Unit -4

Apply Family in R

- Introduction:-
- The apply family belongs to the base package present in R.
- This family basically consists of functions that are used to manipulate parts of data from different datatypes such as arrays, lists, matrices or dataframes in a recursive manner.
- These functions avoid explicit use of loop constructs and also allow crossing of data in a different manner.

- Apply functions are essentially implicit loops that iterate over the elements of an object like a dataframe or list and execute some specified set of commands.
- 4 apply functions are:-
- 1)apply()
- 2)lapply()
- 3)Sapply()
- 4)tapply()
- There are others such as rapply(),vapply(),eapply(), and mapply() which are less commonly used.

Using APPLY in R

- The first member of the apply family of functions is the apply().
- The return type of apply() is an array or a vector or a list of values that are given as aon output
- When a function is applied to the margins of a matrix or an array. The apply function takes a minimum of 3 arguments

- Syntax:-
- Apply(x, MARGIN, FUN,…….)
- X: is an object that programmers will be working with or users can input either a matrix or a higher dimensional array
- Users can have more than 2 dimensions in 2 rows and columns, similar to stacking multiple matrices.
- MARGIN:specifies which dimensions to iterate over specifying 1 means that user want to operate over the first dimesnion row, 2 indicates that users want to apply the function to the 2nd dimension ie., columns) ….
- If user specify c(1,2) it means that the function should be applied over the columns and the rows
- 3rd argument:a function that the programmers would like to apply to the chosen margin.if the programmers would like to use arithmetic functions such as +,%*% etc., then we must ensure that the function name must be quoted.
- Fun is found when match.fun is called may be a function or a symbol or a charcter string that would specify what function is to be searched for from the environment of the call to apply function
- Programmers must name the first 3 arguments

Eg:-apply()

```
> x<-matrix(nrow=6, ncol=5, data=1:30)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    7   13   19   25
[2,]    2    8   14   20   26
[3,]    3    9   15   21   27
[4,]    4   10   16   22   28
[5,]    5   11   17   23   29
[6,]    6   12   18   24   30
> apply(x,2,sum)
[1]  21  57  93 129 165
```

Create amatrix ,add row names and col names.verify that it is a matrix or not.

- > x<-
  matrix(c(10,20,30,40,50,60,70,80,90,100,10,30,50,80,100,150,200,250,270,300,10,30,36,80,96,106,110,130,136,144,10,15,30,
- +
  50,70,86,95,100,105,190,10,40,50,66,78,96,107,120,144,157,10,30,57,98,106,130,160,177,189,198),nrow=10,byrow=FALSE)

- > x
- [,1] [,2] [,3] [,4] [,5] [,6]
- [1,] 10 10 10 10 10 10
- [2,] 20 30 30 15 40 30
- [3,] 30 50 36 30 50 57
- [4,] 40 80 80 50 66 98
- [5,] 50 100 96 70 78 106
- [6,] 60 150 106 86 96 130
- [7,] 70 200 110 95 107 160
- [8,] 80 250 130 100 120 177
- [9,] 90 270 136 105 144 189
- [10,] 100 300 144 190 157 198

- > rownames(x)<-c("day1","day2","day3","day4","day5","day6", "day7","day8","day9","day10")
- > colnames(x)<-c("R1", "R2", "R3", "R4", "R5", "R6")

- > x
- R1 R2 R3 R4 R5 R6
- day1 10 10 10 10 10 10
- day2 20 30 30 15 40 30
- day3 30 50 36 30 50 57
- day4 40 80 80 50 66 98
- day5 50 100 96 70 78 106
- day6 60 150 106 86 96 130
- day7 70 200 110 95 107 160
- day8 80 250 130 100 120 177
- day9 90 270 136 105 144 189
- day10 100 300 144 190 157 198

- > class(x)
- [1] "matrix"
- >

#Find the maximum number of leaves that were counted on each given day out of all the replicates so that would be finding the maximum value in each row.

Users can find the max of the first row in 10 different ways.

The name of the matrix refers to the position of a row and or column with in the matrix, separated by a comma.

