Statistical Computing (36-350)

split, apply, combine *with* `plyr`

Cosma Shalizi and Vincent Vu

October 17, 2011

Agenda

- Abstracting split, apply, combine
- plyr usage
- Examples
- Recommended reading:

  http://plyr.had.co.nz/

Recall

- Many problems can be solved this way:
  - Divide the big problem into smaller pieces
  - Work on each piece *independently*
  - Recombine the pieces

The basic pattern

split apply combine

Example from last time

```
x <- split(df, df$player)
x <- lapply(x, runningTotal)
scores <- do.call(rbind, x)
```

Split the scorecards by player
Compute a running total for each player
Combine the results into an array (and then plot)

*apply() in base R

- apply() - arrays
- lapply() - list
- sapply() - list (simplify)
- vapply() - list (safe simplify)
- tapply() - data frames (tables)
- mapply() - multiple vectors (special case of 2d array)

*apply() in base R

- Functions are useful, but
- Output is inconsistent (lists or array)
- Too much cognitive overhead!

Abstracting the pattern

split apply combine

The `plyr` model

- Essential ingredients
- **Input data structure** (split)
- **Processing function** (apply)
- **Output data structure** (combine)

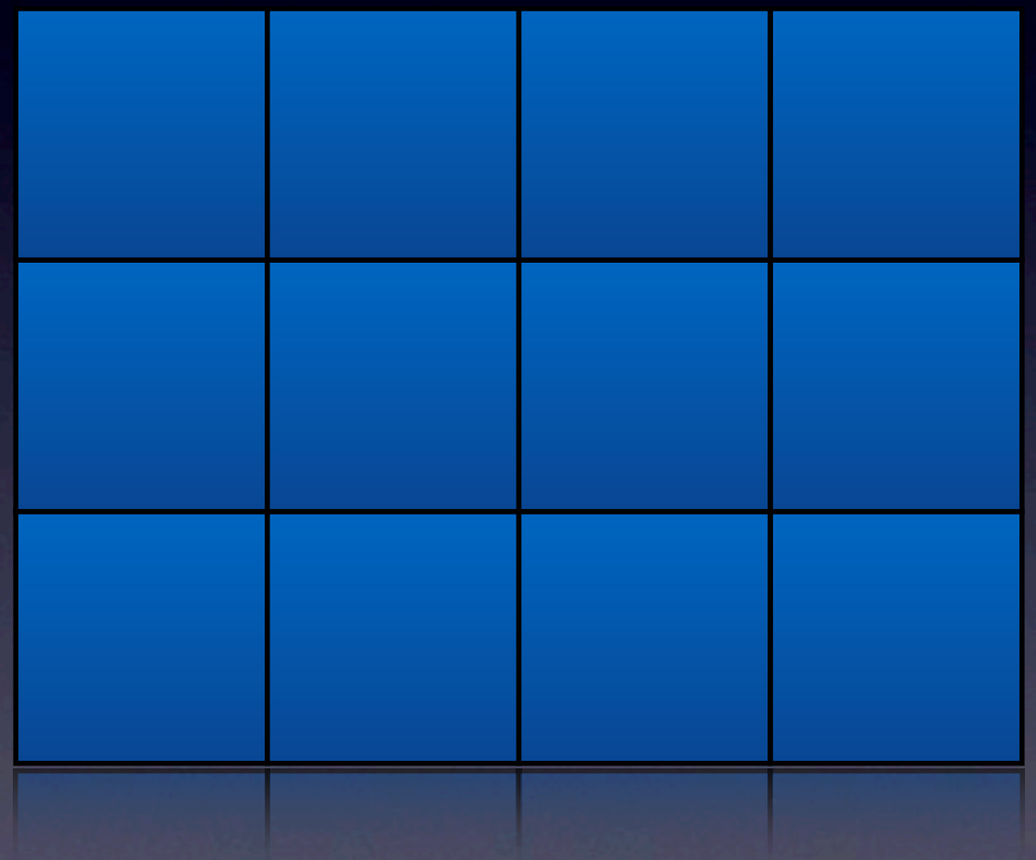
*ply() in plyr

- Functions named and designed consistently
  - *ply() - replace * with 2 characters:
    - first character: input type (a, d, l)
    - second character: output type (a, d, l, _)

output

input	list	array	data frame	discarded
list	`llply()`	`laply()`	`ldply()`	`l_ply()`
array	`alply()`	`aaply()`	`adply()`	`a_ply()`
data frame	`dlply()`	`daply()`	`ddply()`	`d_ply()`

