Lecture on plyr on Oct 22nd 2013

Install plyr

Start R and type

```
> install.packages("plyr")
```

Choose a CRAN mirror, preferably one of US mirrors

Load plyr

```
> library(plyr)
```

Notation: texts after > are R codes

Install and Load R Packages at Custom Location

Create a new directory “R_LIBS” where you want to store R packages permanently, e.g., on ITAP machines,

```
H:/My Documents/R_LIBS
```

Now in R, define a variable for the path to your packages,

```
> PATH_TO_LIBS = "H:/My\ Documents/R_LIBS"
```

Specify the location when install the package

```
> install.packages("plyr", lib=PATH_TO_LIBS)
```

Specify the location when load the package

```
> library(plyr, lib.loc=PATH_TO_LIBS)
```

Change Default Location of Packages

R function .libPaths() gets and sets the search path of R packages

Call .libPaths() with no arguments shows the current search path

```
> .libPaths()
```

By default, R installs packages to the first element of .libPaths()

When load packages, R searches in all elements of .libPaths()

Change Default Location of Packages Cont.

Add your custom location to the search path

```
> .libPaths(c("H:/My\ Documents/R_LIBS",.libPaths()))
```

Now packages are installed to your custom location by default

```
> install.packages("plyr")
```

And packages are searched and loaded from your custom location

```
> library(plyr)
```

Change Default Location of Packages Cont.

To make the change permanent, edit the ".Rprofile" file under R startup directory

Codes in ".Rprofile" will be executed during R startup

When you start R, function getwd() shows current working directory, unless you have called function setwd(), this will be the startup directory

```
> getwd()
```

On ITAP machines, the startup directory is

```
H:/My Documents
```

Create a new file named ".Rprofile" with the following line

```
.libPaths(c("H:/My\ Documents/R_LIBS",.libPaths()))
```

Split-Apply-Combine

Split-apply-combine is a common data analysis pattern/strategy

Split

Break up a big problem into manageable pieces

Apply

Operate on each piece independently

Combine

Put all pieces together

plyr Core Function Names

Function names

aaply	adply	alply	a_ply
daply	ddply	dlply	d_ply
laply	ldply	llply	l_ply
raply	rdply	rlply	r_ply
maply	mdply	mlply	m_ply

Functions are named according to input type and output type

Fist character for input Second character for output

Input types:

a = array, d = data frame, l = list, r = number of iterations, m = a data frame of parameter values

Output types:

a, d, l, _ means output discarded

Effects of input type and output type are orthogonal

My Own Print() Function

Define my own print function for better display

```
> myprint = function(x, ...) {
    cat("\n")
    print(x, ...)
    cat("------End of Print------\n")
}
```

myprint() does nothing more than highlighting space between printed objects

Regular print()

```
> for (x in 1:3) {
    print(x)
}
```

myprint()

```
> for (x in 1:3) {
    myprint(x)
}
```

Functions a*ply()

Input type: array

Arrays are sliced by dimension into lower-d arrays

```
a*ply(.data, .margins, .fun, ...)
```

Functions a*ply() Cont.

Slicing and printing a 2-d array

Make up an array

```
> a2 = array(data=1:6, dim=c(2,3))
```

Split by one dimension

```
> a_ply(.data=a2, .margins=1, .fun=myprint)
> a_ply(.data=a2, .margins=2, .fun=myprint)
```

Split by two dimensions

```
> a_ply(.data=a2, .margins=c(1,2), .fun=myprint)
```

Functions a*ply() Cont.

Slicing and printing a 3-d array

Make up an array

```
> a3 = array(data=1:24, dim=c(2,3,4))
```

Split by one dimension

```
> a_ply(.data=a3, .margins=3, .fun=myprint)
> a_ply(.data=a3, .margins=2, .fun=myprint)
```

Split by two dimensions

```
> a_ply(.data=a3, .margins=c(2,3), .fun=myprint)
```

Split by all three dimensions

```
> a_ply(.data=a3, .margins=c(1,2,3), .fun=myprint)
```

Functions l*ply()

Input type: list

Lists are split by element

```
l*ply(.data, .fun, ...)
```

Make up a list

```
> l3 = list(a=1, b=2:3, c=4:6)
```

Split by element

```
> l_ply(.data=l3, .fun=myprint)
```

Functions d*ply()

Input type: data.frame

Data frame are subsetted by combinations of variables

```
d*ply(.data, .variables, .fun, ...)
```

Make up a data.frame

```
> df = data.frame(
    gender  = rep(c("M","F"),times=c(3,2)),
    grades  = c("A","A","B","A","B"),
    score   = 1:5
)
```

Functions d*ply() Cont.

Split by gender

```
> d_ply(
    .data        = df,
    .variables   = "gender",
    .fun         = myprint
)
```

Split by both gender and grades

```
> d_ply(
    .data        = df,
    .variables   = c("gender","grades"),
    .fun         = myprint
)
```

Functions r*ply()

Input type: Iteration

Evaluate an expression a number of times

```
r*ply(.n, .expr)
```

3 iterations, generate 2 Normal(0,1) values in each iteration

```
> rdply(
    .n      = 3,
    .expr   = rnorm(2)
)
```

Multi-line expression, use curly brackets

```
> rdply(
    .n      = 3,
    .expr   = {
        x = rnorm(100)
        c(mean(x), sd(x))
    }
)
```

Functions m*ply()

Input type: data frame of parameter values

Call function with arguments in a data frame or matrix

```
m*ply(.data, .fun, ...)
```

