R Grouping functions: sapply vs. lapply vs. apply. vs. tapply vs. by vs. aggregate

Whenever I want to do something "map"py in `R`, I usually try to use a function in the `apply` family. (Side question: I still haven't learned `plyr` or `reshape` -- would `plyr` or `reshape` replace all of these entirely?)

However, I've never quite understood the differences between them [how { `sapply`, `lapply`, etc.} apply the function to the input/grouped input, what the output will look like, or even what the input can be], so I often just go through them all until I get what I want.

Can someone explain how to use which one when?

[My current (probably incorrect/incomplete) understanding is...

1. `sapply(vec, f)` : input is a vector. output is a vector/matrix, where element `i` is `f(vec[i])` [giving you a matrix if `f` has a multi-element output]
2. `lapply(vec, f)` : same as `sapply`, but output is a list?
3. `apply(matrix, 1/2, f)` : input is a matrix. output is a vector, where element `i` is f(row/col i of the matrix)
4. `tapply(vector, grouping, f)` : output is a matrix/array, where an element in the matrix/array is the value of `f` at a grouping `g` of the vector, and `g` gets pushed to the row/col names
5. `by(dataframe, grouping, f)` : let `g` be a grouping. apply `f` to each column of the group/dataframe. pretty print the grouping and the value of `f` at each column.
6. `aggregate(matrix, grouping, f)` : similar to `by`, but instead of pretty printing the output, aggregate sticks everything into a dataframe.]

r plyr r-faq sapply tapply

edited Apr 24 at 23:51
Rup
17.2k 4 37 62

asked Aug 17 '10 at 18:31
grautur
4,895 11 51 80

10 to your side question: for many things plyr is a direct replacement for `*apply()` and `by`. plyr (at least to me) seems much more consistent in that I always know exactly what data format it expects and exactly what it will spit out. That saves me a lot of hassle. – JD Long Aug 17 '10 at 18:40

6 Also, I'd recommend adding: `doBy` and the selection & apply capabilities of `data.table`. – Iterator Oct 10 '11 at 15:23

`sapply` is just `lapply` with the addition of `simplify2array` on the output. `apply` does coerce to atomic vector, but output can be vector or list. `by` splits dataframes into sub-dataframes, but it doesn't use `f` on columns separately. Only if there is a method for 'data.frame'-class might `f` get column-wise applied by `by`. `aggregate` is generic so different methods exist for different classes of the first argument. – BondedDust Jan 24 '13 at 21:18

add comment

4 Answers

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
  ```

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

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.

edited Apr 22 at 22:09

answered Aug 21 '11 at 22:50
joran
62k 9 92 136

36 +10 if I could. Splendid answer. – Ari B. Friedman Sep 14 '11 at 3:38

11 Believe you will find that `by` is pure split-lapply and `aggregate` is `tapply` at their cores. I think black sheep make excellent fabric. – BondedDust Sep 14 '11 at 3:42

5 Fantastic response! This should be part of the official R documentation :). One tiny suggestion: perhaps add some bullets on using `aggregate` and `by` as well? (I finally understand them after your description!, but they're pretty common, so it might be useful to separate out and have some specific examples for those two functions.) – grautur Sep 14 '11 at 18:54

1 @grautur I was actively pruning things from this answer to avoid it being (a) too long and (b) a re-write of the documentation. I decided that while `aggregate` , `by` , etc. are based on *apply functions, the way you approach using them is different enough from a users perspective that they ought to be summarized in a separate answer. I may attempt that if I have time, or maybe someone else will beat me to it and earn my upvote. – joran Sep 14 '11 at 23:03

Yeah ... what is a ragged array, anyway? – isomorphismes Oct 10 '11 at 6:01

show **7** more comments

On the side note, here is how the various `plyr` functions correspond to the base `*apply` functions (from the intro to plyr document from the plyr webpage http://had.co.nz/plyr/)

