Looping functions

lapply()

The lapply() function does the following simple series of operations:

1. it loops over a list, iterating over each element in that list
2. it applies a function to each element of the list (a function that you specify)
3. and returns a list (the l is for "list").

This function 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()**.

Note that the actual looping is done internally in C code for efficiency reasons.

It's important to remember that lapply() always returns a list, regardless of the class of the input.

ex1

```
x <- list(a = 1:5, b = rnorm(100))
x$a
```

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

```
x$b
```

```
##   [1] -0.02698945 -0.57339220  1.94382345 -1.37368865  0.48210678  0.01775926
##   [7]  0.51133953 -2.67242820 -1.48568088  1.07755552  1.08405428 -0.27269128
##  [13] -1.31973606  0.59680322 -0.78942681 -1.97511177  0.63118125  0.98263817
##  [19]  1.02625138 -0.12142457 -0.77913229  0.73620550  0.38949008  0.52044911
##  [25]  0.19842096  0.85912722  0.64649988 -2.66523841  0.62715490 -1.91020006
##  [31] -0.63477252 -0.76590938 -0.87792806 -0.45011534  2.13813576  0.93331380
##  [37] -1.79543686 -0.13219375  1.68297108  1.06199540  1.05507009 -0.33639435
##  [43]  0.39993814 -0.87170204 -0.69054320  0.50158073 -0.30683650  1.09462625
##  [49] -0.91334402 -0.59469899  1.23608606  1.21684554  0.15100312 -0.89603952
##  [55]  0.44103197 -0.12774429 -2.22849748  1.19367943  0.02408211 -0.48895322
##  [61] -0.19333686  0.88458505 -0.84766923 -0.91856175  0.75636613  0.65764465
##  [67] -0.49802987  0.46805596  0.03450439  1.07495327 -0.56798462  0.52788813
##  [73] -0.37453395  1.54188141  0.15823403 -0.23064537 -0.01995195 -1.98484170
##  [79]  0.48452407 -0.31304488  0.54555759 -1.19696695 -1.24470667  0.76319616
##  [85]  1.64358799 -0.10749037  0.26586339  1.03795502 -0.79367618 -1.71624002
##  [91]  0.17204156 -0.34683434 -1.27865630  1.36252339  0.57450191  0.01724134
##  [97]  0.44569398  1.35889016 -0.86641928 -0.32995874
```

```
lapply(x, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] -0.0166889
```

ex2

let‘s try to understand how we pass arguments

```
?runif
?lapply
```

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

```
## [[1]]
## [1] 0.5805881
##
## [[2]]
## [1] 0.87024803 0.07813333
##
## [[3]]
## [1] 0.5544263 0.7139209 0.6339550
##
## [[4]]
## [1] 0.24724254 0.22536921 0.27846659 0.04209132
```

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

```
## [[1]]
## [1] 0.7167535
##
## [[2]]
## [1] 7.296165 3.953218
##
## [[3]]
## [1] 8.455292 6.979090 2.159895
##
## [[4]]
## [1] 2.041108 3.938083 6.013747 0.540893
```

The lapply() function and its 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(x, function(mat) { mat[,1] })
```

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

You can put an arbitrary complicated function definition inside lapply(), but if it's going to be more complicated, it' sprobably a better idea to define the function separately.

Another rule is if you want to use fuction many times define it separately if you want it only once for this particular situation use anonimous function.

sapply()

The sapply() function behaves similarly to lapply(); the only real difference is in the return value. sapply() will try to simplify the result of lapply() if possible. Essentially, sapply() calls lapply() on its input and then applies the following algorithm:

- 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] 0.1244324
##
## $c
## [1] 1.355702
##
## $d
## [1] 5.046712
```

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

```
##         a         b         c         d
## 2.5000000 0.1244324 1.3557018 5.0467117
```

split()

It is not a looping function but is very handy when use in conjunction with looping functions.

```
str(split)
```

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

- x is a vector, 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

ex 1

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

```
## $`1`
##  [1]  0.39044480 -0.92935117 -0.99020340 -1.73697971  0.82439163  0.02781468
##  [7]  0.03743138 -0.36269113  0.16799447 -0.17099355
##
## $`2`
##  [1] 0.35476032 0.47356453 0.28270430 0.03155839 0.13929673 0.10737129
##  [7] 0.14578917 0.57086144 0.09251920 0.22945200
## 
## $`3`
##  [1]  1.09189917  1.76382000  1.54375558  1.21673091  1.52757613  0.64250093
##  [7]  1.63759403  0.25803892 -0.09318403  1.25970008
```

```r
sapply(split(x, f), mean)
```

```
##          1          2          3 
## -0.2742142  0.2427877  1.0848432
```

ex 2 splitting a data frame

```r
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
```

```r
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
```

```r
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
```

```r
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
```

ex 3 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
## Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 2.4 1.5 2.5
```

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

