CODING LOOPS WITH "for" - LECTURE

DSC 205 - Advanced introduction to data science

Marcus Birkenkrahe

February 15, 2023

README

Figure 1: Photo by La-Rel Easter on Unsplash

Download the **raw** practice file from GitHub and save it as `5_loop_for_practice.org`.

To test your Emacs mettle, open it on the CMD line with the command `emacs -nw` (no graphics - not needed for this exercise).

Coding loops overview

- Loops repeat a specified section of code often while incrementing an index or counter variable.
- R knows three ways of looping: `for` and `while` as in other C-type languages, and `apply` to run a function over different data structures.
- A `for` loop repeates code while going through a vector until a condition is no longer met:

```
for (loopindex in loopvector) {
  do any code in here
}
```

- A `while` loop repeats code until a condition evaluates as `FALSE`:

```
while (loopcondition) {
  do any code in here
}
```

- The `apply` family of functions allows implicit looping over subsets of vectors, matrices or arrays to apply a function to all elements:

```
apply( X = data
       MARGIN = subset,
       FUN = function)
```

`for` loops

- A `for` loop repeats code while going through a vector until a condition is no longer met:

```
for (loopindex in loopvector) {
  do any code in here
}
```

- `loopindex` represents an element in `loopvector`
- Simple example:

```
for (myitem in 5:7) {
  cat("--BRACED AREA BEGINS--\n")
  cat("the current item is", myitem, "\n")
  cat("--BRACED AREA ENDS--\n")
}

--BRACED AREA BEGINS--
the current item is 5
--BRACED AREA ENDS--
--BRACED AREA BEGINS--
the current item is 6
--BRACED AREA ENDS--
--BRACED AREA BEGINS--
the current item is 7
--BRACED AREA ENDS--
```

- You can manipulate objects outside the loop, i.e. the braced area is not a local environment:

```
counter <- 0
for (myitem in 5:7) {
  counter <- counter + 1
  cat("The item in run",counter,"is",myitem,"\n")
}

The item in run 1 is 5
The item in run 2 is 6
The item in run 3 is 7
```

Looping via index or value

- There is a difference using a *loopindex* inside a `for` statement

vs. representing *indices* of a vector.

- **Loop index:** The loop below uses the *loopindex* `i` to directly

represent the elements in `myvec`:

```
myvec <- c(0.4, 1.1, 0.34, 0.55)
for (i in myvec) {
  print(2 * i)
}
```

```
[1] 0.8
[1] 2.2
[1] 0.68
[1] 1.1
```

- **Vector index:** The loop below uses `i` to represent `integer` values in

the sequence `1:length(myvec)`, which form all vector index positions of `myvec`. The same indices are then used to extract elements from `myvec`.

```
for (i in 1:length(myvec)) {
  print(2 * myvec[i])
}
```

```
[1] 0.8
[1] 2.2
[1] 0.68
[1] 1.1
```

- Vector indices take a longer form but offer more flexibility for

more complicated `for` loops.

Extended example: stepping through a `list`

- You want to write code that will inspect any `list` object and gather information about any `matrix` objects stored as `list` members.
- Sample data:

  ```
  foo <- list(
    aa=c(3,4,1),
    bb=matrix(1:4,2,2),
    cc=matrix(c(T,T,F,T,F,F),3,2),
    dd="string here",
    ee=matrix(c("red","green","blue","yellow")))
  foo
  ```

```
$aa
[1] 3 4 1

$bb
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$cc
      [,1]  [,2]
[1,]  TRUE  TRUE
[2,]  TRUE FALSE
[3,] FALSE FALSE

$dd
[1] "string here"

$ee
     [,1]
[1,] "red"
[2,] "green"
[3,] "blue"
[4,] "yellow"
```

- Problem:
  1. Go through every member of the list
  2. Check whether the member is matrix
  3. If it is a matrix, retrieve number of rows and columns and the data type of the matrix.
- Solution: create vectors to store `list` member information:
  1. `name` of the `list` member name
  2. `is.mat` ("Yes" or "No") to indicate if it is a `matrix`
  3. `nc` and `nr` for numbers of rows and columns for each matrix
  4. `data.type` to store the data type of each matrix

```
name <- names(foo); name
```

```
is.mat <- rep(NA,length(foo)); is.mat
nr <- is.mat
nc <- is.mat
data.type <- is.mat
```

```
[1] "aa" "bb" "cc" "dd" "ee"
[1] NA NA NA NA NA
```

- All sought variables are initialized with "missing" values `NA` and updated in the `for` loop. The results are put in a data frame `bar`.

```
for (i in 1:length(foo)) {
  member <- foo[[i]] # pick list element
  if (is.matrix(member)) {
    is.mat[i] <- "Yes"     # update matrix indicator
    nr[i] <- nrow(member)  # update row counter
    nc[i] <- ncol(member)  # update column counter
    data.type[i] <- class(as.vector(member)) # conversion!
  } else {
    is.mat[i] <- "No"
  }
}
bar <- data.frame(name,
                  is.mat,
                  nr,
                  nc,
                  data.type)
bar
```

```
  name is.mat nr nc data.type
1   aa     No NA NA      <NA>
2   bb    Yes  2  2   integer
3   cc    Yes  3  2   logical
4   dd     No NA NA      <NA>
5   ee    Yes  4  1 character
```

