Practice Problems: Set 1

Anthony Tetreault

2025-02-26

Review Sections 2.1-2.5 exercises:

Rstudio is an interactive interface for the R programming language. It can perform many mathematical and statistical operations on datasets through predominantly manipulation of arrays. An object in R is essentially a name used to call up stored data that has been assigned to a specific variable (the object name.) Conventions for naming objects state to only use alphanumeric characters, no spaces, characters are case_sensitive, and to use the assign operator " <- " to "load" a variable." You can then perform a variety of operations on those objects, including element-wise operations on vectors and matrix operations, as well.

Functions expand our functionality within R. Both from provided functions within R, ex. round() or sample(), and user-created functions. A function can take multiple arguments to further refine the operation carried out by the code, ex.(arguments in bold) sample(**data**=x, **size**=y, **replace**=TRUE). To determine which arguments to use with a specific function, you can pass the function through the args() function, which will give you a list of all arguments associated witht hat function, ex. args(round) or args(sample). Not all arguments need to be fully written out and declared. The function will run without the arguments explicitly declared, but will run through them in order as they should be declared. Due to this, it is recommended that you write out most argument tags, especially if you know there may be some confusion generated.

To write our own functions, we first assign the variable "func_name <- function() {}" within the (), we should include any default values or variables we would like to declare initially to be used in the function. Within the curly braces {}, we will write our function body. Function() {} will build a function out of whatever R code you place between the braces. When calling the function, "function()" the "trigger" that causes R to run the function is the (). If you type a function's name without parentheses, R will show the code for the function, instead. Within the function, arguments are declared within the "x <- function(**here**)" in the function declaration line. You can either supply the default value within the declaration itself "x <- function(name="Maha")" would then always produce default value of "Maha" for the "name" argument without anything provided when calling the function. To override and input our own value, simply call function with x("Steve") to use "Steve" as the "name."

Roll Function

```r
roll <- function() {
  die <- 1:6
  dice <- sample(die, size = 2, replace = TRUE)
  sum(dice)
}
```

Roll2 Function

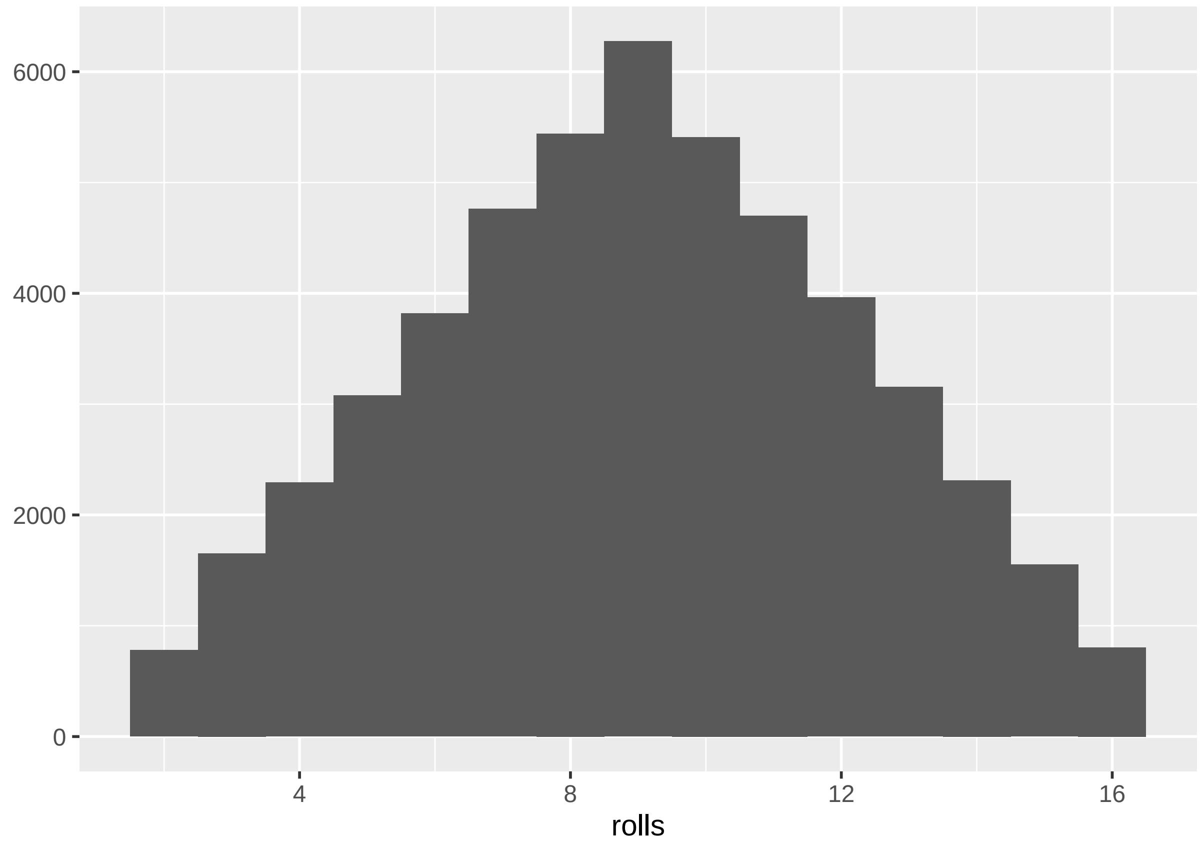
```r
roll2 <- function(bones = 1:6) {
  dice <- sample(bones, size =2, replace = TRUE)
  sum(dice)
}
```

Inputs for roll2 should be limited to a vector of numbers (x:y) within some range. If we wanted to simulate some common die formats (ex. tabletop gaming,) we would use 1:20 (d20), 1:8 (d8), 1:6 (d6), etc.

