CODING LOOPS WITH "apply" - LECTURE

DSC 205 - Advanced introduction to data science

Marcus Birkenkrahe

February 24, 2023

README

Figure 1: Photo by Brett Jordan on Unsplash

- Download the **raw** file to practice during the lecture from GitHub, save it as `7_loop_apply_practice.org` and upload it to Canvas later.
- To test your Emacs mettle, open it on the CMD line with the command `emacs -nw` (no graphics - not needed for this exercise).

`apply`, `lapply`, `sapply`, `tapply`, `vapply`, `mapply`

Figure 2: Family by Rajiv Perera on Unsplash

- The `apply` family of functions allows implicit looping over subsets of vectors, matrices or arrays to apply a function a subset of their elements.
- The different flavors of `apply` are:
  1. `apply` applies a function to a dataset's margin (segment)
  2. `tapply` to apply a function to subsets defined in terms of `factor` vectors, i.e. sliced by categorical variable values.
  3. `lapply` operates member by member on a `list`.
  4. `sapply` to return simplified `lapply` results.
  5. `vapply` if you know the data type you're expecting as a return.
  6. `mapply` as a multi-variate version of `sapply`

- All apply type functions simplify coding enormously by replacing for loop constructions.
- All apply type functions allow for additional arguments (...) to be passed to FUN.

apply - implicit looping over arrays

- The apply function returns a vector or array or list of values by applying a function to the MARGIN of an array or matrix.

```
apply( X = data  # array with positive dim(X)
       MARGIN = subset, # aka row/column/layer etc.
       FUN = function)  # includes your own function
```

- Does apply have any other arguments? Find out!

```
args(apply)
```

```
function (X, MARGIN, FUN, ..., simplify = TRUE)
NULL
```

What's an array?

- scalars: 0-dim arrays, e.g. the number 0
- vectors: 1-dim arrays, e.g. c("a","vector")
- matrices: 2-dim arrays, e.g. matrix(1:9,3,3)
- arrays: n-dim, e.g. array(1:9,dim=c(3,3,2))

```
array(1:18,dim=c(3,3,2))
```

```
, , 1

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

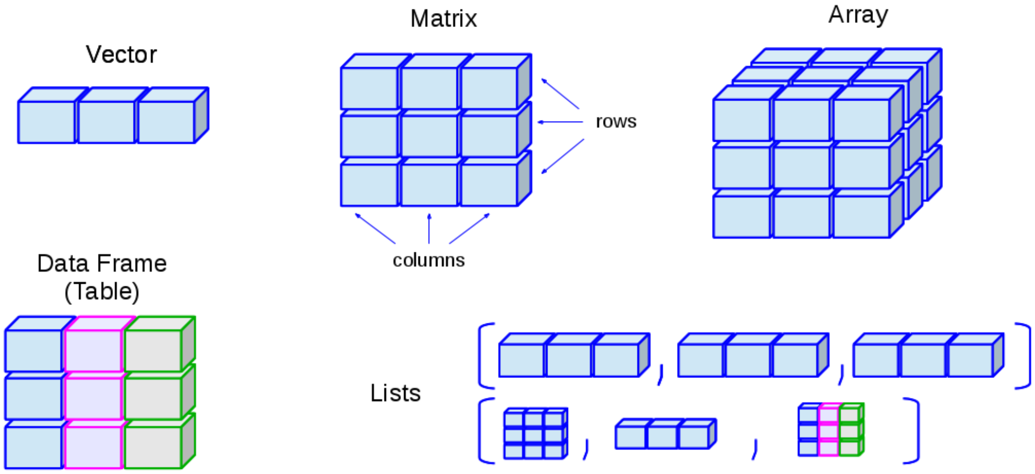

Figure 3: R data structures (source: Ceballos, 2013)

```
, , 2

     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18
```

- The `MARGIN` index follows the positional order of the dimension for matrices and arrays:

MARGIN	DATA STRUCTURE
1	rows
2	columns
3	layers
4	blocks

`apply` example: `matrix`

- Create a 4 x 3 matrix with the elements `1:12`:

```
foo <- matrix(1:12,4,3) # default byrow=FALSE
foo
```

```
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

- Find the sum of each row of `foo`. What about `sum(foo)`?

  ```
  sum(foo)  # this sums up ALL elements
  ```

  ```
  [1] 78
  ```

- Loop over the rows of `foo`:

  ```
  row.totals <- rep(NA, times=nrow(foo)) # initialize counter
  for (i in 1:nrow(foo)) {
    row.totals[i] <- sum(foo[i,]) # sum over i-th row
  }
  row.totals
  ```

  ```
  [1] 15 18 21 24
  ```