Input Data Structure



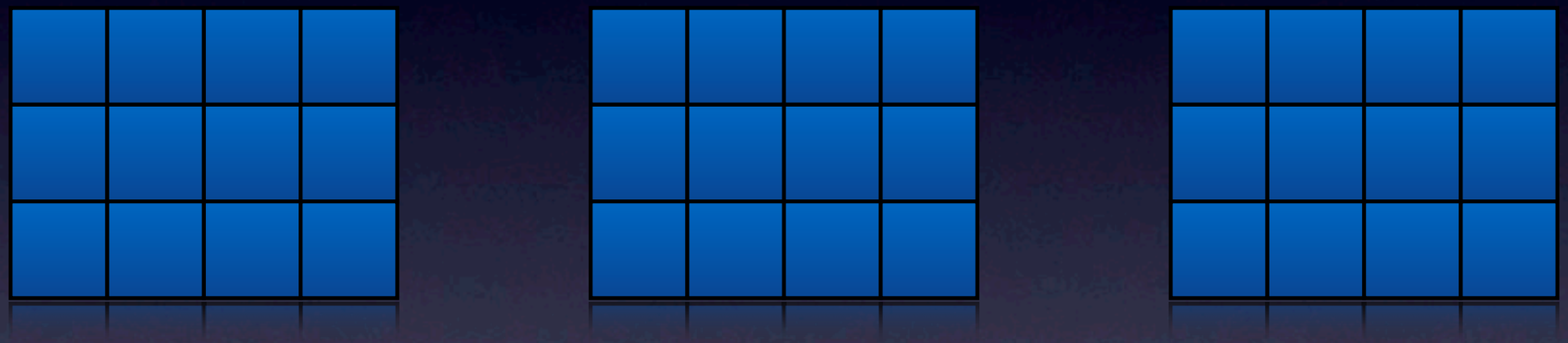
Input Data Structure

- Types
  - Array
  - List
  - Data frame
- Each type has different ways of being split

Arrays

- d-dimensional arrays
  - have *d* dimensions that can be subscripted independently
  - can be split 2^d - 1 different ways