The row address comes first and the column address comes second . This can be done for all 10 rows

- > max(x[1,])
- [1] 10
- > max(x[2,]
- +)
- [1] 40
- > max(x[3,])
- [1] 57
- > max(x[4,])
- [1] 98
- > max(x[5,])
- [1] 106
- > max(x[6,])
- [1] 150
- > max(x[7,])
- [1] 200
- > max(x[8,])
- [1] 250
- > max(x[9,])
- [1] 270
- > max(x[10,])
- [1] 300

Alternate method by ausing a for loop

- > for(i in 1:10){
- + row<-x[i,]
- + max<-max(row)
- + print(max)
- + }
- [1] 10
- [1] 40
- [1] 57
- [1] 98
- [1] 106
- [1] 150
- [1] 200
- [1] 250
- [1] 270
- [1] 300

- Apply function allows us to do the same thing just by writing a very small bit of code.x is an object it is the first argument.
- Finding the max value for each day,hence the function is applied to each row and therefore a margin argument is 1 and the function that we want to apply is the max function
- We pass these values to the apply() and obtain a vector that tells us the maximum number of leaves that were counted on any given day
- Max function takes each row as input one by one and then all of the individual outputs were compiled in to a single output.
- We could do same thing for the column if required by simply specifying 2 as margin instead of 1.

- > x
- R1 R2 R3 R4 R5 R6
- day1 10 10 10 10 10 10
- day2 20 30 30 15 40 30
- day3 30 50 36 30 50 57
- day4 40 80 80 50 66 98
- day5 50 100 96 70 78 106
- day6 60 150 106 86 96 130
- day7 70 200 110 95 107 160
- day8 80 250 130 100 120 177
- day9 90 270 136 105 144 189
- day10 100 300 144 190 157 198

> apply(x,1,max)

- day1 day2 day3 day4 day5 day6 day7 day8 day9 day10
- 10 40 57 98 106 150 200 250 270 300
- > apply(x,2,max)
- R1 R2 R3 R4 R5 R6
- 100 300 144 190 157 198

Building a dataframe

- Dataframe is similar to a matrix.But the differences between matrix and dataframes is that dataframes contain elements that are different types
- We can use apply() on dataframes also.

```
> y<-data.frame(R1=c(10,20,30,40,50,60,70,80,90,100),
+ R2=c(10,30,50,80,100,150,200,250,270,300),
+ R3=c(10,30,36,80,96,106,110,130,136,144),
+ R4=c(10,15,30,50,70,86,95,100,105,190),
+ R5=c(10,40,50,66,78,96,107,120,144,157),
+ R6=c(10,30,57,98,106,130,160,177,189,198))
> y
   R1  R2  R3  R4  R5  R6
1   10  10  10  10  10  10
2   20  30  30  15  40  30
3   30  50  36  30  50  57
4   40  80  80  50  66  98
5   50 100  96  70  78 106
6   60 150 106  86  96 130
7   70 200 110  95 107 160
8   80 250 130 100 120 177
9   90 270 136 105 144 189
10 100 300 144 190 157 198
```

- > class(y)
- [1] "data.frame"
- >
- #write the commands in R console to find mean number of leaves for each day using apply()
- > apply(y,1,mean)
- [1] 10.00000 27.50000 42.16667 69.00000 83.33333 104.66667 123.66667
- [8] 142.83333 155.66667 181.50000
- >
- # add a column day and number it from 1 to 10.by using factors.

```
> y$day<-as.factor(1:10)
> y<-y[,c(7,1:6)]
> y
   day  R1  R2  R3  R4  R5  R6
1    1  10  10  10  10  10  10
2    2  20  30  30  15  40  30
3    3  30  50  36  30  50  57
4    4  40  80  80  50  66  98
5    5  50 100  96  70  78 106
6    6  60 150 106  86  96 130
7    7  70 200 110  95 107 160
8    8  80 250 130 100 120 177
9    9  90 270 136 105 144 189
10  10 100 300 144 190 157 198
>
```

- The command allows to rearrange the columns of the dataframes such that the day is the first column instead of being tagged on to the end of the dataframe as the seventh column.