- Much shorter with `apply`:

  ```
  apply(
    X = foo,
    MARGIN = 1, # MARGIN = 1 (rows), 2 (cols), 3 (layers), 4 (blocks)
    FUN = sum)
  ```

  ```
  [1] 15 18 21 24
  ```

- To `sum` over columns instead, change `MARGIN` to `2`.

  ```
  apply(
    X = foo,
    MARGIN = 2,
    FUN = sum)
  ```

  ```
  [1] 10 26 42
  ```

- You can pass additional arguments to any `apply` function: e.g. you can use the function `sort` and specify it to be `decreasing`:

  ```
  apply(
    X = foo,
    MARGIN = 1,
    FUN = sort,
    decreasing = TRUE)
  
       [,1] [,2] [,3] [,4]
  [1,]    9   10   11   12
  [2,]    5    6    7    8
  [3,]    1    2    3    4
  ```

`apply` example: `array`

- Create a 3 x 2 x 2 array `bar` with the elements `1:18`

  ```
  bar <- array(1:18, dim=c(3,3,2))
  bar
  
  , , 1
  
       [,1] [,2] [,3]
  [1,]    1    4    7
  [2,]    2    5    8
  [3,]    3    6    9
  
  , , 2
  
       [,1] [,2] [,3]
  [1,]   10   13   16
  [2,]   11   14   17
  [3,]   12   15   18
  ```

- Put differently, `bar` has 2 layers of 3 x 3 matrices. What does the following call do?[1]

[1] The `apply` call extracts the diagonal elements for each of the 2 layers with `diag`. Each call to `diag` of a matrix returns a vector and these vectors are returned as columns of a new matrix.

```
baz <- apply(bar,3,FUN=diag)
baz
```

```
     [,1] [,2]
[1,]    1   10
[2,]    5   14
[3,]    9   18
```

- Check the dimensions and class of baz:

```
dim(baz)
class(baz)
is.matrix(baz)
is.array(baz)
```

```
[1] 3 2
[1] "matrix" "array"
[1] TRUE
[1] TRUE
```

tapply - slicing data by categories

- tapply performs operations on subsets defined by factor vectors
- Simple example: compute the mean tooth length by supply category in the ToothGrowth dataset:

```
tapply(X = ToothGrowth$len,   # length of guinea pig teeth
       INDEX = ToothGrowth$supp,  # OJ or VC supply
       FUN = mean)   # arithmetic average
```

```
      OJ       VC
20.66333 16.96333
```

- The result returns the average length for guinea pigs supplied with orange juice (OJ) and vitamin C (VC).
- Here's another example (data source: Kaggle)[2]:

[2] Astonishingly, some websites are trying to sell these (freely available) data for US$100.00 (see here).

1. read web data on diamond pricing (with strings as factors)
2. display structure of data table
3. display first five records

```
dia.url <- "https://raw.githubusercontent.com/birkenkrahe/ds2/main/data/diamonds.c
diamonds <- read.csv(dia.url, stringsAsFactors=TRUE)
str(diamonds)
head(diamonds)
```

```
'data.frame': 53943 obs. of  11 variables:
 $ X       : int  1 2 3 4 5 6 7 8 9 10 ...
 $ carat   : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
 $ cut     : Factor w/ 5 levels "Fair","Good",..: 3 4 2 4 2 5 5 5 1 5 ...
 $ color   : Factor w/ 7 levels "D","E","F","G",..: 2 2 2 6 7 7 6 5 2 5 ...
 $ clarity: Factor w/ 8 levels "I1","IF","SI1",..: 4 3 5 6 4 8 7 3 6 5 ...
 $ depth   : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
 $ table   : num  55 61 65 58 58 57 57 55 61 61 ...
 $ price   : int  326 326 327 334 335 336 336 337 337 338 ...
 $ x       : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
 $ y       : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
 $ z       : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
  X carat       cut color clarity depth table price    x    y    z
1 1  0.23     Ideal     E     SI2  61.5    55   326 3.95 3.98 2.43
2 2  0.21   Premium     E     SI1  59.8    61   326 3.89 3.84 2.31
3 3  0.23      Good     E     VS1  56.9    65   327 4.05 4.07 2.31
4 4  0.29   Premium     I     VS2  62.4    58   334 4.20 4.23 2.63
5 5  0.31      Good     J     SI2  63.3    58   335 4.34 4.35 2.75
6 6  0.24 Very Good     J    VVS2  62.8    57   336 3.94 3.96 2.48
```