2 dimensional array



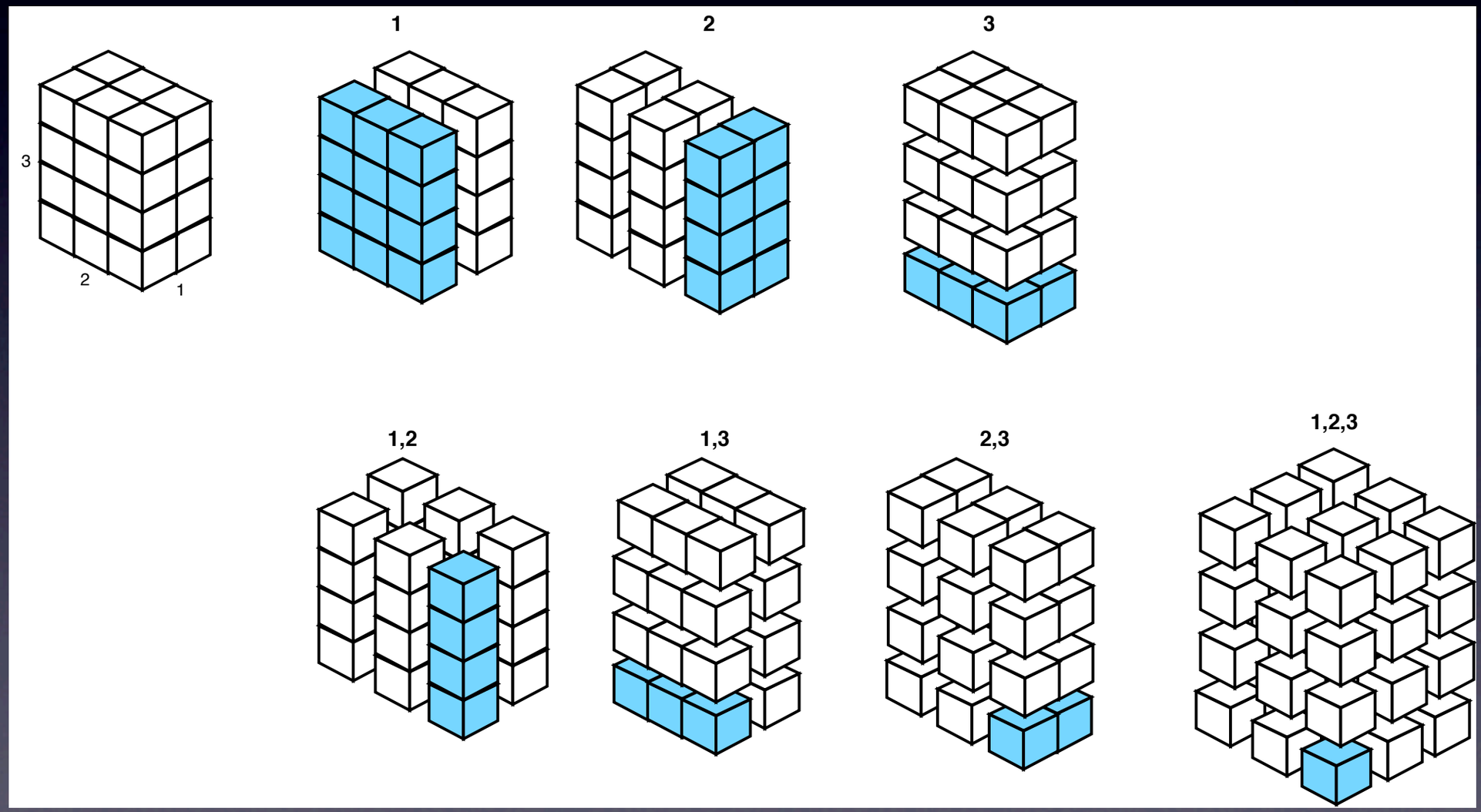
Three ways to split

2 dimensional array

1 2 1,2

Three ways to split

3 dimensional array



Seven ways to split

figure from Wickham (2011)

a*ply()

```
y <- a*ply(.data, .margins, .fun, ...)
```

- `.data` an array
- `.margins` vector of subscripts which the function will be applied over
- `.fun` the function to be applied
- `...` additional arguments to function
- Returns a (*: a = array, d = data frame, l = list)

Lists

Only one way to split

Lists

Only one way to split

l*ply()

```
y <- l*ply(.data, .fun, ...)
```

- `.data` a list
- `.fun` the function to be applied
- `...` additional arguments to function
- Returns a (*: a = array, d = data frame, l = list)

Pop quiz

```
x <- list(a = 10, b = 'joe')
```

What's the difference between

`x[[1]]` and `x[1]` ?

Pop quiz

`x[[1]]`

is the 1st component of x (it is a numeric)

`x[1]`

is a subset of x (it is a list)

Data frames

- Think of a data frame as a table
- Can be split into groups according to the values of variables (columns)
- Useful for ragged data, where groups have possibly different sizes

data frame

round	player	country
4	Tiger W.	USA
4	Jason D.	Australia
3	Tiger W.	USA
3	Charl S.	South Africa
4	Charl S.	South Africa
4	Adam S.	Australia

6 different ways to split

data frame

round	player	country
4	Tiger W.	USA
4	Jason D.	Australia
3	Tiger W.	USA
3	Charl S.	South Africa
4	Charl S.	South Africa
4	Adam S.	Australia

round	player	country
4	Tiger W.	USA
3	Tiger W.	USA

round	player	country
4	Jason D.	Australia

round	player	country
4	Charl S.	South Africa
3	Charl S.	South Africa

round	player	country
4	Adam S.	Australia

split by player

data frame

round	player	country
4	Tiger W.	USA
4	Jason D.	Australia
3	Tiger W.	USA
3	Charl S.	South Africa
4	Charl S.	South Africa
4	Adam S.	Australia