- Elements in the day column are being treated as factors.
- To find the mean number of leaves counted everyday in every row, we can see that the earlier command will not work because we have added a factor to every row and we can not take the mean of a mixture of numeric values and factors.

- > apply(y,1,mean)
- [1] NA NA NA NA NA NA NA NA NA NA
- Warning messages:
- 1: In mean.default(newX[, i], ...) :
- argument is not numeric or logical: returning NA
- 2: In mean.default(newX[, i], ...) :
- argument is not numeric or logical: returning NA
- 3: In mean.default(newX[, i], ...) :
- argument is not numeric or logical: returning NA
- 4: In mean.default(newX[, i], ...) :
- argument is not numeric or logical: returning NA
- 5: In mean.default(newX[, i], ...) :
- argument is not numeric or logical: returning NA
- 6: In mean.default(newX[, i], ...) :
- argument is not numeric or logical: returning NA
- 7: In mean.default(newX[, i], ...) :
- argument is not numeric or logical: returning NA
- 8: In mean.default(newX[, i], ...) :
- argument is not numeric or logical: returning NA
- 9: In mean.default(newX[, i], ...) :
- argument is not numeric or logical: returning NA
- 10: In mean.default(newX[, i], ...) :

Excluding day column to determine mean

- Exclude day column from the apply() by referring to the indices of columns with in square brackets.
- > apply(y[,2:7],1,mean)
- [1] 10.00000 27.50000 42.16667 69.00000 83.33333 104.66667 123.66667
- [8] 142.83333 155.66667 181.50000
- >

Write the commands in R console to specify the columns that needs to be excluded in the apply()

- > apply(y[,-1],1,mean)
- [1] 10.00000 27.50000 42.16667 69.00000 83.33333 104.66667 123.66667
- [8] 142.83333 155.66667 181.50000
- >
- > apply(y[,-c(1,2,4,6)],1,mean)
- [1] 10.00000 25.00000 45.66667 76.00000 92.00000 122.00000 151.66667
- [8] 175.66667 188.00000 229.33333
- >

Write the commands in R console to change a value in the y dataframe

- If we want to exclude several columns, we can use the negative sign in front of the vector containing the indices of all the columns that we want to exclude.
- Exclude values from the 1^{st} ,2^{nd},4^{th}, and 6^{th} columns from the calculation .
- Sometimes during data collection there might be some missing values.
- Assume that on the sixth day users did not collect data from the sixth replicant.

- 6th row ,7th column override the current value of 130 with NA
- h[6,7]<-NA
- > h
- Day R1 R2 R3 R4 R5 R6
- 1 1 10 10 10 10 10 10
- 2 2 20 30 30 15 40 30
- 3 3 30 50 36 30 50 57
- 4 4 40 80 80 50 66 98
- 5 5 50 100 96 70 78 106
- 6 6 60 150 106 86 96 NA
- 7 7 70 200 110 95 107 160
- 8 8 80 250 130 100 120 177
- 9 9 90 270 136 105 144 189
- 10 10 100 300 144 190 157 198
- >

- If we try to apply the function we get NA for sixth row
- > apply(h[,-1],1,mean)
- [1] 10.00000 27.50000 42.16667 69.00000 83.33333 NA 123.66667
- [8] 142.83333 155.66667 181.50000
- >
- REMOVEING NA FROM DATASET
- We can remove the NA in the dataset using na.rm which stands for NA remove.
- na.rm is an argument of the mean function that is specified with in brackets when using the function.
- Assume we have a vector X calculate mean excluding the NA element
- > hk<-c(1,3,6,NA,2)
- > mean(hk,na.rm=TRUE)
- [1] 3

```
> h
   Day  R1  R2  R3  R4  R5  R6
1    1  10  10  10  10  10  10
2    2  20  30  30  15  40  30
3    3  30  50  36  30  50  57
4    4  40  80  80  50  66  98
5    5  50 100  96  70  78 106
6    6  60 150 106  86  96  NA
7    7  70 200 110  95 107 160
8    8  80 250 130 100 120 177
9    9  90 270 136 105 144 189
10  10 100 300 144 190 157 198
> apply(h[,-1],1,mean)
 [1]  10.00000  27.50000  42.16667  69.00000  83.33333       NA 123.66667
 [8] 142.83333 155.66667 181.50000
> apply(h[,-1],1,mean,na.rm=TRUE)
 [1]  10.00000  27.50000  42.16667  69.00000  83.33333  99.60000 123.66667
 [8] 142.83333 155.66667 181.50000
>
```

