# Week 3: Loop Functions and Debugging

Katherine Ramírez Cubillos

21/8/2020

## Loop Funcitons

Loop functions are some of the most powerful funcitons in the R language and they make it kind of very easy to use, especially in an interactive setting.

### Looping on the command line

Writing for, while loops is useful when programming but not particularly easy working interactively on the command line. There are some functions which implement looping to make life easier.

- **lapply**: Loop over a list and evaluate a function on each element.
- **sapply**: Same as **lapply** but try to simplify the result.
- **apply**: Apply a function over the margins of an array.
- **tapply**: Apply a function over subset of a vector.
- **mapply**: Multivariate version of **lapply**

An auxiliary function **split** is also useful, particularly in conjunction with **lapply**.

#### Lapply

The idea behind lapply is that you have a list of objects and you want to loop over the list of objects and apply a function to every element of that list.

**lapply** takes three arguments: (1) a list **x**; (2) a function (or the name of a function) **FUN**, (3) other arguments via its … argument. If **x** is not a list, it will be coerced to a list using **as.list**.

```
lapply
```

```
## function (X, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     if (!is.vector(X) || is.object(X))
##         X <- as.list(X)
##     .Internal(lapply(X, FUN))
## }
## <bytecode: 0x000000000694e358>
## <environment: namespace:base>
```

The actual looping is done internally in C code.

**lapply** always returns a list, regardless of the class of the input.

```
x <- list(a=1:5, b=rnorm(10))
lapply(x, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] 0.1326336
```

```
x <- list(a=1:4, b=rnorm(10), c=rnorm(20,1), d=rnorm(100,5))
lapply(x, mean)
```

```
## $a
## [1] 2.5
##
## $b
## [1] 0.5542493
##
## $c
## [1] 0.7964816
##
## $d
## [1] 4.96137
```

```
x <- 1:4
lapply(x, runif)
```

```
## [[1]]
## [1] 0.5381652
##
## [[2]]
## [1] 0.3171926 0.3633443
##
## [[3]]
## [1] 0.07608729 0.54212169 0.94641856
##
## [[4]]
## [1] 0.77497663 0.99051351 0.96770183 0.04252631
```

```
x <- 1:4
lapply(x, runif, min = 0, max = 10)
```

```
## [[1]]
## [1] 8.90858
##
## [[2]]
## [1] 3.975878 5.211960
##
```

```
## [[3]]
## [1] 8.859777 4.678851 3.779923
##
## [[4]]
## [1] 6.081779 1.034820 5.458470 9.173858
```

**lapply** and friends make have use of *anonymous* functions.

```
x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
x
```

```
## $a
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $b
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

An anonymous funciton for extracting the first column of each matrix

```
lapply(x, function(elt) elt[,1])
```

```
## $a
## [1] 1 2
##
## $b
## [1] 1 2 3
```

### Sapply

**sapply** will try to simplify the result of **lapply** if possible.

- If the result is a list where every element is length 1, then a vector is returned.
- If the result is a list where every element is a vector of the same length (>1), a matrix is returned.
- If it can't figure things out, a list is returned.

```
x <- list(a =1:4, b=rnorm(10), c=rnorm(20,1), d=rnorm(100,5))
lapply(x, mean)
```

```
## $a
## [1] 2.5
##
## $b
## [1] 0.09113882
##
## $c
## [1] 1.135171
```

```
##
## $d
## [1] 4.89027
```

```
sapply(x, mean)
```

```
##          a          b          c          d
## 2.50000000 0.09113882 1.13517057 4.89026965
```

```
#mean(x)
# [1] NA
# Warning message:
# In mean.default(x): argument is not numeric or logical: returning NA.
```

## Apply

**apply** is used to a evaluate a funciton (often an anonumous one) over the margins of an array.

- It is most often used to apply function to the rows or columns of a matrix.
- It can be used with general arrays, e-g. taking the average of an array of matrices.
- It is not really faster than writing a loop, but it works in one line!.

