4 The apply family

Whenever you're using a for loop, you may want to revise your code to see whether you can use the lapply function instead. Learn all about this intuitive way of applying a function over a list or a vector, and how to use its variants, sapply and vapply.

lapply

When you're using R, you will often encounter vectors and lists containing all sorts of information. You've seen before that the for loop is there to help us iterate over all kinds of data structures. But I'm glad to tell you there's an even easier way! Let's have a look at some information related to New York City. It is stored in a list. Suppose we want to find out the class of each element of this list. As you already know, you could use a for loop that iterates over the different elements of nyc, and call the class() function. As expected, the first element is numeric, the second one is a character string, and the third is a logical. This however is quite a bit of code just to find out the class of elements of a list, isn't it?

```
NYC: for
nyc <- list(pop = 8405837,
            boroughs = c("Manhattan", "Bronx", "Brooklyn",
            "Queens", "Staten Island"),
            capital = FALSE
for(info in nyc) {
  print(class(info))
}
```

```
"numeric"
"character"
"logical"
```

lapply is here to rescue us from writing too much code for simple tasks such as these. I non't want to overwhelm you with a complex definition of lapply, so let us simply rewrite our previous statement. Let's replace the for loop with an lapply and see what happens. Under the hood, lapply iterated over the inputted list, nyc, and on each element of the list, applied the function class(). The output of lapply includes the results of calling the class function over each of the list elements. The first element in the output now contains the class of the first element in nyc, namely "numeric". The second element in the output contains the class of the second element in nyc, "character", and the third element contains logical, the class of the third input element. Also notice that the names of the list, pop, boroughs and capital are maintained here, that's pretty useful!

```
NYC: lapply()
nyc <- list(pop = 8405837,
            boroughs = c("Manhattan", "Bronx", "Brooklyn",
            "Queens", "Staten Island"),
            capital = FALSE
lapply(nyc, class)
```

Let's have a look at another example, now with a vector of city names. Suppose we want to build a vector of the same length as cities, containing the number of characters of each city name. We can do that by using the following for loop. First, we create an empty vector, calles num_chars. Inside the for loop, using a looping index, we gradually fill this vector up using the nchar function. We get the wanted result, but it's a lot of code to do a simple thing, in my opinion. Fortunately, refactoring this code to use lapply is easy. As the input, we use the cities vector as well as the function we want to apply on each element in this vector. We get a list containing the number of characters of the corresponding in the input vector. Notice here that the output is a list,

```
Cities: for
cities <- c("New York", "Paris", "London", "Tokyo",
            "Rio de Janeiro", "Cape Town")
num_chars <- c()
for (i in 1:length(cities)) {
num_chars[i] <- ncar(cities[i])
}
num_chars
```

```
8 5 6 5 14 9
```

although the input was a vector. This is something important about the lapply function: it always returns a list, irrespective of the input data structure. If you want to convert this list to a vector, you can simply wrap this lapply function inside the unlist() function which turns a list into a vector.

```
Cities: lapply()
cities <- c("New York", "Paris", "London", "Tokyo",
            "Rio de Janeiro", "Cape Town")
lapply(cities, nchar)
```

```
[[1]]
[1] 8

[[2]]
[1] 5
...
[[6]]
[1] 9
```

```
Cities: lapply()
cities <- c("New York", "Paris", "London", "Tokyo",
            "Rio de Janeiro", "Cape Town")
unlist(lapply(cities, nchar))
```

```
8 5 6 5 14 9
```

In a single one-liner, we have achieved the exact same thing as the bulky for loop. Moreover, by using lapply our code looks much more intuitive and readable.