- Compare with the original list `foo` where the structure output indirectly suggests `matrix` objects through the indexing:

```
str(foo)
```

```
List of 5
 $ aa: num [1:3] 3 4 1
 $ bb: int [1:2, 1:2] 1 2 3 4
 $ cc: logi [1:3, 1:2] TRUE TRUE FALSE TRUE FALSE FALSE
 $ dd: chr "string here"
 $ ee: chr [1:4, 1] "red" "green" "blue" "yellow"
```

TODO Nesting `for` loops

- `for` loops can be nested just like `if` statements.
- The inner loop is executed in full before the outer loop *loopindex* is incremented. Then the inner loop is executed all over again.
- Example: write code that loops over rows and columns of a matrix and update the matrix elements as the product of outer and inner loopindex.
- Solution: first, create loopindices and a base matrix:
  1. Create a `loopvec1` as loopindex vector `5 6 7`
  2. Create a `loopvec2` as loopindex vector `9 8 7 6`
  3. Create a `matrix foo` of missing values whose row and column numbers correspond to `loopvec1` and `loopvec2`, respectively

```
loopvec1 <- 5:7; loopvec1
loopvec2 <- 9:6; loopvec2
foo <- matrix(NA,length(loopvec1),length(loopvec2)); foo
baz <- foo # make copy of foo for later
```

```
[1] 5 6 7
[1] 9 8 7 6
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
```

- The outer `for` loop should run over as many elements as `loopvec1` has, and the inner `for` loop should run over as many elements as `loopvec2` has. `foo` is then updated accordingly:

```
for (i in 1:length(loopvec1)) {
  for (j in 1:length(loopvec2)) {
    foo[i,j] <- loopvec1[i] * loopvec2[j]
  }
}
foo

     [,1] [,2] [,3] [,4]
[1,]   45   40   35   30
[2,]   54   48   42   36
[3,]   63   56   49   42
```

☐ With this code, is the matrix traversed by row or by column?[1]

- Inner loopvectors can be defined to match the current value of the loopindex of the outer loop:

```
for (i in 1:length(loopvec1)) {
  for (j in 1:i) {
    baz[i,j] <- loopvec1[i] * loopvec2[j]
  }
}
baz

     [,1] [,2] [,3] [,4]
[1,]   45   NA   NA   NA
[2,]   54   48   NA   NA
[3,]   63   56   49   NA
```

- Note that inner loop indices are decided based on the outer loop index: for example, when `i=1`, the inner loopvector is `1:1` so it is executed only once before moving on to the next row.

☐ The code will fail if `length(loopvec1) > length(loopvec2)` - why?[2]

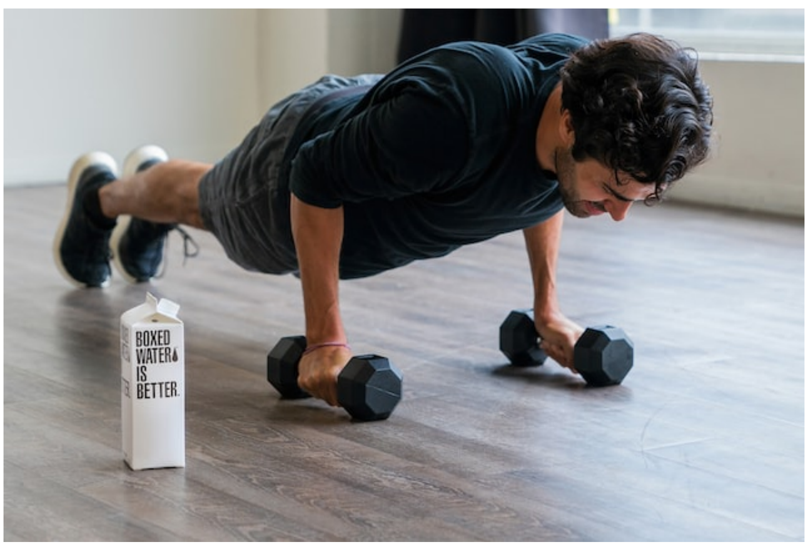
[1] By column - in the example, the sequence of matrix elements filled is: `foo[1,1]`, `foo[1,2]`, `foo[1,3]`, `foo[2,1]` etc.

[2] Because the inner loopvector `1:i` will exceed the number of elements of `loopvec2` - "subscript out of bounds".

```
loopvec1 <- 1:4
loopvec2 <- 9:7
qux <- matrix(NA,length(loopvec1),length(loopvec2)); foo
for (i in 1:length(loopvec1)) {
  for (j in 1:i) {
    qux[i,j] <- loopvec1[i] * loopvec2[j]
  }
}
```

```
     [,1] [,2] [,3] [,4]
[1,]   45   40   35   30
[2,]   54   48   42   36
[3,]   63   56   49   42
Error in `[<-`(`*tmp*`, i, j, value = loopvec1[i] * loopvec2[j]) :
  subscript out of bounds
```

TODO Exercises



Download the raw exercise file from GitHub and save it as `5_loop_forexercise.org`.

TODO Glossary

TERM	MEANING

References

- Davies, T.D. (2016). The Book of R. NoStarch Press.