Looping, Vectorization, and User-Defined Functions

Justin French

2024-08-20

## A Vectorization Pre-Game

R supports vectorized operations natively, so you end doing them without thinking about it all the time. **Vectorization** is just the simultaneous execution of a task across multiple object elements. For example, if you were to multiply a numeric vector by a single constant, this is what happens:

# Make a vector of numbers  
 vec <- c(4:44)  
  
 # Apply multiplication to all elements  
 vec \* 2

## [1] 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56  
## [26] 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88

This principle makes R very efficient from the user’s point of view. It seems simple on the surface, but it actually gets pretty complicated when applied to more complex situations or more sophisticated objects.

The proper sequence of ideas to tackle this is to begin with **sequential operations** before really diving into vectorized operations. So, that’s the progression we will follow.

## Looping

**Looping** is the act of moving sequentially through a set of values of a a variable and repeatedly executing the same code with updated values of the variable. At least, that’s what for loops do. It is an example of an **iterator** (an operation that repeats things), but it only considers the sequence of values you feed it. Other iterators consider different things. The basic syntax of an R for loop looks like this:

for(i in vec){   
 cat(i \* 2, ' ')  
 }

## 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88

A for loop begins with the for, then in parenthesis you specify an arbitrary variable name, provide an in, then tell it what vector to pull values of the variable from. You then open curly braces ({ }) and write the code sequence to be repeated inside. Anywhere the variable you specified for the loop shows up will have the value updated each time the loop runs.

When you want to store succesive outputs of a loop in a single object you need to use an **accumulator**. This isn’t a distinct code object, but more of a technique. For example, if I want to store all the outputs of our multiplication in a new vector I need to do this:

# Make an empty vector  
 vec2 <- c()  
  
 # Fill empty vector with values  
 for(i in vec){   
 # vec2 <- c(vec2, i \* 2)  
 # or  
 vec2[i] <- i \* 2  
 }

If the vector you are feeding the loop to iterate over are numbers that start at 1 and increase, you can use the vector its self as an index when storing outputs in the accumulator. An example of this would be 1:nrow(data.frame) where you wanted to do something specific to each row od a data frame, or 1:ncol(data.frame) to do something to every column.

Another important concept with loops is **nesting**, or putting one iterator inside another. The most common is a nested for loop. These are handy when you have to work across higher dimensional objects. For example, if we loop across a matrix we need to iterate over columns and then rows, or vice versa:

# Make a matrix  
 mat <- matrix(nrow = 5,  
 byrow = T,  
 data = c(0:24))  
  
 for(i in 1:nrow(mat)){  
 for(j in 1:ncol(mat)){  
 mat[i,j] <- mat[i,j]/5  
 }  
 }  
  
 mat

## [,1] [,2] [,3] [,4] [,5]  
## [1,] 0 0.2 0.4 0.6 0.8  
## [2,] 1 1.2 1.4 1.6 1.8  
## [3,] 2 2.2 2.4 2.6 2.8  
## [4,] 3 3.2 3.4 3.6 3.8  
## [5,] 4 4.2 4.4 4.6 4.8

In the case above, the inner loop runs for every value within a single run of the outer loop. This has the effect of making loop start on a row, do all the columns, then move to the next row, and then run through all the columns again, and so on.

We incorporated an example of **soft coding** when we specified the vectors that each loop ran over. This is when you make a function learn some of the necessary values for it to run from the object you run it on. The nrow() and ncol() functions learn the dimensions of the matrix without you having to manually specify them.

## While loops

Another iterator that is under-appreciated by R users is the while loop. Instead of updating a value of something, it keeps the loop running until a condition is met. To write one, you need to know what has to happen for the loop to terminate. For example, if we want to see how high R can count in 3 seconds, we could write a while loop like this:

# Start the counter  
 i <- 0  
   
 # Get the stop time  
 stop <- Sys.time() + 3  
   
 # Keep counting until time runs out  
 while(Sys.time() < stop){  
 i <- i + 1  
 }  
  
 # Print the result  
 i

## [1] 88334

You can make a while loop stop on an event that may not occur at a pre-specified time or sequence. For example, if you want the loop to continue until a certain value comes up, you could do this:

# Set a start "time"  
 t <- 0  
  
 # How many draws to get 15?  
 while(i != 15){  
 i <- rpois(n = 1, lambda = 5)  
 t = t + 1  
 }

### Logical comparisons

Logical comparisons are a special set of functions in R that only return TRUE or FALSE. AS you can see above, we used one in our while loop. The logical comparisons are as follows: <, >, <=, >=, ==,!=, and %in%. They all look to see if the relationship between what is on the left side of the opperator holds with whatever is on the right. The while loop will run until the comparison returns a FALSE.

For clarity, here are a few examples of logical comparisons:

4 < 5

## [1] TRUE

4 > 5

## [1] FALSE

4 == 5

## [1] FALSE

4 != 5

## [1] TRUE

4 %in% c(3,6,4,7)

## [1] TRUE

The double-equal differentiates the logical comparison from assigning a value to an argument in a function.

## Vectorized Operations

While a lot of R objects have intrinsic vectorized methods (e.g. a vector being multiplied by a number), you often need to write custom code to do vectorized operations for more complex stuff.

