A quick introduction to plyr

Sean Anderson

November 7, 2012

plyr is an R package that makes it simple to split data apart, do stuff to it, and mash it back together. This is a common data-manipulation step. Importantly, plyr makes it easy to control the input and output data format from a syntactically consistent set of functions.

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 see Hadley Wickham’s introductory guide The split-apply-combine strategy for data analysis (2011, Journal of Statistical Software, Vol 40). There’s quite a bit of discussion online in general, and especially on stackoverflow.com.

1 Why use apply functions instead of for loops?

1. The code is cleaner (once you’re familiar with the concept). The code can be 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 can be faster than for loops, 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 easily be run in parallel — faster

3 **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:

<table>
<thead>
<tr>
<th></th>
<th>data frame</th>
<th>list</th>
<th>array</th>
</tr>
</thead>
<tbody>
<tr>
<td>data frame</td>
<td>ddply</td>
<td>1dply</td>
<td>adply</td>
</tr>
<tr>
<td>list</td>
<td>dlply</td>
<td>1lply</td>
<td>alply</td>
</tr>
<tr>
<td>array</td>
<td>daply</td>
<td>1aply</td>
<td>aaply</td>
</tr>
</tbody>
</table>

I’ve ignored some less common format options:
1. \( m \) = multi-argument function input

2. \( r \) = replicate a function \( n \) times.

3. \( _ \) = throw away the output

For plotting, you might find the underscore (\( _ \)) option useful. It will do something with the data (say add line segments to a plot) and then throw away the output (e.g., \( d\_ply() \)).

4 Base R apply functions and plyr

plyr provides a consistent and easy-to-work-with format for apply functions with control over the input and output formats. Some of the functionality can be duplicated with base R functions (but with less consistent syntax). Also, few R apply functions work directly with data frames as input and output and data frames are a common object class to work with.

Base R apply functions (from a presentation given by Hadley):

<table>
<thead>
<tr>
<th></th>
<th>array</th>
<th>data frame</th>
<th>list</th>
<th>nothing</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>apply</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>data frame</td>
<td>.</td>
<td>aggregate</td>
<td>by</td>
<td>.</td>
</tr>
<tr>
<td>list</td>
<td>sapply</td>
<td>.</td>
<td>lapply</td>
<td>.</td>
</tr>
<tr>
<td>n replicates</td>
<td>replicate</td>
<td>.</td>
<td>replicate</td>
<td>.</td>
</tr>
<tr>
<td>function arguments</td>
<td>mapply</td>
<td>.</td>
<td>mapply</td>
<td>.</td>
</tr>
</tbody>
</table>

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.

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

<table>
<thead>
<tr>
<th>year</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>5</td>
</tr>
<tr>
<td>2000</td>
<td>7</td>
</tr>
<tr>
<td>2000</td>
<td>11</td>
</tr>
<tr>
<td>2001</td>
<td>18</td>
</tr>
<tr>
<td>2001</td>
<td>4</td>
</tr>
<tr>
<td>2001</td>
<td>18</td>
</tr>
<tr>
<td>2002</td>
<td>19</td>
</tr>
<tr>
<td>2002</td>
<td>13</td>
</tr>
<tr>
<td>2002</td>
<td>13</td>
</tr>
</tbody>
</table>

> 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)
+ })

<table>
<thead>
<tr>
<th>year</th>
<th>cv.count</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>0.3984848</td>
</tr>
<tr>
<td>2001</td>
<td>0.6062178</td>
</tr>
<tr>
<td>2002</td>
<td>0.2309401</td>
</tr>
</tbody>
</table>

### 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))


4
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

Bonus function: mutate. mutate works like transform but lets you build on columns you build.

> ddply(d, "year", mutate, mu = mean(count), sigma = sd(count),
+    cv = sigma/mu)

   year count   mu    sigma      cv
 1 2000   5 7.666667 3.055050 0.3984848
 2 2000   7 7.666667 3.055050 0.3984848
 3 2000  11 7.666667 3.055050 0.3984848
 4 2001  18 13.333333 8.082904 0.6062178
 5 2001   4 13.333333 8.082904 0.6062178
 6 2001  18 13.333333 8.082904 0.6062178
 7 2002  19 15.000000 3.464102 0.2309401
 8 2002  13 15.000000 3.464102 0.2309401
 9 2002  13 15.000000 3.464102 0.2309401
7 Plotting with plyr

You can use plyr to plot data by throwing away the output with an underscore (_). This is a bit cleaner than a for loop since you don’t have to subset the data manually.

```r
> par(mfrow = c(1, 3), mar = c(2, 2, 1, 1), oma = c(3, 3, 0, 0))
> d_ply(d, "year", transform, plot(count, main = unique(year), type = "o"))
> mtext("count", side = 1, outer = TRUE, line = 1)
> mtext("frequency", side = 2, outer = TRUE, line = 1)
```

8 Nested chunking of the data

The basic syntax can be easily extended to break apart the data based on multiple columns:

```r
> baseball.dat <- subset(baseball, year > 2000) # data from the plyr package
> x <- ddply(baseball.dat, c("year", "team"), summarize,
+    homeruns = sum(hr))
> head(x)

    year team  homeruns
1 2001 ANA        4
2 2001 ARI       155
3 2001 ATL       163
4 2001 CIN       112
5 2001 CLE       126
6 2001 COL       168
```
9 Other useful options

9.1 Dealing with errors

You can use the \texttt{failwith} function to control how errors are dealt with.

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

[[1]]
[1] NA

[[2]]
[1] 1
\end{verbatim}

9.2 Parallel processing

In conjunction with \texttt{doMC} (or \texttt{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 \texttt{.parallel = TRUE}.

\begin{verbatim}
> x <- c(1:10)
> wait <- function(i) Sys.sleep(0.1)
> system.time(llply(x, wait))

     user  system elapsed
   0.001   0.000   1.007

> system.time(sapply(x, wait))
\end{verbatim}
library(doMC)
registerDoMC(2)
system.time(llply(x, wait, .parallel = TRUE))

user  system elapsed
0.021  0.006  0.533

10 So, why would I *not* want to use plyr?

plyr can be slow — particularly if you are working with very large datasets that involve a lot of subsetting. Hadley is working on this and recent development versions of plyr run much faster.

It's important to remember that typically the speed that you can write code and understand it later is the rate-limiting step.

Three faster options:

1. Use a base R apply function:

   > system.time(ddply(baseball, "id", summarize, length(year)))
   user  system elapsed
   1.169  0.013  1.188

   > system.time(tapply(baseball$year, baseball$id, function(x) length(x)))
   user  system elapsed
   0.019  0.000  0.020

2. Use an immutable data frame. An immutable data frame (idata.frame) returns pointers to the original object when subset instead of creating a copy of itself each time. This is often the rate-limiting step in an apply function.

   > system.time(idply(baseball, "id", x -> length(x)))
   user  system elapsed
   0.017  0.000  0.018

   > system.time(idtapply(baseball$year, baseball$id, function(x) length(x)))
   user  system elapsed
   0.019  0.000  0.020
> system.time(ddply(idata.frame(baseball), "id", sumarize, length(year)))

         user  system elapsed
0.956   0.003   0.960

(3) Use the data.table package:

> library(data.table)
> dt <- data.table(baseball, key="id")

> system.time(dt[, length(year), by=list(id)])

         user  system elapsed
0.005   0.000   0.005

The next version of plyr lets you work with objects of class data.table:

https://gist.github.com/4007552