

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>

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, let's 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 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 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 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 NA
```

The reason you got a vector of all NAs is that NA is not really a value, but just a 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 of TRUEs 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] 52
```

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 -0.96571293 0.00862410 NA -0.96408732 NA
## [7] 0.25970512 -1.03385814 0.81399435 NA NA 0.97494942
## [13] 0.54328248 NA NA NA 0.50613596 NA
## [19] -0.30308631 0.21058265 NA -0.32162901 -1.02434187 1.53586873
## [25] NA NA NA NA NA 0.32729719
## [31] -0.22305358 1.38623167 NA NA NA 1.53316585
```

```
## [37]      NA  0.22431275      NA      NA -1.41565600      NA
## [43]      NA  0.49577961  0.10850041      NA      NA -1.77849146
## [49]      NA -0.30436598  1.09603717 -1.69163395 -0.59006333      NA
## [55]      NA      NA -0.05470551      NA -0.72742954 -0.95353993
## [61]      NA -1.26192847 -1.80468177 -3.01761126      NA -0.62452963
## [67]      NA      NA -0.97388358  1.15986356  0.32043316      NA
## [73]      NA      NA -0.83379776      NA      NA      NA
## [79]      NA  1.59249387      NA  0.57035787 -0.55711420      NA
## [85]      NA  0.35955910      NA  0.55514035  0.26567392  1.56980595
## [91]      NA      NA -1.17640178      NA      NA  0.21558425
## [97]      NA      NA -0.37553895      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
```