Lesson 9: Scaling with purrr to loop through a vector

Randall Julian, Adam Zabell

1/3/2018

Looping Commands

Reducing duplication in your typed code makes it easier to see the purpose of the code, easier to change the calculation, and easier to remove bugs. The three typical methods in R to iterate with a specific piece of code are functions, for-loops, and vector transformations with the **purrr** package.

Looping with user functions

Every command in **R** is consistent with the function syntax *command(inputA,inputB,...)* and writing your own function is no different. Here, we create two functions that will return a median value across every column, or across every row, of a data.frame.

```
tableOfNumbers <- tibble(
  a = c(53, 1, 51, 23, 28, 12, 87, 0, 47),
  b = floor(100 * runif(9)),
  c = as.integer(100 * rnorm(9)),
  d = rep(5, 9),
  e = seq(1, 9)
)
medianByColumn <- function(tb) {
  recordOfValues <- c()
  for (i in 1:ncol(tb)) {
    byColumn <- unlist(tb[, i])
    recordOfValues %<>% append(median(byColumn))
  }
  return(recordOfValues)
}
medianByRow <- function(tb) {
  recordOfValues <- c()
  for (i in 1:nrow(tb)) {
    byRow <- unlist(tb[i, ])
    recordOfValues %<>% append(median(byRow))
  }
  return(recordOfValues)
}
```

Both of these functions use a for-loop to cycle through the `byColumn` or `byRow` vector of numbers. The logic is basically identical except for the placement of a single comma, which means a single function could be written to handle both tasks. Finally, the `recordOfValues` empty vector needs to be initalized before it can be used, which is slightly confusing. Each of these comments are addressed below.

Looping with *for* and *while*

These loops are explicitly sequential, showing the input and output of each command in exactly the same way as the primary code you work with. The *for* loop follows a three component format: create the output

vector, define the looping sequence, and perform the body of work. In our example functions, that body is two lines long, but it could be as complex as necessary to produce the intended output. When working with long loops (100s to 1000s of iterations), the inefficiency of growing the output vector at each iteration will slow down the code. Instead, the output should be initialized as an empty vector of the expected length. Modifying our `medianByColumn` to resolve this inefficiency:

```
medianByColumn <- function(tb) {
  recordOfValues <- vector("double", ncol(tb))
  for (i in 1:ncol(tb)) {
    byColumn <- unlist(tb[, i])
    recordOfValues[i] <- median(byColumn)
  }
  return(recordOfValues)
}
```

> Exercise 1: Making a two-variable function that 'switches'
> Create a function `medianBy( )` that accepts both the data.frame and a logical value to indicate whether the result should be by row or column.

```
medianBy <- function(tb, byRow=FALSE) {

}
```

The *while* loop is used when the work being performed should continue until a logical condition occurs. The obvious risk is that the condition never occurs, prompting the need for a failsafe. They are rarely used, but here are two examples.

```
numberOfFlips <- 0
numberOfHeads <- 0
while (numberOfHeads < 3) { # risky
  if (sample(c("H", "T"), 1) == "H") {
    numberOfHeads <- numberOfHeads + 1
  }
  numberOfFlips <- numberOfFlips + 1
}

countedFlips <- 0
numberOfTails <- 0
while (numberOfTails < 3 | countedFlips == 1000) { # safer
  if (sample(c("H", "T"), 1) == "T") {
    numberOfTails <- numberOfTails + 1
  }
  countedFlips <- countedFlips + 1
}
```

Looping with *map*

These loops are explicitly vectored, performing the action on every element in that vector. The **purrr** package has five functions which return a vector for each type of output: list, logical, integer, double, and character. It comes with a significant increase in speed and further improved readability, but requires a different seeming workflow. Most notably, for-loops often have bookkeeping variables for the intermediate steps in the body of work, while `map` functions generate the output vector directly and are designed to work well with pipes.

The *map* structure builds on the idea that a solution to "how do I solve for a single set of data" is simpler to visualize. After breaking the problem down to small steps on one set, enclosing those steps in the function

will iterate on each column of data in the data.frame to report the result. Revisiting our example functions to use *map_dbl* which will return a real (double precision) number:

```
medianByColumn <- function(tb) {
  recordOfValues <- map_dbl(tb, median) %>% unname()
  return(recordOfValues)
}
medianByRow <- function(tb) {
  transposedTibble <- as_tibble(t(tb))
  recordOfValues <- map_dbl(transposedTibble, median) %>% unname()
  return(recordOfValues)
}
```

Exercise 2: Improve the `medianBy( )` function to use map looping.

```
medianBy <- function(tb, byRow=FALSE) {

}
```

Nested Looping

Having built one loop, a second (or more) iteration may still be necessary to do the work. This is usually done for simple counting variables like `i` and `j` as shown below.

```
vectorList <- vector("list", 10)
for (i in 1:10) {
  oneVector <- vector("double", 20)
  for (j in 1:20) {
    oneVector[j] <- i * j
  }
  vectorList[[i]] <- oneVector
}
nestingTibble <- vectorList %>%
  set_names(map_chr(1:10, paste0, "column")) %>%
  bind_cols()
```

It can also be done using the map functions. In this example, *map_dfc* returns a data.frame created by column-binding each entry.

```
betterNestingTibble <- c(1:10) %>%
  map_dfc( ~ tibble(x = . * 1:20) ) %>%
  set_names(map_chr(1:10, paste0, "column"))
```

Deciding when to use a function instead of a for-loop is often an aesthetic choice focused on the readability of the code. The `betterNestingTibble` may not be as clear as code that keeps for-loop(i) and uses the map command to replace the for-loop(j).

Summary

- `functions( )` capture regularly used segments of code so the main workflow can focus on the result instead of the process
- `for( ) { }` loops retain the serial nature of the workflow and avoid the need to copy/paste the iterations
- `map( )` functions are incredibly fast and powerful, and the preferred looping mechanism in R once the format is understood