```
str(apply)
```

```
## function (X, MARGIN, FUN, ...)
```

- **x** is an array.
- **MARGIN** is an integer vector indicating which margin should be "retained".
- **FUN** is a funciton to be applied.
- ... is for other arguments to be passed to **FUN**

```
x <- matrix(rnorm(200), 20, 10)

# This takes the mean across all the rows in each column, limiting the rows from the array
apply(x, 2, mean)
```

```
##  [1]  0.02169154  0.13922249 -0.20375618  0.12052383  0.08737966 -0.23048191
##  [7]  0.10621195 -0.40861231 -0.23471852 -0.14353233
```

```
# In this case it takes a vector of 20 rows, because there's 20 rows, and inside each and for each row,
apply(x, 1, sum)
```

```
##  [1] -1.9737617  2.1521392 -2.9409619 -1.8129529  0.2958432 -6.8008543
##  [7] -0.9449133  2.7104431  0.1753749  1.7126719  2.6576376  1.0932647
## [13]  4.5834426 -7.3748885  0.6499071 -1.1293584 -0.2614908 -6.1999085
## [19]  0.4564260 -1.9694954
```

### Col/row sums and means

For sums and means of matrix dimensions, we have some shortcuts.

- **rowSums= apply(x, 1, sum)**
- **rowMeans = apply(x, 1, mean)**
- **colSums = apply(x, 2, sum)**
- **colMeans = apply(x, 2, mean)**

The shortcut functions are *much* faster, but you won't notice you're using a large matrix. ### Other ways to apply

Quantiles of the rows of a matrix

```
# This funciton goes through each row of the matrix and calculates the twenty-fifth,
# and the seventy-fifth percentile of that row.


x <- matrix(rnorm(200), 20, 10)
apply(x, 1, quantile, probs = c(0.25, 0.75))
```

```
##           [,1]       [,2]       [,3]       [,4]        [,5]       [,6]
## 25% -0.5009909 -0.1331143 -0.6553949 -0.7751224 -0.79754548 -0.4978180
## 75%  0.4374800  0.6822177  0.5520384  1.1935064  0.09027135  0.9980355
##           [,7]         [,8]       [,9]      [,10]      [,11]      [,12]
## 25% -0.1504221 -1.663072798 -0.8001892 -0.3258872 -0.2553195 -0.5257433
## 75%  0.7893039 -0.008103091  0.9727034  0.6261530  1.0301507  0.8350963
##          [,13]      [,14]      [,15]      [,16]      [,17]      [,18]
## 25% -0.3042640 -0.4143477 0.01252546 -0.1583507 -0.2373569 -0.6036866
## 75%  0.7867219  0.8177394 1.14049719  0.5557315  1.2456791  0.8464726
##          [,19]        [,20]
## 25% -0.9851387 -1.191476016
## 75%  0.4638109 -0.008519891
```

Average matrix in an array

```
a <- array(rnorm(2*2*10), c(2,2,10))
apply(a, c(1,2), mean)
```

```
##             [,1]        [,2]
## [1,] -0.3533975  0.07690822
## [2,]  0.1369292 -0.16706545
```

```
rowMeans(a, dims = 2)
```

```
##             [,1]        [,2]
## [1,] -0.3533975  0.07690822
## [2,]  0.1369292 -0.16706545
```

### Mapply

**mapply** is a multivariate apply of sorts which applies a funciton unparallel over a set of arguments.

- **FUN** is a function to apply.
- **...** contains arguments to apply over.
- **MoreArgs** is a list of other arguments to **FUN**
- **SIMPLIFY** indicates whether the result should be simplified.

The following is tedious to type: **list(rep(1,4), rep(2,3), rep(3,2), rep(4,1))**

Instead we can do

```
mapply(rep, 1:4, 4:1)
```

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

```
noise <- function(n, mean, sd){
  rnorm(n, mean, sd)
}
noise(5,1,2)
```

### Vectorizing a function