In the previous examples we applied the functions class() and nchar() on every element of the input vector or list, but lapply can also be used with functions that you've written yourself. Let's try to write some code to illustrate this. Assume we have a list of oil prices per gallon. We will write a function that triples every element of oil_prices. We can write the function triple that simply calculates the triple of its input. Now, all we need to do is applying this triple function on each element of oil_prices using lapply. Works like a charm! Again, you can unlist the result, if you want your output as a vector rather than as a list.

```
Oil
oil_prices <- list(2.37, 2.49, 2.18,
                   2.22, 2.47, 2.32)
triple <- function(x) {
    3 * x
  }
result <- lapply(oil_prices, triple)
str(result)
```

```
List of 6
 $ : num 7.11
 $ : num 7.47
 $ : num 6.54
 $ : num 6.66
 $ : num 7.41
 $ : num 6.96
```

```
unlist(result)
```

```
7.11 7.47 6.54
6.66 7.41 6.96
```

Now let's try to make this triple function a bit more generic, and instead call it multiply, and have it use an additional argument, factor. We can now chose with which factor we want to multiply our input. Calling triple is now the same as calling multiply with the factor argument equal to three. But wait, what if we want to use multiply inside the lapply function? How can we specify this additional argument? Well, lapply allows you to add additional arguments to the function: just include them right after the function you want to apply to your list or vector. Now, on every element of oil_prices, the function multiply is called, with x equal to each element of the input list, and a factor of three every time. See what happens with the oil prices when we change the factor of the multiply function: they all get multiplied by four now.

```
oil_prices <- list(2.37, 2.49, 2.18, 2.22, 2.47, 2.32)
multiply <- function(x, factor) {
    x * factor
    }
times3 <- lapply(oil_prices, multiply, factor = 3)
unlist(times3)
```

```
7.11 7.47 6.54 6.66 7.41 6.96
```

```
times4 <- lapply(oil_prices, multiply, factor = 4)
unlist(times4)
```

```
9.48 9.96 8.72 8.88 9.88 9.28
```

sapply

Before, I talked about how lapply can be used to apply a function over each and every element of a list or a vector. They key thing of the lapply function is that its output is always a list. That's because an R function can return any R object. Also, the class of the R object it returns can differ depending on the input. When lapply is used to apply such a function over all elements in an input list or vector, it needs a list to store these results, because a list is able to contain heterogeneous content. However, you can think of many cases where the function always returns the same type of object over and over. We had a vector, cities. We used lapply with the nchar function. As a result, we obtained a list with the length of each of the cities names. But wait! These values could very well fit into a simple vector as well! They all have the same type! We already tried to solve this by using the unlist function to convert a list to a vector as follows. But behold! There's an easier way to tackle the case in which all the results have the same type by using the sapply function. It's short for 'simplify apply'. Awesome, right? The result is a named vector, which contains the same information as the vector we obtained earlier using unlist and lapply together. Under the hood, something slightly more complex is going on.

```
Cities: lapply()
cities <- c("New York", "Paris", "London", "Tokyo",
            "Rio de Janeiro", "Cape Town")
result <- lapply(cities, nchar)
str(result)
```

```
List of 6
 $ : int 8
 $ : int 5
 $ : int 6
 $ : int 5
 $ : int 14
 $ : int 9
```

```
unlist(lapply(cities, nchar))
```

```
8 5 6 5 14 9
```

```
sapply(cities, nchar)
```

```
New York  Paris  London  Tokyo  Rio de Janeiro  Cape Town
       8      5       6      5              14          9
```

sapply calls lapply to apply the nchar function over each element of the cities vector, and then uses the simplify2array function to convert list lapply generated to an array. In our case, sapply managed to convert the result to a one dimensional array, which is a vector. It's pretty awesome to see how R takes care of all this for us! On top of all that, sapply even found a sensible way of naming this vector. You can chose not to name the output of sapply, by setting USE.NAMES argument to FALSE. Voila, city names are gone! Remember that USE.NAMES is TRUE by default.

```
sapply(cities, nchar, USE.NAMES = FALSE)
```

```
8 5 6 5 14 9
```

