Post01: The beauty of for loops (and extended applications)

Jun Seo Park

October 27, 2017

Introduction

In many ways, for loops are the most fundamental coding structure in programming language. It allows the programmer to move from crude hard-coding practices to a substantial improvement in efficiency through automation. A basic example: let's say that a programmer wishes to print out all the values from 2 to 6. Without the help of for loops, the code would look something like this:

```
print(2)
print(3)
print(4)
print(5)
print(6)
```

```
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
```

Alternatively, the smart R programmer has the option of using a for loop:

```
for (value in 2:6) {
  print(value)
}
```

```
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
```

In this simple example, the for loop does not present a significant difference; the archaic method of printing each number individually only takes 2 more lines of code than the for loop. However, when this is extended to a situation in which the action must be completed 1000 times (e.g. print every number from 1 to 1000), the for loop is significantly more time-efficient. It is in these scenarios in which the for loop is best utilized.

As is with several other functions in R, there are different ways to take advantage of the power of the for loop. This post will cover these different methods; it will also touch upon situations where for loops are not ideal, as well as more powerful modifications of the for loop.

Structure of a for loop

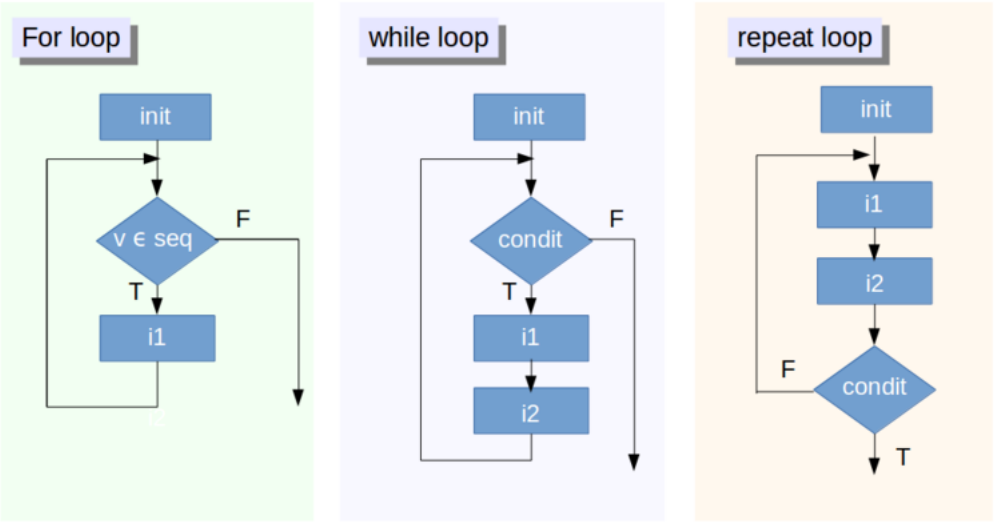

Image courtesy of datacamp.com.

The above image is a visual understanding of how for loops, while loops, and repeat loops operate (we will be focusing on the for loop). The structure of a for loop is as follows:

1. The for loop begins with an initialization - i.e. a variable `v` is set to an initial value `init`.
2. If `v` (currently set to `init`) is not in the specified sequence `seq`, the loop terminates due to the `FALSE` output.
3. If `v` is in `seq`, then the contents of the for loop (i.e. `i1`) are executed.
4. The loop revisits the logical to re-verify if `v` is in `seq`. The entire process is repeated until Step 2 returns a `FALSE` output.