```
## [1] -0.04194341 -3.07209494 -0.48814177 -0.95531379  1.18090985
```

```
noise(1:5, 1:5, 2)
```

```
## [1] 3.359213 4.469374 1.073062 5.039904 5.768158
```

```
mapply(noise, 1:5, 1:5, 2)
```

### Instant Vectorization

```
## [[1]]
## [1] 2.822324
##
## [[2]]
## [1] -0.7710859  1.1120549
##
## [[3]]
## [1] 2.944346 1.633635 7.178824
##
## [[4]]
## [1] 6.217689 3.474943 4.300103 2.984424
##
## [[5]]
## [1] 4.827398 3.224944 5.437558 3.544052 8.458756
```

Which is the same as

```
list(noise(1,1,2), noise(2,2,2), noise(3,3,2), noise(4,4,2), noise(5,5,2))
```

```
## [[1]]
## [1] -1.12934
##
## [[2]]
## [1] 3.9458732 0.9251276
##
## [[3]]
## [1] 2.316497 5.559849 2.223582
##
## [[4]]
## [1] 1.967756 5.179325 4.243808 8.108317
##
## [[5]]
## [1] 6.186422 6.242714 7.863547 5.687742 9.645751
```

### Tapply

**tapply** is used to apply a functon over subsets of a vector.

```
str(tapply)
```

```
## function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

- **x** is a vector
- **INDEX** is a factor or a list of factors (or else they are coerced to factors)
- **FUN** is a function to be applied
- **...** contains other arguments to be passed **FUN**
- **simplify**, should we simplify the result?

Takes groups means

```
x <- c(rnorm(10), runif(10), rnorm(10))
f <- gl(3,10)
f
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
## Levels: 1 2 3
```

```
tapply(x,f,mean)
```

```
##          1          2          3
## -0.3635288  0.2959831  0.2184690
```

Take group means without simplification

```
tapply(x, f, mean, simplify = FALSE)
```

```
## $`1`
## [1] -0.3635288
##
## $`2`
## [1] 0.2959831
##
## $`3`
## [1] 0.218469
```

Find group ranges

```
tapply(x, f, range)
```

```
## $`1`
## [1] -2.356069  1.539507
##
## $`2`
## [1] 0.04127045 0.63638000
##
## $`3`
## [1] -1.939503  2.281755
```

### Split

**split** takes a vector or other objects and split it into groups determined by a factor or list of factors.

```
str(split)
```

```
## function (x, f, drop = FALSE, ...)
```

- **x** is a vector (or list) or data frame.
- **f** is a factor (or coerced to one) or a list of factors.
- **drop** indicates whether empty factors levels should be dropped.

```
x <- c(rnorm(10), runif(10), rnorm(10,1))
f <- gl(3,10)
split(x,f)
```

```
## $‘1‘
##  [1]  0.22521386  0.66500318  0.51480494 -0.48627804 -0.10245916 -0.02565023
##  [7] -1.41400170  0.40328725  1.14470190 -1.83800752
##
## $‘2‘
##  [1] 0.42562108 0.58772288 0.42449865 0.59034199 0.91682932 0.57025422
##  [7] 0.05636467 0.33936407 0.80848095 0.77875423
##
## $‘3‘
##  [1] -0.42832137  1.36754462  0.94008956 -0.05622289  1.49061023  0.83750712
##  [7]  1.54706688  0.37831867  0.34085195  1.02636502
```

A common idiom is **split** followed by an **lapply**

```
lapply(split(x,f), mean)
```

```
## $‘1‘
## [1] -0.09133855
##
## $‘2‘
## [1] 0.5498232
##
## $‘3‘
## [1] 0.744381
```

```


library(datasets)
head(airquality)
```