Determine the proportion of the totalnumber of leaves counted on each day

- There is no function that solve the problem so users can make up the own function that solve the problem.
- The function returns the value of the element/the max value of each vector
- This function is applied on matrix

- > ak
- R1 R2 R3 R4 R5 R6
- day1 10 10 10 10 10 10
- day2 20 30 30 15 40 30
- day3 30 50 36 30 50 57
- day4 40 80 80 50 66 98
- day5 50 100 96 70 78 106
- day6 60 150 106 86 96 130
- day7 70 200 110 95 107 160
- day8 80 250 130 100 120 177
- day9 90 270 136 105 144 189
- day10 100 300 144 190 157 198

- > prop<-function(x){
- + x/max(x)
- + }

- > apply(ak,2,prop)
- R1 R2 R3 R4 R5 R6
- day1 0.1 0.03333333 0.06944444 0.05263158 0.06369427 0.05050505
- day2 0.2 0.10000000 0.20833333 0.07894737 0.25477707 0.15151515
- day3 0.3 0.16666667 0.25000000 0.15789474 0.31847134 0.28787879
- day4 0.4 0.26666667 0.55555556 0.26315789 0.42038217 0.49494949
- day5 0.5 0.33333333 0.66666667 0.36842105 0.49681529 0.53535354
- day6 0.6 0.50000000 0.73611111 0.45263158 0.61146497 0.65656566
- day7 0.7 0.66666667 0.76388889 0.50000000 0.68152866 0.80808081
- day8 0.8 0.83333333 0.90277778 0.52631579 0.76433121 0.89393939
- day9 0.9 0.90000000 0.94444444 0.55263158 0.91719745 0.95454545
- day10 1.0 1.00000000 1.00000000 1.00000000 1.00000000 1.00000000

- Above output is whole matrix instead of just a vector.
- This is one of the features of the apply()
- It can return different objects(including vectors, arrays, matrices or lists). Depending on what function is being applied.

Using LAPPLY IN R

- LAPPLY() is a function that loops over a list and evaluates a function on each element.
- SAPPLY () is also similar to LAPPLY() but it also ensures that the results are more simplified.
- Output for lapply() is always a list
- Output for sapply() is simplified to vectors and matrices
- Lapply() &sapply() are functions that allow programmers to apply other functions to vectors,matrices,arrays,or lists.
- Lappy takes 3 arguments—*X*:-which is a list. *FUN*---function, and other *arguments*
- Function(X,FUN,….)

- If X is not a list then X would be coerced to a list through the function *as.list*
- If the argument X passed is not a list then it will be coerced to a list only if possible.
- Suppose it is impossible to coerce the object X to a list, then a *NULL* value is returned to the programmers.

If object X can be coerced to a list, this is done through a function *as.list, and then the* rest of the lapply function is implemented internally in C code to make it more efficient.

- Lapply() function always returns a list, irrespective of the class of the input given by programmer.
- ***Structure of the lapply***
- > lapply
- function (X, FUN, ...)
- {
- FUN <- match.fun(FUN)
- if (!is.vector(X) || is.object(X))
- X <- as.list(X)
- .Internal(lapply(X, FUN))
- }
- <bytecode: 0x000000001a80dee8>
- <environment: namespace:base>
- >

Eg:-summing the columns usinglapply()

```
> my.data<-data.frame(data1=rnorm(10),data2=rnorm(10),data3=rnorm(10))
> my.data
       data1      data2       data3
1  2.1769916  0.2744092 -0.30175961
2  1.1940334 -0.7229859 -0.42833236
3  0.4804799 -0.1873488  0.56673187
4 -0.7191441 -1.1187763 -0.77909427
5  0.8219887 -1.1780342 -0.78744929
6  0.6747213 -0.6583715 -0.09784757
7 -0.8546326  1.1706198 -1.39865546
8  1.3453265  0.9377545  0.80358649
9  0.2912998  0.1038053 -1.21331744
10 -0.4815525 -1.5125504 -0.39332429
> lapply(my.data,sum)
$data1
[1] 4.929512

$data2
[1] -2.891478

$data3
[1] -4.029462
```

