R : Loop Functions

May 26, 2016

1 lapply

- **lapply** returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.
- Usage:
  $lapply(X, FUN, ...)$
  where X is a vector and FUN is the function to be applied to X.
- Ex:

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

returns

```
$a
[1] 3
$b
[1] -0.06275585
```

and

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

returns

```
$a
[1] 5.91608
$b
[1] 0.8135231
$c
[1] 1.220144
$d
[1] 0.8248216
```

- **runif** is a command that generates random deviates.
  Usage: $runif(n, min = 0, max = 1)$.

- Here is another example using *lapply* and *runif*:

```
> test <- 1:4
> lapply(test, runif)
```

returns

```
[[1]]
[1] 0.4929043
[[2]]
[1] 0.3779104 0.4434126
[[3]]
[1] 0.5540859 0.6271506 0.5331803
[[4]]
[1] 0.6045924 0.7118674 0.7373680 0.5829549
```

- **Anonymous functions** are used when it isn't worth actually naming the function. For ex:

```
lapply(mtcars, function(x) length(unique(x)))
Filter(function(x) !is.numeric(x), mtcars)
integrate(function(x) sin(x) ^ 2, 0, pi)
```

- If we create some matrices, we can write some anonymous functions to pull the first column or row using *lapply*.

```
> xMatrices <- list(a = matrix(2:7,3,2), b = matrix(1:6,2,3))
> xMatrices
$a
     [,1] [,2]
[1,]    2    5
[2,]    3    6
[3,]    4    7

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

Then *lapply* using (to pull first row):

```
> lapply(xMatrices, function(anon) anon[1,])
$a
[1] 2 5
$b
[1] 1 3 5
```

Or to pull the first column:

```
> lapply(xMatrices, function(anon) anon[,1])
$a
[1] 2 3 4
$b
[1] 1 2
```

2 sapply

- **sapply** will try to simplify the result of *lapply* , if possible.
- Usage:
  $sapply(X, FUN, ...)$
  where X is a vector and FUN is the function to be applied to X.
  * If the result is a list where each element has length 1, a vector is returned.
  * If the result is a list where every element is a vector, a matrix is returned.
  * If it can't figure out what to do, a list is returned.
- For example:

```
> testVec <- list(a = 1:5, b = rnorm(10),
c = rnorm(20,1), d = rnorm(100,mean=0,sd=1))
> sapply(testVec, sd)
```

returns

```
       a        b        c        d
1.581139 1.333568 1.101684 1.014829
```

3 apply

- **apply** will evaluate a function (often an anonymous one) over the dimensions of an array or matrix.
  * Often used to apply a function to rows or columns of a matrix.
  * It can be used with general arrays – ex: taking the mean of an array of matrices.
  * It may not be faster than writing a loop – but is much simpler!
- Usage:
  $apply(X, MARGIN, FUN, ...)$
  where X is an array and FUN is the function to be applied to X. $MARGIN$ is an integer vector indicating which margin to 'retain.'
- For example, we can find operations on rows and columns of a matrix. The names explain the function.

```
> matrixA <- matrix(rnorm(50),10,5)
> matrixA
            [,1]        [,2]        [,3]       [,4]       [,5]
 [1,]  0.6614258  1.56365973  0.76967625  0.2289903 -1.1830024
 [2,] -0.8572734  0.30399205  0.06186408 -0.7208525 -0.5110569
 [3,] -0.1346467  0.73066714  1.08114358  0.3334223 -0.5029956
 [4,] -0.1360720 -1.70059026 -0.93222551  0.4135450  0.9184752
 [5,] -0.2783048 -0.65104513  0.61224467  0.5213020  0.8180062
 [6,] -1.7026745  0.00533423  0.65958393 -0.7841496  0.5819655
 [7,] -0.6877963 -1.18392575 -0.19740452 -0.8816024 -0.5274435
 [8,] -1.4668603  0.15200376  0.38061756  0.1605821 -0.2141954
 [9,] -1.6184856 -0.71234306  0.71605579 -1.1336989 -1.0638763
[10,]  0.3163083  1.73161583 -0.74194732  0.4277605  0.4367956
```

Row Sums and Means:

```
> rowSums <- apply(matrixA, 1, sum)
> rowSums
 [1]  2.0407496 -1.7233266  1.5075907 -1.4368676  1.0222030 -1.2399403
 [7] -3.4781725 -0.9878524 -3.8123480  2.1705329
> rowMeans <- apply(matrixA, 1, mean)
> rowMeans
 [1]  0.4081499 -0.3446653  0.3015181 -0.2873735  0.2044406 -0.2479881
 [7] -0.6956345 -0.1975705 -0.7624696  0.4341066
```

Column Sums and Means:

```
> colSums <- apply(matrixA, 2, sum)
> colSums
[1] -5.9043795  0.2393686  2.4096085 -1.4347012 -1.2473276
> colMeans <- apply(matrixA, 2, mean)
> colMeans
[1] -0.59043795  0.02393686  0.24096085 -0.14347012 -0.12473276
```

- Here is another way to use *apply*. This calculates the quantiles of the rows of a matrix. This uses *matrixA* again.

