

Package ‘plyr’

June 23, 2009

Type Package

Title Tools for splitting, applying and combining data

Version 0.1.9

Author Hadley Wickham <h.wickham@gmail.com>

Maintainer Hadley Wickham <h.wickham@gmail.com>

Description 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. For example, you might want to fit a model to each spatial location or time point in your study, summarise data by panels or collapse high-dimensional arrays to simpler summary statistics.

URL <http://had.co.nz/plyr>

Depends R (>= 2.8)

Suggests RUnit, abind, tcltk

License GPL

LazyData true

Repository CRAN

Date/Publication 2009-06-23 13:20:14

R topics documented:

.....	2
aapply	3
adply	5
alply	6
as.data.frame.function	7
as.quoted	7
a_ply	8

Baseball batting	9
colwise	10
compact	11
create_progress_bar	12
dapply	13
ddply	14
defaults	15
dlply	16
d_ply	17
each	17
failwith	18
laply	19
ldply	20
llply	21
l_ply	22
maply	22
mdply	23
mlply	24
m_ply	25
Ozone measurements	26
progress_text	27
progress_tk	28
progress_win	28
raply	29
rbind.fill	30
rdply	31
rlply	32
r_ply	33
splat	34
splitter_a	35
splitter_d	36
summarise	37
Index	38

.

*Quote variables***Description**

Create a list of unevaluated expressions for later evaluation

Usage

`. (...)`

Arguments

... unevaluated expressions to be recorded. Specify names if you want the set the names of the resultant variables

Details

This function is similar to `~` in that it is used to capture the name of variables, not their current value. This is used throughout `plyr` to specify the names of variables (or more complicated expressions).

Similar tricks can be performed with `substitute`, but when functions can be called in multiple ways it becomes increasingly tricky to ensure that the values are extracted from the correct frame. Substitute tricks also make it difficult to program against the functions that use them, while the `quoted` class provides `as.quoted.character` to convert strings to the appropriate data structure.

Value

list of symbol and language primitives

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
.(a, b, c)
.(first = a, second = b, third = c)
.(a ^ 2, b - d, log(c))
as.quoted(~ a + b + c)
as.quoted(a ~ b + c)
as.quoted(c("a", "b", "c"))

# Some examples using ddply - look at the column names
ddply(mtcars, "cyl", each(nrow, ncol))
ddply(mtcars, ~ cyl, each(nrow, ncol))
ddply(mtcars, .(cyl), each(nrow, ncol))
ddply(mtcars, .(log(cyl)), each(nrow, ncol))
ddply(mtcars, .(logcyl = log(cyl)), each(nrow, ncol))
ddply(mtcars, .(vs + am), each(nrow, ncol))
ddply(mtcars, .(vsam = vs + am), each(nrow, ncol))
```

aapply

Split array, apply function, and return results in an array

Description

For each slice of an array, apply function then combine results into an array

Usage

```
aapply(.data, .margins, .fun = NULL, ..., .progress = "none", .drop = TRUE)
```

Arguments

<code>.data</code>	matrix, array or data frame to be processed
<code>.margins</code>	a vector giving the subscripts to split up data by. 1 splits up by rows, 2 by columns and <code>c(1,2)</code> by rows and columns, and so on for higher dimensions
<code>.fun</code>	function to apply to each piece
<code>...</code>	other arguments passed on to <code>.fun</code>
<code>.progress</code>	name of the progress bar to use, see create_progress_bar
<code>.drop</code>	should extra dimensions of length 1 be dropped, simplifying the output. Defaults to TRUE

Details

All plyr functions use the same split-apply-combine strategy: they split the input into simpler pieces, apply `.fun` to each piece, and then combine the pieces into a single data structure. This function splits matrices, arrays and data frames by dimensions and combines the result into an array. If there are no results, then this function will return a vector of length 0 (`vector()`).

This function is very similar to [apply](#), except that it will always return an array, and when the function returns >1 d data structures, those dimensions are added on to the highest dimensions, rather than the lowest dimensions. This makes `aapply` idempotent, so that `apply(input, X, identity)` is equivalent to `aperm(input, X)`.

Value

if results are atomic with same type and dimensionality, a vector, matrix or array; otherwise, a list-array (a list with dimensions)

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
dim(ozone)
aapply(ozone, 1, mean)
aapply(ozone, 1, mean, .drop = FALSE)
aapply(ozone, 3, mean)
aapply(ozone, c(1,2), mean)

dim(aapply(ozone, c(1,2), mean))
dim(aapply(ozone, c(1,2), mean, .drop = FALSE))

aapply(ozone, 1, each(min, max))
aapply(ozone, 3, each(min, max))
```

```
standardise <- function(x) (x - min(x)) / (max(x) - min(x))
aapply(ozone, 3, standardise)
aapply(ozone, 1:2, standardise)

aapply(ozone, 1:2, diff)
```

adply

Split array, apply function, and return results in a data frame

Description

For each slice of an array, apply function then combine results into a data frame

Usage

```
adply(.data, .margins, .fun = NULL, ..., .progress = "none")
```

Arguments

<code>.data</code>	matrix, array or data frame to be processed
<code>.margins</code>	a vector giving the subscripts to split up <code>data</code> by. 1 splits up by rows, 2 by columns and <code>c(1,2)</code> by rows and columns, and so on for higher dimensions
<code>.fun</code>	function to apply to each piece
<code>...</code>	other arguments passed on to <code>.fun</code>
<code>.progress</code>	name of the progress bar to use, see create_progress_bar

Details

All `plyr` functions use the same split-apply-combine strategy: they split the input into simpler pieces, apply `.fun` to each piece, and then combine the pieces into a single data structure. This function splits matrices, arrays and data frames by dimensions and combines the result into a data frame. If there are no results, then this function will return a data frame with zero rows and columns (`data.frame()`).