Another way to use for loops

Our first example of a for loop iterated over each object; in other words, the `v` variable was set to the object itself - beginning from 2, then each object in the vector created by `2:6`. For loops can also operate based on indices, as follows:

```
x <- 2:6
for (i in 1:length(x)) {
  x[i] <- x[i] + 3
}

x
```

```
[1] 5 6 7 8 9
```

The goal of this for loop is to add 3 to each element in the vector `x = 2:6`. The for loop iterates over the index `i` from 1 to the length of the vector `x` (i.e. 5). This is particularly useful in situations in which we need to adjust the values inside the object of iteration. If we used the same format as earlier (in which we iterated over objects), we would not be able to change the values in `x`. Case in point:

```
x <- 2:6
for (value in x) {
  value <- value + 3
}

x
```

```
[1] 2 3 4 5 6
```

We can see that the initial object `x` is not changed at all (in contrast to what the other for loop was able to do). This was because we temporarily assigned the variable `value` to the corresponding number in the vector `x`; when we added 3 to `value`, it was modifying the variable `value`, not the object `x`.

Note: instead of `1:length(x)`, a common practice is to use the `seq_along` function (documentation found here). In this case, this would be implemented with the following code:

```
x <- 2:6
for (i in seq_along(x)) {
  x[i] <- x[i] + 3
}

x
```

```
[1] 5 6 7 8 9
```

Yet another way to use for loops

There is one more way to use for loops (albeit relatively uncommon). Let's return to our vector `x`, and suppose we name each of the elements in `x`. Then we can iterate over the names of the elements - achieving a similar effect as the numeric index discussed above.

```
x <- 2:6
names(x) <- c('a', 'b', 'c', 'd', 'e')

x
```

```
a b c d e
2 3 4 5 6
```

```
for (name in names(x)) {
  x[name] <- x[name] + 3
}

x
```

```
a b c d e
5 6 7 8 9
```

The efficiency of different iteration methods

This section is a peek into the world of more advanced R coding practices. In some cases, the method of iteration in the for loop is more than functionality or preference - it could affect how quickly/efficiently the code runs. Let us say we have `x = 2:6` and we wish to create a new vector `y` to store the values in which we have added 3 to each element of x. When iterating by element:

```r
x <- 2:6
y <- c()
for (value in x) {
  y <- c(y, value+3)
}

y
```

```
[1] 5 6 7 8 9
```

This is inefficient because each time R runs through the for loop, it has to copy each element of the vector y. By iterating over the indices, we can eliminate this issue:

```r
x <- 2:6
y <- numeric(length(x))
for (i in 1:length(x)) {
  y[i] <- x[i] + 3
}

y
```

```
[1] 5 6 7 8 9
```

This code only requires R to access the individual element that is being indexed; while this may seem trivial for a small vector of 5 elements, it can make a large difference when performing large operations (e.g. simulating 5 million random die rolls).

Nested for loops

Next, let us consider the nested for loop. As the name implies, this is a scenario in which one for loop is "nested" inside another. For each iteration of the outer for loop, the inner for loop runs in its entirety before iterating to the next element of the outer for loop. Let us begin with a 3x3 matrix of the numbers 1 through 12:

```r
z <- matrix(1:12, 3, 4, byrow=TRUE)

z
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

If we wanted to find the square of each element inside the matrix, we could use a nested for loop as follows:

```r
for (i in 1:nrow(z)) {
  for (j in 1:ncol(z)) {
    z[i,j] <- z[i,j]^2
  }
}

z
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    9   16
[2,]   25   36   49   64
[3,]   81  100  121  144
```

In words: the outer for loop iterates from 1 to 3, while the inner for loop iterates from 1 to 4. Once the outer for loop is initialized to `i=1`, the inner for loop must iterate from `j=1` to `j=4` before the first iteration of the outer for loop is complete. Then the outer for loop is increased to `i=2`, and the inner for loop iterates again from `j=1` to `j=4`. Then again, the process is repeated with `i=3`. Intuitively, R is going through first row and squaring the value at that row's intersection with each of the four columns before repeating the same task on the next two rows.

Note that in this example, the order of the for loops can be switched around to iterate from `j=1` to `j=ncol(y)` on the outer for loop, and `i=1` to `i=nrow(y)` on the inner for loop. This would yield the same results, but the intuition changes: this time, R would go through the first column and squaring the value at that row's intersection with each of the three rows before moving on to the remaining three columns.

Nested for loops are good for iterating through two-dimensional data objects such as matrices and data frames, or even objects with three or more dimensions. However, there is a quicker way to perform the above example in R, which we will cover in the next section.

When not to use for loops

As amazing as for loops are, R provides a built-in functionality that may diminish the power of for loops in certain cases. Let us revisit the for loop in which we add 3 to each element in the vector `x = 2:6`. Because operations in R are vectorized, there is a simpler way to write this for loop:

```r
x <- 2:6
x <- x + 3

x
```

```
[1] 5 6 7 8 9
```

Vectorization of operations means that when performing an operation (e.g. add, subtract, multiply, divide, raise to exponent, etc.) to an object, R will automatically perform the operation on each individual element of the object. Contrary to what the name of the feature may suggest, this is not limited to just vectors; in fact, it gets extremely useful when working with matrices and data frames. Returning to the example in which we square each element of the matrix, we can simply write the following:

```
z <- matrix(1:12, 3, 4, byrow=TRUE)
z <- z^2

z
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    9   16
[2,]   25   36   49   64
[3,]   81  100  121  144
```

The vectorization reduces our nested for loop to one line of extremely simple code. In theory, vectorization is in fact a for loop; but it makes writing code easier, because all the manipulation of the for loop is done behind the scenes for us by R.

Extending for loops: lapply

The culmination of this post is the `lapply` function and its wrapper functions, `sapply` and `vapply`. Consider a list of vectors as follows:

```
w <- list(c(1,2,3,4), c(5,6,7,8), c(9,10,11,12))

w
```

```
[[1]]
[1] 1 2 3 4

[[2]]
[1] 5 6 7 8

[[3]]
[1]  9 10 11 12
```

If we wanted to take the mean of each individual vector in `w`, we could write a for loop to iterate through each vector in the list. However, we have the option of using the `lapply` function to automatically do this for us. Here's a diagram of how `lapply` works:

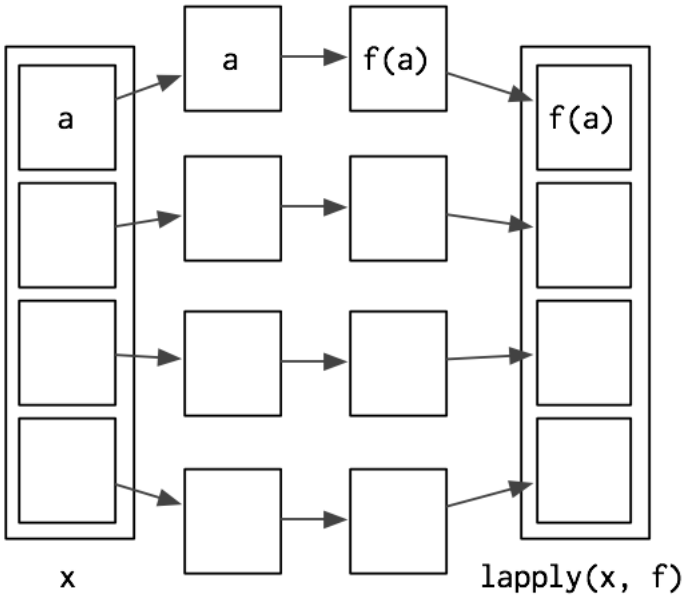

Image courtesy of Advanced R by Hadley Wickham.

As the diagram indicates, `lapply` takes in a list of several elements (in this case, 4). It then extracts each element separately and applies the function `f`. Lastly, it reassembles the results as a list and returns it.

Going back to our example with the mean of the three vectors, we can write the following:

```
lapply(w, mean)
```

```
[[1]]
[1] 2.5

[[2]]
[1] 6.5

[[3]]
[1] 10.5
```

The `lapply` function takes in `w` and uses the built-in function `mean` to apply that to each vector in `w`, then returns the three values as a list.

Custom functions in lapply

Let's say we want to find the square of each element in each vector of the list. There aren't any built-in functions in R that return a square (typically, to return a squared value, we use `^2`). Are there any ways to skirt this issue? Of course - we can write our own function! Better yet, this can be written directly inside the `lapply` call.

