Tutorial 4 | Loop functions

Loop functions

Using for/while loops is useful for writing scripts, but aren't 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 previous one, but simplifies the results
- **apply**: apply a function of a margins of array
- **tapply**: apply a function over subsets of a vector
- **mapply**: multivariate version of apply

Lapply

lapply takes three arguments:

- a list **X** (note that if X is not a list, it will be coerced to a list using as.list)
- a function (or the name of the function) FUN
- other arguments via its `...` argument

```
lapply
```

```
## function (X, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     if (!is.vector(X) || is.object(X))
##         X <- as.list(X)
##     .Internal(lapply(X, FUN))
## }
## <bytecode: 0x0000000012e24708>
## <environment: namespace:base>
```

So, it takes an object and a function that it needs to apply and returns a list, regardless of the class of the input object:

```
x<-list(a=1:5,b=rnorm(10)) # a: ints from 1 to 5,
# b: 10 random normal numbers
lapply(x,mean)
```

```
## $a
## [1] 3
##
## $b
## [1] -0.2621715
```

If there are default values of a function that you want to specify, then you can also do it in the lapply function after the name of the function:

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

```
## [[1]]
## [1] 8.572517
##
## [[2]]
## [1] 1.441079 3.637378
##
## [[3]]
## [1] 0.5252517 2.5732040 8.1910439
##
## [[4]]
## [1] 4.739430 3.637551 4.549540 8.140184
```

```
#runif is a function that takes N as an argument and generates N
#random normal variables (if max is not defined the default range is 0 to 1)
```

What you can also do is you can use anonymous functions in lapply. That means that you can create functions on the go if they don't have a name already. For example I want to extract the first row of each matrix in a list of matrices:

```
x<-list(a=matrix(1:4,2,2), b= matrix(1:6,3,2))
lapply(x,function(elt) elt[ ,1])
```

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

Sapply

sapply is a version of lapply simplifies a result of laplly 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 regular list is returned

For example recall the first example with lapply:

```
x<-list(a=1:5,b=rnorm(10)) # a: ints from 1 to 5,
# b: 10 random normal numbers
#This is what lapply does
lapply(x,mean)
```

```
## $a
## [1] 3
##
## $b
## [1] 0.3858027
```

```
#This is what sapply does
sapply(x,mean)
```

```
##         a         b
## 3.0000000 0.3858027
```

Apply

apply used to evaluate a function (often an anonymous one) over the margins of 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 (which was the case in the old versions of S language), but it works in one line, so.

```
str(apply)
```

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

So here:

- X is an array
- MARGIN is an integer vector indicating which margins should be used when applying function to them
- FUN is a function to be applied
- ... is for other arguments to be passed to FUN

```
x<- matrix(rnorm(200),20,10)
#This gives the mean of each row of the matrix
apply(x,1,mean)
```

```
##  [1]  0.39266202  0.09802102  0.29618968 -0.09607433  0.33665386 -0.02918708
##  [7]  0.12141301  0.28881580 -0.02446458  0.08771761  0.09688016 -0.10891160
## [13] -0.38177788 -0.10430269  0.15575636  0.49293726  0.66660590  0.12998759
## [19]  0.09604966  0.04511698
```

```
#This gives the sum of elements in of each column
apply(x,2,sum)
```

```
##  [1]  6.5701894 -2.0367157  1.7160580  4.4665218  4.9407146  8.4037436
##  [7] -0.2012029 -7.1848537 10.5792303 -1.6527979
```

So to explain why 1 and why 2 think about this: Matrix is defined usually by rows(1) x columns(2), so the number represents the number of dimension across which the function should be applied. Meaning that in this case, 1 represents first dimension of the matrix (namely across each row), while the 2 represents second dimension, namely across each column.

However, for simple operations like calculating means of the rows, or for example sums of elements in each columns there exist special functions to do so:

- rowSums: same as using apply(x,1,sum)
- colSums: same as using apply(x,2,sum)
- rowMeans: same as using apply(x,1,mean)
- colMeans: same as using apply(x,2,mean)

These are way easier to use and memorize than using plain apply with arguments.

Mapply

mapply is a multivariable version of presious functions. Meaning that, for example in lapply the input argument needed to be the list and the function is applied over the elements of that list, but with mapply you can pass multiple lists and the function would be applied to them in parralel (at the same time)

```
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 that are needed to be passed to a function
- SIMPLIFY indicates whether the result should be simplified

For example you need to create a list of four 1's, three 2's, two 3's and one 4, namely something that looks like this:

```
j<-list(rep(1,4),rep(2,3),rep(3,2),rep(4,1))
#See how tefious it is to type this in?
j
```

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

Now using mapply it would be way shorter and harder to mess things up:

```
j<-mapply(rep,1:4,4:1)
j
```

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

Tapply

tapply is used to apply the function over subsets of a vector.(idk why it's called Tapply tho)

```
str(tapply)
```

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

- X is a vector
- INDEX is a fector or a list of factors(or else they would be coerced to factors)
- FUN is the function to be applied
- ... contains other arguments to be passed to FUN
- simplify tells whether we should simplify the result or not

```
x<-c(rnorm(10), runif(10), rnorm(10,1))
fact<-gl(3,10)
#these are the factors
#using factors we define categories
fact
```

```
##  [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
```

```
#meaning that we want to find means of three categories corresponding to factors 'fact' in data vector
tapply(x,fact,mean)
```

```
##           1           2           3
## -0.09046131  0.42588742  0.72674737
```

Split

Split takes a vector or other object and splits it into groups determined by a factor or list of factors.