Value

a data frame

Author(s)

Hadley Wickham <h.wickham@gmail.com>

`aply`*Split array, apply function, and return results in a list*

Description

For each slice of an array, apply function then combine results into a list

Usage

```
aply(.data, .margins, .fun = NULL, ..., .progress = "none")
```

Arguments

<code>.data</code>	matrix, array or data frame to be processed
<code>.margins</code>	a vector giving the subscripts to split up data by. 1 splits up by rows, 2 by columns and <code>c(1,2)</code> by rows and columns, and so on for higher dimensions
<code>.fun</code>	function to apply to each piece
<code>...</code>	other arguments passed on to <code>.fun</code>
<code>.progress</code>	name of the progress bar to use, see create_progress_bar

Details

All plyr functions use the same split-apply-combine strategy: they split the input into simpler pieces, apply `.fun` to each piece, and then combine the pieces into a single data structure. This function splits matrices, arrays and data frames by dimensions and combines the result into a list. If there are no results, then this function will return a list of length 0 (`list()`).

`aply` is somewhat similar to [apply](#) for cases where the results are not atomic.

Value

list of results

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
aply(ozone, 3, quantile)
aply(ozone, 3, function(x) table(round(x)))
```

```
as.data.frame.function
```

Make a function return a data frame

Description

Create a new function that returns the existing function wrapped in a data.frame

Usage

```
as.data.frame.function(x, row.names, optional, ...)
```

Arguments

x
row.names
optional
...

Details

This is useful when calling *dply functions with a function that returns a vector, and you want the output in rows, rather than columns

Author(s)

Hadley Wickham <h.wickham@gmail.com>

```
as.quoted
```

Convert input to quoted variables

Description

Convert characters, formulas and calls to quoted .variables

Usage

```
as.quoted(x)
```

Arguments

x

Details

This method is called by default on all plyr functions that take a `.variables` argument, so that equivalent forms can be used anywhere.

Currently conversions exist for character vectors, formulas and call objects.

Value

a list of quoted variables

Author(s)

Hadley Wickham <h.wickham@gmail.com>

See Also

.

Examples

```
as.quoted(c("a", "b", "log(d)"))
as.quoted(a ~ b + log(d))
```

a_ply

Split array, apply function, and discard results

Description

For each slice of an array, apply function and discard results

Usage

```
a_ply(.data, .margins, .fun = NULL, ..., .progress = "none", .print = FALSE)
```

Arguments

<code>.data</code>	matrix, array or data frame to be processed
<code>.margins</code>	a vector giving the subscripts to split up <code>data</code> by. 1 splits up by rows, 2 by columns and <code>c(1,2)</code> by rows and columns, and so on for higher dimensions
<code>.fun</code>	function to apply to each piece
<code>...</code>	other arguments passed on to <code>.fun</code>
<code>.progress</code>	name of the progress bar to use, see create_progress_bar
<code>.print</code>	

Details

All plyr functions use the same split-apply-combine strategy: they split the input into simpler pieces, apply `.fun` to each piece, and then combine the pieces into a single data structure. This function splits matrices, arrays and data frames by dimensions and discards the output. This is useful for functions that you are calling purely for their side effects like display plots and saving output.

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Baseball batting *Yearly batting records for all major league baseball players*

Description

This data frame contains batting statistics for a subset of players collected from <http://www.baseball-databank.org/>. There are a total of 21,699 records, covering 1,228 players from 1871 to 2007. Only players with more 15 seasons of play are included.

Variables:

- id, unique player id
- year, year of data
- stint
- team, team played for
- lg, league
- g, number of games
- ab, number of times at bat
- r, number of runs
- h, hits, times reached base because of a batted, fair ball without error by the defense
- X2b, hits on which the batter reached second base safely
- X3b, hits on which the batter reached third base safely
- hr, number of home runs
- rbi, runs batted in
- sb, stolen bases
- cs, caught stealing
- bb, base on balls (walk)
- so, strike outs
- ibb, intentional base on balls
- hbp, hits by pitch
- sh, sacrifice hits
- sf, sacrifice flies
- gidp, ground into double play

Usage

```
data(baseball)
```

Format

A 21699 x 22 data frame

References

<http://www.baseball-databank.org/>

Examples

```

baberruth <- subset(baseball, id == "ruthba01")
baberruth$cyear <- baberruth$year - min(baberruth$year) + 1

calculate_cyear <- function(df) {
  transform(df,
    cyear = year - min(year),
    cpercent = (year - min(year)) / (max(year) - min(year))
  )
}

baseball <- ddply(baseball, .(id), calculate_cyear)
baseball <- subset(baseball, ab >= 25)

model <- function(df) {
  lm(rbi / ab ~ cyear, data=df)
}
model(baberruth)
models <- dplyr::dply(baseball, .(id), model)

```

colwise

Column-wise function

Description

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

Usage

