Module 3

Loop functions

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; a list, a function (or name of a function) and other arguments via its ... argument. The ... is used to pass arguments that go with the function that will be applied to each element of the list. If the first argument is not a list, it will be coerced to a list using as.list(). If coercion is not possible, R will return an error. The actual looping is done internally in C code.

```
lapply
```

```
## function (X, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     if (!is.vector(X) || is.object(X))
##         X <- as.list(X)
##     .Internal(lapply(X, FUN))
## }
## <bytecode: 0x000001ba89bab628>
## <environment: namespace:base>
```

lapply will always return a list. An illustration is as follows:

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

```
## $a
## [1] 3
##
## $b
## [1] -0.3758086
```

The result is a list; with the mean function applied to each element of the list.

```
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] -0.04329238
##
## $c
## [1] 0.9502288
##
## $d
## [1] 5.248976
```

While each element of the list was a numeric vector, lapply will return a vector with a single number for each element of the list.

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

```
## [[1]]
## [1] 0.6833012
##
## [[2]]
## [1] 0.1519491 0.9951957
##
## [[3]]
## [1] 0.74144687 0.05796814 0.17669344
##
## [[4]]
## [1] 0.6047729 0.1071902 0.9728057 0.1233955
```

runif generates uniform random variables. The first argument to runif is the number of random variables to generate. So the result is a list of vectors, with 1,2,3,4 random numbers respectively. By default runif will generate numbers between zero and one only.

We can give additional arguments to the function by appending those arguments to the lapply function.

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

```
## [[1]]
## [1] 0.9380474
##
## [[2]]
## [1] 2.285472 4.385604
##
## [[3]]
## [1] 7.704052 6.573029 1.955804
```

```
##
## [[4]]
## [1] 0.7545832 9.2337974 3.0718191 1.6421544
```

A list with two matrices

```
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 function for extracting the first column if each matrix; defined inside lapply.

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

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

sapply

The `sapply` function will try to simplify the result of `lapply` if possible. If the result is a list where every element is of length 1, a vector is returned. If the result is a list where every element is a vector of the same length (>1) then a matrix is returned. If it cannot figure things out, a list is returned. An example is shown below

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

```
##          a          b          c          d
##  2.5000000 -0.3270942  1.0511099  5.0337304
```

apply

The `apply` function 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 also be used with general arrays. It is not much faster than a loop for arrays but does the job in one line

```
str(apply)
```

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

Here, X is an array. MARGIN is an integer vector indicating which margin is to be retained. FUN is the function to be applied and the .. .is for arguments to be passed to FUN.

```
## Create a matrix of 20 rows and 10 columns
x <- matrix(rnorm(200), 20, 10)

## Collapse the rows, retain the columns and apply mean on all rows of each column
apply(x, 2, mean)
```

```
##  [1]  0.35549379  0.12091764  0.19318919  0.03491817  0.09817818 -0.02879452
##  [7] -0.12744339  0.17681462 -0.21769831 -0.27668424
```

```
## Collapse the columns, retain the rows and apply mean on all columns of each  row
apply(x, 1, mean)
```

```
##  [1] -0.03252291 -0.61760704  0.24044621 -0.36026324  0.32082801 -0.07498860
##  [7]  0.20165787  0.28129739 -0.16946538  0.17541404  0.02088495 -0.01581479
## [13]  0.03084978  0.56837932 -0.36898915 -0.20498876  0.31591867 -0.18011331
## [19] -0.07863475  0.60549396
```

For sums and meansa on matrix dimensions, the following functions exist : rowSums, colSums, rowMeans, colMeans.

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

While they are equivalent to the apply functions, they are much faster as they have been optimized specifically for that operation. However, unless the matrix is particularly large, the speed gain will not be noticeable.

Get the quantiles (25th and 75th) of each row of a matrix