This is accomplished with the apply() family of functions. Each of these takes an input object and applies a prespecified function to each element of it *simultaneously*. The most basic apply statement is just apply(). It is intended to run on arrays, matrices, and data frames (i.e., 2-dimensional objects). For example, if we wanted to see what the data type was for each column of a data frame, we could use the typeof() function on the data frame with an apply() statement:

# Uses mtcars data frame  
 apply(X = mtcars, # Specify data object  
 MARGIN = 2, # Rows (1) or Columns (2)?  
 FUN = typeof) # Specify function

## mpg cyl disp hp drat wt qsec vs   
## "double" "double" "double" "double" "double" "double" "double" "double"   
## am gear carb   
## "double" "double" "double"

We could have accomplished the same task with a for loop, but it would have taken 4-5 lines of code. Anything a vectorized operator can do could be accomplished with a loop. The main benefit of apply statements is compactness of the code. In previous versions of R, vectorized statements had a clear speed advantage. However, that is largely academic in modern R releases; the difference is usually negligible.

There is a variety of apply functions to suit different purposes. The most common variants are lapply(), which works on lists, and sapply() which simplifies the output from working on a list.

### An advanced example

Remember from yesterday that we are automotive engineers who are very concerned with fuel economy. However, our cheap supervisors will only let us measure one criterion about a vehicle to estimate it’s fuel economy. We have to figure out which single attribute of a vehicle best predicts fuel economy. Fortunately, we have the mtcars data set to use for this. We’ll fit 3 models and store their outputs in a list, then apply the AIC() function to determine which has the lowest score:

# Make a list to hold models  
 modList <- list()  
  
 # First model  
 modList[['mod.disp']] <- # Make it a named element  
 lm(mpg~disp, data = mtcars) # Fit the model  
  
 # Second model  
 modList[['mod.cyl']] <- # Make it a named element  
 lm(mpg~cyl, data = mtcars) # Fit the model  
   
 # Third model   
 modList[['mod.wt']] <- # Make it a named element  
 lm(mpg~wt, data = mtcars) # Fit the model

Now that we have a bunch of models, let’s figure out how to get the AIC score of all 3 as a vector. We need to follow a set of steps to decide what all is needed:

* Step 1: What function am I gonna use on each element and what arguments does it have?
* Step 2: What is the object structure of the input and what is the desired output? Which apply statement is correct to accomplish this? If it isn’t a matrix, data frame, or some other array, you can’t use apply(). If the input is a list, you need lapply() or sapply(). If you want the output as a list, lapply() is the way to go. If you want the output simplified to an easier object to work with, use sapply().

Our input is a list, and we want a vector as the output. That means we should try sapply():

sapply(X = modList,  
 FUN = AIC)

## mod.disp mod.cyl mod.wt   
## 170.2094 169.3064 166.0294

## Writing Your Own Functions

All a function is is a block of code that take a certain input, does a certain task with it, at returns a certain output. Calling it a function just means you gave it a name. This is actually very easy to do. The basic syntax is as follows:

myFunction <- function(arg1, arg2 = FALSE){  
 internalThing <- arg1 < 9  
 return(internalThing == arg2)  
 }

Like any R obect (which functions are), you start by assigning it to a name. You then open the function() function and specify what the arguments of the function will be. If you want one to have a default value, you just set it equal to something. Once you are done setting arguments inside the parenthesis, you open curly braces and write code that uses the arguments as variables. Any information that your code needs should be specified in the arguments (i.e., the code should be self-contained). At the end of the code, you need to use the return() function to specify what your function will provide as output. The guts can get quite complicated, but all functions follow this design.

Let’s write a function that performs our AIC comparison on a list of models. we’ll start simple, but make it progressively fancier. For now, let’s just make it do exactly what our initial version did:

aicComp <-   
 function(modList){ # Specify input argument  
 res <- sapply(modList, # Use the argument in func.  
 AIC) # Specify the function  
 return(res) # Return the output  
 }  
  
 aicComp(modList = modList) # Run the function

## mod.disp mod.cyl mod.wt   
## 170.2094 169.3064 166.0294

This is a lot of extra code to do what we accomplished with one line earlier. How about we make our function produce a whole table representing our AIC comparison in a way we could use for our paper/thesis/dissertation?

aicComp <-   
 function(modList){ # Specify input argument  
 modName <- names(modList)  
   
 nCovs <- sapply(modList,  
 function(x){  
 length(x$coefficients)-1  
 })  
   
 modAIC <-sapply(modList, AIC)   
   
 modR2 <- sapply(modList,  
 function(x){  
 summary(x)$r.squared  
 })  
   
 res <- data.frame(modName = modName,  
 covariates = nCovs,  
 AIC = modAIC,  
 R2 = modR2  
 )  
   
 rownames(res) <- 1:nrow(res) # Set the rownames attribute  
   
 return(res) # Return the output  
 }  
  
 aicComp(modList = modList) # Run the function

## modName covariates AIC R2  
## 1 mod.disp 1 170.2094 0.7183433  
## 2 mod.cyl 1 169.3064 0.7261800  
## 3 mod.wt 1 166.0294 0.7528328