```
colwise(.fun, .cols = function(x) TRUE)
```

Arguments

<code>.fun</code>	function
<code>.cols</code>	either function that tests columns for inclusion, or a quoted object giving which columns to process

Details

catcolwise and numcolwise provide version that only operate on discrete and numeric variables respectively

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
# Count number of missing values
nmissing <- function(x) sum(is.na(x))

# Apply to every column in a data frame
colwise(nmissing)(baseball)
# This syntax looks a little different. It is shorthand for the
# the following:
f <- colwise(nmissing)
f(baseball)

# This is particularly useful in conjunction with dply
ddply(baseball, .(year), colwise(nmissing))

# To operate only on specified columns, supply them as the second
# argument. Many different forms are accepted.
ddply(baseball, .(year), colwise(nmissing, .(sb, cs, so)))
ddply(baseball, .(year), colwise(nmissing, c("sb", "cs", "so")))
ddply(baseball, .(year), colwise(nmissing, ~ sb + cs + so))

# Alternatively, you can specify a boolean function that determines
# whether or not a column should be included
ddply(baseball, .(year), colwise(nmissing, is.character))
ddply(baseball, .(year), colwise(nmissing, is.numeric))
ddply(baseball, .(year), colwise(nmissing, is.discrete))

# These last two cases are particularly common, so some shortcuts are
# provided:
ddply(baseball, .(year), numcolwise(nmissing))
ddply(baseball, .(year), catcolwise(nmissing))
```

compact

Compact list

Description

Remove all NULL entries from a list

Usage

```
compact(l)
```

Arguments

1 list

Author(s)

Hadley Wickham <h.wickham@gmail.com>

```
create_progress_bar
```

Create progress bar

Description

Create progress bar object from text string.

Usage

```
create_progress_bar(name = "none")
```

Arguments

name type of progress bar to create

Details

Progress bars give feedback on how apply step is proceeding. This is mainly useful for long running functions, as for short functions, the time taken up by splitting and combining may be on the same order (or longer) as the apply step. Additionally, for short functions, the time needed to update the progress bar can significantly slow down the process. For the trivial examples below, using the tk progress bar slows things down by a factor of a thousand.

Note that progress bar is approximate, and if the time taken by individual function applications is highly non-uniform it may not be very informative of the time left.

There are currently four types of progress bar: "none", "text", "tk", and "win". See the individual documentation for more details. In plyr functions, these can either be specified by name, or you can create the progress bar object yourself if you want more control over its appearance. See the examples.

Author(s)

Hadley Wickham <h.wickham@gmail.com>

See Also

[progress_none](#), [progress_text](#), [progress_tk](#), [progress_win](#)

Examples

```
l_ply(1:1000, identity, .progress = "none")
l_ply(1:1000, identity, .progress = "tk")
l_ply(1:1000, identity, .progress = "text")
l_ply(1:1000, identity, .progress = progress_text(char = "-"))
```

dply

Split data frame, apply function, and return results in an array

Description

For each subset of data frame, apply function then combine results into an array

Usage

```
dply(.data, .variables, .fun = NULL, ..., .progress = "none", .drop = TRUE)
```

Arguments

<code>.data</code>	data frame to be processed
<code>.variables</code>	variables to split data frame by, as quoted variables, a formula or character vector
<code>.fun</code>	function to apply to each piece
<code>...</code>	other arguments passed on to <code>.fun</code>
<code>.progress</code>	name of the progress bar to use, see create_progress_bar
<code>.drop</code>	should extra dimensions of length 1 be dropped, simplifying the output. Defaults to TRUE

Details

All plyr functions use the same split-apply-combine strategy: they split the input into simpler pieces, apply `.fun` to each piece, and then combine the pieces into a single data structure. This function splits data frames by variable and combines the result into an array. If there are no results, then this function will return a vector of length 0 (`vector()`).

`dply` with a function that operates column-wise is similar to [aggregate](#).

Value

if results are atomic with same type and dimensionality, a vector, matrix or array; otherwise, a list-array (a list with dimensions)

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```

dapply(baseball, .(year), nrow)

# Several different ways of summarising by variables that should not be
# included in the summary

dapply(baseball[, c(2, 6:9)], .(year), mean)
dapply(baseball[, 6:9], .(baseball$year), mean)
dapply(baseball, .(year), function(df) mean(df[, 6:9]))

```

ddply

Split data frame, apply function, and return results in a data frame

Description

For each subset of a data frame, apply function then combine results into a data frame

Usage