```
## Create a matrix of 20 rows and 10 columns
x <- matrix(rnorm(200), 20, 10)

apply(x, 1,quantile, probs=c(0.25,0.75))
```

```
##           [,1]       [,2]       [,3]       [,4]       [,5]      [,6]       [,7]
## 25% -0.5891224 -1.1619405 -0.4934316 -1.0791529 -0.3958315 0.1545752 -1.6558678
## 75%  1.0491513  0.4338634  0.8724790  0.3972719  0.7047646 1.3049785  0.2894552
##            [,8]       [,9]      [,10]        [,11]      [,12]      [,13]
## 25% -0.79241028 -0.4130179 -0.3144924 -0.882429520 -0.6814766 -0.4558255
## 75% -0.02604946  0.5670495  0.8916043 -0.009615689 -0.2729458  0.6663475
##          [,14]      [,15]     [,16]      [,17]       [,18]      [,19]
## 25% -0.4642692 -0.2916939 0.1469827 -0.6675432 -0.08314923 -0.9204144
## 75%  0.7075940  0.2928857 0.6149438  1.4947909  0.64093398  0.1697462
##          [,20]
## 25% -0.8289519
## 75%  0.5084833
```

Creating an array with 2 rows and 2 columns with its third dimension being 10. Imagine this to be a set of ten 2 x 2 matrices stacked together.

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

```
## , , 1
##
##            [,1]       [,2]
## [1,]  0.3782499  1.9664579
## [2,] -0.5771299 -0.4038712
##
## , , 2
##
##           [,1]       [,2]
## [1,] 0.3722707  1.1322928
## [2,] 0.2717886 -0.4971583
##
## , , 3
##
##           [,1]       [,2]
## [1,] 0.4302202 -0.5494457
## [2,] 1.3832094 -1.5095408
##
## , , 4
##
##           [,1]      [,2]
## [1,] 1.1116496 -1.414977
## [2,] 0.7394459  1.214308
##
## , , 5
##
##             [,1]      [,2]
## [1,] -1.76967256 0.3141692
## [2,] -0.09124919 0.8521758
##
## , , 6
##
##           [,1]      [,2]
## [1,] 0.8068965 -1.330010
## [2,] 0.9943955  1.220092
##
## , , 7
##
##           [,1]       [,2]
## [1,] 1.0124227  0.4218244
## [2,] 0.7843084 -1.0900454
##
## , , 8
##
##             [,1]       [,2]
## [1,]  0.14223519 -0.7165665
## [2,] -0.08806757  1.0994394
##
```

```
## , , 9
##
##             [,1]       [,2]
## [1,] -1.12514183  0.3463636
## [2,] -0.07284498 -1.3430777
##
## , , 10
##
##          [,1]       [,2]
## [1,] 1.398628 0.06798259
## [2,] 1.262913 1.50103744
```

Now, I want to find the averages of each position in the 2 x 2 matrix. So when using the apply function, we'll need to retain the first two dimensions

```
apply(a, c(1,2), mean)
```

```
##           [,1]       [,2]
## [1,] 0.2757758 0.02380911
## [2,] 0.4606769 0.10433593
```

tapply

This is used to apply functions over a subset of a vector.

```
str(tapply)
```

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

Here, X is an vector INDEX is a factor or list of factors. FUN is the function to be applied and the .. .is for arguments to be passed to FUN. simplify is a boolean argument that decides whether the result of tapply should be simplified futher.

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

```
##  [1]  0.12069207  0.35262571 -0.58775991 -0.02976840 -2.24115083 -0.50074518
##  [7] -0.70387159 -1.76496593  1.40558043  0.78195992  0.18407109  0.70999747
## [13]  0.64036491  0.20758429  0.24425145  0.38763227  0.47153775  0.61816744
## [19]  0.11123679  0.72344227  0.99639582  1.97954235 -1.30339156  1.09286706
## [25]  2.08136482  0.04134847 -0.17638168  1.24842607  0.10764591  1.52900888
```

```
factor <- gl(3,10)
factor
```

```
##  [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
```

Using the tapply function on the vector x with factor

```
tapply(x, factor, mean)
```

```
##          1          2          3
## -0.3167404  0.4298286  0.7596826
```

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

```
## $`1`
## [1] -0.3167404
##
## $`2`
## [1] 0.4298286
##
## $`3`
## [1] 0.7596826
```

```
tapply(x, factor, range)
```

```
## $`1`
## [1] -2.241151  1.405580
##
## $`2`
## [1] 0.1112368 0.7234423
##
## $`3`
## [1] -1.303392  2.081365
```

mapply

The`mapply` function is a multi-variate version of the apply function which applies a function in parallel over a set of arguments.

