



Introduction to the R Language

Loop Functions

Roger D. Peng, Associate Professor of Biostatistics
Johns Hopkins Bloomberg School of Public Health

Looping on the Command Line

Writing `for`, `while` loops is useful when programming but not particularly easy when 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 subsets of a vector
- `mapply`: Multivariate version of `lapply`

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

lapply

`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: 0x7ff7a1951c00>
## <environment: namespace:base>
```

The actual looping is done internally in C code.

lapply

`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] 0.4671
```

lapply

```
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] 0.5261  
##  
## $c  
## [1] 1.421  
##  
## $d  
## [1] 4.927
```

lapply

```
> x <- 1:4
> lapply(x, runif)
[[1]]
[1] 0.2675082

[[2]]
[1] 0.2186453 0.5167968

[[3]]
[1] 0.2689506 0.1811683 0.5185761

[[4]]
[1] 0.5627829 0.1291569 0.2563676 0.7179353
```

lapply

```
> x <- 1:4
> lapply(x, runif, min = 0, max = 10)
[[1]]
[1] 3.302142

[[2]]
[1] 6.848960 7.195282

[[3]]
[1] 3.5031416 0.8465707 9.7421014

[[4]]
[1] 1.195114 3.594027 2.930794 2.766946
```

lapply

`lapply` and friends make heavy 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
```


lapply

An anonymous function 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

sapply

```
> 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] 0.06082667

$c
[1] 1.467083

$d
[1] 5.074749
```

sapply

```
> sapply(x, mean)
      a      b      c      d
2.50000000 0.06082667 1.46708277 5.07474950

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



Introduction to the R Language

Loop Functions - apply

Roger Peng, Associate Professor
Johns Hopkins Bloomberg School of Public Health

apply

`apply` is used to evaluate a function (often an anonymous one) over the margins of an array.

- It is most often used to apply a 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!

apply

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

- **X** is an array
- **MARGIN** is an integer vector indicating which margins should be “retained”.
- **FUN** is a function to be applied
- ... is for other arguments to be passed to **FUN**

apply

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)
 [1]  0.04868268  0.35743615 -0.09104379
 [4] -0.05381370 -0.16552070 -0.18192493
 [7]  0.10285727  0.36519270  0.14898850
[10]  0.26767260

> apply(x, 1, sum)
 [1] -1.94843314  2.60601195  1.51772391
 [4] -2.80386816  3.73728682 -1.69371360
 [7]  0.02359932  3.91874808 -2.39902859
[10]  0.48685925 -1.77576824 -3.34016277
[13]  4.04101009  0.46515429  1.83687755
[16]  4.36744690  2.21993789  2.60983764
[19] -1.48607630  3.58709251
```


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 unless you're using a large matrix.

Other Ways to Apply

Quantiles of the rows of a matrix.

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 1, quantile, probs = c(0.25, 0.75))
      [,1]      [,2]      [,3]      [,4]
25% -0.3304284 -0.99812467 -0.9186279 -0.49711686
75%  0.9258157  0.07065724  0.3050407 -0.06585436
      [,5]      [,6]      [,7]      [,8]
25% -0.05999553 -0.6588380 -0.653250  0.01749997
75%  0.52928743  0.3727449  1.255089  0.72318419
      [,9]      [,10]      [,11]      [,12]
25% -1.2467955 -0.8378429 -1.0488430 -0.7054902
75%  0.3352377  0.7297176  0.3113434  0.4581150
      [,13]      [,14]      [,15]      [,16]
25% -0.1895108 -0.5729407 -0.5968578 -0.9517069
75%  0.5326299  0.5064267  0.4933852  0.8868922
      [,17]      [,18]      [,19]      [,20]
```

apply

Average matrix in an array

```
> a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
> apply(a, c(1, 2), mean)
      [,1]      [,2]
[1,] -0.2353245 -0.03980211
[2,] -0.3339748  0.04364908

> rowMeans(a, dims = 2)
      [,1]      [,2]
[1,] -0.2353245 -0.03980211
[2,] -0.3339748  0.04364908
```



Introduction to the R Language

Loop Functions - tapply

Roger Peng, Associate Professor
Johns Hopkins Bloomberg School of Public Health

tapply

tapply is used to apply a function over subsets of a vector. I don't know why it's called tapply.