Generate random Normal values with different means and standard deviations

Make up a data frame of parameters

```
> param = expand.grid(
    mean    = 1:3,
    sd      = 1:2
)
```

Call a function with these values of parameters

```
> mdply(
    .data   = param,
    .fun    = rnorm,
    n       = 2
)
```

Core Functions Output Types

_ for discarded output

l for list

a for array

d for data.frame

Output type depends on the output of the “Apply” step

Functions *_ply()

Output type: output discarded

The outputs of “Apply” are discarded

Side effects, e.g., print graphs to files

```
> r_ply(
    .n      = 1,
    .expr   = {
        pdf(file="Rplots.pdf")
        plot(1:10, 1:10)
        dev.off()
    }
)
```

Functions *lply()

Output type: list

The output of “Apply” can be anything, each one is made an element of a final list

```
> rlply(
    .n      = 3,
    .expr   = rnorm(2)
)
```

Functions *aply()

Output type: array

The output of “Apply” need to be array (vector, matrix, array), its dimensions are included in the final output array after the split dimensions

Vector (1-d array)

```
> raply(
    .n      = 3,
    .expr   = rnorm(2)
)
```

Functions *aply() Cont.

Matrix (2-d array)

```
> raply(
    .n      = 3,
    .expr   = matrix(rnorm(6), ncol=2)
)
```

3-d array

```
> raply(
    .n      = 3,
    .expr   = array(1:24, dim=c(2,3,4))
)
```

Functions *dply()

Output type: data frame

The output of “Apply” need to be vector or data frame

Output is a vector

```
> rdply(
    .n      = 3,
    .expr   = rnorm(2)
)
```

Output is a named vector

```
> rdply(
    .n      = 3,
    .expr   = c(x1=rnorm(1), x2=rnorm(1))
)
```

Output is a data.frame

```
> rdply(
    .n      = 3,
    .expr   = data.frame(x=rnorm(2))
)
```

Other useful functions

count()

arrange()

summarise()

colwise()

We will use Barley data in the lattice package to demonstrate the usage of these functions

Barley Data

Load data

```
> library(lattice)
> ?barley
> barley
> head(barley)
> tail(barley)
```

Yield for 10 varieties of barley at 6 sites in each of two years

120 records

4 variables: yield, variety, year, site

Function count()

Count the number of occurences

```
count(df, vars, wt_var)
```

Number of observations for each site

```
> count(df=barley, vars="site")
```

Number of observations for each site and year combination

```
> count(df=barley, vars=c("site", "year"))
```

Number of observations for each site, again but with weight

```
> tmp = count(df=barley, vars=c("site", "year"))
> tmp
> count(df=tmp, vars="site", wt_var="freq")
```

Function arrange()

Order a data frame by its columns

```
arrange(df, ...)
```

Order by one column: by yield from largest to smallest

```
> arrange(df=barley, -yield)
```

Order by multiple columns: first by year and site, then by yield from largest to smallest

```
> arrange(df=barley, year, site, -yield)
```

Function summarise()

Summarise a data frame

```
summarise(.data, ...)
```

Summarise the whole data frame

```
> summarise(.data=barley,
    max=max(yield), min=min(yield)
)
```

Group-wise summaries

```
> ddply(
    .data        = barley,
    .variables   = c("year", "site"),
    .fun         = summarise,
    max          = max(yield),
    min          = min(yield)
)
```

Function colwise()

Column-wise function

```
colwise(.fun, .cols)
```

Turn a function that operates on a vector into a function that operates column-wise on a data frame

Add a column to Barley data

```
> barley$noise = rnorm(nrow(barley))
```

Compute the mean for both yield and noise

```
> colwise(.fun=mean,
    .cols=c("yield","noise")
)
> colwise(.fun=mean,
    .cols=c("yield","noise")
)(barley)
```

Carrying Out Split-Apply-Combine

Split and combine are taken care of by plyr

Analyst needs only think about applying methods

Goal: compute the five number summary of yield at each site in each year

Yield at one site in one year is a working unit

Subset data at one site in one year

```
> unit = subset(barley,
    subset=(site=="University Farm" & year==1931)
)
```

Apply the Analysis

Compute the five number summary

```
> result = quantile(unit$yield)
```

Make it a function

```
> five.num = function(data) {
    quantile(data$yield)
}
> result = five.num(unit)
```

Yields at Every Site in Every year: User plyr Functions

Use ddply()

```
> results.plyr = ddply(
    .data        = barley,
    .variables   = c("site","year"),
    .fun         = five.num
)
```

Yields at Every Site in Every year: User Base R Functions

Split to pieces

```
> pieces = split(
    x = barley,
    f = list(barley$site, barley$year)
)
```

Initialize results

```
> results = list()
```

Apply to pieces

```
> for(i in seq_along(pieces)) {
    piece              = pieces[[i]]
    results[[i]]       = five.num(piece)
}
```

Combine pieces

```
> results = do.call("rbind", results)
> results = as.data.frame(results)
```

Yields at Every Site in Every year: User Base R Functions Cont.

Not done yet, need proper labels

Obtain the names of pieces

```
> groups = names(pieces)
```

Split the names by dot

```
> groups = strsplit(groups, split="\\.")
```

Make the names a data.frame

```
> groups       = do.call("rbind", groups)
> groups       = as.data.frame(groups)
> names(groups) = c("site", "year")
```

Merge with the five number summary data.frame

```
> results.r = cbind(groups, results)
```