```
## List of 10
##  $ 1.1: num [1:2] 1.376 0.222
##  $ 2.1: num(0)
##  $ 1.2: num [1:2] 0.402 0.212
##  $ 2.2: num(0)
##  $ 1.3: num 1.17
##  $ 2.3: num 0.828
##  $ 1.4: num(0)
##  $ 2.4: num [1:2] -0.165 1.286
##  $ 1.5: num(0)
##  $ 2.5: num [1:2] 0.158 -0.189
```

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

```
## List of 6
##  $ 1.1: num [1:2] 1.376 0.222
##  $ 1.2: num [1:2] 0.402 0.212
##  $ 1.3: num 1.17
##  $ 2.3: num 0.828
##  $ 2.4: num [1:2] -0.165 1.286
##  $ 2.5: num [1:2] 0.158 -0.189
```

```
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
```

```
s <- split(airquality, list(airquality$Month, airquality$Day))
```

```
sapply(s, function(x) if (length(x[,'Temp']) != 0) x[,'Temp'] else 0)
```

```
##  5.1  6.1  7.1  8.1  9.1  5.2  6.2  7.2  8.2  9.2  5.3  6.3  7.3  8.3  9.3  5.4
##   67   78   84   81   91   72   74   85   81   92   74   67   81   82   93   62
##  6.4  7.4  8.4  9.4  5.5  6.5  7.5  8.5  9.5  5.6  6.6  7.6  8.6  9.6  5.7  6.7
##   84   84   86   93   56   85   83   85   87   66   79   83   87   84   65   82
##  7.7  8.7  9.7  5.8  6.8  7.8  8.8  9.8  5.9  6.9  7.9  8.9  9.9 5.10 6.10 7.10
##   88   89   80   59   87   92   90   78   61   90   92   90   75   69   87   89
## 8.10 9.10 5.11 6.11 7.11 8.11 9.11 5.12 6.12 7.12 8.12 9.12 5.13 6.13 7.13 8.13
##   92   73   74   93   82   86   81   69   92   73   86   76   66   82   81   82
## 9.13 5.14 6.14 7.14 8.14 9.14 5.15 6.15 7.15 8.15 9.15 5.16 6.16 7.16 8.16 9.16
##   77   68   80   91   80   71   58   79   80   79   71   64   77   81   77   78
## 5.17 6.17 7.17 8.17 9.17 5.18 6.18 7.18 8.18 9.18 5.19 6.19 7.19 8.19 9.19 5.20
##   66   72   82   79   67   57   65   84   76   76   68   73   87   78   68   62
## 6.20 7.20 8.20 9.20 5.21 6.21 7.21 8.21 9.21 5.22 6.22 7.22 8.22 9.22 5.23 6.23
##   76   85   78   82   59   77   74   77   64   73   76   81   72   71   61   76
## 7.23 8.23 9.23 5.24 6.24 7.24 8.24 9.24 5.25 6.25 7.25 8.25 9.25 5.26 6.26 7.26
##   82   75   81   61   76   86   79   69   57   75   85   81   63   58   78   82
## 8.26 9.26 5.27 6.27 7.27 8.27 9.27 5.28 6.28 7.28 8.28 9.28 5.29 6.29 7.29 8.29
##   86   70   57   73   86   88   77   67   80   88   97   75   81   77   86   94
## 9.29 5.30 6.30 7.30 8.30 9.30 5.31 6.31 7.31 8.31 9.31
##   76   79   83   83   96   68   76    0   81   94    0
```

tapply()

tapply() is used to apply a function over subsets of a vector. It can be thought of as a combination of split() and sapply() for **vectors only**.

```
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?

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

tapply(x, f, mean) # output simplified vector
```

```
##          1          2          3
## -0.2790954  0.3851786  0.7437352
```

```
tapply(x, f, range) # output list
```

```
## $`1`
## [1] -1.637045  1.492424
##
## $`2`
## [1] 0.0968159 0.7997152
##
## $`3`
## [1] -1.237046  3.166571
```

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

```
## $`1`
## [1] -0.2790954
##
## $`2`
## [1] 0.3851786
##
## $`3`
## [1] 0.7437352
```

apply()

The apply() function is used to a 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 (which is just a 2-dimensional array).

However, it can be used with general arrays, for example, to take the average of an array of matrices. Using apply() is not really faster than writing a loop, but it works in one line and is highly compact.

