R Programming apply family of functions

In this experiment we will learn about how to use apply family of functions to do data wrangling and avaoid use of looping constructs

The R base manual tells you that it's called as follows:

```
apply(X, MARGIN, FUN, ...)
```

`X` is an array or a matrix if the dimension of the array is `2`

`MARGIN` is a variable defining how the function is applied:

- When `MARGIN=1`, it applies over rows
- Whereas with `MARGIN=2`, it works over columns
- `MARGIN=c(1,2)`, it applies to both rows and columns;

and `FUN`, represents the function that you want to apply to the data. It can be any R function, including a User Defined Function (UDF).

```
# Complete the following code by using apropriate apply family function
# Construct a 5x6 matrix
X <-matrix(rnorm(30), nrow=5, ncol=6)
X
# Sum the values of each column with apply()
apply(X, 2, sum)
# Sum the values of each row with apply()
rowSums(X)
```

```
                         A matrix: 5 × 6 of type dbl
 1.56577694   1.1423489   0.1599963  -0.4322737  -0.83472254  -1.2529742
 1.08803489   1.4491379  -0.1103644  -1.4007716   0.01456015  -0.5491222
-0.27716507   0.1375821  -0.4678829   1.1487384   1.03995178   1.2177147
-0.44962484  -0.2380314  -0.8561092  -0.6595325  -2.67695723  -1.0984328
 0.09695042  -0.9669232  -0.3800127   1.9285163  -1.16640992   0.9962987
2.02397234823781 · 1.52411439494027 · -1.65437291593552 · 0.584676968443811 · -3.6235777531755
-0.686515738535445
0.348151784965056 · 0.49147474770796 · 2.7989390707566 · -5.9786879371 · 0.508419637645727
```

```
apply(X,2,sum)
```

```
-2.13990092067223 · -0.207127265177643 · -2.10256795261022 · 2.86351430852767 · 0.112667744743
-1.07162628459544
```

```
apply(X, 1, sum)
```

```
2.70746793552948 · -5.7394046667717 · 0.177901170368714 · 0.111976178612437 · 0.19701901247632
```

```
# Complete the following code by using looping construct
# Construct a 5x6 matrix
X <-matrix(rnorm(30), nrow=5, ncol=6)
X
```

A matrix: 5 × 6 of type dbl

```
 0.4175995  1.4825187 -1.0564286 -0.7176037 -1.0060144 -0.68582203
-0.1928479 -0.7788001  0.7216013  1.7237513 -1.9654844  0.55986950
-0.5153336  1.5036463  0.9669224 -3.2057635 -1.3784739 -0.04552271
-0.7787829  1.7077944 -0.8948594  0.4978330 -2.0695836 -0.83058000
-0.9648506  0.9579158  2.4260971 -1.4408037 -0.7508842 -0.27442032
```

```
a <- round(rnorm(6),3)
b <- round(rnorm(6),3)
c <- round(rnorm(6),3)
d <- round(rnorm(6),3)
e <- round(rnorm(6),3)


mat <- matrix(c(a,b,c,d,e),ncol=6)
mat
```

A matrix: 5 × 6 of type dbl

```
-0.732  0.534  0.128  0.514  1.480 -1.872
-1.406 -1.639  1.367  0.642 -0.700 -0.191
-1.121 -0.736  0.333 -0.800 -0.996 -0.483
-0.398 -0.970 -0.771  0.108 -1.278 -0.103
-1.193  0.224 -0.058 -0.512 -1.029  0.223
```

```
# Sum the values of each column with apply()
df <- as.data.frame(mat)
colSum <- 0
for(i in 1:6){
  collSum <- apply(df[i],2,sum)
  print(collSum)
}
```

```
    V1
-4.85
     V2
-2.587
    V3
0.999
     V4
-0.048
     V5
-2.523
```

```
        V6
    -2.426
```

The **lapply()** Function is used when You want to apply a given function to every element of a list and obtain a list as a result.

Syntax of **lapply()** looks like the **apply()** function.

There is a slight difference in way they work, which is :

1. **lapply()** can be used for other objects like dataframes, lists or vectors
2. The output returned by **lapply()** is a list (which explains the “l” in the function name), which has the same number of elements as the object passed to it.