- Above eg:-generate a dataframe with 3 elements, data1,data2,data3.
- Compute the sum of each element.
- This can be done by applying either lapply() or sapply() to my.data
- 2nd argument is the function that we wish to apply.
- o/p from lapply()is a list with 3 elements.
- Suppose that the data we have to work with is a list consisting of 3 matrices each of different dimensions

#Applying lapply() and sapply() to lists

- > A<-matrix(1:9,nrow=3);B<-matrix(1:16,nrow=4);C<-A<-matrix(1:8,nrow=4);

- > k<-list(A=A,B=B,C=C)

- > k
- $`A`
- [,1] [,2]
- [1,] 1 5
- [2,] 2 6
- [3,] 3 7
- [4,] 4 8

- $B
- [,1] [,2] [,3] [,4]
- [1,] 1 5 9 13
- [2,] 2 6 10 14
- [3,] 3 7 11 15
- [4,] 4 8 12 16

- $C
- [,1] [,2]
- [1,] 1 5
- [2,] 2 6
- [3,] 3 7
- [4,] 4 8

Using bracket function to extract a column

- Suppose we would like to extract the second column of these matrices.
- It can be done with the bracket function, where we specify that we want all rows by not writing anything for the rows argument.but only column 2 by writing a ‘2’ at the columns argument
- The result is a list with 3 elements which are vectors of different lengths..collapsing these into a vector will result in loss of information.
- Eg:-> lapply(k,"[",,2)
- $`A`
- [1] 5 6 7 8
- $B
- [1] 5 6 7 8
- $C
- [1] 5 6 7 8

USING SAPPLY IN R

- The sapply() is a user-friendly version and a wrapper of the lapply()function.
- It returns a vector by default.
- Sapply() ensures that the result of the lapply() is simplified as much as possible.
- Suppose the result of lapply is list where the length of each element is 1,then sapply returns a vector, which is simplified.
- Suppose the result of lapply is a list such that each element is a vector of the same length(>1), then, the returned result of sapply would be a matrix.
- If the function is not able to identify the return type of the output.then a list is returned

Using sapply() to get accurate results

- > x<-list(a=1:4,b=rnorm(10),c=rnorm(20,1),d=rnorm(100,5))
- > x
- $`a`
- [1] 1 2 3 4

- $b
- [1] 0.5598018 -0.3407920 0.7144057 -0.1131741 0.7816717 -0.4765731
- [7] -1.0755795 0.7916670 -0.1946836 0.7102384

- $c
- [1] 2.2024843 0.8252657 3.1203180 0.9696472 1.7251239 0.4415068
- [7] 1.4386779 0.4946751 0.8385002 0.1286719 -0.4070503 1.5405843
- [13] 1.1542330 1.2196965 2.4719070 0.2058418 1.2938130 0.3876710
- [19] 1.5116611 1.7532755

- $d
- [1] 5.260024 4.068104 4.753210 2.219746 4.657682 4.104573 4.407395 5.096485
- [9] 6.338178 5.955521 4.209788 5.614897 4.066367 4.575470 4.896653 4.377454
- [17] 5.751464 5.521800 3.839684 4.321781 6.014341 4.928251 4.992427 3.084627
- [25] 3.095278 3.597495 5.864624 5.171301 5.986438 5.445200 4.734843 5.673137
- [33] 5.979122 4.795974 4.199123 5.541993 3.283493 3.063936 5.396178 4.652216
- [41] 3.971815 5.375200 5.008409 6.158932 2.975247 4.484979 5.264711 4.482008
- [49] 5.900559 6.461866 5.055817 2.689314 4.431945 4.541720 6.355052 5.475631
- [57] 3.277466 4.619909 4.689707 7.380063 5.984682 6.096761 5.082242 6.031350
- [65] 6.298678 4.443082 4.482099 5.316130 3.462372 4.420648 3.942861 5.960458
- [73] 4.355473 5.871826 3.571279 4.918036 6.172947 4.595888 3.360167 6.174050
- [81] 6.679379 7.384775 7.032067 7.253911 4.894945 6.440747 4.089095 5.485229
- [89] 5.357886 5.304604 4.966726 3.781327 6.021441 5.869640 5.205953 2.780804
- [97] 4.576001 6.568543 6.424264 4.004960

