What You're Doing Is Rather Desperate

Notes from the life of a computational biologist

A brief introduction to “apply” in R

At any R Q&A site, you’ll frequently see an exchange like this one:

> *Q: How can I use a loop to [...insert task here...] ?*
> *A: Don’t. Use one of the apply functions.*

So, what are these wondrous *apply* functions and how do they work? I think the best way to figure out anything in R is to learn by experimentation, using embarrassingly trivial data and functions.

If you fire up your R console, type “??apply” and scroll down to the functions in the *base* package, you’ll see something like this:

```
1  base::apply            Apply Functions Over Array Margins
2  base::by               Apply a Function to a Data Frame Split by Factors
3  base::eapply           Apply a Function Over Values in an Environment
4  base::lapply           Apply a Function over a List or Vector
5  base::mapply           Apply a Function to Multiple List or Vector Argum
6  base::rapply           Recursively Apply a Function to a List
7  base::tapply           Apply a Function Over a Ragged Array
```

Let’s examine each of those.

1. apply
Description: “Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.”

OK – we know about vectors/arrays and functions, but what are these “margins”? Simple: either the rows (1), the columns (2) or both (1:2). By “both”, we mean “apply the function to each individual value.” An example:

```
# create a matrix of 10 rows x 2 columns
m <- matrix(c(1:10, 11:20), nrow = 10, ncol = 2)
# mean of the rows
apply(m, 1, mean)
 [1]  6  7  8  9 10 11 12 13 14 15
# mean of the columns
apply(m, 2, mean)
[1]  5.5 15.5
# divide all values by 2
apply(m, 1:2, function(x) x/2)
      [,1] [,2]
 [1,]  0.5  5.5
 [2,]  1.0  6.0
 [3,]  1.5  6.5
 [4,]  2.0  7.0
 [5,]  2.5  7.5
 [6,]  3.0  8.0
 [7,]  3.5  8.5
 [8,]  4.0  9.0
 [9,]  4.5  9.5
[10,]  5.0 10.0
```

That last example was rather trivial; you could just as easily do “m[, 1:2]/2” – but you get the idea.

2. by

Updated 27/2/14*: note that the original example in this section no longer works; use colMeans now instead of mean.*
Description: “Function ‘by’ is an object-oriented wrapper for ‘tapply’ applied to data frames.”

The *by* function is a little more complex than that. Read a little further and the documentation tells you that “a data frame is split by row into data frames subsetted by the values of one or more factors, and function ‘FUN’ is applied to each subset in turn.” So, we use this one where factors are involved.

To illustrate, we can load up the classic R dataset “iris”, which contains a bunch of flower measurements:

```
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

# get the mean of the first 4 variables, by species
by(iris[, 1:4], Species, colMeans)
Species: setosa
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       5.006        3.428        1.462        0.246
------------------------------------------------------------
Species: versicolor
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       5.936        2.770        4.260        1.326
------------------------------------------------------------
Species: virginica
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       6.588        2.974        5.552        2.026
```

Essentially, *by* provides a way to split your data by factors and do calculations on each subset. It returns an object of class “by” and there are many, more complex ways to use it.

3. eapply

Description: “eapply applies FUN to the named values from an environment and returns the results as a list.”

This one is a little trickier, since you need to know something about *environments* in R. An environment, as the name suggests, is a self-contained object with its own variables and functions. To continue using our very simple example:

```
# a new environment
e <- new.env()
# two environment variables, a and b
e$a <- 1:10
e$b <- 11:20
# mean of the variables
eapply(e, mean)
$b
[1] 15.5

$a
[1] 5.5
```

I don’t often create my own environments, but they’re commonly used by R packages such as Bioconductor so it’s good to know how to handle them.

4. lapply

Description: “lapply 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.”

That’s a nice, clear description which makes *lapply* one of the easier apply functions to understand. A simple example:

```
# create a list with 2 elements
l <- list(a = 1:10, b = 11:20)
# the mean of the values in each element
```

```
lapply(l, mean)
$a
[1] 5.5

$b
[1] 15.5

# the sum of the values in each element
lapply(l, sum)
$a
[1] 55

$b
[1] 155
```

The *lapply* documentation tells us to consult further documentation for *sapply*, *vapply* and *replicate*. Let’s do that.

4.1 sapply

Description: “sapply is a user-friendly version of lapply by default returning a vector or matrix if appropriate.”

