Grouping functions (tapply, by, aggregate) and the *apply family

Whenever I want to do something "map"py in R, I usually try to use a function in the `apply` family.

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.

Side question: I still haven't learned plyr or reshape -- would `plyr` or `reshape` replace all of these entirely?

r sapply tapply r-faq

edited Sep 19 at 20:28
Frank
44.3k 5 41 103

asked Aug 17 '10 at 18:31
grautur
10.5k 26 78 112

25 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

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

5 `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. – 42- Jan 24 '13 at 21:18

5 Mnemonic: l is for 'list', s is for 'simplifying', t is for 'per type' (each level of the grouping is a type) – Lutz Prechelt Sep 16 '14 at 13:20

9 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); not generally advisable for data frames as it will coerce to a matrix first.*

```
# 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.

  edited May 23 at 12:34
  Community ♦
  1 1

  answered Aug 21 '11 at 22:50
  
  joran
  116k 14 264 322

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

15 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

3 @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

2 also, `?Map` as a relative of `mapply` – baptiste Feb 16 '12 at 5:53

3 @jsanders - I wouldn't agree with that at all. `data.frame` s are an absolutely central part of R and as a `list` object are frequently manipulated using `lapply` particularly. They also act as containers for grouping vectors/factors of many types together in a traditional rectangular dataset. While `data.table` and `plyr` might add a certain type of syntax that some might find more comfortable, they are extending and acting on `data.frame` s respectively. – thelatemail Aug 20 '14 at 6:08

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
6,407 3 24 29

13 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

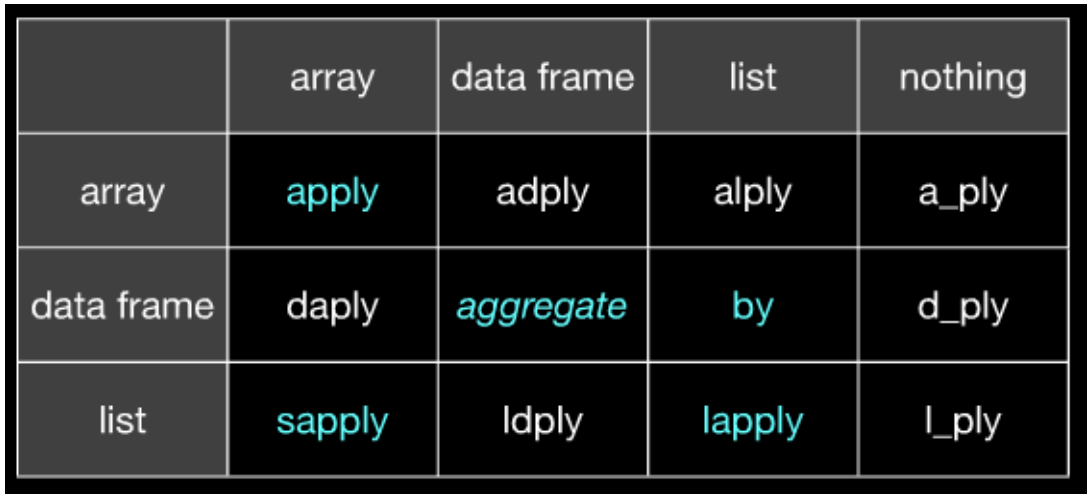
1 I guess I figured that the concept behind `plyr` functions is similar to `*apply` functions, so if you can do one, you can do the other, but `plyr` functions are easier to remember. But I totally agree on the `ddply()` hammer! – JoFrhwld Aug 17 '10 at 19:36

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. – marbel Jan 2 '14 at 23:04

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



(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

userJT
3,294 6 38 61

answered Oct 9 '11 at 5:29

isomorphismes
4,262 7 36 56

2 is there a typo in the slide? The top left cell should be aaply – JHowIX Sep 16 '16 at 18:16

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

Then the following mnemonics may help to 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 which acts on a list or vector and returns a list.
- `sapply` is a *simple* `lapply` (function defaults to returning a vector or matrix when possible)
- `vapply` is a *verified apply* (allows the return object type to be prespecified)
- `rapply` is a *recursive* apply for nested lists, i.e. lists within lists
- `tapply` is a *tagged* apply where the tags identify the subsets
- `apply` is *generic*: applies a function to a matrix's rows or columns (or, more generally, to dimensions of an array)

Building the Right Background

If using the `apply` family still feels a bit alien to you, then it might be that you're missing a key point of view.

These two articles can help. They provide the necessary background to motivate the **functional programming techniques** that are being provided by the `apply` family of functions.

Users of Lisp will recognise the paradigm immediately. If you're not familiar with Lisp, once you get your head around FP, you'll have gained a powerful point of view for use in R -- and `apply` will make a lot more sense.

- Advanced R: Functional Programming, by Hadley Wickham
- Simple Functional Programming in R, by Michael Barton

edited May 23 at 10:31
Community ♦
1 1

answered Apr 25 '14 at 0:20

Assad Ebrahim
3,467 5 28 59

mnemonic for `vapply` is off... how about *verified* apply since the output type is certain. – MichaelChirico Jan 7 '16 at 2:32

Since I realized that (the very excellent) answers of this post lack of `by` and `aggregate` explanations. Here is my contribution.

BY

The `by` function, as stated in the documentation can be though, as a "wrapper" for `tapply`. The power of `by` arises when we want to compute a task that `tapply` can't handle. One example is this code:

```
ct <- tapply(iris$Sepal.Width , iris$Species , summary )
cb <- by(iris$Sepal.Width , iris$Species , summary )

 cb
iris$Species: setosa
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.300   3.200   3.400   3.428   3.675   4.400
------------------------------------------------------------
iris$Species: versicolor
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.000   2.525   2.800   2.770   3.000   3.400
------------------------------------------------------------
iris$Species: virginica
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.200   2.800   3.000   2.974   3.175   3.800


ct
$setosa
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.300   3.200   3.400   3.428   3.675   4.400
```

```
$versicolor
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.000   2.525   2.800   2.770   3.000   3.400

$virginica
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.200   2.800   3.000   2.974   3.175   3.800
```

If we print these two objects, `ct` and `cb`, we "essentially" have the same results and the only differences are in how they are shown and the different `class` attributes, respectively `by` for `cb` and `array` for `ct`.

As I've said, the power of `by` arises when we can't use `tapply`; the following code is one example:

```
 tapply(iris, iris$Species, summary )
Error in tapply(iris, iris$Species, summary) :
  arguments must have same length
```

R says that arguments must have the same lengths, say "we want to calculate the `summary` of all variable in `iris` along the factor `Species`": but R just can't do that because it does not know how to handle.

With the `by` function R dispatch a specific method for `data frame` class and then let the `summary` function works even if the length of the first argument (and the type too) are different.

```
bywork <- by(iris, iris$Species, summary )

bywork
iris$Species: setosa
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
 Min.   :4.300   Min.   :2.300   Min.   :1.000   Min.   :0.100   setosa    :50
 1st Qu.:4.800   1st Qu.:3.200   1st Qu.:1.400   1st Qu.:0.200   versicolor: 0
 Median :5.000   Median :3.400   Median :1.500   Median :0.200   virginica : 0
 Mean   :5.006   Mean   :3.428   Mean   :1.462   Mean   :0.246
 3rd Qu.:5.200   3rd Qu.:3.675   3rd Qu.:1.575   3rd Qu.:0.300
 Max.   :5.800   Max.   :4.400   Max.   :1.900   Max.   :0.600
------------------------------------------------------------
iris$Species: versicolor
  Sepal.Length    Sepal.Width     Petal.Length   Petal.Width          Species
 Min.   :4.900   Min.   :2.000   Min.   :3.00   Min.   :1.000   setosa    : 0
 1st Qu.:5.600   1st Qu.:2.525   1st Qu.:4.00   1st Qu.:1.200   versicolor:50
 Median :5.900   Median :2.800   Median :4.35   Median :1.300   virginica : 0
 Mean   :5.936   Mean   :2.770   Mean   :4.26   Mean   :1.326
 3rd Qu.:6.300   3rd Qu.:3.000   3rd Qu.:4.60   3rd Qu.:1.500
```

```
 Max.   :7.000   Max.   :3.400   Max.   :5.10   Max.   :1.800
------------------------------------------------------------
iris$Species: virginica
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
 Min.   :4.900   Min.   :2.200   Min.   :4.500   Min.   :1.400   setosa    : 0
 1st Qu.:6.225   1st Qu.:2.800   1st Qu.:5.100   1st Qu.:1.800   versicolor: 0
 Median :6.500   Median :3.000   Median :5.550   Median :2.000   virginica :50
 Mean   :6.588   Mean   :2.974   Mean   :5.552   Mean   :2.026
 3rd Qu.:6.900   3rd Qu.:3.175   3rd Qu.:5.875   3rd Qu.:2.300
 Max.   :7.900   Max.   :3.800   Max.   :6.900   Max.   :2.500
```

it works indeed and the result is very surprising. It is an object of class `by` that along `Species` (say, for each of them) computes the `summary` of each variable.

Note that if the first argument is a `data frame` , the dispatched function must have a method for that class of objects. For example is we use this code with the `mean` function we will have this code that has no sense at all:

```
 by(iris, iris$Species, mean)
iris$Species: setosa
[1] NA
------------------------------------------
iris$Species: versicolor
[1] NA
------------------------------------------
iris$Species: virginica
[1] NA
Warning messages:
1: In mean.default(data[x, , drop = FALSE], ...) :
  argument is not numeric or logical: returning NA
2: In mean.default(data[x, , drop = FALSE], ...) :
  argument is not numeric or logical: returning NA
3: In mean.default(data[x, , drop = FALSE], ...) :
  argument is not numeric or logical: returning NA
```

AGGREGATE

`aggregate` can be seen as another a different way of use `tapply` if we use it in such a way.

```
at <- tapply(iris$Sepal.Length , iris$Species , mean)
ag <- aggregate(iris$Sepal.Length , list(iris$Species), mean)

 at
    setosa versicolor  virginica
     5.006      5.936      6.588
 ag
```

```
     Group.1     x
1     setosa 5.006
2 versicolor 5.936
3  virginica 6.588
```

The two immediate differences are that the second argument of `aggregate` **must** be a list while `tapply` **can** (not mandatory) be a list and that the output of `aggregate` is a data frame while the one of `tapply` is an `array`.

The power of `aggregate` is that it can handle easily subsets of the data with `subset` argument and that it has methods for `ts` objects and `formula` as well.

These elements make `aggregate` easier to work with that `tapply` in some situations. Here are some examples (available in documentation):

```
ag <- aggregate(len ~ ., data = ToothGrowth, mean)

 ag
  supp dose   len
1   OJ  0.5 13.23
2   VC  0.5  7.98
3   OJ  1.0 22.70
4   VC  1.0 16.77
5   OJ  2.0 26.06
6   VC  2.0 26.14
```

We can achieve the same with `tapply` but the syntax is slightly harder and the output (in some circumstances) less readable:

```
att <- tapply(ToothGrowth$len, list(ToothGrowth$dose, ToothGrowth$supp), mean)

 att
       OJ    VC
0.5 13.23  7.98
1   22.70 16.77
2   26.06 26.14
```

There are other times when we can't use `by` or `tapply` and we have to use `aggregate`.

```
 ag1 <- aggregate(cbind(Ozone, Temp) ~ Month, data = airquality, mean)

 ag1
  Month    Ozone     Temp
1     5 23.61538 66.73077
2     6 29.44444 78.22222
```

```
3     7 59.11538 83.88462
4     8 59.96154 83.96154
5     9 31.44828 76.89655
```

We cannot obtain the previous result with `tapply` in one call but we have to calculate the mean along `Month` for each elements and then combine them (also note that we have to call the `na.rm = TRUE` , because the `formula` methods of the `aggregate` function has by default the `na.action = na.omit`):

```
ta1 <- tapply(airquality$Ozone, airquality$Month, mean, na.rm = TRUE)
ta2 <- tapply(airquality$Temp, airquality$Month, mean, na.rm = TRUE)

 cbind(ta1, ta2)
       ta1      ta2
5 23.61538 65.54839
6 29.44444 79.10000
7 59.11538 83.90323
8 59.96154 83.96774
9 31.44828 76.90000
```

while with `by` we just can't achieve that in fact the following function call returns an error (but most likely it is related to the supplied function, `mean`):

```
by(airquality[c("Ozone", "Temp")], airquality$Month, mean, na.rm = TRUE)
```

Other times the results are the same and the differences are just in the class (and then how it is shown/printed and not only -- example, how to subset it) object:

```
byagg <- by(airquality[c("Ozone", "Temp")], airquality$Month, summary)
aggagg <- aggregate(cbind(Ozone, Temp) ~ Month, data = airquality, summary)
```

The previous code achieve the same goal and results, at some points what tool to use is just a matter of personal tastes and needs; the previous two objects have very different needs in terms of subsetting.

edited Aug 28 '15 at 10:03

answered Aug 28 '15 at 2:28

SabDeM

5,210 1 11 32

There are lots of great answers which discuss differences in the use cases for each function.

None of the answer discuss the differences in performance. That is reasonable cause various functions expects various input and produces various output, yet most of them have a general common objective to evaluate by series/groups. My answer is going to focus on performance. Due to above the input creation from the vectors is included in the timing, also the `apply` function is not measured.

I have tested two different functions `sum` and `length` at once. Volume tested is 50M on input and 50K on output. I have also included two currently popular packages which were not widely used at the time when question was asked, `data.table` and `dplyr` . Both are definitely worth to look if you are aiming for good performance.

