A quick introduction to plyr

Sean Anderson

April 14, 2011

What is `plyr`? It's a bundle of awesomeness (i.e. an R package) that makes it simple to split apart data, do stuff to it, and mash it back together. This is a common data manipulation step.

Or, from the documentation:

"`plyr` is a set of tools that solves a common set of problems: you need to break a big problem down into manageable pieces, operate on each pieces and then put all the pieces back together. It's already possible to do this with split and the apply functions, but `plyr` just makes it all a bit easier…"

This is a very quick introduction to `plyr`. For more details have a look at the `plyr` site: `http://had.co.nz/plyr/` and particularly Hadley Wickham's introductory guide *The split-apply-combine strategy for data analysis*.

`http://had.co.nz/plyr/plyr-intro-090510.pdf`

1 Why use `apply` functions instead of `for` loops?

1. the code is cleaner – easier to code and read, and less error prone because:
   (a) you don't have to deal with subsetting
   (b) you don't have to deal with saving your results
2. apply functions are often faster, sometimes dramatically

2 Why use `plyr` over base `apply` functions?

1. `plyr` has a common syntax — easier to remember
2. `plyr` requires less code since it takes care of the input and output format
3. `plyr` can be run in parallel — faster

3 The basic idea behind apply functions

`apply` functions work by applying a function to a set of values and returning the output in some format. Here's about as simple an example as possible:

```
> y <- c(1, 2, 3)
> f <- function(x) x^2
> sapply(y, f)
```

```
[1] 1 4 9
```

Here, I have applied the function `f` to the values of `y`. Note that the `sapply` function was unnecessary here. This would have been better done in a vectorized format. If `y` was large, and the function more complex, the vectorized format could be noticeably faster.

```
> f(y)
```

```
[1] 1 4 9
```

But, it isn't always possible (or easy) to vectorize a function, particularly when you're dealing with groupings of data as in the following examples.

4 plyr basics

plyr builds on the built in apply functions by giving you control over the input and output formats and keeping the syntax consistent across all variations. It also adds some niceties like error processing, parallel processing, and progress bars.

The basic format is 2 letters followed by ply(). The first letter refers to the format in and the second to the format out.

The 3 main letters are:

1. d = data frame
2. a = array (includes matrices)
3. l = list

So, ddply means: take a data frame, split it up, do something to it, and return a data frame. I find I use this the majority of the time since I often work with data frames.

ldply means: take a list, split it up, do something to it, and return a data frame. This extends to all combinations. The columns are the input formats and the rows are the output format:

	data frame	list	array
data frame	ddply	ldply	adply
list	dlply	llply	alply
array	daply	laply	aaply

I've ignored a couple other format options. One that you might find useful is the underscore (_) which will throw away the output (e.g., d_ply()). This can be useful when plotting.

5 A general example with plyr

Let's take a simple example. Take a data frame, split it up (by year), calculate the coefficient of variation of the count, and return a data frame. This could easily be done on one line, but I'm expanding it here to show the format a more complex function could take.

```
> set.seed(1)
> d <- data.frame(year = rep(2000:2002, each = 3), count = round(runif(9,
+     0, 20)))
> print(d)

  year count
1 2000     5
2 2000     7
3 2000    11
4 2001    18
5 2001     4
6 2001    18
7 2002    19
8 2002    13
9 2002    13

> library(plyr)
> ddply(d, "year", function(x) {
+     mean.count <- mean(x$count)
+     sd.count <- sd(x$count)
+     cv <- sd.count/mean.count
+     data.frame(cv.count = cv)
+ })

  year  cv.count
1 2000 0.3984848
2 2001 0.6062178
3 2002 0.2309401
```

6 `transform` and `summarise`

It is often convenient to use these functions within `plyr`. `transform` acts as it would normally as the base R function and modifies an existing data frame. `summarise` creates a new (usually) condensed data frame.

```
> ddply(d, "year", summarise, mean.count = mean(count))

  year mean.count
1 2000   7.666667
2 2001  13.333333
3 2002  15.000000

> ddply(d, "year", transform, total.count = sum(count))

  year count total.count
1 2000     5          23
2 2000     7          23
3 2000    11          23
4 2001    18          40
5 2001     4          40
6 2001    18          40
7 2002    19          45
8 2002    13          45
9 2002    13          45
```

7 Other useful options

7.1 Dealing with errors

You can use the `failwith` function to control how errors are dealt with.

```
> f <- function(x) if (x == 1) stop("Error!") else 1
> safe.f <- failwith(NA, f, quiet = TRUE)
> llply(1:2, safe.f)

[[1]]
[1] NA

[[2]]
[1] 1
```

7.2 Parallel processing

In conjunction with `doMC` (or `doSMP` on Windows) you can run your function separately on each core of your computer. On a dual core machine this could double your speed in some situations. Set `.parallel = TRUE`.

```
> x <- c(1:10)
> wait <- function(i) Sys.sleep(0.1)
> system.time(llply(x, wait))

   user  system elapsed
  0.001   0.001   1.001

> system.time(sapply(x, wait))

   user  system elapsed
  0.001   0.001   1.002

> library(doMC)
> registerDoMC(2)
> system.time(llply(x, wait, .parallel = TRUE))

   user  system elapsed
  0.016   0.010   0.528
```