```
ddply(.data, .variables, .fun = NULL, ..., .progress = "none", .drop = TRUE)
```

Arguments

<code>.data</code>	data frame to be processed
<code>.variables</code>	variables to split data frame by, as quoted variables, a formula or character vector
<code>.fun</code>	function to apply to each piece
<code>...</code>	other arguments passed on to <code>.fun</code>
<code>.progress</code>	name of the progress bar to use, see create_progress_bar
<code>.drop</code>	

Details

All plyr functions use the same split-apply-combine strategy: they split the input into simpler pieces, apply `.fun` to each piece, and then combine the pieces into a single data structure. This function splits data frames by variables and combines the result into a data frame. If there are no results, then this function will return a data frame with zero rows and columns (`data.frame()`).

The most unambiguous behaviour is achieved when `.fun` returns a data frame - in that case pieces will be combined with `rbind.fill`. If `.fun` returns an atomic vector of fixed length, it will be `rbinded` together and converted to a data frame. Any other values will result in an error.

Value

a data frame

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
ddply(baseball, .(year), "nrow")
ddply(baseball, .(lg), c("nrow", "ncol"))

mean_rbi <- function(df) mean(df$rbi, na.rm=TRUE)
rbi <- ddply(baseball, .(year), mean_rbi)
with(rbi, plot(year, V1, type="l"))
rbi <- ddply(baseball, .(year), "mean_rbi")

mean_rbi <- function(rbi, ...) mean(rbi, na.rm=TRUE)
rbi <- ddply(baseball, .(year), splat(mean_rbi))

ddply(baseball, .(year), numcolwise(mean), na.rm=TRUE)
base2 <- ddply(baseball, .(id), function(df) {
  transform(df, career_year = year - min(year) + 1)
})
```

defaults

Set defaults

Description

Convenient method for combining a list of values with their defaults.

Usage

```
defaults(x, y)
```

Arguments

x	list of values
y	defaults

Author(s)

Hadley Wickham <h.wickham@gmail.com>

dply

Split data frame, apply function, and return results in a list

Description

For each subset of a data frame, apply function then combine results into a list

Usage

```
dply(.data, .variables, .fun = NULL, ..., .progress = "none", .drop = TRUE)
```

Arguments

<code>.data</code>	data frame to be processed
<code>.variables</code>	variables to split data frame by, as quoted variables, a formula or character vector
<code>.fun</code>	function to apply to each piece
<code>...</code>	other arguments passed on to <code>.fun</code>
<code>.progress</code>	name of the progress bar to use, see create_progress_bar
<code>.drop</code>	

Details

All plyr functions use the same split-apply-combine strategy: they split the input into simpler pieces, apply `.fun` to each piece, and then combine the pieces into a single data structure. This function splits data frames by variables and combines the result into a list. If there are no results, then this function will return a list of length 0 (`list()`).

`dply` is similar to `by` except that the results are returned in a different format.

Value

if results are atomic with same type and dimensionality, a vector, matrix or array; otherwise, a list-array (a list with dimensions)

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
linmod <- function(df) lm(rbi ~ year, data = transform(df, year = year - min(year)))
models <- dply(baseball, .(id), linmod)
models[[1]]

coef <- ldply(models, coef)
with(coef, plot(`(Intercept)`, year))
qual <- laply(models, function(mod) summary(mod)$r.squared)
hist(qual)
```

d_ply	<i>Split data frame, apply function, and discard results</i>
-------	--

Description

For each subset of a data frame, apply function and discard results

Usage

```
d_ply(.data, .variables, .fun = NULL, ..., .progress = "none", .print = FALSE)
```

Arguments

.data	data frame to be processed
.variables	variables to split data frame by, as quoted variables, a formula or character vector
.fun	function to apply to each piece
...	other arguments passed on to .fun
.progress	name of the progress bar to use, see create_progress_bar
.print	

Details

All plyr functions use the same split-apply-combine strategy: they split the input into simpler pieces, apply .fun to each piece, and then combine the pieces into a single data structure. This function splits data frames by variable and discards the output. This is useful for functions that you are calling purely for their side effects like display plots and saving output.

Author(s)

Hadley Wickham <h.wickham@gmail.com>

each	<i>Aggregate multiple functions into a single function</i>
------	--

Description

Combine multiple functions to a single function returning a named vector of outputs

Usage

```
each(...)
```

Arguments

...	functions to combine
-----	----------------------

Details

Each function should produce a single number as output

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
each(min, max) (1:10)
each("min", "max") (1:10)
each(c("min", "max")) (1:10)
each(c(min, max)) (1:10)
each(length, mean, var) (rnorm(100))
```

failwith

Fail with

Description

Modify a function so that it returns a default value when there is an error.

Usage

```
failwith(default = NULL, f, quiet = FALSE)
```

Arguments

default	default value
f	function
quiet	

Value

a function

Author(s)

Hadley Wickham <h.wickham@gmail.com>

See Also

[try_default](#)

Examples

```
f <- function(x) if (x == 1) stop("Error!") else 1
## Not run:
f(1)
f(2)
## End(Not run)

safef <- failwith(NULL, f)
safef(1)
safef(2)
```

lapply

Split list, apply function, and return results in an array

Description

For each element of a list, apply function then combine results into an array

Usage

```
lapply(.data, .fun = NULL, ..., .progress = "none", .drop = TRUE)
```

Arguments

<code>.data</code>	input list
<code>.fun</code>	function to apply to each piece
<code>...</code>	other arguments passed on to <code>.fun</code>
<code>.progress</code>	name of the progress bar to use, see create_progress_bar
<code>.drop</code>	should extra dimensions of length 1 be dropped, simplifying the output. Defaults to TRUE

Details

All plyr functions use the same split-apply-combine strategy: they split the input into simpler pieces, apply `.fun` to each piece, and then combine the pieces into a single data structure. This function splits lists by elements and combines the result into an array. If there are no results, then this function will return a vector of length 0 (`vector()`).

`lapply` is very similar in spirit to [sapply](#) except that it will always return an array, and the output is transposed with respect `sapply` - each element of the list corresponds to a column, not a row.