```
str(apply)
```

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

- 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

```
x <- matrix(rnorm(200), 20, 10)
apply(x, 2, mean)  ## Take the mean of each column
```

```
##  [1]  0.2540847 -0.1000687  0.3784536  0.4851913  0.3156337 -0.3179366
##  [7]  0.1784956  0.2293983  0.4731950 -0.2859670
```

```
apply(x, 1, sum)   ## Take the mean of each row
```

```
##  [1]  8.1082282  2.0529365  2.7681565  0.5954836  2.5520442  1.2741487
##  [7]  3.0596121 -3.3380872  2.0607716  1.8985147  4.0948887 -0.5403996
## [13] -5.6974167  1.3047239  9.0082761  3.8902057  5.0738945 -2.9252717
## [19] -1.1548575 -1.8762529
```

The MARGIN argument essentially indicates to apply() which dimension of the array you want to preserve or retain.

1 is for rows

2 is for columns

For the special case of column/row sums and column/row means of matrices, we have some useful 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 heavily optimized and hence are much faster, but you probably won't notice unless you're using a large matrix. Another nice aspect of these functions is that they are a bit more descriptive. It's arguably more clear to write colMeans(x) in your code than apply(x, 2, mean).

```
a <- array(rnorm(200 * 200 * 100), c(200, 200, 100))
```

```
now <<- Sys.time()

res <- apply(a, c(1, 2), mean)

difftime(Sys.time(), now)
```

```
## Time difference of 0.2686424 secs
```

```
now <<- Sys.time()

res <- rowMeans(a, dims = 2)

difftime(Sys.time(), now)
```

```
## Time difference of 0.01687932 secs
```

mapply()

The mapply() function is a multivariate apply of sorts which applies a function in parallel over a set of arguments. Recall that lapply() and friends only iterate over a single R object. What if you want to iterate over multiple R objects in parallel? This is what mapply() is for.

```
str(mapply)
```

```
## function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

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

ex1

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

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

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

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

ex 2 Simulating random normal variables

```r
str(rnorm)
```

```
## function (n, mean = 0, sd = 1)
```

```r
rnorm(5, 1, 2)
```

```
## [1] -2.7900904  0.2738156  3.3041369  1.7500724 -2.7665916
```

```r
rnorm(1:5, 1:5, 2)
```

```
## [1] -0.8923675 -1.8135396  4.5947986  5.2458252  5.9093274
```

```r
mapply(rnorm, 1:5, 1:5, 2)
```

```
## [[1]]
## [1] -2.915993
##
## [[2]]
## [1] 1.544843 1.648675
##
## [[3]]
## [1] 2.8619114 1.4374180 0.4142283
##
## [[4]]
## [1] -0.8293758  3.4337980  3.3422173  2.5931708
##
## [[5]]
## [1] 2.134222 3.033412 4.053117 9.515634 4.282133
```

```r
# the same as:
list(rnorm(1, 1, 2), rnorm(2, 2, 2),
      rnorm(3, 3, 2), rnorm(4, 4, 2),
      rnorm(5, 5, 2))
```

```
## [[1]]
## [1] 2.914101
##
## [[2]]
## [1] 2.620947 4.167201
##
## [[3]]
## [1] 2.375809 3.387040 2.108104
##
## [[4]]
## [1] 1.477563 2.009357 2.188974 1.719010
##
## [[5]]
## [1]  1.956054  4.521252  4.088351  6.580161 10.167062
```

Function vectorization

The mapply() function can be use to automatically "vectorize" a function. What this means is that it can be used to take a function that typically only takes single arguments and create a new function that can take vector arguments. This is often needed when you want to plot functions.

$$\sum_{i=1}^{n}(x_i - \mu)^2/\sigma^2$$

```
sumsq <- function(mu, sigma, x) {
        sum(((x - mu) / sigma)^2)
}
```

```
x <- rnorm(100)
sumsq(1:10, 1:10, x)
```

```
## [1] 119.6204
```

```
mapply(sumsq, 1:10, 1:10, MoreArgs = list(x = x))
```

```
##  [1] 211.1276 132.6150 116.6436 110.5703 107.5383 105.7720 104.6352 103.8509
##  [9] 103.2813 102.8512
```

There's even a function in R called Vectorize() that automatically can create a vectorized version of your function. So we could create a vsumsq() function that is fully vectorized as follows.

```
vsumsq <- Vectorize(sumsq, c("mu", "sigma"))
vsumsq(1:10, 1:10, x)
```

```
##  [1] 211.1276 132.6150 116.6436 110.5703 107.5383 105.7720 104.6352 103.8509
##  [9] 103.2813 102.8512
```

Summary

- The loop functions in R are very powerful because they allow you to conduct a series of operations on data using a compact form
- The operation of a loop function involves iterating over an R object (e.g. a list or vector or matrix), applying a function to each element of the object, and the collating the results and returning the collated results.
- Loop functions make heavy use of anonymous functions, which exist for the life of the loop function but are not stored anywhere
- The split() function can be used to divide an R object in to subsets determined by another variable which can subsequently be looped over using loop functions.