### Splitting a Data Frame

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

```
s <- split(airquality, airquality$Month)
lapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
```

```
## $‘5‘
##    Ozone  Solar.R     Wind
##       NA       NA 11.62258
```

```
##
## $‘6‘
##     Ozone   Solar.R      Wind
##        NA 190.16667  10.26667
##
## $‘7‘
##      Ozone    Solar.R       Wind
##         NA 216.483871   8.941935
##
## $‘8‘
##    Ozone  Solar.R     Wind
##       NA       NA 8.793548
##
## $‘9‘
##    Ozone  Solar.R     Wind
##       NA 167.4333  10.1800
```

```
sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
```

```
##                5         6          7        8        9
## Ozone         NA        NA         NA       NA       NA
## Solar.R       NA 190.16667 216.483871       NA 167.4333
## Wind    11.62258  10.26667   8.941935 8.793548  10.1800
```

```
sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")], na.rm = TRUE))
```

```
##                 5         6          7          8         9
## Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
## Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
## Wind     11.62258  10.26667   8.941935   8.793548  10.18000
```

```
x <- rnorm(10)
f1 <- gl(2,5)
f2 <- gl(5,2)
f1
```

### Siplitting on More than One Level

```
##  [1] 1 1 1 1 1 2 2 2 2 2
## Levels: 1 2
```

```
f2
```

```
##  [1] 1 1 2 2 3 3 4 4 5 5
## Levels: 1 2 3 4 5
```

```
interaction(f1, f2)
```

```
##  [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
## Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 2.4 1.5 2.5
```

Interactions can create empty levels.

```
str(split(x, list(f1, f2)))
```

```
## List of 10
##  $ 1.1: num [1:2] -1.58 1.25
##  $ 2.1: num(0)
##  $ 1.2: num [1:2] -0.41154 0.00517
##  $ 2.2: num(0)
##  $ 1.3: num 0.424
##  $ 2.3: num 1.91
##  $ 1.4: num(0)
##  $ 2.4: num [1:2] -0.0211 1.6837
##  $ 1.5: num(0)
##  $ 2.5: num [1:2] -0.631 -0.385
```

Empty levels can be dropped

```
str(split(x, list(f1, f2), drop = TRUE))
```

```
## List of 6
##  $ 1.1: num [1:2] -1.58 1.25
##  $ 1.2: num [1:2] -0.41154 0.00517
##  $ 1.3: num 0.424
##  $ 2.3: num 1.91
##  $ 2.4: num [1:2] -0.0211 1.6837
##  $ 2.5: num [1:2] -0.631 -0.385
```

## Debugging

### Something's Wrong

Indications that something's not right:

- **Message**: A generic notification/diagnostic message produced by the **message** funciton; execution of the function continues.
- **Warning**: An indication that something is wrong but not necessarily fatal; execution of the function continues; generated by the **warning** funciton.
- **error**: An indication that a fatal problem has occurred; execution stops; produced by the **stop** funciton.
- **condition**: A generic concept for indicating that something unexpected can occur; programmers can create their own conditions.

### How do you know that something is wrong with your function?

- What was your input? How did you call the function?
- What were you expecting? Output, messages, other results?
- What did you get?
- How does what you get differ from what you were expecting?

- Were your expectations correct in the first place?
- Can you reproduce the problem (exactly)?

**The primary tools for debugging functions in R are**

- **traceback**: prints out the function call stack after an error occurs; does nothing if there's no error.
- **debug**: flags a function for "debug" mode which allows you to step through execution of a function one line at a time.
- **browser**: Suspends the execution of a function wherever it is called and puts the function in debug mode.
- **trace**: allows you to insert debugging code into a function a specific places.
- **recover**: allows you to modify the error behavior so that you can crows the funciton call stack.

These are interactive tools specifically designed to allow you to pick through a function. There's also the more blunt technique of inserting print/call statements in the function.

### Summary

- There are main indications of a problem/condition: **massage, warning, error**
- Only an **error** is fatal.
- When analyzing a function with a problem, make sure you can reproduce the problem, clearly state your expectations and how the output differs from your expecation.
- Interactive debugging tools **traceback, debug, crowser, trace** and **recover** can be use yo find problematics code in functions.
- Debugging tools are not a substitute for thinking.