Value

if results are atomic with same type and dimensionality, a vector, matrix or array; otherwise, a list-array (a list with dimensions)

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```

laply(baseball, is.factor)
# cf
ldply(baseball, is.factor)
colwise(is.factor)(baseball)

laply(seq_len(10), identity)
laply(seq_len(10), rep, times = 4)
laply(seq_len(10), matrix, nrow = 2, ncol = 2)

```

 ldply

Split list, apply function, and return results in a data frame

Description

For each element of a list, apply function then combine results into a data frame

Usage

```
ldply(.data, .fun = NULL, ..., .progress = "none")
```

Arguments

<code>.data</code>	list to be processed
<code>.fun</code>	function to apply to each piece
<code>...</code>	other arguments passed on to <code>.fun</code>
<code>.progress</code>	name of the progress bar to use, see create_progress_bar

Details

All plyr functions use the same split-apply-combine strategy: they split the input into simpler pieces, apply `.fun` to each piece, and then combine the pieces into a single data structure. This function splits lists by elements and combines the result into a data frame. If there are no results, then this function will return a data frame with zero rows and columns (`data.frame()`).

The most unambiguous behaviour is achieved when `.fun` returns a data frame - in that case pieces will be combined with `rbind.fill`. If `.fun` returns an atomic vector of fixed length, it will be `rbinded` together and converted to a data frame. Any other values will result in an error.

Value

a data frame

Author(s)

Hadley Wickham <h.wickham@gmail.com>

lply

Split list, apply function, and return results in a list

Description

For each element of a list, apply function then combine results into a list

Usage

```
lply(.data, .fun = NULL, ..., .progress = "none", .inform = FALSE)
```

Arguments

<code>.data</code>	list to be processed
<code>.fun</code>	function to apply to each piece
<code>...</code>	other arguments passed on to <code>.fun</code>
<code>.progress</code>	name of the progress bar to use, see create_progress_bar
<code>.inform</code>	

Details

All plyr functions use the same split-apply-combine strategy: they split the input into simpler pieces, apply `.fun` to each piece, and then combine the pieces into a single data structure. This function splits lists by elements and combines the result into a list. If there are no results, then this function will return a list of length 0 (`list()`).

`lply` is equivalent to `lapply` except that it will preserve labels and can display a progress bar.

Value

list of results

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
lply(lply(mtcars, round), table)
lply(baseball, summary)
# Examples from ?lapply
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))

lply(x, mean)
lply(x, quantile, probs = 1:3/4)
```

`l_ply`*Split list, apply function, and discard results*

Description

For each element of a list, apply function and discard results

Usage

```
l_ply(.data, .fun = NULL, ..., .progress = "none", .print = FALSE)
```

Arguments

<code>.data</code>	list to be processed
<code>.fun</code>	function to apply to each piece
<code>...</code>	other arguments passed on to <code>.fun</code>
<code>.progress</code>	name of the progress bar to use, see create_progress_bar
<code>.print</code>	

Details

All plyr functions use the same split-apply-combine strategy: they split the input into simpler pieces, apply `.fun` to each piece, and then combine the pieces into a single data structure. This function splits lists by elements and discards the output. This is useful for functions that you are calling purely for their side effects like display plots and saving output.

Author(s)

Hadley Wickham <h.wickham@gmail.com>

`maply`*Call function with arguments in array or data frame, returning an array*

Description

Call a multi-argument function with values taken from columns of an data frame or array, and combine results into an array

Usage

```
maply(.data, .fun = NULL, ..., .progress = "none")
```

Arguments

`.data` matrix or data frame to use as source of arguments
`.fun` function to be called with varying arguments
`...` other arguments passed on to `.fun`
`.progress` name of the progress bar to use, see [create_progress_bar](#)

Details

The `m*ply` functions are the `plyr` version of `mapply`, specialised according to the type of output they produce. These functions are just a convenient wrapper around `a*ply` with `margins = 1` and `.fun` wrapped in `splat`.

This function combines the result into an array. If there are no results, then this function will return a vector of length 0 (`vector()`).

Value

if results are atomic with same type and dimensionality, a vector, matrix or array; otherwise, a list-array (a list with dimensions)

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
maply(cbind(mean = 1:5, sd = 1:5), rnorm, n = 5)
maply(cbind(1:5, 1:5), rnorm, n = 5)
maply(expand.grid(mean = 1:5, sd = 1:5), rnorm, n = 5)
```

mdply	<i>Call function with arguments in array or data frame, returning a data frame</i>
-------	--

Description

Call a multi-argument function with values taken from columns of an data frame or array, and combine results into a data frame

Usage

```
mdply(.data, .fun = NULL, ..., .progress = "none")
```

Arguments

<code>.data</code>	matrix or data frame to use as source of arguments
<code>.fun</code>	function to be called with varying arguments
<code>...</code>	other arguments passed on to <code>.fun</code>
<code>.progress</code>	name of the progress bar to use, see create_progress_bar

Details

The `m*ply` functions are the `plyr` version of `mapply`, specialised according to the type of output they produce. These functions are just a convenient wrapper around `a*ply` with `margins = 1` and `.fun` wrapped in `splat`.

This function combines the result into a data frame. If there are no results, then this function will return a data frame with zero rows and columns (`data.frame()`).

Value

a data frame

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
mdply(data.frame(mean = 1:5, sd = 1:5), rnorm, n = 2)
mdply(expand.grid(mean = 1:5, sd = 1:5), rnorm, n = 2)
mdply(cbind(mean = 1:5, sd = 1:5), rnorm, n = 5)
```

mply

Call function with arguments in array or data frame, returning a list

Description

Call a multi-argument function with values taken from columns of an data frame or array, and combine results into a list

