# SLG - apply() Implementation

*Todd Wilson*

*November 28, 2016*

We create some fake data to implement the various functions in the `apply()` family. Note that `datavec1` is a list of three vectors of length 10, `datavec2` is another list of three vectors of length 10 (final vector is the first ten lowercase letters), and `datamat` is the $6 \times 5$ matrix of the first 30 positive integers by column.

```
#some data
data1    <-  1:10
data2    <- 11:20
data3    <- 21:30

#datavec1 is a list of three vectors
datavec1 <- list(data1, data2, data3)
datavec1
```

```
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[2]]
##  [1] 11 12 13 14 15 16 17 18 19 20
##
## [[3]]
##  [1] 21 22 23 24 25 26 27 28 29 30
```

```
length(datavec1)
```

```
## [1] 3
```

```
#a, b, ..., j
datal    <- letters[1:10]

#datavec2 is a list of three vectors (datal is the final )
datavec2 <- list(data1, data2, datal)
datavec2
```

```
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[2]]
##  [1] 11 12 13 14 15 16 17 18 19 20
##
## [[3]]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```
#data2 is all the data
datafull <- 1:30
```

```r
#datamat is a 6x5 matrix
datamat  <- matrix(datafull, nrow=6)
datamat
```

```
##      [,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
```

According to the desciption in the R help pages, `apply()` "returns a vector or array or list of values obtained by applying a function to margins of an array or matrix." We input our data matrix and see that it returns the row means (MARGIN = 1) and column means (MARGIN = 2). Of course, we could find these using `rowMeans()` and `colMeans()`, respectively. These are faster, newer functions. Of course, we are not retricted to using only the mean function.

```r
#apply()

#row means (second argument = 1)
apply(datamat, 1, mean)
```

```
## [1] 13 14 15 16 17 18
```

```r
#column means (second argument = 2)
apply(datamat, 2, mean)
```

```
## [1]  3.5  9.5 15.5 21.5 27.5
```

```r
#could accomplish last two examples with rowMeans() and colMeans()
#this method is newer and faster
rowMeans(datamat)
```

```
## [1] 13 14 15 16 17 18
```

```r
colMeans(datamat)
```

```
## [1]  3.5  9.5 15.5 21.5 27.5
```

```r
#but we can use functions other than the mean
apply(datamat, 2, function(x) x+1)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2    8   14   20   26
## [2,]    3    9   15   21   27
## [3,]    4   10   16   22   28
## [4,]    5   11   17   23   29
## [5,]    6   12   18   24   30
## [6,]    7   13   19   25   31
```

`eapply()`, the environment apply, "applies FUN to the named values from an environment and returns the results as a list." Loosely speaking, we can use the `apply()` function in different environments than the one we are currently in with this function.

```
#eapply()

#define a new environment with variables e1, e2, and e3
new.e    <- new.env()
new.e$e1 <-  1:20
new.e$e2 <- 21:40
new.e$e3 <- 41:60

#apply() cannot grab these means from a different environment
#lapply(new.e, mean)

#...this call actually works, in order to compile Markdown file
#but in a fancier setting it wouldn't

#eapply() can get them
eapply(new.e, mean)
```

```
## $e1
## [1] 10.5
##
## $e2
## [1] 30.5
##
## $e3
## [1] 50.5
```

`lapply()`, the list apply, "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." This is the most basic `apply()` function, and Hadley Wickham's "first functional." This function gets to the heart of what the `apply()` family does.

```
#lapply()

lapply(datavec1, sum)
```

```
## [[1]]
## [1] 55
##
## [[2]]
## [1] 155
##
## [[3]]
## [1] 255
```

`sapply()`, the simple (?) apply, "is a user-friendly version and wrapper of lapply." It is a special version of `lapply()` that, instead of returning in a list format, simply returns just a vector of values.

```
#sapply()

sapply(datavec1, mean)
```

```
## [1]  5.5 15.5 25.5
```

