R has many *apply functions which are ably described in the help files (e.g. `?apply`). There are enough of them, though, that beginning useRs may have difficulty deciding which one is appropriate for their situation or even remembering them all. They may have a general sense that "I should be using an *apply function here", but it can be tough to keep them all straight at first.

Despite the fact (noted in other answers) that much of the functionality of the *apply family is covered by the extremely popular `plyr` package, the base functions remain useful and worth knowing.

This answer is intended to act as a sort of **signpost** for new useRs to help direct them to the correct *apply function for their particular problem. Note, this is **not** intended to simply regurgitate or replace the R documentation! The hope is that this answer helps you to decide which *apply function suits your situation and then it is up to you to research it further. With one exception, performance differences will not be addressed.

- **apply** - *When you want to apply a function to the rows or columns of a matrix (and higher-dimensional analogues).*

  ```
  # Two dimensional matrix
  M <- matrix(seq(1,16), 4, 4)

  # apply min to rows
  apply(M, 1, min)
  [1] 1 2 3 4

  # apply max to columns
  apply(M, 2, max)
  [1]  4  8 12 16

  # 3 dimensional array
  M <- array( seq(32), dim = c(4,4,2))

  # Apply sum across each M[*, , ] - i.e Sum across 2nd and 3rd dimension
  apply(M, 1, sum)
  # Result is one-dimensional
  [1] 120 128 136 144

  # Apply sum across each M[*, *, ] - i.e Sum across 3rd dimension
  apply(M, c(1,2), sum)
  # Result is two-dimensional
       [,1] [,2] [,3] [,4]
  [1,]   18   26   34   42
  [2,]   20   28   36   44
  [3,]   22   30   38   46
  [4,]   24   32   40   48
  ```

  If you want row/column means or sums for a 2D matrix, be sure to investigate the highly optimized, lightning-quick `colMeans`, `rowMeans`, `colSums`, `rowSums`.

- **lapply** - *When you want to apply a function to each element of a list in turn and get a list back.*

  This is the workhorse of many of the other *apply functions. Peel back their code and you will often find `lapply` underneath.

  ```
  x <- list(a = 1, b = 1:3, c = 10:100)
  lapply(x, FUN = length)
  $a
  [1] 1
  $b
  [1] 3
  $c
  [1] 91

  lapply(x, FUN = sum)
  $a
  [1] 1
  $b
  [1] 6
  $c
  [1] 5005
  ```

- **sapply** - *When you want to apply a function to each element of a list in turn, but you want a* ***vector*** *back, rather than a list.*

  If you find yourself typing `unlist(lapply(...))`, stop and consider `sapply`.

  ```
  x <- list(a = 1, b = 1:3, c = 10:100)
  #Compare with above; a named vector, not a list
  sapply(x, FUN = length)
  a  b  c
  1  3 91

  sapply(x, FUN = sum)
  a    b    c
  1    6 5005
  ```

  In more advanced uses of `sapply` it will attempt to coerce the result to a multi-dimensional array, if appropriate. For example, if our function returns vectors of the same length, `sapply` will use them as columns of a matrix:

  ```
  sapply(1:5,function(x) rnorm(3,x))
  ```

  If our function returns a 2 dimensional matrix, `sapply` will do essentially the same thing, treating each returned matrix as a single long vector:

  ```
  sapply(1:5,function(x) matrix(x,2,2))
  ```

  Unless we specify `simplify = "array"`, in which case it will use the individual matrices to build a multi-dimensional array:

  ```
  sapply(1:5,function(x) matrix(x,2,2), simplify = "array")
  ```

  Each of these behaviors is of course contingent on our function returning vectors or matrices of the same length or dimension.

- **vapply** - *When you want to use `sapply` but perhaps need to squeeze some more speed out of your code.*

  For `vapply`, you basically give R an example of what sort of thing your function will return, which can save some time coercing returned values to fit in a single atomic vector.

  ```
  x <- list(a = 1, b = 1:3, c = 10:100)
  #Note that since the advantage here is mainly speed, this
  # example is only for illustration. We're telling R that
  # everything returned by length() should be an integer of
  # length 1.
  vapply(x, FUN = length, FUN.VALUE = 0L)
  a  b  c
  1  3 91
  ```

- **mapply** - *For when you have several data structures (e.g. vectors, lists) and you want to apply a function to the 1st elements of each, and then the 2nd elements of each, etc., coercing the result to a vector/array as in `sapply`.*

  This is multivariate in the sense that your function must accept multiple arguments.

  ```
  #Sums the 1st elements, the 2nd elements, etc.
  mapply(sum, 1:5, 1:5, 1:5)
  [1]  3  6  9 12 15
  #To do rep(1,4), rep(2,3), etc.
  mapply(rep, 1:4, 4:1)
  [[1]]
  [1] 1 1 1 1

  [[2]]
  [1] 2 2 2

  [[3]]
  [1] 3 3

  [[4]]
  [1] 4
  ```

- **Map** - *A wrapper to `mapply` with `SIMPLIFY = FALSE`, so it is guaranteed to return a list.*
- `Map(sum, 1:5, 1:5, 1:5)`
- `[[1]]`
- `[1] 3`
-
- `[[2]]`
- `[1] 6`
-
- `[[3]]`
- `[1] 9`
-
- `[[4]]`
- `[1] 12`
-
- `[[5]]`
  `[1] 15`

- **rapply** - *For when you want to apply a function to each element of a **nested list** structure, recursively.*

  To give you some idea of how uncommon `rapply` is, I forgot about it when first posting this answer! Obviously, I'm sure many people use it, but YMMV. `rapply` is best illustrated with a user-defined function to apply:

  ```
  #Append ! to string, otherwise increment
  myFun <- function(x){
      if (is.character(x)){
      return(paste(x,"!",sep=""))
      }
      else{
      return(x + 1)
      }
  }

  #A nested list structure
  l <- list(a = list(a1 = "Boo", b1 = 2, c1 = "Eeek"),
            b = 3, c = "Yikes",
            d = list(a2 = 1, b2 = list(a3 = "Hey", b3 = 5)))


  #Result is named vector, coerced to character
  rapply(l,myFun)

  #Result is a nested list like l, with values altered
  rapply(l, myFun, how = "replace")
  ```

- **tapply** - *For when you want to apply a function to **subsets** of a vector and the subsets are defined by some other vector, usually a factor.*

  The black sheep of the *apply family, of sorts. The help file's use of the phrase "ragged array" can be a bit confusing, but it is actually quite simple.

  A vector:

  ```
  x <- 1:20
  ```

  A factor (of the same length!) defining groups:

  ```
  y <- factor(rep(letters[1:5], each = 4))
  ```

  Add up the values in `x` within each subgroup defined by `y`:

  ```
  tapply(x, y, sum)
   a  b  c  d  e
  10 26 42 58 74
  ```

  More complex examples can be handled where the subgroups are defined by the unique combinations of a list of several factors. `tapply` is similar in spirit to the split-apply-combine functions that are common in R (`aggregate`, `by`, `ave`, `ddply`, etc.) Hence its black sheep status.