Now, what would happen if the function you want to apply over the input, each time returns a vector containing two values instead of one? Let's find out with another example. The function first_and_last, that I've written for you, splits up a string to its letters, and then returns the 'minimum and maximum' letter according to alphabetical order. If we call this function on the character string "New York", the function returns a vector containing "e" and "Y". These are precisely the first and last letters of the word "New York" when ordered alphabetically. We can now use this function to apply it over every city name in cities. Instead of a vector, we now obtain a matrix, with 2 rows and 6 columns. Can you see how the output is generated? Notice here, that once again sapply assigns meaningful strings to the names of the columns and rows automatically. Both of my previous examples show the power of the sapply function to simplify the output of lapply, but what if this simplification is not possible? There are cases in which the function you want to apply does not always return a vector

```
Cities: sapply()
first_and_last <- function(name) {
    name <- gsub(" ", "", name)
    letters <- strsplit(name, split = "")[[1]]
    c(first = min(letters), last = max(letters))
  }
first_and_last("New York")
```

```
first  last
  "e"   "Y"
```

```
sapply(cities, first_and_last)
```

```
      New York  Paris  London  Tokyo  Rio de Janeiro  Cape Town
first "e"       "a"    "d"     "k"    "a"             "a"
last  "Y"       "s"    "o"     "y"    "R"             "w"
```

of the same length at all times. For these cases, simplification to a vector or a matrix just doesn't make sense. How does sapply respond to that? I've defined a function, unique_letters, that returns a vector of all the letters that are used inside a character string. If we try this function on the character string "London", we get a vector containing the unique letters in "London": "L", "o", "n" and "d". Let us first see how lapply behaves when we use unique_letters on the cities variable.

```
unique_letters <- function(name) {
    name <- gsub(" ", "", name)
    letters <- strsplit(name, split = "")[[1]]
    uniqe(letters)
  }
unique_letters("London")
```

```
"L" "o" "n" "d"
```

As expected, we get a list containing vectors of single letters. We also see that the vectors have varying lengths, so trying to simplify this list could lead to pretty strange results. Let's see how sapply handles this.

```
lapply(cities, unique_letters)
```

```
[[1]]
[1] "N" "e" "w" "Y" "o" "r"
"k"

[[2]]
[1] "P" "a" "r" "i" "s"

[[3]]
[1] "L" "o" "n" "d"

[[4]]
```

The result is the same as the lapply function, we get a list of vectors because R couldn't think of a meaningful way of simplifying the list of vectors. It was only able to give some meaningful names to the results.

```
sapply(cities, unique_letters)
```

```
$`New York`
[1] "N" "e" "w" "Y" "o" "r" "k"

$Paris
[1] "P" "a" "r" "i" "s"

$London
[1] "L" "o" "n" "d"

$Tokyo
[1] "T" "o" "k" "y"
```

That fact that sapply simplifies when possible is quite handy, but it can also lead to problems. When writing a program, you might expect that the result of a sapply() function will be a vector where in fact it's still a list because simplification didn't work out! To solve this, one can also use the R function vapply, which we'll discuss in our next chapter.

vapply

That was some pretty advanced stuff you did there! Before you head over to the final topic of this chapter, let's do a quick recap. First, you learned about lapply. This function allows you to avoid the for loop altogether and apply a function on every element of a list or a vector. The output list has the same length as the input list. The lapply function always returns a list, but there are many cases in which this list can be simplified to an array. That's why R provides the sapply function, short for simplify apply. Whenever possible, sapply tries to convert the list that lapply generates to an array. If this is not possible, however, sapply simply returns the same list that lapply generates. This can be quite dangerous, because the behavior of sapply's output depends on the specifics of the data we're using.

This short overview leads us seamlessly to the vapply function. vapply is quite similar to sapply. Under the hood, it uses lapply and then tries to simplify the result. However, when using vapply, you have to explicitly say what the type of the return value will be. In sapply, this is not required nor possible.

