5. Missing Values

Solutions to Swirl's R Programming Exercises

06-19-2022

Acknowledgements: R Language Concepts and code questions (with minor modifications) are used here from the swirl package. https://www.r-project.org/nosvn/pandoc/swirl.html

Important note: We don't require to use library(swirl) and swirl() here because we are not going to run the R script in RStudio Console.

Missing values play an important role in statistics and data analysis. Often, missing values must not be ignored, but rather they should be carefully studied to see if there's an underlying pattern or cause for their missingness.

In R, NA is used to represent any value that is 'not available' or 'missing' (in the statistical sense). In this lesson, we'll explore missing values further.

Any operation involving NA generally yields NA as the result. To illustrate, let's create a vector c(44, NA, 5, NA) and assign it to a variable x.

```
x <- c(44, NA, 5, NA)
```

Now, let's multiply x by 3.

```
x*3
```

```
## [1] 132  NA  15  NA
```

Notice that the elements of the resulting vector that correspond with the NA values in x are also NA.

To make things a little more interesting, lets create a vector containing 1000 draws from a standard normal distribution with y <- rnorm(1000).

```
y <- rnorm(1000)
```

Next, let's create a vector containing 1000 NAs with z <- rep(NA, 1000).

```
z <- rep(NA, 1000)
```

Finally, let's select 100 elements at random from these 2000 values (combining y and z) such that we don't know how many NAs we'll wind up with or what positions they'll occupy in our final vector – my_data <- sample(c(y, z), 100).

```
z <- rep(NA, 1000)
```

Try again. Getting it right on the first try is boring anyway! Or, type info() for more options.

The sample() function draws a random sample from the data provided as its first argument (in this case c(y, z)) of the size specified by the second argument (100). The command my_data <- sample(c(y, z), 100) will give us what we want.

```
my_data <- sample(c(y, z), 100)
```

Let's first ask the question of where our NAs are located in our data. The is.na() function tells us whether each element of a vector is NA. Call is.na() on my_data and assign the result to my_na.

```
my_na <- is.na(my_data)
```

Everywhere you see a TRUE, you know the corresponding element of my_data is NA. Likewise, everywhere you see a FALSE, you know the corresponding element of my_data is one of our random draws from the standard normal distribution.

In our previous discussion of logical operators, we introduced the `==` operator as a method of testing for equality between two objects. So, you might think the expression my_data == NA yields the same results as is.na(). Give it a try.

```
my_data == NA
```

```
##   [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
##  [26] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
##  [51] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
##  [76] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```

The reason you got a vector of all NAs is that NA is not really a value, but justa placeholder for a quantity that is not available. Therefore the logical expression is incomplete and R has no choice but to return a vector of the samelength as my_data that contains all NAs.

Don't worry if that's a little confusing. The key takeaway is to be cautious when using logical expressions anytime NAs might creep in, since a single NA value can derail the entire thing.

So, back to the task at hand. Now that we have a vector, my_na, that has a TRUE for every NA and FALSE for every numeric value, we can compute the total number of NAs in our data.

The number 1 and FALSE as the number 0. Therefore, if we take the sum of a bunch ofTRUEs and FALSEs, we get the total number of TRUEs.

Let's give that a try here. Call the sum() function on my_na to count the total number of TRUEs in my_na, and thus the total number of NAs in my_data. Don't assign the result to a new variable.

```
sum(my_na)
```

```
## [1] 47
```

Pretty cool, huh? Finally, let's take a look at the data to convince ourselves that everything 'adds up'. Print my_data to the console.

```
my_data
```

```
##   [1]           NA           NA -0.552504675 -0.347080424  0.875900822
##   [6]           NA           NA  2.338169771           NA  1.280522304
##  [11]           NA -0.959725408  0.241980507  0.030797211 -1.992285144
##  [16]  1.086545535 -0.187687975  0.185823647 -2.396332803 -0.036508705
##  [21]           NA           NA           NA           NA -0.177409370
##  [26]  1.668406574  0.213860802 -1.023888559           NA           NA
##  [31]           NA -0.112580488  0.198994100           NA           NA
##  [36]  0.177620538           NA           NA           NA  1.521415525
##  [41]  1.159530561  0.331932233 -0.267595186  1.091841411           NA
##  [46]           NA  1.114643543           NA  0.178378611           NA
##  [51]  1.101762977           NA  1.314990345  1.035309300           NA
##  [56]  0.004786004 -0.238701370           NA -1.115729114  0.107315991
##  [61] -0.612616226           NA           NA  1.884614177  0.382404956
##  [66]           NA  2.790502635 -0.075435845           NA           NA
##  [71] -1.428662669           NA  0.462512707  1.642504477  0.236642797
##  [76]           NA  0.327922816  0.073633566           NA           NA
##  [81]  1.217962082           NA -0.648944430           NA           NA
##  [86]           NA           NA           NA           NA           NA
##  [91] -0.809599548           NA -1.717028976  0.010203246           NA
##  [96]           NA  1.168806299 -0.390807923           NA           NA
```

Now that we've got NAs down pat, let's look at a second type of missing value – NaN, which stands for 'not a number'. To generate NaN, try dividing (using a forward slash) 0 by 0 now.

```
0/0
```

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

Let's do one more, just for fun. In R, Inf stands for infinity. What happens if you subtract Inf from Inf?

```
Inf-Inf
```

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