Apply function family in R with Examples

Becky Lin

2023-02-13

Introduction

Apply functions are a family of functions in base R which allow you to repetitively perform an action on multiple chunks of data. An apply function is essentially a loop, but run faster than loops and often require less code.

The apply functions that this document will address are apply, lapply, sapply, tapply, vapply, mapply and mclapply. There are so many different apply functions because they are meant to operate on different types of data.

The apply() function is the basic model of the family of apply functions in R, which includes specific functions like lapply(), sapply(), tapply(), mapply(), vapply(), rapply(), bapply(), eapply(), and others. All of these functions allow us to iterate over a data structure such as a list, a matrix, an array, a DataFrame, or a selected slice of a given data structure — and perform the same operation at each element.

apply() function

apply() takes Data frame or matrix as an input and gives output in vector, list or array. Apply function in R is primarily used to avoid explicit uses of loop constructs. It is the most basic of all collections can be used over a matrice.

This function takes 3 arguments

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

where

- X: an array or matrix or data frame
- MARGIN: take a value or range between 1 and 2 to define where to apply the function FUN.
  - MARGIN=1: the FUN is performed on rows
  - MARGIN=2: the FUN is performed on columns
  - MARGIN = c(1,2): the FUN is performed on rows and columns.
- FUN: tells which function to apply. Built functions like mean, median, sum, min, max, and even user-defined functions can be applied.

Examples

```
data <- matrix(C<-(1:12),nrow=3, ncol=4)
data
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
apply(data, 2, sum)
```

```
## [1]  6 15 24 33
```

```
apply(data, 1, sum)
```

```
## [1] 22 26 30
```

```
f<-function(x){ max(x)-min(x)}
apply(data,2,f)
```

```
## [1] 2 2 2 2
```

```
apply(data, 2, range)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    3    6    9   12
```

lapply() function

lappy() returns a list of the similar length as input list object, each element of which is the result of applying FUN to the corresponding element of list. The "l" in lappy() stands for *list*. The difference between lapply() and apply() lies between the output return. The output of lapply() is a list. lapply() can be used for other objects like data frames and lists.

This function takes two arguments

```
lapply(X, FUN)
```

where

- X: a list or a vector or a data frame
- FUN: function applied to each element of X

Examples

```
movies <- c("SPYDERMAN","BATMAN","VERTIGO")
lapply(movies,tolower)
```

```
## [[1]]
## [1] "spyderman"
##
## [[2]]
## [1] "batman"
##
## [[3]]
## [1] "vertigo"
```

```
data2 = matrix(c(1:4),nrow=2,ncol=2)
lapply(data2,function(x) x+1)
```

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

```
## [[4]]
## [1] 5
```

sapply() function

- sapply() function takes list, vector or data frame as input and gives output in vector or matrix.
- sapply() is useful for operations on list objects and returns a list object of same length of original set.
- sapply() and lapply() do the same job but sapply() returns a vector while lapply() returns a list.
- In general, sapply() is more efficient than lapply() in the output returned because sapply() store values directly into a vector.

```
sapply(X, FUN)
```

Arguments:

- X: a vector or list or vector or data frame.
- FUN: function applied to each element of X.

Example

```
dt <- cars
lmn_cars <- lapply(dt, mean)
smn_cars <- sapply(dt, mean)
lmn_cars
```

```
## $speed
## [1] 15.4
##
## $dist
## [1] 42.98
```

```
smn_cars
```

```
## speed  dist
## 15.40 42.98
```

```
# FUN: home-made function
avg <- function(x) {
  ( min(x) + max(x) ) / 2}
fcars <- sapply(dt, avg)
fcars
```

```
## speed  dist
##  14.5  61.0
```

Example 2: Slicing vector

We can use lapply() or sapply() interchangeable to slice a data frame. We create a function, below_average(), that takes a vector of numerical values and returns a vector that only contains the values that are strictly above the average. We compare both results with the identical() function.

```
below_ave <- function(x) {
    ave <- mean(x)
    return(x[x > ave])
}
dt_s<- sapply(dt, below_ave)
dt_l<- lapply(dt, below_ave)
identical(dt_s, dt_l)
```

```
## [1] TRUE
```

```
c(length(dt_s$speed),length(dt_s$dist) )
```

```
## [1] 24 21
```

Summarize the difference between apply(), sapply() and lapply()

Function	Arguments	Objective	Input	Output
apply	apply(X, MARGIN, FUN)	Apply a function to the rows or columns or both	Data frame or matrix	vector, list, array
lapply	lapply(X, FUN)	Apply a function to all the elements of the input	list, vector or data frame	list
sapply	sapply(X, FUN)	Apply a function to all the elements of the input	list, vector or data frame	vector or matrix

tapply() function

- tapply() computes a measure (mean, median, min, max, etc..) or a function for each factor variable in a vector
- To modify the outplut class to **list**, simply set the **simplify** argument to FALSE.
- tapply() is a very useful function that lets you create a subset of a vector and then apply some functions to each of the subset.

```
tapply(X, INDEX, FUN = NULL)
```

Arguments

- X: an object, usually a vector
- INDEX: a list containing factor
- FUN: function applied to each element of X

Part of the job of a data scientist or researchers is to compute summaries of variables. For instance, measure the average or group data based on a characteristic. Most of the data are grouped by ID, city, countries, and so on. Summarizing over group reveals more interesting patterns.