That simply means that if *lapply* would have returned a list with elements $a and $b, *sapply* will return either a vector, with elements [[‘a’]] and [[‘b’]], or a matrix with column names “a” and “b”. Returning to our previous simple example:

```
# create a list with 2 elements
l <- list(a = 1:10, b = 11:20)
# mean of values using sapply
l.mean <- sapply(l, mean)
# what type of object was returned?
class(l.mean)
[1] "numeric"
# it's a numeric vector, so we can get element "a" like this
l.mean[['a']]
[1] 5.5
```

4.2 vapply

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

A third argument is supplied to *vapply*, which you can think of as a kind of template for the output. The documentation uses the *fivenum* function as an example, so let’s go with that:

```
l <- list(a = 1:10, b = 11:20)
# fivenum of values using vapply
l.fivenum <- vapply(l, fivenum, c(Min.=0, "1st Qu."=0, Median=0, "3rd Qu
class(l.fivenum)
[1] "matrix"
# let's see it
```

```
l.fivenum
           a    b
Min.     1.0 11.0
1st Qu.  3.0 13.0
Median   5.5 15.5
3rd Qu.  8.0 18.0
Max.    10.0 20.0
```

So, *vapply* returned a matrix, where the column names correspond to the original list elements and the row names to the output template. Nice.

4.3 replicate

Description: “replicate is a wrapper for the common use of sapply for repeated evaluation of an expression (which will usually involve random number generation).”

The replicate function is very useful. Give it two mandatory arguments: the number of replications and the function to replicate; a third optional argument, *simplify* = *T*, tries to simplify the result to a vector or matrix. An example – let’s simulate 10 normal distributions, each with 10 observations:

```
replicate(10, rnorm(10))
             [,1]        [,2]        [,3]       [,4]        [,5]
 [1,]  0.67947001 -1.94649409  0.28144696  0.5872913  2.22715085 -0.2759
 [2,]  1.17298643 -0.01529898 -1.47314092 -1.3274354 -0.04105249  0.5286
 [3,]  0.77272662 -2.36122644  0.06397576  1.5870779 -0.33926083  1.1211
 [4,] -0.42702542 -0.90613885  0.83645668 -0.5462608 -0.87458396 -0.7238
 [5,] -0.73892937 -0.57486661 -0.04418200 -0.1120936  0.08253614  1.3190
 [6,]  2.93827883 -0.33363446  0.55405024 -0.4942736  0.66407615 -0.1536
 [7,]  1.30037496 -0.26207115  0.49818215  1.0774543 -0.28206908  0.8254
 [8,] -0.04153545 -0.23621632 -1.01192741  0.4364413 -2.28991601 -0.0028
 [9,]  0.01262547  0.40247248  0.65816829  0.9541927 -1.63770154  0.3281
[10,]  0.96525278 -0.37850821 -0.85869035 -0.6055622  1.13756753 -0.3719
             [,7]        [,8]       [,9]       [,10]
 [1,]  0.03928297  0.34990909 -0.3159794  1.08871657
 [2,] -0.79258805 -0.30329668 -1.0902070  0.73356542
 [3,]  0.10673459 -0.02849216  0.8094840  0.06446245
 [4,] -0.84584079 -0.57308461 -1.3570979 -0.89801330
 [5,] -1.50226560 -2.35751419  1.2104163  0.74650696
 [6,] -0.32790991  0.80144695 -0.0071844  0.05742356
 [7,]  1.36719970  2.34148354  0.9148911  0.20451421
 [8,] -0.51112579 -0.53658159  1.5194130 -0.94250069
 [9,]  0.52017814 -1.22252527  0.4519702  0.08779704
[10,]  1.35908918  1.09024342  0.5912627 -0.20709053
```

5. mapply

Description: “mapply is a multivariate version of sapply. mapply applies FUN to the first elements of each (...) argument, the second elements, the third elements, and so on.”

The mapply documentation is full of quite complex examples, but here’s a simple, silly one:

```
l1 <- list(a = c(1:10), b = c(11:20))
```

```
l2 <- list(c = c(21:30), d = c(31:40))
# sum the corresponding elements of l1 and l2
mapply(sum, l1$a, l1$b, l2$c, l2$d)
 [1]  64  68  72  76  80  84  88  92  96 100
```

Here, we sum l1$a[1] + l1$b[1] + l2$c[1] + l2$d[1] (1 + 11 + 21 + 31) to get 64, the first element of the returned list. All the way through to l1$a[10] + l1$b[10] + l2$c[10] + l2$d[10] (10 + 20 + 30 + 40) = 100, the last element.

6. rapply

Description: “rapply is a recursive version of lapply.”