Usage

```
mply(.data, .fun = NULL, ..., .progress = "none")
```

Arguments

<code>.data</code>	matrix or data frame to use as source of arguments
<code>.fun</code>	function to be called with varying arguments
<code>...</code>	other arguments passed on to <code>.fun</code>
<code>.progress</code>	name of the progress bar to use, see create_progress_bar

Details

The `m*ply` functions are the `plyr` version of `mapply`, specialised according to the type of output they produce. These functions are just a convenient wrapper around `a*ply` with `margins = 1` and `.fun` wrapped in `splat`.

This function combines the result into a list. If there are no results, then this function will return a list of length 0 (`list()`).

Value

list of results

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
mply(cbind(1:4, 4:1), rep)
mply(cbind(1:4, times = 4:1), rep)

mply(cbind(1:4, 4:1), seq)
mply(cbind(1:4, length = 4:1), seq)
mply(cbind(1:4, by = 4:1), seq, to = 20)
```

`m_ply`

Call function with arguments in array or data frame, discarding results

Description

Call a multi-argument function with values taken from columns of an data frame or array, and discard results

Usage

```
m_ply(.data, .fun = NULL, ..., .progress = "none")
```

Arguments

<code>.data</code>	matrix or data frame to use as source of arguments
<code>.fun</code>	function to be called with varying arguments
<code>...</code>	other arguments passed on to <code>.fun</code>
<code>.progress</code>	name of the progress bar to use, see create_progress_bar

Details

The `m*ply` functions are the `plyr` version of `mapply`, specialised according to the type of output they produce. These functions are just a convenient wrapper around `a*ply` with `margins = 1` and `.fun` wrapped in `splat`.

This function combines the result into a list. If there are no results, then this function will return a list of length 0 (`list()`).

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Ozone measurements *Monthly ozone measurements over Central America*

Description

This data set is a subset of the data from the 2006 ASA Data expo challenge, <http://stat-computing.org/dataexpo/2006/>. The data are monthly ozone averages on a very coarse 24 by 24 grid covering Central America, from Jan 1995 to Dec 2000. The data is stored in a 3d array with the first two dimensions representing latitude and longitude, and the third representing time.

Usage

```
data(ozone)
```

Format

A 24 x 24 x 72 numeric array

References

<http://stat-computing.org/dataexpo/2006/>

Examples

```
value <- ozone[1, 1, ]
time <- 1:72
month.abbr <- c("Jan", "Feb", "Mar", "Apr", "May",
  "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
month <- factor(rep(month.abbr, length = 72), levels = month.abbr)
year <- rep(1:6, each = 12)
deseasf <- function(value) lm(value ~ month - 1)

models <- alply(ozone, 1:2, deseasf)
coefs <- laply(models, coef)
dimnames(coefs)[[3]] <- month.abbr
names(dimnames(coefs))[3] <- "month"
```

```
deseas <- laply(models, resid)
dimnames(deseas)[[3]] <- 1:72
names(dimnames(deseas))[3] <- "time"

dim(coefs)
dim(deseas)
```

progress_text	<i>Text progress bar</i>
---------------	--------------------------

Description

A textual progress bar

Usage

```
progress_text(style = 3, ...)
```

Arguments

```
style
...
```

Details

This progress bar displays a textual progress bar that works on all platforms. It is a thin wrapper around the built-in [setTxtProgressBar](#) and can be customised in the same way.

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
l_ply(1:1000, identity, .progress = "text")
l_ply(1:1000, identity, .progress = progress_text(char = "-"))
```

progress_tk

Graphical progress bar, powered by Tk

Description

A graphical progress bar displayed in a Tk window

Usage

```
progress_tk(title = "plyr progress", label = "Working...", ...)
```

Arguments

title	window title
label	progress bar label (inside window)
...	other arguments passed on to tkProgressBar

Details

This graphical progress will appear in a separate window.

Author(s)

Hadley Wickham <h.wickham@gmail.com>

See Also

[tkProgressBar](#) for the function that powers this progress bar

Examples

```
l_ply(1:1000, identity, .progress = "tk")
l_ply(1:1000, identity, .progress = progress_tk(width=400))
l_ply(1:1000, identity, .progress = progress_tk(label=""))
```

progress_win*Graphical progress bar, powered by Windows*

Description

A graphical progress bar displayed in a separate window

Usage

```
progress_win(title = "plyr progress", ...)
```

Arguments

title window title
 ... other arguments passed on to [winProgressBar](#)

Details

This graphical progress only works on Windows.