```
Base function   Input   Output   plyr function
---------------------------------------
aggregate        d       d       ddply + colwise
apply            a       a/l     aaply / alply
by               d       l       dlply
lapply           l       l       llply
mapply           a       a/l     maply / mlply
replicate        r       a/l     raply / rlply
sapply           l       a       laply
```

One of the goals of `plyr` is to provide consistent naming conventions for each of the functions, encoding the input and output data types in the function name. It also provides consistency in output, in that output from `dlply()` is easily passable to `ldply()` to produce useful output, etc.

Conceptually, learning `plyr` is no more difficult than understanding the base `*apply` functions.

`plyr` and `reshape` functions have replaced almost all of these functions in my every day use. But, also from the Intro to Plyr document:

> Related functions `tapply` and `sweep` have no corresponding function in `plyr`, and remain useful. `merge` is useful for combining summaries with the original data.

answered Aug 17 '10 at 19:20
JoFrhwld
3,877 14 18

8 When I started learning R from scratch I found plyr MUCH easier to learn than the `*apply()` family of functions. For me, `ddply()` was very intuitive as I was familiar with SQL aggregation functions. `ddply()` became my hammer for solving many problems, some of which could have been better solved with other commands. – JD Long Aug 17 '10 at 19:23

1 I guess I figured that the concept behind `plyr` functions is similar to `*apply` functions, so if you can do one, you can do the other, but `plyr` functions are easier to remember. But I totally agree on the `ddply()` hammer! – JoFrhwld Aug 17 '10 at 19:36

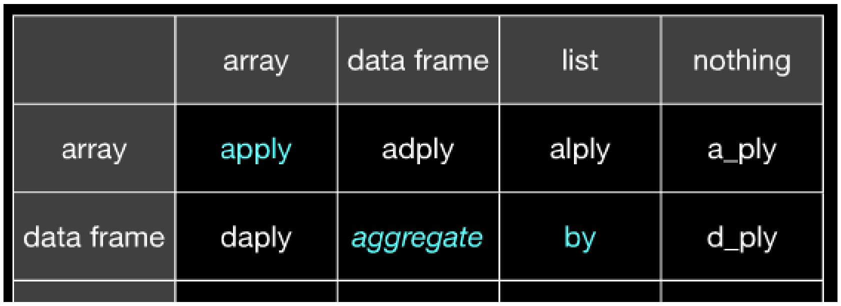
1 Got it, I'll have to finally pick up plyr soon! Its prefix naming alone is gold... – grautur Aug 17 '10 at 22:28

1 +1 For adding the note about tapply and sweep. Great to know both what plyr can and can't do. – John Robertson Jun 22 '12 at 19:01

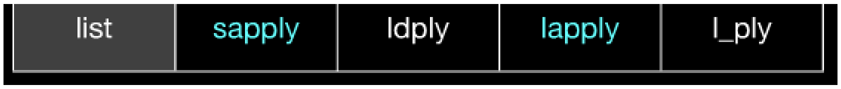
1 The plyr package has the `join()` function that performs tasks similar to merge. Perhaps it's more to the point to mention it in the context of plyr. – Martín Bel Jan 2 at 23:04

show **1** more comment

From slide 21 of http://www.slideshare.net/hadley/plyr-one-data-analytic-strategy:



	array	data frame	list	nothing
array	apply	adply	alply	a_ply
data frame	daply	*aggregate*	by	d_ply



list	sapply	ldply	lapply	l_ply

(Hopefully it's clear that `apply` corresponds to @Hadley's `aaply` and `aggregate` corresponds to @Hadley's `ddply` etc. Slide 20 of the same slideshare will clarify if you don't get it from this image.)

(on the left is input, on the top is output)

edited Feb 15 '12 at 23:42
user56
470 6 19

answered Oct 9 '11 at 5:29
isomorphismes
1,751 1 16 31

add comment

First start with Joran's excellent answer -- doubtful anything can better that.

Then the following mnemonics may help remember the distinctions between each. Whilst some are obvious, others may be less so --- for these you'll find justification in Joran's discussions.

Mnemonics

- `lapply` is a *list* apply (acts on a list and returns a list or list of lists)
- `sapply` is a *shaped* apply (shapes output into a table)
- `vapply` is a *vectored* apply (vectored sapply for speed, returns atomic vectors)
- `rapply` is a *recursive* apply (for recursive lists, i.e. lists within lists)
- `tapply` is a *tagged* apply (the tags identify the subsets)
- `apply` is *generic*: (applies a function to a matrix's rows or columns)

edited Apr 27 at 16:41

answered Apr 25 at 0:20
Assad Ebrahim
1,129 1 12 26

add comment

Not the answer you're looking for? Browse other questions tagged r plyr r-faq sapply tapply or ask your own question.