To understand how it works, let's use the iris dataset. This dataset is very famous in the world of machine learning. The purpose of this dataset is to predict the class of each of the three flower species: Sepal, Versicolor, Virginica. The dataset collects information for each species about their length and width.

```
data(iris)
# find the median Sepal width of each species
tapply(iris$Sepal.Width, iris$Species, median)
```

```
##     setosa versicolor  virginica
##        3.4        2.8        3.0
```

```
tapply(iris$Sepal.Width, iris$Species, mean,simplify=FALSE)
```

```
## $setosa
## [1] 3.428
##
## $versicolor
## [1] 2.77
##
## $virginica
## [1] 2.974
```

```
# if data frame contains some NA values in its column,
# specify na.rm to remove NA values.
# tapply(iris$Sepal.Width, iris$Species, mean,rm.na=TRUE)
```

vapply()

- vapply() is similar to sapply(), but has a pre-specified type of return value, so it can be safer (and sometimes faster) to use.

```
vapply(X, FUN, FUN.VALUE)
```

Arguments:

- X: a vector-object object
- FUN: function applied to each element of X
- FUN.VALUE is where you specify the type of data you are expecting. I am expecting each item in the list to return a single numeric value, so FUN.VALUE = numeric(1).

```
# input is a vector
vec <- c(1:10)
vapply(vec, sum, numeric(1))
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
# input is a list
A<-c(1:9)
B<-c(1:12)
C<-c(1:3)
my.lst<-list(A,B,C)
vapply(my.lst, sum, numeric(1))
```

```
## [1] 45 78  6
```

```
my.lst$C<-c("x", "y", "z")
# the following vapply() fails as the expected
# output type is not numeric for the C element
# vapply(lst, min, numeric(1))
```

mapply()

- mapply() in R can be used to apply a functoin to multiple list or vector arguments.

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

- FUN: the function to apply
- ...: arguments to vectorize over
- MoreArgs: a list of other arguments to FUN
- SIMPLIFY: whether or not to reduce the result to a vector
- USE.NAMES: whether or not to use names if the first... argument has names.

Example 1: to create a matrix

```
mapply(rep, 1:3, times=5)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    1    2    3
```

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

```
# compare to method below
matrix(c(rep(1, 5), rep(2, 5), rep(3, 5)), ncol=3)
```

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

Example 2: to multipy corresponding elements in vectors

```
vec1 <- c(1, 2, 3, 4)
vec2 <- c(2, 4, 6, 8)
vec3 <- c(3, 6, 9, 12)

#find max value of each corresponding elements in vectors
mapply(function(val1, val2, val3) val1*val2*val3, vec1, vec2, vec3)
```

```
## [1]   6  48 162 384
```

```
vec1+vec2+vec3
```

```
## [1]  6 12 18 24
```

mclapply(): Parallel versions of lapply()

- mclapply(): Parallel versions of lapply().
- mclapply() returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.
- mclapply() relies on forking and hence is not available on Windows unless mc.cores = 1. mcmapply is a parallelized version of mapply, and mcMap corresponds to Map.

```
mclapply(X, FUN, ...,
         mc.preschedule = TRUE, mc.set.seed = TRUE,
         mc.silent = FALSE, mc.cores = getOption("mc.cores", 2L),
         mc.cleanup = TRUE, mc.allow.recursive = TRUE)
```

- X: a vector (atomic or list) or an expressions vector. Other objects (including classed objects) will be coerced by as.list.
- FUN: the function to be applied to (mclapply) each element of X or (mcmapply) in parallel to.
- mc.cores: the number of cores to use, i.e. at most how many child processes will be run simultaneously. The option is initialized from environment variable MC_CORES if set. Must be at least one, and parallelization requires at least two cores.

Example: Parallel Streams of random numbers

To get random numbers in parallel, we need to use a special random number generator (RNG) designed for parallelization.

```
library(parallel)
ncores <- detectCores()
ncores
```

```
## [1] 10
```

```r
set.seed(42)
mclapply(1:ncores, function(x) rnorm(5), mc.cores = ncores)
```

```
## [[1]]
## [1] -0.2556723  1.4610642 -1.2606742  0.2381550 -1.1472538
##
## [[2]]
## [1]  1.6273406 -1.1928235  0.3404527 -0.1581612 -0.5706706
##
## [[3]]
## [1] -0.24296450 -0.02370808 -0.45275679  0.72366938  2.25936811
##
## [[4]]
## [1] -0.66799241 -0.05668815  1.19456441  0.86185537 -0.25777609
##
## [[5]]
## [1] -1.5975503  0.5921301 -1.1634075 -0.3557754 -0.4678605
##
## [[6]]
## [1]  1.05223056 -0.02181428  0.74934714  1.49913393 -1.59829372
##
## [[7]]
## [1]  0.6463115 -1.5777625  0.7036888 -1.7573174 -0.1403510
##
## [[8]]
## [1] -0.300972325  0.920095763  0.870108923  0.009374493 -0.184688037
##
## [[9]]
## [1]  1.4660515 -0.2857983  0.3424212 -0.8579082  0.9370406
##
## [[10]]
## [1] -0.82530647 -1.55067100 -1.81355689 -0.08742372  2.42667691
```

Reference

1. https://www.guru99.com/r-apply-sapply-tapply.html
2. https://www.stat.umn.edu/geyer/8054/notes/parallel.html