```
library(dplyr)
library(data.table)
set.seed(123)
n = 5e7
k = 5e5
x = runif(n)
grp = sample(k, n, TRUE)

timing = list()

# sapply
timing[["sapply"]] = system.time({
    lt = split(x, grp)
    r.sapply = sapply(lt, function(x) list(sum(x), length(x)), simplify = FALSE)
})

# lapply
timing[["lapply"]] = system.time({
    lt = split(x, grp)
    r.lapply = lapply(lt, function(x) list(sum(x), length(x)))
})

# tapply
timing[["tapply"]] = system.time(
    r.tapply <- tapply(x, list(grp), function(x) list(sum(x), length(x)))
)

# by
timing[["by"]] = system.time(
    r.by <- by(x, list(grp), function(x) list(sum(x), length(x)), simplify = FALSE)
)

# aggregate
timing[["aggregate"]] = system.time(
    r.aggregate <- aggregate(x, list(grp), function(x) list(sum(x), length(x)),
```

```
    simplify = FALSE)
    )

    # dplyr
    timing[["dplyr"]] = system.time({
        df = data_frame(x, grp)
        r.dplyr = summarise(group_by(df, grp), sum(x), n())
    })

    # data.table
    timing[["data.table"]] = system.time({
        dt = setnames(setDT(list(x, grp)), c("x","grp"))
        r.data.table = dt[, .(sum(x), .N), grp]
    })

    # all output size match to group count
    sapply(list(sapply=r.sapply, lapply=r.lapply, tapply=r.tapply, by=r.by,
    aggregate=r.aggregate, dplyr=r.dplyr, data.table=r.data.table),
           function(x) (if(is.data.frame(x)) nrow else length)(x)==k)
    #    sapply     lapply     tapply         by  aggregate      dplyr data.table
    #      TRUE       TRUE       TRUE       TRUE       TRUE       TRUE       TRUE


    # print timings
    as.data.table(sapply(timing, `[[`, "elapsed"), keep.rownames = TRUE
                  )[,.(fun = V1, elapsed = V2)
                    ][order(-elapsed)]
    #          fun elapsed
    #1:  aggregate 109.139
    #2:         by  25.738
    #3:      dplyr  18.978
    #4:     tapply  17.006
    #5:     lapply  11.524
    #6:     sapply  11.326
    #7: data.table   2.686
```

edited Dec 8 '15 at 22:50

answered Dec 8 '15 at 22:42

jangorecki
6,444 2 22 75

Is it normal that dplyr is lower than the applt functions ? – Dimitri Petrenko Jun 8 '16 at 9:35

1 @DimitriPetrenko I don't think so, not sure why it is here. It is best to test against your own data, as there are many factors that comes into play. – jangorecki Jun 8 '16 at 11:48

It is maybe worth mentioning `ave`. `ave` is `tapply`'s friendly cousin. It returns results in a form that you can plug straight back into your data frame.

```
dfr <- data.frame(a=1:20, f=rep(LETTERS[1:5], each=4))
means <- tapply(dfr$a, dfr$f, mean)
##  A    B    C    D    E
## 2.5  6.5 10.5 14.5 18.5

## great, but putting it back in the data frame is another line:

dfr$m <- means[dfr$f]

dfr$m2 <- ave(dfr$a, dfr$f, FUN=mean) # NB argument name FUN is needed!
dfr
##   a f    m   m2
##   1 A  2.5  2.5
##   2 A  2.5  2.5
##   3 A  2.5  2.5
##   4 A  2.5  2.5
##   5 B  6.5  6.5
##   6 B  6.5  6.5
##   7 B  6.5  6.5
##   ...
```

There is nothing in the base package that works like `ave` for whole data frames (as `by` is like `tapply` for data frames). But you can fudge it:

```
dfr$foo <- ave(1:nrow(dfr), dfr$f, FUN=function(x) {
    x <- dfr[x,]
    sum(x$m*x$m2)
})
dfr
##     a f    m   m2    foo
## 1   1 A  2.5  2.5    25
## 2   2 A  2.5  2.5    25
## 3   3 A  2.5  2.5    25
## ...
```

answered Nov 6 '14 at 0:00

dash2

1,560 7 22

Despite all the great answers here, there are 2 more base functions that deserve to be

mentioned, the useful `outer` function and the obscure `eapply` function

outer

`outer` is a very useful function hidden as a more mundane one. If you read the help for `outer` its description says:

```
The outer product of the arrays X and Y is the array A with dimension
c(dim(X), dim(Y)) where element A[c(arrayindex.x, arrayindex.y)] =
FUN(X[arrayindex.x], Y[arrayindex.y], ...).
```

which makes it seem like this is only useful for linear algebra type things. However, it can be used much like `mapply` to apply a function to two vectors of inputs. The difference is that `mapply` will apply the function to the first two elements and then the second two etc, whereas `outer` will apply the function to every combination of one element from the first vector and one from the second. For example:

```
 A<-c(1,3,5,7,9)
 B<-c(0,3,6,9,12)

mapply(FUN=pmax, A, B)

> mapply(FUN=pmax, A, B)
[1]  1  3  6  9 12

outer(A,B, pmax)

 > outer(A,B, pmax)
      [,1] [,2] [,3] [,4] [,5]
 [1,]    1    3    6    9   12
 [2,]    3    3    6    9   12
 [3,]    5    5    6    9   12
 [4,]    7    7    7    9   12
 [5,]    9    9    9    9   12
```

I have personally used this when I have a vector of values and a vector of conditions and wish to see which values meet which conditions.

eapply

`eapply` is like `lapply` except that rather than applying a function to every element in a list, it applies a function to every element in an environment. For example if you want to find a list of user defined functions in the global environment:

```
A<-c(1,3,5,7,9)
B<-c(0,3,6,9,12)
C<-list(x=1, y=2)
D<-function(x){x+1}

> eapply(.GlobalEnv, is.function)
$A
[1] FALSE

$B
[1] FALSE

$C
[1] FALSE

$D
[1] TRUE
```

Frankly I don't use this very much but if you are building a lot of packages or create a lot of environments it may come in handy.

edited Jun 3 '16 at 13:37

answered May 16 '16 at 3:59

John Paul

5,942 2 26 49

I recently discovered the rather useful `sweep` function and add it here for the sake of completeness:

sweep

The basic idea is to *sweep* through an array row- or column-wise and return a modified array. An example will make this clear (source: datacamp):

Let's say you have a matrix and want to standardize it column-wise:

```
dataPoints <- matrix(4:15, nrow = 4)

# Find means per column with `apply()`
dataPoints_means <- apply(dataPoints, 2, mean)

# Find standard deviation with `apply()`
dataPoints_sdev <- apply(dataPoints, 2, sd)
```

```
# Center the points
dataPoints_Trans1 <- sweep(dataPoints, 2, dataPoints_means,"-")
print(dataPoints_Trans1)
##      [,1] [,2] [,3]
## [1,] -1.5 -1.5 -1.5
## [2,] -0.5 -0.5 -0.5
## [3,]  0.5  0.5  0.5
## [4,]  1.5  1.5  1.5
# Return the result
dataPoints_Trans1
##      [,1] [,2] [,3]
## [1,] -1.5 -1.5 -1.5
## [2,] -0.5 -0.5 -0.5
## [3,]  0.5  0.5  0.5
## [4,]  1.5  1.5  1.5
# Normalize
dataPoints_Trans2 <- sweep(dataPoints_Trans1, 2, dataPoints_sdev, "/")

# Return the result
dataPoints_Trans2
##            [,1]       [,2]       [,3]
## [1,] -1.1618950 -1.1618950 -1.1618950
## [2,] -0.3872983 -0.3872983 -0.3872983
## [3,]  0.3872983  0.3872983  0.3872983
## [4,]  1.1618950  1.1618950  1.1618950
```

NB: for this simple example the same result can of course be achieved more easily by

```
apply(dataPoints, 2, scale)
```

answered Jun 16 at 16:03

vonjd

1,318 2 19 32

1 Is this related to grouping? – Frank Jun 16 at 16:55

1 @Frank: Well, to be honest with you the title of this post is rather misleading: when you read the question itself it is about "the apply family". `sweep` is a higher-order function like all the others mentioned here, e.g. `apply`, `sapply`, `lapply` So the same question could be asked about the accepted answer with over 1,000 upvotes and the examples given therein. Just have a look at the example given for `apply` there. – vonjd Jun 16 at 17:03

1 Oh right, good point. – Frank Jun 16 at 18:08

protected by Bhargav Rao ♦ Feb 2 '16 at 17:54

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?