Author(s)

Hadley Wickham <h.wickham@gmail.com>

See Also

[winProgressBar](#) for the function that powers this progress bar

Examples

```
if(exists("winProgressBar")) {
  l_ply(1:1000, identity, .progress = "win")
  l_ply(1:1000, identity, .progress = progress_win(title="Working..."))
}
```

 rply

Replicate expression and return results in a array

Description

Evaluate expression n times then combine results into an array

Usage

```
rply(.n, .expr, .progress = "none", .drop = TRUE)
```

Arguments

.n number of times to evaluate the expression
 .expr expression to evaluate
 .progress name of the progress bar to use, see [create_progress_bar](#)
 .drop

Details

This function runs an expression multiple times, and combines the result into a data frame. If there are no results, then this function returns a vector of length 0 (`vector(0)`). This function is equivalent to `replicate`, but will always return results as a vector, matrix or array.

@keyword manip @arguments number of times to evaluate the expression @arguments expression to evaluate @arguments name of the progress bar to use, see `create_progress_bar` @value if results are atomic with same type and dimensionality, a vector, matrix or array; otherwise, a list-array (a list with dimensions)

Value

if results are atomic with same type and dimensionality, a vector, matrix or array; otherwise, a list-array (a list with dimensions)

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```

rapply(100, mean(runif(100)))
rapply(100, each(mean, var)(runif(100)))

rapply(10, runif(4))
rapply(10, matrix(runif(4), nrow=2))

# See the central limit theorem in action
hist(rapply(1000, mean(rexp(10))))
hist(rapply(1000, mean(rexp(100))))
hist(rapply(1000, mean(rexp(1000))))

```

rbind.fill

Combine objects by row, filling in missing columns

Description

rbinds a list of data frames filling missing columns with NA

Usage

```
rbind.fill(...)
```

Arguments

... data frames to row bind together

Details

This is a minor enhancement to `rbind` which adds in columns that are not present in all inputs.

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```

rbind.fill(mtcars[c("mpg", "wt")], mtcars[c("wt", "cyl")])

bplayer <- split(baseball, baseball$id)
system.time(b1 <- do.call("rbind", bplayer))
rownames(b1) <- NULL
system.time(b2 <- rbind.fill(bplayer))
stopifnot(all.equal(b1, b2))

a <- data.frame(a = factor(letters[1:3]), b = 1:3, c = date())
b <- data.frame(a = factor(letters[3:5]),
               d = as.Date(c("2008-01-01", "2009-01-01", "2010-01-01")))
ab1 <- rbind.fill(a, b)[, letters[1:4]]
ab2 <- rbind.fill(b, a)[c(4:6, 1:3), letters[1:4]]
ab2$a <- factor(ab2$a, levels(ab1$a))
rownames(ab2) <- NULL
stopifnot(all.equal(ab1, ab2))

```

 rdply

Replicate expression and return results in a data frame

Description

Evaluate expression `n` times then combine results into a data frame

Usage

```
rdply(.n, .expr, .progress = "none")
```

Arguments

<code>.n</code>	number of times to evaluate the expression
<code>.expr</code>	expression to evaluate
<code>.progress</code>	name of the progress bar to use, see create_progress_bar

Details

This function runs an expression multiple times, and combines the result into a data frame. If there are no results, then this function returns a data frame with zero rows and columns (`data.frame()`). This function is equivalent to `replicate`, but will always return results as a data frame.

Value

a data frame

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
rdply(20, mean(runif(100)))  
rdply(20, each(mean, var)(runif(100)))  
rdply(20, data.frame(x = runif(2)))
```

rply

Replicate expression and return results in a list

Description

Evaluate expression `n` times then combine results into a list

Usage

```
rply(.n, .expr, .progress = "none")
```

Arguments

<code>.n</code>	number of times to evaluate the expression
<code>.expr</code>	expression to evaluate
<code>.progress</code>	name of the progress bar to use, see create_progress_bar

Details

This function runs an expression multiple times, and combines the result into a list. If there are no results, then this function will return a list of length 0 (`list()`). This function is equivalent to [replicate](#), but will always return results as a list.

@keyword manip @arguments number of times to evaluate the expression @arguments expression to evaluate @arguments name of the progress bar to use, see [create_progress_bar](#) @value list of results

Value

list of results

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
mods <- rply(100, lm(y ~ x, data=data.frame(x=rnorm(100), y=rnorm(100))))  
hist(lapply(mods, function(x) summary(x)$r.squared))
```

r_ply

Replicate expression and discard results

Description

Evaluate expression n times then discard results

Usage

```
r_ply(.n, .expr, .progress = "none", .print = FALSE)
```

Arguments

.n	number of times to evaluate the expression
.expr	expression to evaluate
.progress	name of the progress bar to use, see create_progress_bar
.print	

Details

This function runs an expression multiple times, discarding the results. This function is equivalent to [replicate](#), but never returns anything

@keyword manip @arguments number of times to evaluate the expression @arguments expression to evaluate @arguments name of the progress bar to use, see [create_progress_bar](#) @argument automatically print each result? (default: FALSE)

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
r_ply(10, plot(runif(50)))  
r_ply(25, hist(runif(1000)))
```

splat	<i>'Splat' arguments to a function</i>
-------	--

Description

Wraps a function in do.call

Usage

```
splat(flat)
```

Arguments

flat function to splat

Details

This is useful when you want to pass a function a row of data frame or array, and don't want to manually pull it apart in your function.

@arguments function to splat @value a function

Value

a function

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
hp_per_cyl <- function(hp, cyl, ...) hp / cyl
splat(hp_per_cyl)(mtcars[1,])
splat(hp_per_cyl)(mtcars)

f <- function(mpg, wt, ...) data.frame(mw = mpg / wt)
ddply(mtcars, .(cyl), splat(f))
```

splitter_a	<i>Split an array by .margins</i>
------------	-----------------------------------

Description

Split a 2d or higher data structure into lower-d pieces based

Usage

```
splitter_a(data, .margins = 1)
```

Arguments

data
.margins

Details

This is the workhorse of the `a*ply` functions. Given a >1 d data structure (matrix, array, data.frame), it splits it into pieces based on the subscripts that you supply. Each piece is a lower dimensional slice.