- > lapply(x,mean)
- $`a`
- [1] 2.5

- $b
- [1] 0.1356982

- $c
- [1] 1.165825

- $d
- [1] 4.988

- >

- > sapply(x,mean)
- a b c d
- 2.5000000 0.1356982 1.1658252 4.9879995
- > mean(x)
- [1] NA
- Warning message:
- In mean.default(x) : argument is not numeric or logical: returning NA
- >

- In the above example:when lapply is called by the user and mean is applied to each element,a list of length of 4 is returned to the user and each element of the list is a single number.
- It is better to receive this result as a vector containing all these numbers and this is done by sapply
- Ie., when object X calls the sapply() with the function being mean, a vector with 4 numbers is returned to the user as output.
- If mean is called on the list by itself, it would not work because mean is not meant to be applied to lists and so programmers will get a warning message with NA

USING TAPPLY IN R

- Tapply is another important function which is used when a function should be applied over subsets of vectors
- Imagine that the users have a numeric vector such that there exists parts of this vector on which the users would want a summary statistics to be calculated.
- So we need some other variable or an object that helps in identifying what element of numeric vector is associated to what group.
- The idea used here is that the summary statistics such as mean and standard deviation are calculated by users for each group in the numeric vector.

Syntax: tapply

- tapply(X, INDEX, FUN = NULL, …, default = NA, simplify = TRUE)
- Tapply() uses that the first argument is a numeric vector or any other vector of some form
- 2nd argument is another vector whose length is the same as that of the first vector and also determines the group in which every element of the numeric vector belongs to.
- Eg:-there are 2 groups of men and women.The 1st observations are recorded by men and the next recorded for women
- Programmer should have a factor variable that would indicate to the users as to which recorded observation corresponds to men and which corresponds to women.

- If we want to extract the mean of the numeric character for men and women then we use the *tapply()* to perform such calculations.
- *FUN* is the function that the users would like to perform
- *Simplify* argument specifies whether we want to simplify the results or not.

Structure of the tapply

- > str(tapply)
- function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)

Write the commands in R console to obtain the mean using tapply function by considering a vector having 10 normal and 10 uniform variables.Assume that these vectors have three groups

- > x<-c(rnorm(10),runif(10),rnorm(10,1))
- > x
- [1] 0.84613861 0.07502533 2.67136791 -1.27438337 0.78632958 1.14318688
- [7] -1.72442285 0.70333541 0.41761834 -2.62092169 0.49462915 0.08914491
- [13] 0.41905081 0.51993484 0.54844754 0.10401121 0.16187964 0.07360538
- [19] 0.02871989 0.26277321 0.16956816 2.23357855 0.79657723 0.70460833
- [25] 0.35038017 0.68519635 -0.32608742 1.48340303 2.32273490 0.09059710
- >

- We can use gl() for creating an additional factor variable containing 3 levels.each level shall be scanned repeatedly for 10 times

Generate Factor Levels

Description

Generate factors by specifying the pattern of their levels.

Usage

```
gl(n, k, length = n*k, labels =
seq_len(n), ordered = FALSE)
```

n	an integer giving the number of levels.
k	an integer giving the number of replications.
length	an integer giving the length of the result.
labels	an optional vector of labels for the resulting factor levels.
ordered	a logical indicating whether the result should be ordered or not.

- > 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
- >

- Tapply is used on X passed as the factor variable f and also the mean function which would return the mean of each group of elements in X.
- > tapply(x,f,mean)
- 1 2 3
- 0.1023274 0.2702197 0.8510556
- >

- If we give the value of *simplify* as FALSE, then *a list* is returned to the user as a result.
- If we apply the tapply() on the same numeric vector and factor where the mean is to be calculated, and give the value of simplify as *FALSE*, we then get *a list of 3 elements*

- > tapply(x,f,mean,simplify=FALSE)
- $`1`
- [1] 0.1023274

- $`2`
- [1] 0.2702197

- $`3`
- [1] 0.8510556

- >
- We can also pass more complex summary statistics in the tapply() instead of calculating the mean,which just returns a number.