`vapply()`, the value-return-is-pre-specified apply, "is similar to sapply, but has a pre-specified type of return value, so it can be safer (and sometimes faster) to use." It is another special version of `lapply()`. An example of this is provided in the documentation using the five-number summary function `fivesum`. We specify that the data will return in the table format seem below. Oddly, we need initial values beside the row headings ("= 0" below) - it would run with any initial value ("= 1" would have worked, for example), but will not run without one.

```
#vapply()

#for some reason, fivenum row names need initial values
vapply(datavec1, fivenum, c(Min = 0, Q1 = 0, Med = 0, Q3 = 0, Max = 0))
```

```
##     [,1] [,2] [,3]
## Min  1.0 11.0 21.0
## Q1   3.0 13.0 23.0
## Med  5.5 15.5 25.5
## Q3   8.0 18.0 28.0
## Max 10.0 20.0 30.0
```

`mapply()`, the multivariate apply, "is a multivariate version of sapply [that] applies FUN to the first elements of each [argument]." Below, we apply across all three odd-numbered columns of `datamat` and ask it to return the maximum value of each index (which always appears in the last column by design) minus the minimum value of each index (which always appears in the first columnn by design). Note that each element of the returned 6-vector is 24 because all the values in the last column for each index are 24 more than the corresponding index in the first column.

```
#mapply()

#just multivariate sapply()
#note that FUN appears first in the syntax
mapply(mean, datavec1)
```

```
## [1]  5.5 15.5 25.5
```

```
mapply(function(a, b, c) {max(a, b, c) - min(a, b, c)}, datamat[,1], datamat[,3], datamat[,5])
```

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

`rapply()`, the recursive apply, "is a recursive version of lapply." Unfortunately, this name is not very intuitive. The magic of `rapply()` is two-fold: (1) the "how" argument can return listed (= "list" or = "replace") or unlisted (= "unlist") values, as with `lapply()` and `sapply()`, respectively; and (2) the "class" argument allows us to specify over what kind of values we wish to apply the function. For example, `datavec1` returns the same thing with `lapply()` and `rapply()` - R can square these integers. But, it cannot square the letters from `datavec2`, so `lapply` fails to render. However, with `rapply()`, we can specify to only square integers, which it gladly does.

```
#rapply()

lapply(datavec1, function(x) {x^2})
```

```
## [[1]]
##  [1]   1   4   9  16  25  36  49  64  81 100
##
## [[2]]
##  [1] 121 144 169 196 225 256 289 324 361 400
##
## [[3]]
##  [1] 441 484 529 576 625 676 729 784 841 900
```

```
rapply(datavec1, function(x) {x^2}, class=c("integer"), how="list")
```

```
## [[1]]
##  [1]   1   4   9  16  25  36  49  64  81 100
##
## [[2]]
##  [1] 121 144 169 196 225 256 289 324 361 400
##
## [[3]]
##  [1] 441 484 529 576 625 676 729 784 841 900
```

```
#for non-numeric input, lapply() gives an error
#lapply(datavec2, function(x) {x^2})
#Error in x^2 : non-numeric argument to binary operator
#rapply() knows to only apply to class integer
rapply(datavec2, function(x) {x^2}, class=c("integer"), how="replace")
```

```
## [[1]]
##  [1]   1   4   9  16  25  36  49  64  81 100
##
## [[2]]
##  [1] 121 144 169 196 225 256 289 324 361 400
##
## [[3]]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

`tapply()`, the table apply, "appl[ies] a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors." This is the function to use when finding means when subsetting by a factor. We reintroduce the iris data set and compute the means of the sepal lengths for each species.

```
#tapply()
attach(iris)
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```r
tail(iris)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 145          6.7         3.3          5.7         2.5 virginica
## 146          6.7         3.0          5.2         2.3 virginica
## 147          6.3         2.5          5.0         1.9 virginica
## 148          6.5         3.0          5.2         2.0 virginica
## 149          6.2         3.4          5.4         2.3 virginica
## 150          5.9         3.0          5.1         1.8 virginica
```

```r
tapply(iris[,1], iris$Species, mean)
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```