Let's take a look at case where sapply and vapply act quite similarly, and then check an example where the power of vapply is more clear.

```
cities <- c("New York", "Paris", "London", "Tokyo",
            "Rio de Janeiro", "Cape Town")
sapply(cities, nchar)
```

```
New York   Paris   London   Tokyo   Rio de Janeiro   Cape Town
       8       5        6       5               14           9
```

We’ll be using the cities example from before. We had a vector of city names, cities, over which we apply different functions. For example, calling the nchar function with sapply gives us a vector with the length of each character string. How can we write this using vapply? Well, when we check the documentation of they vapply function, we can see it can be used as follows. X, FUN and USE.NAMES are arguments that you already know from the sapply() function, but the FUN.VALUE argument is new here. This argument should be a general template for the return value of FUN, the function that you want to apply over the input X. In our example, we want to apply the nchar function over cities. nchar is a function that returns a single number, which is a numeric vector of length 1. We can template this output using the numeric() function, by setting FUN.VALUE to numeric(1), which tells the vapply function that nchar() should return a single numerical value.

```
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
vapply(cities, nchar, numeric(1))
```

```
New York   Paris   London   Tokyo   Rio de Janeiro   Cape Town
       8       5        6       5               14           9
```

The result is exactly the same as the sapply function from before. However, this ‘pre-specification’ of FUN’s return value makes vapply a safer alternative to sapply. To understand this, let’s re-use another example from our discussion of sapply, where we extracted the first and last letters of the cities’ names.

```
first_and_last <- function(name) {
    name <- gsub(" ", "", name)
    letters <- strsplit(name, split = "")[[1]]
    c(first = min(letters), last = max(letters))
  }
sapply(cities, first_and_last)
```

Here, sapply works like a charm again.

```
      New York  Paris  London  Tokyo  Rio de Janeiro  Cape Town
first  "e"        "a"    "d"     "k"     "a"              "a"
last   "Y"        "s"    "o"     "y"     "R"              "w"
```

To write this using vapply, we’ll need to set FUN.NAMES again. This time, the FUN we want to apply, first_and_last returns a character vector of length two, which can be expressed as character(2).

```
vapply(cities, first_and_last, character(2))
```

Works great!

```
      New York  Paris  London  Tokyo  Rio de Janeiro  Cape Town
first  "e"        "a"    "d"     "k"     "a"              "a"
last   "Y"        "s"    "o"     "y"     "R"              "w"
```

But let’s see what happens if we told vapply() that we expect first_and_last to returns a character vector of length 1.

```
vapply(cities, first_and_last, character(1))
```

This generates an error. The output of the first_and_last function is not expected, so R complains.

```
Error in vapply(cities, first_and_last, character(1)) :
  values must be length 1,
but FUN(X[[1]]) result is length 2
```

A similar error pops up if we tell vapply() that the output of first_and_last will be a numerical vector of length 2.

```
vapply(cities, first_and_last, numeric(2))
```

```
Error in vapply(cities, first_and_last, numeric(2)) :
  values must be type 'double',
but FUN(X[[1]]) result is 'character'
```

This little bit of extra work in defining the FUN.VALUES arguments has the benefit that you really have to think about what your function will return without blindly assuming that the sapply function will handle every case for you! Let’s have a look at a final example. Remember the

```
unique_letters <- function(name) {
    name <- gsub(" ", "", name)
    letters <- strsplit(name, split = "")[[1]]
    uniqe(letters)
}
```

function we wrote to get the unique letters in a string? Here it is again. We can call this unique_letters() function and apply it over the cities vector using sapply. At this point, we could have incorrectly assumed that sapply would be successful at simplifying the result to a vector, but this is not the case because the unique_letters function returns vectors of different sizes. If we try to do something similar with vapply, we have to specify the FUN.VALUE argument. Let's assume that unique_letters() always returns a vector of 4 charcter strings. As before, we get an error, because the unique_letters() function doesn't always return a vector of character strings of length 4. This stresses our main point: vapply() is safer than sapply() if you want to simplify the result that lapply() generates. Wow, that's a lot of applying that you can do in R now! In fact, there's even more. The apply, tapply, mapply and rapply functions exist as well. You'll learn more about these in the advanced R course.

```
sapply(cities, unique_letters)
```

```
$`New York`
[1] "N" "e" "w" "Y" "o" "r" "k"
...
$`Cape Town`
[1] "C" "a" "p" "e" "T" "o" "w" "n"
```

```
vapply(cities, unique_letters, numeric(2))
```

```
Error in vapply(cities, unique_letters, character(4)) :
  values must be type length 4,
but FUN(X[[1]]) result is length 7
```