I think “recursive” is a little misleading. What *rapply* does is apply functions to lists in different ways, depending on the arguments supplied. Best illustrated by examples:

```
# let's start with our usual simple list example
l <- list(a = 1:10, b = 11:20)
# log2 of each value in the list
rapply(l, log2)
       a1        a2        a3        a4        a5        a6        a7        a8
0.000000  1.000000  1.584963  2.000000  2.321928  2.584963  2.807355  3.000000
       a9       a10        b1        b2        b3        b4        b5        b6
3.169925  3.321928  3.459432  3.584963  3.700440  3.807355  3.906891  4.000000
       b7        b8        b9       b10
4.087463  4.169925  4.247928  4.321928
# log2 of each value in each list
rapply(l, log2, how = "list")
$a
 [1] 0.000000 1.000000 1.584963 2.000000 2.321928 2.584963 2.807355 3.00
 [9] 3.169925 3.321928

$b
 [1] 3.459432 3.584963 3.700440 3.807355 3.906891 4.000000 4.087463 4.16
 [9] 4.247928 4.321928

# what if the function is the mean?
rapply(l, mean)
   a    b
 5.5 15.5

rapply(l, mean, how = "list")
$a
[1] 5.5

$b
[1] 15.5
```

So, the output of *rapply* depends on both the function and the *how* argument. When how = “list” (or “replace”), the original list structure is preserved. Otherwise, the default is to *unlist*, which results in a vector.

You can also pass a “classes=” argument to *rapply*. For example, in a mixed list of numeric and character variables, you could specify that the function act only on the numeric values with “classes = numeric”.

7. tapply

Description: “Apply 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.”

Woah there. That sounds complicated. Don’t panic though, it becomes clearer when the required arguments are described. Usage is “tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)”, where X is “an atomic object, typically a vector” and INDEX is “a list of factors, each of same length as X”.

So, to go back to the famous iris data, “Species” might be a factor and “iris$Petal.Width” would give us a vector of values. We could then run something like:

```
attach(iris)
# mean petal length by species
tapply(iris$Petal.Length, Species, mean)
    setosa versicolor  virginica
     1.462      4.260      5.552
```

Summary

I’ve used very simple examples here, with contrived data and standard functions (such as mean and sum). For me, this is the easiest way to learn what a function does: I can look at the original data, then the result and figure out what happened. However, the “apply” family is a much more powerful than these illustrations – I encourage you to play around with it.

The things to consider when choosing an apply function are basically:

- What class is my input data? – vector, matrix, data frame...
- On which subsets of that data do I want the function to act? – rows, columns, all values...
- What class will the function return? How is the original data structure transformed?

It’s the usual input-process-output story: what do I have, what do I want and what lies inbetween?

Related

Configuring the R BatchJobs package for Torque batch queues
In "programming"

R/ggplot2 tip: aes_string
In "programming"

Bioinformatics journals: time from submission to acceptance, revisited
In "bioinformatics"

This entry was posted in computing, R, statistics and tagged r-project, statistics, tutorial on August 20, 2010 [https://nsaunders.wordpress.com/2010/08/20/a-brief-introduction-to-apply-in-r/] .

9 thoughts on “A brief introduction to “apply” in R”

Harlan
August 20, 2010 at 00:57

This is a really good summary. An alternative to these built-in functions, which can be tricky to remember, is to use Hadley Wickham’s plyr package which includes a set of functions that generalize the general split/apply/join process to most standard types of R data in a consistent way. Not always as fast as the core functions, but much easier on the programmer!

nsaunders Post author
August 20, 2010 at 09:48

Yes, I like plyr a lot and its functions have a nice naming scheme which indicates the data structures, *e.g.* ddply is data frame in -> data frame out. That will probably be the subject of the next post.

Daniel Jurczak
August 20, 2010 at 08:15

I really, really like this entry. Short descriptions, a little example and the corresponding source code.

Nice job Neil.

Larry (IEOR Tools)
August 20, 2010 at 23:15

Excellent tutorial. I’ve thought about doing this as a topic at one of our R user groups. I will definitely use this as a reference.

darrenjw
August 21, 2010 at 00:27

Good job! I too had this on my list of things to do, but never quite got around to it. Now I don’t have to! ;-)

Vijai
August 23, 2010 at 12:59

Very neat explanations for beginners of R like me.
Please know that there is a large audience for this type of blog entries and we; while not always verbose about it; appreciate it immensely.
Sincere thanks
JVJAI

aL3xa
August 23, 2010 at 22:25

Sublime resource! Thank you for your effort! This is essential for coding in R, and yet, was missing in the R blogosphere.

respiratoryclub
September 3, 2010 at 05:10

A very helpful post. Thanks!

Bob Muenchen
September 3, 2010 at 10:32

Nicely done! I remember pulling my hair out trying to understand those help files the first time through.

For completeness you might add the aggregate function.

After learning what lapply does, I like to cover what do.call does. For a simple example, go to Amazon.com & search inside my book, R for Stata Users for an object named “myBYout”. That’ll take you to pages 228-232 where I do lapply followed by a do.call. The plyr package does make that kind of processing much easier though.

There’s a minor typo in your sapply example where it says:

l.mean[['a]]

you need a close quote:

l.mean[['a']]

I would write it with single brackets and spaces for readability:

l.mean['a']

Keep up the good work!

Bob

Comments are closed.