```
str(mapply)
```

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

Here, X is an array, FUN is the function, MoreArgs is for arguments to be passed to FUN. The ... is the various arguments to apply the function over.

```
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
```

Instead, we can use

```
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) # 5 variables with mean 1 and standard deviation 2
```

```
## [1] 3.586706 5.012983 4.540546 2.839475 2.074193
```

Vectorizing a function

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

```
## [1] 0.08530021 2.75691844 3.22027555 2.27238618 3.59217369
```

Here, it was expected to have 1 number with mean 1, 2 numbers with mean 2 and so on; and each set would have a standard deviation of 2. However it just generates some random normal noise. The function does not work with a vector of arguments.

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

```
## [[1]]
## [1] -1.147069
##
## [[2]]
## [1] 3.173470 2.999684
##
## [[3]]
## [1] 2.806505 2.413480 1.227121
##
## [[4]]
## [1] 6.075078 3.262731 3.848277 3.245179
##
## [[5]]
## [1] 6.864854 7.674337 1.845301 5.814098 5.571249
```

mapply therefore allows you to vectorize a function that does not allow for vector inputs.

split

Whie `split` is not a loop function itself, it is often used in conjunction with functions like lapply or sapply. It 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, list or data frame. f is a factor 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] -0.4656507  0.4451863  0.7595831  1.0412210 -0.3025976 -0.7070519
##  [7]  0.4855345 -0.4919238  1.4148641  0.2319513
##
## $`2`
##  [1] 0.23849912 0.61242849 0.56739901 0.03105924 0.08108323 0.47379150
##  [7] 0.42917244 0.50881787 0.32566122 0.73698313
##
## $`3`
##  [1]  1.1512493  1.0769162 -0.8190440  2.0808013  1.0480776  2.0204777
##  [7]  1.6062068  0.3427616 -0.3960639  0.6129208
```

Using `lapply` in conjunction with the split

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

```
## $`1`
## [1] 0.2411116
##
## $`2`
## [1] 0.4004895
##
## $`3`
## [1] 0.8724303
```

Consider the following dataset:

```
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
```

Now lets try to get the mean values for each month in the dataset.

```
data <- airquality
factor <- airquality$Month
s <- split(data, factor)
## Don't directly apply mean function because we need only the mean for Ozone Solar Wind Temp
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
```

Alternatively, since each list has 3 elements the sapply function will create a matrix with 3 rows.

```
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
```

When dealing with multiple factors (combination of levels within those factors), use the interaction method to see the number of unique levels possible.

```
x <- rnorm(10)
f1 <- gl(2,5)
f2 <- gl(5,2)
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] -0.0102 -0.7192
##  $ 2.1: num(0)
##  $ 1.2: num [1:2] 1.035 0.452
##  $ 2.2: num(0)
##  $ 1.3: num 1.22
##  $ 2.3: num -1.42
##  $ 1.4: num(0)
##  $ 2.4: num [1:2] -0.479 -0.921
##  $ 1.5: num(0)
##  $ 2.5: num [1:2] 0.241 -0.812
```

Empty levels can be dropped.

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

```
## List of 6
##  $ 1.1: num [1:2] -0.0102 -0.7192
##  $ 1.2: num [1:2] 1.035 0.452
##  $ 1.3: num 1.22
##  $ 2.3: num -1.42
##  $ 2.4: num [1:2] -0.479 -0.921
##  $ 2.5: num [1:2] 0.241 -0.812
```

Debugging

Diagnosing the problem

Basic tools

Using the tools