The margins are specified in the same way as `apply`, but `splitter_a` just splits up the data, while `apply` also applies a function and combines the pieces back together. This function also includes enough information to recreate the split from attributes on the list of pieces.

Value

a list of lower-d slices, with attributes that record split details

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
splitter_a(mtcars, 1)
splitter_a(mtcars, 2)

splitter_a(ozone, 2)
splitter_a(ozone, 3)
splitter_a(ozone, 1:2)
```

splitter_d *Split a data frame by variables*

Description

Split a data frame into pieces based on variable contained in that data frame

Usage

```
splitter_d(data, .variables = NULL, drop = TRUE)
```

Arguments

data data frame
.variables a [quoted](#) list of variables, a formula, or character vector
drop

Details

This is the workhorse of the `d*ply` functions. Based on the variables you supply, it breaks up a single data frame into a list of data frames, each containing a single combination from the levels of the specified variables.

This is basically a thin wrapper around [split](#) which evaluates the variables in the context of the data, and includes enough information to reconstruct the labelling of the data frame after other operations.

Value

a list of data.frames, with attributes that record split details

Author(s)

Hadley Wickham <h.wickham@gmail.com>

See Also

[.](#) for quoting variables, [split](#)

Examples

```
splitter_d(mtcars, .(cyl))
splitter_d(mtcars, .(vs, am))
splitter_d(mtcars, .(am, vs))

mtcars$cyl2 <- factor(mtcars$cyl, levels = c(2, 4, 6, 8, 10))
splitter_d(mtcars, .(cyl2), drop = TRUE)
splitter_d(mtcars, .(cyl2), drop = FALSE)
```

```
mtcars$cyl3 <- ifelse(mtcars$vs == 1, NA, mtcars$cyl)
splitter_d(mtcars, .(cyl3))
splitter_d(mtcars, .(cyl3, vs))
splitter_d(mtcars, .(cyl3, vs), drop = FALSE)
```

summarise

Summarise a data frame.

Description

Create a new data frame that summarises an existing data frame.

Usage

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

Arguments

<code>.data</code>	The data frame to be summarised
<code>...</code>	Further arguments of the form <code>var = value</code>

Details

Summarise works in an analogous way to `transform`, except instead of adding columns to an existing data frame, it creates a new one. This is particularly useful in conjunction with `ddply` as it makes it easy to perform group-wise summaries.

Author(s)

Hadley Wickham <h.wickham@gmail.com>

Examples

```
summarise(baseball,
  duration = max(year) - min(year),
  nteams = length(unique(team)))
ddply(baseball, "id", summarise,
  duration = max(year) - min(year),
  nteams = length(unique(team)))
```

Index

*Topic **datasets**

Baseball batting, 8
Ozone measurements, 25

*Topic **debugging**

failwith, 17

*Topic **manip**

a_ply, 7
aapply, 3
adply, 4
alply, 5
as.data.frame.function, 6
compact, 10
d_ply, 16
daply, 12
ddply, 13
defaults, 14
dlply, 15
each, 16
l_ply, 21
laply, 18
ldply, 19
llply, 20
m_ply, 24
maply, 21
mdply, 22
mlply, 23
r_ply, 32
raply, 28
rbind.fill, 29
rdply, 30
rlply, 31
summarise, 36

*Topic **utilities**

create_progress_bar, 11

., 2, 7, 35

~, 2

a_ply, 7
aapply, 3
adply, 4

aggregate, 12

alply, 5

apply, 3, 5, 34

as.data.frame.function, 6

as.quoted, 6

as.quoted.character, 2

baseball(*Baseball batting*), 8

Baseball batting, 8

by, 15

c.quoted(*as.quoted*), 6

catcolwise(*colwise*), 9

colwise, 9

compact, 10

create_progress_bar, 3–5, 7, 11, 12,
13, 15, 16, 18–24, 28–32

d_ply, 16

daply, 12

ddply, 13, 36

defaults, 14

dlply, 15

each, 16

failwith, 17

l_ply, 21

laply, 18

lapply, 20

ldply, 19

llply, 20

m_ply, 24

maply, 21

mdply, 22

mlply, 23

numcolwise(*colwise*), 9

ozone(*Ozone measurements*), 25

Ozone measurements, 25

progress_none, 11
progress_text, 11, 26
progress_tk, 11, 27
progress_win, 11, 27

quoted, 35
quoted(.), 2

r_ply, 32
rply, 28
rbind, 30
rbind.fill, 13, 19, 29
rdply, 30
replicate, 29–32
rlply, 31

sapply, 18
setTxtProgressBar, 26
splat, 22–25, 33
split, 35
splitter_a, 34
splitter_d, 35
substitute, 2
summarise, 36
summarize(*summarise*), 36

tkProgressBar, 27
try_default, 17

winProgressBar, 28