Calculate the range of observation which gives the min value and the max value of the observations of the subset of vector X

- #find group ranges
- > tapply(x, f , range)
- $`1`
- [1] -2.620922 2.671368

- $`2`
- [1] 0.02871989 0.54844754

- $`3`
- [1] -0.3260874 2.3227349

Above o/p is a list where each element is a vector having its length equal to 2

- Tapply is a very helpful function because the vector is split in to small parts and a summary statistic or a function is applied to each of these parts.
- after this the function is applied on them and all the parts are then collected and connected together.
- Split function:-
- Split() can be used by combining 2 or more functions such as lapply or sapply.
- It is not considered a loop function.
- Syntax:-
- > str(split)
- function (x, f, drop = FALSE, ...)
- Split() takes a vector or an object x as its 1st argument
- f: represents group levels
- Split() split the object x in to a number of groups that are identified by factor f.

- Eg:-f consists of 3 levels for representing three different groups.
- Use split() which splits the object x into 3 different groups.
- Once if we have these groups split apart ,we can use lapply(),sapply() to apply a function to each of these individual groups.
- Eg:-programmers had a simulated 10 normal random variables with mean o and 10 uniform and 10 normal variables with mean 1.
- f consists of 3 levels the vector is split into 3 parts
- 1^{st} element being 10 normals
- 2^{nd} element being 10 uniforms
- 3^{rd} element being 10 normals is returned.

#genrating a collection having normal random variables

```
> x<-c(rnorm(10),runif(10),rnorm(10,1))
> x
 [1]  0.7373115296 -0.8472848431  0.8590583411  0.5606110710 -1.0661772110
 [6] -0.2684579220 -0.5962508998 -0.3626668122 -0.7380673748 -0.0006589424
[11]  0.2040063113  0.6225134828  0.8703633002  0.2093047446  0.5443125837
[16]  0.0301094926  0.9682989123  0.3035488105  0.8737872790  0.1225081645
[21]  1.7233152580 -0.4197623812  1.8866453567  1.3353931663  1.0659835390
[26]  0.3661346394  0.1559020434 -1.1420061736  0.8600028360  1.1456713573
> 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
> split(x,f)
$`1`
 [1]  0.7373115296 -0.8472848431  0.8590583411  0.5606110710 -1.0661772110
 [6] -0.2684579220 -0.5962508998 -0.3626668122 -0.7380673748 -0.0006589424

$`2`
 [1] 0.20400631 0.62251348 0.87036330 0.20930474 0.54431258 0.03010949
 [7] 0.96829891 0.30354881 0.87378728 0.12250816

$`3`
 [1]  1.7233153 -0.4197624  1.8866454  1.3353932  1.0659835  0.3661346
 [7]  0.1559020 -1.1420062  0.8600028  1.1456714
```

- The common procedure is to use split followed by lapply.
- Split() is used because it makes it easier to split complex types of objects.
- Suppose we have a datasets package containing a dataframe called *airquality.*
- > 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
- >

Calculate the mean of ozone,solar radiation and wind within each month using lapply for airquality dataframe

- In each month,30 observations exist and we calculate the mean with in each month.
- As a solution the dataframe is split into tiny monthly parts and column mean values are calculated using one of the functions such as *apply* for *colmeans* on those variables

Calculating the mean of ozone,solar radiation and wind using lapply

```
> 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

>
```

- We split the airquality dataframe using the month variable,though the month variable present in dataframe is not a factor variable.
- It is also possible that this variable may be converted in to a factor variable.
- This is due to reason that it takes only the following 5 values-5,6,7,8,9 and the values are recorded only during the warm months of the year.
- Then perform a split of the airquality variable according to the month variable and then apply an anonymous function which considers the mean value of the columns of the ozone,solarradiation, and wind.
- Column means all these 3 variables are taken for each of the monthly dataframes.
- The result is a list where each element of the list is a vector with length equal to 3 ie.,would be mean one each for ozone,solar radiation,a nd wind with in that month.

