Functions

Raphael Carvalho

01/06/2019

lapply and sapply

The lapply() function takes a list as input, applies a function to each element of the list, then returns a list of the same length as the original one. Since a data frame is really just a list of vectors (you can see this with as.list(flags)), we can use lapply() to apply the class() function to each column of the flags dataset. Let's see it in action!

Type cls_list <- lapply(flags, class) to apply the class() function to each column of the flags dataset and store the result in a variable called cls_list. Note that you just supply the name of the function you want to apply (i.e. class), without the usual parentheses after it.

```
y <- "https://archive.ics.uci.edu/ml/machine-learning-databases/flags/flag.data"
flags <- read.table(y, header = FALSE, sep = ",")

cls_list <- lapply(flags, class)
```

You may remember from a previous lesson that lists are most helpful for storing multiple classes of data. In this case, since every element of the list returned by lapply() is a character vector of length one (i.e. "integer" and "vector"), cls_list can be simplified to a character vector. To do this manually, type as.character(cls_list).

```
as.character(cls_list)
```

```
##  [1] "factor"  "integer" "integer" "integer" "integer" "integer" "integer"
##  [8] "integer" "integer" "integer" "integer" "integer" "integer" "integer"
## [15] "integer" "integer" "integer" "factor"  "integer" "integer" "integer"
## [22] "integer" "integer" "integer" "integer" "integer" "integer" "integer"
## [29] "factor"  "factor"
```

sapply() allows you to automate this process by calling lapply() behind the scenes, but then attempting to simplify (hence the 's' in 'sapply') the result for you. Use sapply() the same way you used lapply() to get the class of each column of the flags dataset and store the result in cls_vect. If you need help, type ?sapply to bring up the documentation.

```
cls_vect <- sapply(flags, class)
```

Columns 11 through 17 of our dataset are indicator variables, each representing a different color. The value of the indicator variable is 1 if the color is present in a country's flag and 0 otherwise.

Therefore, if we want to know the total number of countries (in our dataset) with, for example, the color orange on their flag, we can just add up all of the 1s and 0s in the 'orange' column. Try sum(flags$orange) to see this.

```
sum(flags$orange)
```

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

First, use flag_colors <- flags[, 11:17] to extract the columns containing the color data and store them in a new data frame called flag_colors. (Note the comma before 11:17. This subsetting command tells R that we want all rows, but only columns 11 through 17.)

```
 flag_colors <- flags[, 11:17]
```

To get a list containing the sum of each column of flag_colors, call the lapply() function with two arguments. The first argument is the object over which we are looping (i.e. flag_colors) and the second argument is the name of the function we wish to apply to each column (i.e. sum). Remember that the second argument is just the name of the function with no parentheses, etc.

```
lapply(flag_colors, sum)
```

```
## $V11
## [1] 153
##
## $V12
## [1] 91
##
## $V13
## [1] 99
##
## $V14
## [1] 91
##
## $V15
## [1] 146
##
## $V16
## [1] 52
##
## $V17
## [1] 26
```

The result is a list, since lapply() always returns a list. Each element of this list is of length one, so the result can be simplified to a vector by calling sapply() instead of lapply(). Try it now.

```
sapply(flag_colors, sum)
```

```
## V11 V12 V13 V14 V15 V16 V17
## 153  91  99  91 146  52  26
```

Perhaps it's more informative to find the proportion of flags (out of 194) containing each color. Since each column is just a bunch of 1s and 0s, the arithmetic mean of each column will give us the proportion of 1s. (If it's not clear why, think of a simpler situation where you have three 1s and two 0s – $(1 + 1 + 1 + 0 + 0)/5 = 3/5 = 0.6$).

```
sapply(flag_colors, mean)
```

```
##       V11       V12       V13       V14       V15       V16       V17
## 0.7886598 0.4690722 0.5103093 0.4690722 0.7525773 0.2680412 0.1340206
```

In the examples we've looked at so far, sapply() has been able to simplify the result to vector. That's because each element of the list returned by lapply() was a vector of length one. Recall that sapply() instead returns a matrix when each element of the list returned by lapply() is a vector of the same length (> 1). To illustrate this, let's extract columns 19 through 23 from the flags dataset and store the result in a new data frame called flag_shapes. flag_shapes <- flags[, 19:23] will do it.

```
flag_shapes <- flags[, 19:23]
```

