbuzz(27)
```

```
## [1] "fizz"
```

The next input is divisible by five:

```
fizzbuzz(25)
```

```
## [1] "buzz"
```

This input is divisible by both three and five:

```
fizzbuzz(45)
```

```
## [1] "fizzbuzz"
```

Finally, consider an input that is neither divisible by three nor by five:

```
fizzbuzz(11)
```

```
## [1] 11
```

Question 5

Rewrite the function below using `cut` to simplify the set of nested if-else statements.

```
get_temp_desc <- function(temp) {
  if (temp <= 0) {
    "freezing"
  } else if (temp <= 10) {
    "cold"
  } else if (temp <= 20) {
    "cool"
  } else if (temp <= 30) {
    "warm"
  } else {
    "hot"
  }
}
```

Function implementation

```
get_temp_desc <- function(temp) {
  cut(
    temp,
    breaks = c(-Inf, 10 * (0:3), Inf),
    labels = c("freezing", "cold", "cool", "warm", "hot")
  ) %>% as.character()
}
```

Test cases

A couple of cases in which the output should be "freezing":

```
get_temp_desc(-5)
```

```
## [1] "freezing"
```

```
get_temp_desc(0)
```

```
## [1] "freezing"
```

A couple of cases in which the output should be "cold":

```
get_temp_desc(5)
```

```
## [1] "cold"
```

```
get_temp_desc(10)
```

```
## [1] "cold"
```

A couple of cases in which the output should be “cool”:

```
get_temp_desc(15)
```

```
## [1] "cool"
```

```
get_temp_desc(20)
```

```
## [1] "cool"
```

A couple of cases in which the output should be “warm”:

```
get_temp_desc(25)
```

```
## [1] "warm"
```

```
get_temp_desc(30)
```

```
## [1] "warm"
```

A case in which the output should be “hot”:

```
get_temp_desc(35)
```

```
## [1] "hot"
```