round	player	country
4	Tiger W.	USA
4	Jason D.	Australia
4	Charl S.	South Africa
4	Adam S.	Australia

round	player	country
3	Tiger W.	USA
3	Charl S.	South Africa

split by round

data frame

round	player	country
4	Tiger W.	USA
4	Jason D.	Australia
3	Tiger W.	USA
3	Charl S.	South Africa
4	Charl S.	South Africa
4	Adam S.	Australia

round	player	country
4	Tiger W.	USA

round	player	country
4	Jason D.	Australia
4	Adam S.	Australia

round	player	country
4	Charl S.	South Africa

round	player	country
3	Tiger W.	USA

round	player	country
3	Charl S.	South Africa

split by (round, country)

d*ply()

```
y <- d*ply(.data, .variables, .fun, ...)
```

- `.data` a data frame
- `.variables` variables used to define groups
- `.fun` the function to be applied
- `...` additional arguments to function
- Returns a (*: a = array, d = data frame, l = list)

d*ply()

```
y <- d*ply(.data, .variables, .fun, ...)
```

- **.variables**
  - can be of the following forms
    - .(var1, var2)
    - c('var1', 'var2)
  - d*ply() will search the data frame for those variables, and then global environment

Data frames

- Data frame is actually a list of vectors
  - => Can be split into separate columns
  - => Can be used with l*ply()
- Responds to array-like indexing
  - => Can be split like a 2D array
  - => Can be used with a*ply()

Processing function

```
function(x, ...)
```

Processing function

- Function that is applied to each piece
- Should:
  - Take a piece as its first argument
  - Return same type as eventual output (but there are exceptions)
- Sometimes cause side effects (plot, save, ...)

Output Data Structure

Output Data Structure

- Defines how results are combined and labelled
- Types
  - Array (a)
  - List (l)
  - Data frame (d)
- Discarded (_) – For side effects, e.g. plotting

Arrays

- Output organized in the expected way.
- Processing function should return an object of same type each time it is called.
- If processing function returns a list, then output will be a list-array (list with dimensions)

Data frames

- Output will contain results with additional label columns indicating which group the result corresponds to.

Applying the pattern to *your* problem

- What is the data type of
  - input data structure?
  - output data structure?
- Use a built-in function, or write a processing function and test it on one piece.

Examples

Remember to install plyr

```
> install.packages('plyr', dependencies = T)
```

Remember to load plyr

```
> library(plyr)
```

Regularly sampled spatial data

```
x <- array(..., dim = c(10, 10, 40))
```

- Data:
  - 10 x 10 grid of locations
  - 40 measurements at each location
- Problem:
  - Standardize measurements at each location

Standardize

one location

```
z <- scale(x[1, 1, ])
```

Standardize

iteration

```
y <- array(dim = dim(x))
for(i in 1:dim(x)[1]) {
    for(j in 1:dim(x)[2]) {
        y[i, j, ] <- scale(x[i, j, ])
    }
}
```

plyr

```
y <- aaply(x, 1:2, scale)
```

Ragged spatial data

```
x <- data.frame(loc.x = ...,
                loc.y = ...,
                value = ...)
```

- Data:
  - irregularly sampled (x,y) locations
  - different numbers of measurements at each location
- Problem:
  - Standardize measurements at each location

Standardize

one location

```
df <- subset(x, loc.x = 1 & loc.y = 1)
z <- scale(df$value)
```

Standardize

iteration

```
### TOO PAINFUL ###
```

plyr

```
y <- ddply(x, .(loc.x, loc.y),
           function(x) scale(x$value))
```

2011 Masters Golf

Example with `plyr`

Input

holes 1–18	round	player	country
...	4	Charl S.	South Africa
...	3	Charl S.	South Africa
...	2	Charl S.	South Africa
...	1	Charl S.	South Africa
...	4	Jason D.	Australia
...	3	Jason D.	Australia
...	...	...	...