The range() function returns the minimum and maximum of its first argument, which should be a numeric vector. Use lapply() to apply the range function to each column of flag_shapes. Don't worry about storing the result in a new variable. By now, we know that lapply() always returns a list.

```
lapply(flag_shapes, range)
```

```
## $V19
## [1] 0 4
##
## $V20
## [1] 0 2
##
## $V21
## [1] 0 1
##
## $V22
## [1] 0 4
##
## $V23
## [1]  0 50
```

Do the same operation, but using sapply() and store the result ina variable called shape_mat.

```
shape_mat <- sapply(flag_shapes, range)
shape_mat
```

```
##      V19 V20 V21 V22 V23
## [1,]   0   0   0   0   0
## [2,]   4   2   1   4  50
```

As we've seen, sapply() always attempts to simplify the result given by lapply(). It has been successful in doing so for each of the examples we've looked at so far. Let's look at an example where sapply() can't figure out how to simplify the result and thus returns a list, no different from lapply().

When given a vector, the unique() function returns a vector with all duplicate elements removed. In other words, unique() returns a vector of only the 'unique' elements. To see how it works, try unique(c(3, 4, 5, 5, 5, 6, 6)).

```
unique(c(3, 4, 5, 5, 5, 6, 6))
```

```
## [1] 3 4 5 6
```

We want to know the unique values for each variable in the flags dataset. To accomplish this, use lapply() to apply the unique() function to each column in the flags dataset, storing the result in a variable called unique_vals.

```
unique_vals <- lapply(flags, unique)
#unique_vals
```

Since unique_vals is a list, you can use what you've learned to determine the length of each element of unique_vals (i.e. the number of unique values for each variable). Simplify the result, if possible. Hint: Apply the length() function to each element of unique_vals.

```
sapply(unique_vals, length)
```

```
##  V1  V2  V3  V4  V5  V6  V7  V8  V9 V10 V11 V12 V13 V14 V15 V16 V17 V18
## 194   6   4 136  48  10   8   5  12   8   2   2   2   2   2   2   2   8
## V19 V20 V21 V22 V23 V24 V25 V26 V27 V28 V29 V30
##   4   3   2   3  14   2   2   2   2   2   7   8
```

Occasionally, you may need to apply a function that is not yet defined, thus requiring you to write your own. Writing functions in R is beyond the scope of this lesson, but let's look at a quick example of how you might do so in the context of loop functions.

Pretend you are interested in only the second item from each element of the unique_vals list that you just created. Since each element of the unique_vals list is a vector and we're not aware of any built-in function in R that returns the second element of a vector, we will construct our own function.

lapply(unique_vals, function(elem) elem[2]) will return a list containing the second item from each element of the unique_vals list. Note that our function takes one argument, elem, which is just a 'dummy variable' that takes on the value of each element of unique_vals, in turn.

```
lapply(unique_vals, function(elem) elem[2])
```

```
## $V1
## [1] Albania
## 194 Levels: Afghanistan Albania Algeria American-Samoa Andorra ... Zimbabwe
##
## $V2
## [1] 3
##
## $V3
## [1] 3
##
## $V4
## [1] 29
##
## $V5
## [1] 3
##
## $V6
## [1] 6
##
## $V7
## [1] 6
##
## $V8
## [1] 2
##
## $V9
## [1] 0
##
## $V10
## [1] 3
##
## $V11
## [1] 0
##
## $V12
## [1] 0
##
## $V13
## [1] 1
##
## $V14
## [1] 0
##
## $V15
## [1] 0
```

```
##
## $V16
## [1] 0
##
## $V17
## [1] 1
##
## $V18
## [1] red
## Levels: black blue brown gold green orange red white
##
## $V19
## [1] 1
##
## $V20
## [1] 1
##
## $V21
## [1] 1
##
## $V22
## [1] 1
##
## $V23
## [1] 0
##
## $V24
## [1] 1
##
## $V25
## [1] 1
##
## $V26
## [1] 0
##
## $V27
## [1] 1
##
## $V28
## [1] 1
##
## $V29
## [1] red
## Levels: black blue gold green orange red white
##
## $V30
## [1] red
## Levels: black blue brown gold green orange red white
```