```r
str(split)
```

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

- x is a vector, list or a dataframe
- f is factor or the list of factors
- drop indicates whether empty factors levels should be dropped

```r
#Let's take vectors from the previous example
split(x,fact)
```

```
## $`1`
##  [1]  0.1588878 -0.3298285 -0.9580563 -0.8266082 -2.0387768  1.9557932
##  [7]  0.9589433 -0.5653286 -0.2824168  1.0227778
##
## $`2`
##  [1] 0.67999194 0.81117946 0.17361178 0.34138557 0.04823729 0.66944878
##  [7] 0.44765508 0.44879663 0.11742572 0.52114196
##
## $`3`
##  [1]  0.3697763 -0.5867064  2.0467389 -1.3515210  0.2780518  1.8242697
##  [7]  2.5591135  1.6403435  0.3267378  0.1606696
```

Even though it is not initially clear how split could be helpful, keep in mind that this function is able to manipulate much more complicated types of data. For example

```r
#Let's load standard dataset "airquality" from the "datasets" library
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
```

Let's assume you need to calculate the means of the ozone, colar radiation, wind and temperature data in each month. Here is where split function comes in-handy:

```r
#So first we use split on airquality dataset, and we take Month
#as a list of factor for categorization
spl<-split(airquality,airquality$Month)
#And now we can use lapply on each of the months in splitted
#function and calculate their means using collMeans
lapply(spl,function(x) colMeans(x[,c("Ozone","Solar.R", "Wind","Temp")]))
```

```
## $`5`
##    Ozone  Solar.R     Wind     Temp
```

```
##       NA       NA 11.62258 65.54839
##
## $`6`
##     Ozone   Solar.R      Wind      Temp
##        NA 190.16667  10.26667  79.10000
##
## $`7`
##      Ozone    Solar.R       Wind       Temp
##         NA 216.483871   8.941935  83.903226
##
## $`8`
##     Ozone   Solar.R      Wind      Temp
##        NA        NA  8.793548 83.967742
##
## $`9`
##    Ozone  Solar.R     Wind     Temp
##       NA 167.4333  10.1800  76.9000
```

Now you can probably see that this formulation of the result isn't really comfortable to read and the matrix would be much better. For that we can use sapply:

```
#Notice that we also add na.rm=TRUE so that we consider only
#complete cases
spl_m<-sapply(spl,function(x) colMeans(x[,c("Ozone","Solar.R", "Wind","Temp")], na.rm=TRUE))
spl_m
```

```
##                 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
## Temp     65.54839  79.10000  83.903226  83.967742  76.90000
```

And now just for convenience we can change the names of the columns so that it would be easier to understand the data:

```
colnames(spl_m)<-c("May","June","July","August","September")
spl_m
```

```
##                May      June       July     August September
## 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
## Temp     65.54839  79.10000  83.903226  83.967742  76.90000
```

```
#Much better!
```

One thing to note is that split can be used with more than just one factor:

```
x<-rnorm(10)
#Fator 1:
f1<-gl(2,5)
f1
```

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

```
#Factor 2:
f2<-gl(5,2)
f2
```

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

```
#Now just to see how the combination of those factors would
#look like:
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
```

```
#So in other words it contains every possible combimation of all factor lists
str(split(x,list(f1,f2)))
```

```
## List of 10
##  $ 1.1: num [1:2] 0.576 1.493
##  $ 2.1: num(0)
##  $ 1.2: num [1:2] -0.812 -2.047
##  $ 2.2: num(0)
##  $ 1.3: num -0.582
##  $ 2.3: num 0.0881
##  $ 1.4: num(0)
##  $ 2.4: num [1:2] -0.186 -1.252
##  $ 1.5: num(0)
##  $ 2.5: num [1:2] 0.367 0.376
```

```
#Now as you can see some levels are empty, because as well as in
#real datasets you would probably have some combinations of
#factors that just don't exist in your dataset. And to eliminate
#those cases from the results we use 'drop' argument (set to
#TRUE):
str(split(x,list(f1,f2), drop=TRUE))
```

```
## List of 6
##  $ 1.1: num [1:2] 0.576 1.493
##  $ 1.2: num [1:2] -0.812 -2.047
##  $ 1.3: num -0.582
##  $ 2.3: num 0.0881
##  $ 2.4: num [1:2] -0.186 -1.252
##  $ 2.5: num [1:2] 0.367 0.376
```