Another one of those blog posts about apply

Silas Tittes

31 August, 2016

The most excellent Hadley Wickham has a great presentation about functional programming where he compares the redundancies in cupcake recipes to the construction of for loops. The goal of the presentation is to promote his new package `purrr`, which provides a more comprehensive set of functional programming tools for `R`, and to show audience members the ease and power of a functional programming approach. I found the analogy and overall point very persuasive, so much so, I changed my twitter bio to include the phrase, "recovering for loop addict". I'm no stranger to using the `apply` family, but I never considered it my go-to approach, especially if the task was complex. But the cupcakes put me over the edge, and I decided to take the plunge as much as I could.

I'm still learning `purrr`, so I'll avoid discussing the `map` functions just yet. Instead I'll talk about using `lapply()` cause it's the `apply` function I use most often. The thing I always find most difficult is nesting `lapply()` calls in an effort to replicate nested loop structures. I thought I would compare a nested loop and a nested `lapply()` call side-by-side to see how they compare in syntax and efficiency.

```
#vectors to loop over
x_vec <- 1:5 #change size to see differences in process time
y_vec <- x_vec

c_time <- Sys.time()
#store output
storage_loop <- array(NA, c(length(x_vec), length(y_vec)))
#set_along, following Wickham suggestion
for(i in seq_along(x_vec)){
  for(j in seq_along(y_vec)){
    storage_loop[i, j] <- paste(i, j) #fill storage
  }
}

loop_time <- Sys.time() - c_time #check duration


c_time <- Sys.time()
storage_apply <- sapply(as.list(x_vec),
                        function(x){sapply(as.list(y_vec),
                                           function(y) paste(y, x))})
apply_time <- Sys.time() - c_time

loop_time - apply_time
```

```
## Time difference of 0.003070593 secs
```

```
#is the same output is produced?
mean(storage_loop == storage_apply)
```

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

This is obviously an overly simplistic example, but I think it demonstrates the syntax and efficiency differences quite well. Replacing `paste()` with something you care about iterating over should be easy once you know how to structure the syntax.

Notice `lapply()` requires a list as input, hence the conversion of the index vectors to list with `as.list()`. I like creating line breaks between the the list input and the function input. There may be prettier ways to do this, but I find this fairly readable.There is also a lot of flexibility concerning the data input/output options with `apply`, which is definitely not true for `for` loops.

For example, if we wanted to output a linear 1 x n instead of matrix:

```
#changed storage to 1d array, changed the length from n by n to n*n
#added counter to keep track of storage vector index
storage_loop <- array(NA, length(x_vec)*length(y_vec))
#set_along, following Wickham suggestion
count <- 1
for(i in seq_along(x_vec)){
  for(j in seq_along(y_vec)){
    storage_loop[count] <- paste(i, j) #fill storagecount
    count <- count + 1
  }
}


#changed outter apply to sapply, and added unlist()
#flipped x and y order in paste to make comparable to loop
storage_apply <- unlist(lapply(as.list(x_vec),
                        function(x){sapply(as.list(y_vec),
                                           function(y) paste(x,y))}))

#is the same output is produced?
mean(storage_loop == storage_apply)
```

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

There may not be many reasons to do this, but it shows the ease of changing the structure of the data output from the `lapply()`/`sapply()` call compared to the loop. Not to mention, you can create all sorts of other list structures with calls to `apply`. These can be useful depending on downstream applications. all by changing between `sapply()` and `lapply()`.

```
storage_apply <- lapply(as.list(x_vec),
                        function(x){lapply(as.list(y_vec),
                                           function(y) paste(x,y))})


#just the first chunk
storage_apply[[1]]
```

```
## [[1]]
## [1] "1 1"
##
## [[2]]
## [1] "1 2"
##
```

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

```
storage_apply <- sapply(as.list(x_vec),
                        function(x){lapply(as.list(y_vec),
                                           function(y) paste(x,y))})
storage_apply
```

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

```
storage_apply <- lapply(as.list(x_vec),
                        function(x){sapply(as.list(y_vec),
                                           function(y) paste(x,y))})
storage_apply[[1]]
```

```
## [1] "1 1" "1 2" "1 3" "1 4" "1 5"
```

```
storage_apply[[2]]
```

```
## [1] "2 1" "2 2" "2 3" "2 4" "2 5"
```

Lastly, if you have a multi-core machine, you can speed up calls to `apply` with the `parallel` package without any extra work (unless you count typing "library(parallel)" and "mc". Only do this if you have a large process though, as there is a bit of a "start up" cost.

```
library(parallel)

#vectors to loop over
x_vec <- 1:20#00 #change size to see differences in process time
y_vec <- x_vec

c_time <- Sys.time()
storage_apply <- lapply(as.list(x_vec),
                        function(x){lapply(as.list(y_vec),
                                           function(y) paste(y, x))})
apply_time <- Sys.time() - c_time

c_time <- Sys.time()
storage_apply <- mclapply(as.list(x_vec),
                        function(x){lapply(as.list(y_vec),
```

```
                                        function(y) paste(y, x))})
mcapply_time <- Sys.time() - c_time

apply_time - mcapply_time
```

```
## Time difference of -0.006709099 secs
```

Here is a summary of my current feelings about for loops version functional programming approaches:

	pro 'for' loop	**con 'for' loop**	**pro 'apply'**	**con 'apply'**
Syntax	Straight forward, similar across programming languages. Not unpleasant to look at. Easier to make one-off test cases to ensure the script does what you intended.	Requires more typing. Requires explicitly allotting storage.	Far more compact.	Sometimes difficult to write and interpret. Hard to make "pretty" especially with anonymous function calls. Less dealing with storage.
Comp. Efficiency		slow	fast! parallelizable. Even faster! (sometimes)	

In summary, if you're new to programming and using `R`, learn how to use loop structures, which will make you more prepared to learn other languages. Once you know how to use them, ditch them in `R`. If you just can't get the syntax to work in `apply` calls (or `map` calls), go back to your old loopy ways.