Review Sections 3.1 - 3.4 exercises:

Histogram of 50,000 rolls of three 8-sided fair dice.

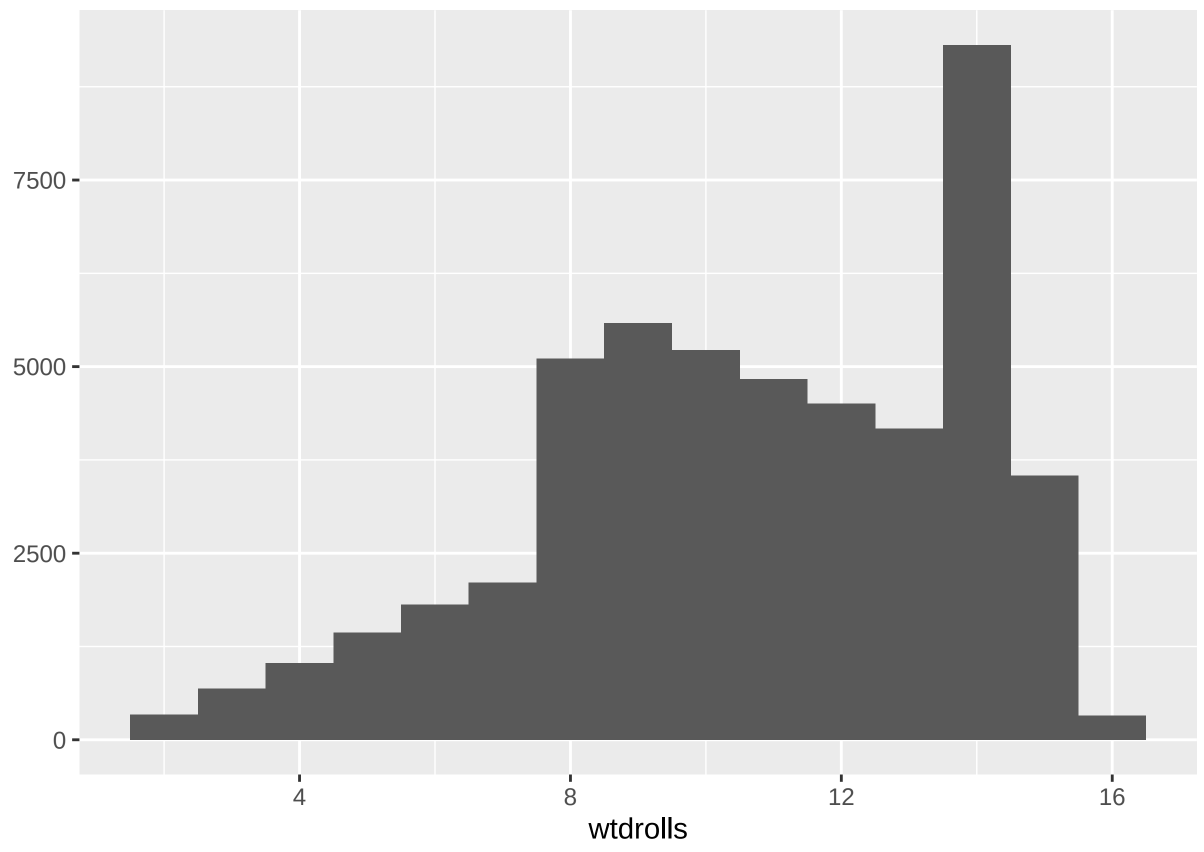
```r
roll3 <- function(die=1:8) {
  dice <- sample(die, size = 2, replace = TRUE)
  sum(dice)
}
rolls <- replicate(50000, roll3())
qplot(rolls, binwidth=1)
```



Histogram of 50,000 rolls of three 8-sided loaded dice.

(7 weighted 5/10)

```r
wtdroll <- function(die=1:8) {
  dice <- sample(die, size = 2, replace = TRUE,
                 prob = c(0.1,0.1,0.1,0.1,0.1,0.1,0.5,0.1))
  sum(dice)
}
wtdrolls <- replicate(50000, wtdroll())
qplot(wtdrolls, binwidth = 1)
```



Finally:

Rewrite the "rescale01()" function such that -Inf is 0 and Inf is 1.

```r
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE, finite =TRUE)
  y <- (x- rng[1]) / (rng[2] - rng[1])
  y[y == -Inf] <- 0
  y[y == Inf] <- 1
  y
}
z <- c(-Inf, 0, 3, 5, 7, 45, 6, 42, Inf)
rescale01(z)
```

```
## [1] 0.00000000 0.00000000 0.06666667 0.11111111 0.15555556 1.00000000 0.13333333
## [8] 0.93333333 1.00000000
```

Write both_na(), take two vectors of the same length and return number of positions that are NA in both.

```
both_na <- function(x, y) {
  sum(is.na(x) & is.na(y))
}
x <- c(NA, 1, 3, NA, NA, NA, 4)
y <- c(NA, 3, 2, 4, NA, NA, NA)
both_na(x, y)
```

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