Scaling up with Apply Functions

Marguerite A. Butler

March 12, 2018

1 Apply Functions

`for` loops are straightforward to understand, but they are usually slower in R and sometimes not very elegant. Another way to repeatedly execute code is via the `apply()` functions. `apply` functions are unique to R, and in some situations can operate on an entire object at once, which can make them fast. This is called **vectorization**. For example, think of a very simple function that calculates the square of a number:

```
> square <- function( x ) {
+   return (x*x)
+   }
```

If you wanted to apply it to the vector 1:10, using a for loop, it would look like this:

```
> xx <- vector(length=10)   ## create a container for output
> for ( i in 1:10 ) {       ## step through is from 1 to 10
+   xx[i] <- square( i )    ## run square function for each i
+   }
> xx

 [1]   1   4   9  16  25  36  49  64  81 100
```

This runs the `square()` function 10 times, once for each value of `i` from 1 to 10. Importantly, notice that it works by going through `i` one element at a time.

Alternatively we can use an `apply` function, let's try `sapply()`:

```
> sapply( 1:10, square )

 [1]   1   4   9  16  25  36  49  64  81 100
```

There are several different "flavors" of apply functions, but they all have similar forms:

```
> sapply( X, FUN, ...)
```

Where `X` is an object, and `FUN` is a function. The function is applied to each element of `X`, often simultaneously (whether this happens simultaneously or not depends on whether the function written with vectorization in mind, you have to just try).

Another common type is `lapply`, which operates on list objects and returns a list. `sapply` ('s' for simplify) is almost identical to `lapply`, but tries to make prettier output by returning a vector or a matrix if possible (instead of a list):

```
> sapply( 1:5, square )
```

```
[1]  1  4  9 16 25
```

```
> lapply( 1:5, square )
```

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

[[2]]
[1] 4

[[3]]
[1] 9

[[4]]
[1] 16

[[5]]
[1] 25
```

There is also `apply()` which works on matrices or arrays, and has an index argument for whether it should apply the function over rows or columns etc., `tapply` to apply the function across a grouping index or treatments, `mapply` to apply to multiple lists simultaneously, `outer` which applies the function to an outer product of two arrays, and more. `aggregate` is actually a user-friendly wrapper for `tapply`. All of the `apply` functions work in the same way. Don't get overwhelmed - I mainly use `sapply` or `lapply`, and `aggregate`, and occasionally `apply` if I need to work over rows. That's all you need to remember, consult the help page when you need.

1.1 Additional Arguments

If the function needs additional arguments, you just provide them separated by commas:

```
> sapply( X, FUN, arg1, arg2, ...)
```

For example, let's say we wanted to sample with replacement from the vector `1:5`. To do it once, we would do:

```
> sample(5, replace=T)
```

```
[1] 4 2 1 2 5
```

To do it 4 times, you could do:

```
> sapply( c(5, 5, 5, 5), sample, replace=T)
```

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

`sapply` took the vector of fives and created a sample for each one.

1.2 Using homemade functions

Sometimes the function that you want to run inside of an apply function is more complicated and requires many lines. Suppose you wanted to run several functions or have many lines of code. You have two choices. First, you can write a function definition and then pass it to an apply function:

```
> myfunction <- function (file, y=NULL, z=NULL) {
+   xx <- read.csv(file)
+   plot(xx, ...)
+   zz <- some_other_function (x,y,z)
+   ...
+   return (out)
+   }
> sapply(  list_of_filenames ,  myfunction, y=blah1, z=blah2)
```

Alternatively you could define the function within the apply function:

```
> sapply( input, function(x) {
+   ...lines_of_code...
+   })
```

Where `x` is a single element of the `input` object, so if `input` is a vector, `x` would be one element of the vector. But if `input` is a list, it would be the first list element, etc. Apply functions work really nicely with lists, and many times they handle dataframes nicely as well.

To return to one of our first examples, to code the `square` function inside of the `sapply` it would simply be:

```
> sapply ( 1:10, function(x)  x*x )

 [1]   1   4   9  16  25  36  49  64  81 100
```

Where `{}` around `x*x` are optional here because it's only one line. This is much cleaner and more elegant than:

```
> xx <- vector(length=10)   ## create a container for output
> for ( i in 1:10 ) {       ## step through is from 1 to 10
+   xx[i] <- square( i )    ## run square function for each i
+   }
> xx

 [1]   1   4   9  16  25  36  49  64  81 100
```

Furthermore, it's often easier to understand assigning the output object, because the entire object is returned, not filled element by element:

```
> xx <- sapply ( 1:10, function(x)  x*x )
```

This is another advantage of thinking of the manipulation on the whole object rather than pieces of it.

2 Exercises

1. Perform the following computation using an apply function.

```
> mylist <- vector("list")   ## creates a null (empty) list
> for (i in 1:4) {
+    mylist[i] <- list(data.frame(x=rnorm(3), y=rnorm(3)))
+ }
```

2. Plot x as a function of y for each dataframe using an apply function.

3. Using an apply function, compute an anova on `x` $\sim$ `y` on each dataframe, and save the anova output (there should be 4 of them) to a list or dataframe.

4. Write a for loop that finds the sum of the sequence of integers from 1 to 100, then accomplish the same computation with an apply function.