data frame

Desired Output

	1	2	3	4	…
Charl S.	…	…	…	…	…
Jason D.	…	…	…	…	…
Adam S.	…	…	…	…	…
Tiger W.	…	…	…	…	…
…	…	…	…	…	…

array

Processing function

```
runningTotal <- function(df, par = masters2011$course$par)
{
    # Reorder the rows of the data frame by round
    # (so that the scores are in chronological order)
    # and extract the scores as a matrix
    x <- as.matrix(df[order(df$round), 1:18])
    n <- nrow(x)

    # Convert from an 18 x n matrix to a vector of
    # length 18*n, row-wise
    x <- as.vector(t(x))

    # Calculate the running over/under score
    x <- cumsum(x - rep(par, n))

    return(x)
}
```

returns a vector

base R

```
x <- split(df, df$player)
x <- lapply(x, runningTotal)
scores <- do.call(rbind, x)
```

plyr

```
scores <- daply(df, .(player), runningTotal)
```

Computing group summaries

- How many players in each country?

Input

holes 1–18	round	player	country
...	4	Charl S.	South Africa
...	3	Charl S.	South Africa
...	2	Charl S.	South Africa
...	1	Charl S.	South Africa
...	4	Jason D.	Australia
...	3	Jason D.	Australia
...	...	...	...

data frame

Desired Output

country	count
USA	...
South Africa	...
Ireland	...
...	...

data frame

Why doesn't this work?

```
ddply(df, .(country), nrow)
```

How many players in each country?

```
countPlayers <- function(x) {
    data.frame(count = length(unique(x$player)))
}

ddply(df, .(country), countPlayers)
```

How many players in each country?

	country	count
1	South Africa	3
2	Australia	4
3	United States	21
4	England	6
5	Argentina	1
6	South Korea	3
7	Italy	1
8	N. Ireland	1
9	Japan	2
10	Scotland	1
11	Sweden	1
12	Spain	3
13	Germany	1
14	Colombia	1

Computing group summaries

- What was the median score on each round for each country?

Input

holes 1–18	round	player	country
...	4	Charl S.	South Africa
...	3	Charl S.	South Africa
...	2	Charl S.	South Africa
...	1	Charl S.	South Africa
...	4	Jason D.	Australia
...	3	Jason D.	Australia
...	...	...	...

data frame

Desired Output

	1	2	3	4
USA	...	...	...	...
South Africa	...	...	...	...
Australia	...	...	...	...
...	...	...	...	...

array

Strategy 1

- Split by (country, round)
  - Within each group:
    - Split by player
      - Compute score for each player
    - Median of player scores

Strategy 2

- Split by (player, round)
  - Compute score for each player
- Then split by (country, round)
  - Compute median

simpler

Strategy 2

```
scorePlayer <- function(x) {
    par <- masters2011$course$par
    data.frame(score = sum(x[1:18] - par))
}

scores <- ddply(df, .(player, round), scorePlayer)
scores <- merge(df, scores)
daply(scores, .(country, round),
      function(x) median(x$score))
```

Strategy 2

	round			
country	1	2	3	4
South Africa	-3.0	-1.0	1.0	-3.0
Australia	0.0	-2.5	0.5	-4.5
United States	0.0	-1.0	0.0	0.0
England	0.0	-2.0	-1.0	-1.5
Argentina	-1.0	-2.0	-5.0	-1.0
South Korea	-5.0	0.0	1.0	0.0
Italy	2.0	-2.0	-3.0	-2.0
N. Ireland	-7.0	-3.0	-2.0	8.0
Japan	-0.5	0.0	-1.5	0.0
Scotland	2.0	-3.0	-3.0	1.0
Sweden	0.0	-2.0	2.0	-1.0
Spain	-3.0	1.0	3.0	1.0
Germany	0.0	-1.0	3.0	-2.0
Colombia	-2.0	3.0	1.0	4.0

Warning

- Many problems can be fit into the split, apply, combine pattern
- But don’t force it

Don’t do this

```
sapply(1:100, function(i) {

    #### LOTS OF STUFF HERE ####

})
```

```
l_ply(1:708, function(i) {

    #### COMPLEX THINGS ####

})
```

Summary

- 3 ingredients: input type, output type, processing function
- Start by writing a processing function for a single piece – do it by hand and then generalize
- Built-in functions are often useful for computing summaries