To see how this works, complete the following code block.

lapply() is a quite common operation performed on real data when making comparisons or aggregations from different dataframes.

```
# Using sequence 1:9 create a matrix A with size 3*3
A <- matrix(1:9,3,3)
#Using sequence 4:15 create a matrix B with size 4*3
B <- matrix(4:15,4,3)
#Using sequnce 8:10 create a matrix C with size 3*2
C <- matrix(8:10,3,2)
# create variable MyList as list of matrix A, B, C
MyList <- list(A,B,C)
MyList
```

```
# Using lapply and [ operator Extract the 1st row from `MyList` with the selection operator
```

```
        A matrix:
        3 × 3 of
        type int
1.      1   4   7

        2   5   8

        3   6   9
        A matrix: 4 ×
        3 of type int
2.      4    8   12
```

```
# Using lapply and [ operator Extract the 2nd column from `MyList` with the selection opera
paste("second column:")
lapply(MyList,"[",,2)
```

```
'second column:'

    1. 4 · 5 · 6
    2. 8 · 9 · 10 · 11
    3. 8 · 9 · 10


       9    9
```

```
# Using lapply and [ operator Extract the 1st row from `MyList` with the selection operator
paste("First column:")
lapply(MyList,"[",1,)
```

```
'First column:'

    1. 1 · 4 · 7
    2. 4 · 8 · 12
    3. 8 · 8
```

```
# Using lapply and [ operator Extract the 4th row from `MyList` with the selection operator
paste("Fourth column:")
lapply(MyList,"[",4,)
```

```
'Fourth column:'
Error in FUN(X[[i]], ...): subscript out of bounds
Traceback:
```

SEARCH STACK OVERFLOW

The **sapply()** Function

The **sapply()** function works like **lapply()**, but it tries to simplify the output to the most elementary data structure that is possible. And indeed, `sapply()` is a **'wrapper'** function for **lapply()**.

Let's repeat the extraction operation of a single element as in the last example, but now take the first element of the second row (indexes 2 and 1) for each matrix.

Applying the **lapply()** function would give us a list unless you pass `simplify=FALSE` as a parameter to **sapply()**. Then, a list will be returned. See how it works in the code chunk below:

```
# Return a list with `lapply()`
lapply(MyList,"[", 2, 1 )

# Return a vector with `sapply()`
sapply(MyList,"[", 2, 1 )

# Return a list with `sapply()`
sapply(MyList,"[", 2, 1, simplify=F)

# Return a vector with `unlist()`
unlist(lapply(MyList,"[", 2, 1 ))
```

```
    1. 2
    2. 5
    3. 9

2 · 5 · 9

    1. 2
    2. 5
    3. 9

2 · 5 · 9
```

In this Text Block Write note on mapply function and give exampe code and exaplain use of mapply function.

1.

```
# Example use of mapply function
```

In the 5th cell of this notebook, I created 5 different variables in order to create 5 vectors to pass it to matrix() function. Now, this might seem simple and easy because I wanted only 5 vectors (5 rows) in my matrix. But this kind of approach isn't optimal. If I need to have 1000 vectors to convert into a matrix, I'll have to use 1000 variables to store.

```
a <- round(rnorm(6),3)
b <- round(rnorm(6),3)
c <- round(rnorm(6),3)
d <- round(rnorm(6),3)
e <- round(rnorm(6),3)
mat <- matrix(c(a,b,c,d,e),ncol=6)
mat
```

```
                 A matrix: 5 × 6 of type dbl
 0.108   0.083   0.356   2.179   0.390  -0.093
-0.249  -0.734   0.023   1.366  -0.024   0.589
```

However, with the help of mapply() (and rep()) function, I can achieve the above objective in an optimal way. Here, not only the number of lines got reduced, but the memory required to store intermediate variables is saved as well.

```
mat1 <- matrix(rep(mapply(rep, round(rnorm(6), 3), times=1), times=5), ncol=6)
mat1
```

```
                 A matrix: 5 × 6 of type dbl
 0.914  -1.023   2.134  -0.403  -0.959  -1.280
-1.280   0.914  -1.023   2.134  -0.403  -0.959
-0.959  -1.280   0.914  -1.023   2.134  -0.403
-0.403  -0.959  -1.280   0.914  -1.023   2.134
 2.134  -0.403  -0.959  -1.280   0.914  -1.023
```

The rep() function gives 5 vectors to the matrix() function and ncol=6 helps to create a matrix of size 5x6. Inside the rep() , mapply() comes to the picture to apply another rep() to generate random normal numbers (rounded off to 3 decimal places). This whole process is repeated 5 times.

✓ 0s completed at 01:50