```
lapply(w, function(x) x^2)
```

```
[[1]]
[1]  1  4  9 16

[[2]]
[1] 25 36 49 64

[[3]]
[1]  81 100 121 144
```

The function takes the parameter `x` and returns a square value `x^2`. When `lapply` runs this function on each vector inside `w`, each vector returns four values (remember our exercise with vectorized operations?). Since there are three vectors in the list, the final return value from `lapply` is a list of three vectors, each containing 4 values that are squares of the elements in the original list of vectors.

In terms of code etiquette, a simple function such as `function(x) x^2` is readable enough to call inside the `lapply` function; however, if the function is complicated (e.g. requires more than one or two line of code), it is common practice to create a custom function prior to running `lapply`, name the function, then pass it as the second parameter in `lapply`. More information about writing functions in R can be found on this statsmethods.net tutorial.

Siblings of lapply

`lapply` has two commonly used wrappers - `sapply` and `vapply` - that accomplish similar goals in that they simplify the for loop process. However, they are different in key ways:

sapply

Whereas `lapply` returns exclusively as a list, `sapply` guesses the best format of returning the data (i.e. a vector or matrix). Returning to our examples above, we can observe:

```
sapply(w, mean)
```

```
[1]  2.5  6.5 10.5
```

```
sapply(w, function(x) x^2)
```

```
     [,1] [,2] [,3]
[1,]    1   25   81
[2,]    4   36  100
[3,]    9   49  121
[4,]   16   64  144
```

In the first `sapply` call, the means of each vector are combined into a vector (as opposed to a list of three vectors in `lapply`). In the second `sapply` call, the squared values are returned in matrix format. It is important to note that `w` was originally a list of 3 vectors of 4 elements each, or a 3x4 matrix; however, when `sapply` guessed our desired return format to be a matrix, it reformatted the data as a 4x3 matrix. While `sapply` is generally seen as user-friendly, such errors can be devastating in large-scale data analysis.

vapply

To use `vapply`, the user must specify the return format of each function call. This is better explained by example:

```
vapply(w, mean, numeric(1))
```

```
[1]  2.5  6.5 10.5
```

```
vapply(w, function(x) x^2, numeric(4))
```

```
     [,1] [,2] [,3]
[1,]    1   25   81
[2,]    4   36  100
[3,]    9   49  121
[4,]   16   64  144
```

The third parameter specifies what each iteration of the function on each vector in the list should return; the mean function returns a numeric vector of length 1, whereas the custom `x^2` function returns a numeric vector of length 4. This can be a crucial safety valve, and since R does not have to guess the format of the data return, at times it can be a more efficient call than `sapply`. The details about its efficiency and safety can be found on this StackExchange post.

apply and mapply

`apply` is similar to `sapply`; however, the user must specify the margin on which the function should be performed. Clearly, this is geared towards matrices. More information regarding `apply` can be found this comparison article on r-bloggers.com.

`mapply` is also similar to `sapply`; however, it is beyond the scope of this post (and this class). For those interested, more documentation can be found here.

Conclusion

We began with a simple overview of for loops; however, there are several different ways to iterate over an object in a for loop - by object, by index, or by name. Furthermore, some methods can be more efficient than others depending on the contents of the for loop. We also learned that for loops can be more time-consuming to write since vectorization is such a core concept in R. Lastly, we covered the `lapply` family of functions that allows the advanced R programmer to apply built-in or custom user-defined functions over a list of objects without having to write a for loop. For loops are not simply limited to just the classic for loop, but have key differences in their appearance/efficiency, and can be wrapped in other functions in which R does the work for the programmer under the hood. With this knowledge, we can be better R programmers who write reproducible and intuitive code.