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., 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?

tapply

Take group means.

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> 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
[24] 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)
      1      2      3
0.1144464 0.5163468 1.2463678
```

tapply

Take group means without simplification.

```
> tapply(x, f, mean, simplify = FALSE)
$'1'
[1] 0.1144464

$'2'
[1] 0.5163468

$'3'
[1] 1.246368
```

tapply

Find group ranges.

```
> tapply(x, f, range)
$'1'
[1] -1.097309  2.694970

$'2'
[1] 0.09479023 0.79107293

$'3'
[1] 0.4717443 2.5887025
```




Introduction to the R Language

Loop Functions - mapply

Roger Peng, Associate Professor
Johns Hopkins Bloomberg School of Public Health

mapply

`mapply` is a multivariate apply of sorts which applies a function in parallel over a set of arguments.

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,
         USE.NAMES = TRUE)
```

- `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

mapply

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
```

Vectorizing a Function

```
> noise <- function(n, mean, sd) {  
+ rnorm(n, mean, sd)  
+ }  
> noise(5, 1, 2)  
[1] 2.4831198 2.4790100 0.4855190 -1.2117759  
[5] -0.2743532  
  
> noise(1:5, 1:5, 2)  
[1] -4.2128648 -0.3989266 4.2507057 1.1572738  
[5] 3.7413584
```

Instant Vectorization

```
> mapply(noise, 1:5, 1:5, 2)
[[1]]
[1] 1.037658

[[2]]
[1] 0.7113482 2.7555797

[[3]]
[1] 2.769527 1.643568 4.597882

[[4]]
[1] 4.476741 5.658653 3.962813 1.204284

[[5]]
[1] 4.797123 6.314616 4.969892 6.530432 6.723254
```

Instant Vectorization

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))
```

split

`split` takes a vector or other objects and splits 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

split

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> split(x, f)
$'1'
 [1] -0.8493038 -0.5699717 -0.8385255 -0.8842019
 [5]  0.2849881  0.9383361 -1.0973089  2.6949703
 [9]  1.5976789 -0.1321970

$'2'
 [1] 0.09479023 0.79107293 0.45857419 0.74849293
 [5] 0.34936491 0.35842084 0.78541705 0.57732081
 [9] 0.46817559 0.53183823

$'3'
 [1] 0.6795651 0.9293171 1.0318103 0.4717443
 [5] 2.5887025 1.5975774 1.3246333 1.4372701
```


split

A common idiom is `split` followed by an `lapply`.

```
> lapply(split(x, f), mean)
$'1'
[1] 0.1144464

$'2'
[1] 0.5163468

$'3'
[1] 1.246368
```

Splitting a Data Frame

```
> library(datasets)
> head(airquality)
  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
```

Splitting a Data Frame

```
> 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 |

Splitting a Data Frame

```
> 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 |

Splitting on More than One Level

```
> x <- rnorm(10)
> f1 <- gl(2, 5)
> f2 <- gl(5, 2)
> f1
 [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
10 Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 ... 2.5
```

Splitting on More than One Level

Interactions can create empty levels.

```
> str(split(x, list(f1, f2)))  
List of 10  
 $ 1.1: num [1:2] -0.378  0.445  
 $ 2.1: num(0)  
 $ 1.2: num [1:2] 1.4066 0.0166  
 $ 2.2: num(0)  
 $ 1.3: num -0.355  
 $ 2.3: num 0.315  
 $ 1.4: num(0)  
 $ 2.4: num [1:2] -0.907  0.723  
 $ 1.5: num(0)  
 $ 2.5: num [1:2] 0.732 0.360
```

split

Empty levels can be dropped.

```
> str(split(x, list(f1, f2), drop = TRUE))  
List of 6  
 $ 1.1: num [1:2] -0.378  0.445  
 $ 1.2: num [1:2]  1.4066 0.0166  
 $ 1.3: num -0.355  
 $ 2.3: num  0.315  
 $ 2.4: num [1:2] -0.907  0.723  
 $ 2.5: num [1:2]  0.732  0.360
```