- Most of the ozone values in month are observed to be NA. This is beacause there are missing values in the column (NA) and heance mean can not be taken.
- *#calculate the mean of ozone, solar radiation and wind with in each month using sapply for airquality dataframe.*
- Sapply can be used . It simplifies the output since every element belonging to the returned list contains a vector of length3.
- It puts all the numbers into a matrix of 3 rows and 5 columns.
- Monthly mean value of each of these 3 variables is observed in a much more compact format ie., in the form of a matrix instead of a list.
- Before calculating the mean ,we must ensure that the missing values are removed.
- To remove such missing values from each of the columns, we may pass the na.rm argument to the function and
- then on applying the sapply() on the resultant split list,the mean values of the observed values for each of the 3 varaibles for each of the 3 variables for each of the 5 months are returned as an output

Calculating the mean of Ozone,solar radiation and wind using sapply

```
> 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
>
```

Using MAPPLY IN R

mapply is a multivariate version of the lapply and sapply functions.

This function is applied in parallel over sets of different arguments.

Lapply,sapply, and tapply consist of applying a function only on the elements of a single object.

For eg:-consider a situation where we have 2 lists and we would like to apply a function on them such that the elements of first list shall be taken as 1 argument in the function and the elements belonging to the 2 list shall be taken as another argument in the function.

In such situations we cannot use the lapply or sapply functions.

This can be done by using the for loop, so that it indexes all the elements belonging to each of the two lists and later each of the elements is passed to a function that are present in the list.

Using the for loop is not a good idea and the code becomes tedious.

- One alternative to do this is by using the *mapply().*
- *Mapply* takes multiplelists as arguments and then a function is applied to the 2 elements belonging to the multiple lists in parallel
- ***Syntax of mapply***
- Structure of mapply is obtained by giving command as:
- > str(mapply)
- function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
- >
- *FUN:* function to apply
- *MoreArgs:* if user want to pass more arguments that are needed to the function FUN.
- (eg:-if there are 3 lists and if a user passess 3 objects the defined function takes atleast 3 arguments in the function)
- *SIMPLIFY*: if this is given as TRUE by the user it signifies that the result should be simplified.

Create a list that repeats integer 1 --4 times,2—3 times,3—2times and 4—1 time.

- #replicating values using mapply and rep
- #the following is tedious to type
- > 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 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

- >

- We used *mapply(rep)* where rep is repeat function which takes 2 arguments.
- The initial set of argument is given as 1:4 and final set of arguments is given as 4:1.the result is our desired list
- *Mapply* is used when a function has to be applied to multiple sets of arguments.
- #consider another example:-use a function that will generate few random normal noise.
- This function consists of 3 arguments :
- n: number of observations made
- Mean
- Standard deviation
- Noise is applied to the single set of arguments 5,1,2
- 5 is random normal random variables.
- Mean value is 1; standard deviation value is 2.

Applying noise on single set of arguments

- #this function will not work correctly and accurately if a vector of 5 elements(1:5 and 1:5) is passed as an argument .
- We expect is to have 1 random variable whose mean=1
- 2 random variable whose mean=2
- 3 random variables whose mean=3 and so on……
- Till random variable with mean=5 but we donot get this output:
- > noise<-function(n,mean,sd){
- + rnorm(n,mean,sd)
- + }
- > noise(5,1,2)
- [1] -3.4119619 1.7847785 -1.0621909 -0.4050456 2.3475628
- >

Mapply function on vectorized noise function

- When we use mapply function with 3 sets of arguments 1:5, 1:5 and 2. we get desired output

```
> mapply(noise,1:5,1:5,2)
[[1]]
[1] -0.8000631
[[2]]
[1]  1.632366 -1.488433
[[3]]
[1] 1.729325 1.999961 4.138120
[[4]]
[1] 2.699145 4.788870 6.403534 4.264246
[[5]]
[1] 3.759027 4.500436 6.786153 5.722771 5.354308
```

That's how a function that does not allow for vector arguments is instantly vectorized