```
> apply(matrixA, 1, quantile, probs = c(.25,.75))
```

returns

```
         [,1]        [,2]       [,3]       [,4]       [,5]       [,6]
25% 0.2289903 -0.72085245 -0.1346467 -0.9322255 -0.2783048 -0.7841496
75% 0.7696763  0.06186408  0.7306671  0.4135450  0.6122447  0.5819655
          [,7]       [,8]       [,9]     [,10]
25% -0.8816024 -0.2141954 -1.1336989 0.3163083
75% -0.5274435  0.1605821 -0.7123431 0.4367956
```

A column-oriented version of this is:

```
> apply(matrixA, 2, quantile, probs = c(.25,.75))
```

returns

```
          [,1]       [,2]       [,3]       [,4]       [,5]
25% -1.3144636 -0.6970186 -0.1325874 -0.7683253 -0.5233469
75% -0.1350031  0.6239984  0.7019378  0.3935143  0.5456730
```

- *apply* can also be used to look at a whole array of objects. For example, this array of matrices.

```
> mArray2 <- array(rnorm(2*2*10), c(2,2,3))
> mArray2
, , 1
         [,1]     [,2]
[1,] 1.451297 1.236377
[2,] 1.086570 1.463750

, , 2
          [,1]       [,2]
[1,] -2.375930  0.2472643
[2,] -1.255108 -0.8155285

, , 3
           [,1]      [,2]
[1,] -0.4692694 0.1939882
[2,] -1.6442363 1.4534122
```

Then we may use *apply* to operate on this matrix array. This function finds the mean across each row and column of the array.

```
> apply(mArray2, c(1,2), mean)
           [,1]      [,2]
[1,] -0.4646341 0.5592098
[2,] -0.6042579 0.7005447
```

4 mapply

- **mapply** is a multivariate apply of sorts which applies a function in parallel over a set of arguments.
- Usage:
  $mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)$
  where
  * FUN is a function to apply
  * "..." contains arguments to apply over
  * $MoreArgs$ is a list of other arguments for FUN
  * and $SIMPLIFY$ indicates whether the result should be simplified.
- Ex: Instead of typing

  ```
  > list(rep(1,4), rep(2,3), rep(3,2), rep(4,1))
  ```

  We can use *mapply* like this:

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

  which returns

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

  [[2]]
  [1] 2 2 2

  [[3]]
  [1] 3 3

  [[4]]
  [1] 4
  ```

- Here is another example. Here's a function that makes a little noise:

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

  We can vectorize this function by using *mapply*:

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

  to get some vectors

  ```
  [[1]]
  [1] 2.331235

  [[2]]
  [1] -0.0007545863  3.3751239750

  [[3]]
  [1]  5.2463796 -0.4334525  4.8754105

  [[4]]
  [1] 4.600540 1.675570 2.210837 6.511701

  [[5]]
  [1] 7.730601 6.199313 1.684892 4.162158 4.675275
  ```

5 tapply

- **tapply** is used to apply a function over subsets of a vector.
- Usage:
  $tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)$
  where
  * X is a vector
  * $INDEX$ is a factor or list of factors
  * "..." contains other arguments to be passed to FUN
  * $simplify$ can be $TRUE$ or $FALSE$.
- Ex:

```
> x <- c(rnorm(10), runif(10), rnorm(10,1))
> x
 [1]  1.88028510 -0.20865279  0.78897517 -1.14355406  0.54381818 -1.19289028
 [7] -1.23076189  0.09285183 -0.17325715  1.54863086  0.75253856  0.93077604
[13]  0.84016727  0.91670794  0.50379103  0.94338293  0.01547062  0.26189705
[19]  0.63686966  0.89848117  1.13802204  0.21009540  0.30497234  1.42228040
[25] -0.05485161  1.87818212 -0.27187853  0.42479587  0.90921599  2.19011112
```

We can use gl to apply factor levels. Here, $gl(3, 10)$ creates three factors - each with 10 elements.

```
> 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
```

We may use *tapply* to apply these factors to the set x, and find the mean.

```
> tapply(x, f, mean)
        1         2         3
0.0905445 0.6700082 0.8150945
```

We may also use *tapply* to find group ranges for x:

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

which returns

```
$`1`
[1] -1.230762  1.880285

$`2`
[1] 0.01547062 0.94338293

$`3`
[1] -0.2718785  2.1901111
```