- Using `tapply`, you can add up the total value of the diamonds for the full data set but separated according to `color` (key coded):

  ```
  tapply(
    X = diamonds$price,
    INDEX = diamonds$color,
    FUN = sum)
  ```

  ```
         D        E        F        G        H        I        J
  21476439 30148457 35545622 45158240 37257301 27608146 14949281
  ```

`lapply` - cycling through lists

- `lapply` operates member by member on a `list` and returns a `list`:

  ```
  baz <- 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")))
  ```

- Check for matrices in the list `baz`:

  ```
  baz <- 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")))
  lapply(
    X = baz,
    FUN = is.matrix)

  $aa
  [1] FALSE

  $bb
  [1] TRUE

  $cc
  [1] TRUE

  $dd
  [1] FALSE

  $ee
  [1] TRUE
  ```

- No margin or index information is required. R knows how to apply `FUN` to each member of the list, and returns a `list`. Fun!

sapply - simplified cycling

- sapply (s = "simplified") returns the same results as lapply but in an array form:

  ```
  baz <- 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")))
  sap <- sapply(
          X = baz,
          FUN = is.matrix)
  sap
  is.vector(sap)   # sap is a named vector
  ```

  ```
     aa    bb    cc    dd    ee
  FALSE  TRUE  TRUE FALSE  TRUE
  [1] TRUE
  ```

- baz has a names attribute that is copies to the corresponding entries of the returned object:

  ```
  attributes(sap)
  names(sap)
  str(sap)
  ```

  ```
  $names
  [1] "aa" "bb" "cc" "dd" "ee"
  [1] "aa" "bb" "cc" "dd" "ee"
   Named logi [1:5] FALSE TRUE TRUE FALSE TRUE
   - attr(*, "names")= chr [1:5] "aa" "bb" "cc" "dd" ...
  ```

- If we did not have sapply, you could unlist the result of lapply:

  ```
  unlist(lapply(baz,is.matrix))
  sapply(baz,is.matrix)
  ```

  ```
     aa    bb    cc    dd    ee
  FALSE  TRUE  TRUE FALSE  TRUE
     aa    bb    cc    dd    ee
  FALSE  TRUE  TRUE FALSE  TRUE
  ```

SOMEDAY `vapply` - simplified cycling with safety check

- Read the help file and this tutorial (Treadway, 2020).

SOMEDAY `mapply` - multivariate version of `sapply`

- Read the `help` file and this tutorial (Zach, 2021).

TODO Exercises

1. Write an implicit loop that calculates the product of all the column elements of the matrix returned by the call to `apply(foo, 1, sort, decreasing=TRUE)` where `foo` is `matrix(1:12,4.3)`.

   Tip: To multiply numbers, you can use the function `prod`.

2. Convert the following `for` loop to an implicit loop that does exactly the same thing. Here, `t` transposes its matrix argument.

Bonus: compare the results of the two operations without looking.

```
matlist <- list(
  matrix(c(T,F,T,T),2,2),
  matrix(c("a","c","b","z","p","q"),3,2),
  matrix(1:8,2,4))
matlist # original list

for (i in 1:length(matlist)) {
  matlist[[i]] <- t(matlist[[i]])
}
matlist  # transposed list
```

```
[[1]]
      [,1] [,2]
[1,]  TRUE TRUE
[2,] FALSE TRUE

[[2]]
     [,1] [,2]
[1,] "a"  "z"
[2,] "c"  "p"
[3,] "b"  "q"

[[3]]
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
[[1]]
     [,1]  [,2]
[1,] TRUE FALSE
[2,] TRUE  TRUE

[[2]]
     [,1] [,2] [,3]
[1,] "a"  "c"  "b"
[2,] "z"  "p"  "q"

[[3]]
     [,1] [,2]
```

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

Glossary

TERM	MEANING
`apply`	apply `function` to the `margin` of a dataset `X`
`tapply`	apply `function` to subsets grouped by `factor`
`lapply`	apply `function` to `list` members, return `list`
`sapply`	simplified `lapply`, returns vector
`vapply`	`apply` when you know the return datatype
`mapply`	multivariate version of `sapply`

References

- Ceballos, M. (2013). Data structure. URL: venus.ifca.unican.es.

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

- Treadway, A. (20 Oct 2020). Why you should use vapply in R. URL: theautomatic.net.

- Zach (Dec 7, 2021). How to Use the mapply() Function in R (With Examples). URL: statology.org.