Package 'plyr'

January 8, 2012

Type Package

Title Tools for splitting, applying and combining data

Version 1.7.1

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. The development of plyr has been generously supported by BD (Becton Dickinson).

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

Depends R (>= 2.11.0)

Suggests abind, testthat (>= 0.2), tcltk, foreach, itertools,iterators

License MIT

LazyData true

Collate 'dimensions.r' 'helper-arrange.r' 'helper-col-wise.r''helper-count.r' 'helper-data-frame.r' 'helper-each.r''helper-match-df.r' 'helper-mutate.r' 'helper-quick-df.r''helper-rename.r' 'helper-round-any.r' 'helper-splat.r''helper-strip-splits.r' 'helper-summarise.r' 'helper-try.r''helper-vaggregate.r' 'id.r' 'immutable.r' 'indexed-array.r''indexed-data-frame.r' 'indexed.r' 'join.r' 'loop-apply.r''ply-array.r' 'ply-data-frame.r' 'ply-iterator.r' 'ply-list.r''ply-mapply.r' 'ply-null.r' 'ply-replicate.r' 'progress.r''quote.r' 'rbind-matrix.r' 'rbind.r' 'simplify-array.r''simplify-data-frame.r' 'simplify-vector.r' 'split-array.r''split-data-frame.r' 'split-indices.r' 'split.r' 'utils.r''data.r' 'helper-defaults.r'

Repository CRAN

Date/Publication 2012-01-08 14:36:10

R topics documented:

. *Quote variables to create a list of unevaluated expressions for later evaluation.*

Description

This function is similar to `\textasciitilde` 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).

Usage

```
.(..., .env = parent.frame())
```

Arguments

`...`	unevaluated expressions to be recorded. Specify names if you want the set the names of the resultant variables
`.env`	environment in which unbound symbols in `...` should be evaluated. Defaults to the environment in which `.` was executed.

Details

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

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))
```

---

aaply *Split array, apply function, and return results in an array.*

Description

For each slice of an array, apply function, keeping results as an array. 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 aaply idempotent, so that apply(input, X, identity) is equivalent to aperm(input, X).

Usage

```
aaply(.data, .margins, .fun = NULL, ..., .expand = TRUE,
  .progress = "none", .drop = TRUE, .parallel = FALSE)
```

Arguments

.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
.data	matrix, array or data frame to be processed
.margins	a vector giving the subscripts to split up data by. 1 splits up by rows, 2 by columns and c(1,2) by rows and columns, and so on for higher dimensions
.expand	if .data is a data frame, should output be 1d (expand = FALSE), with an element for each row; or nd (expand = TRUE), with a
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach dimension for each variable.
.drop	should extra dimensions of length 1 in the output be dropped, simplifying the output. Defaults to TRUE

Value

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

Input

This function splits matrices, arrays and data frames by dimensions

Output

If there are no results, then this function will return a vector of length 0 (`vector()`).

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

See Also

Other array input: `adply`, `alply`

Other array output: `daply`, `laply`

Examples

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

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

aaply(ozone, 1, each(min, max))
aaply(ozone, 3, each(min, max))

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

aaply(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, ..., .expand = TRUE,
  .progress = "none", .parallel = FALSE)
```

Arguments

.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
.data	matrix, array or data frame to be processed
.margins	a vector giving the subscripts to split up data by. 1 splits up by rows, 2 by columns and c(1,2) by rows and columns, and so on for higher dimensions
.expand	if .data is a data frame, should output be 1d (expand = FALSE), with an element for each row; or nd (expand = TRUE), with a
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach dimension for each variable.

Value

A data frame, as described in the output section.

Input

This function splits matrices, arrays and data frames by dimensions

Output

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.

If there are no results, then this function will return a data frame with zero rows and columns (data.frame()).

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

See Also

Other array input: aaply, alply

Other data frame output: ddply, ldply

alply *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. `alply` is somewhat similar to `apply` for cases where the results are not atomic.

Usage

```
alply(.data, .margins, .fun = NULL, ..., .expand = TRUE,
  .progress = "none", .parallel = FALSE)
```

Arguments

`.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`
`.data`	matrix, array or data frame to be processed
`.margins`	a vector giving the subscripts to split up `data` by. 1 splits up by rows, 2 by columns and c(1,2) by rows and columns, and so on for higher dimensions
`.expand`	if `.data` is a data frame, should output be 1d (expand = FALSE), with an element for each row; or nd (expand = TRUE), with a
`.parallel`	if `TRUE`, apply function in parallel, using parallel backend provided by foreach dimension for each variable.

Value

list of results

Input

This function splits matrices, arrays and data frames by dimensions

Output

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

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

See Also

Other array input: `aaply`, `adply`

Other list output: `dlply`, `llply`

Examples

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

`arrange` *Order a data frame by its colums.*

Description

This function completes the subsetting, transforming and ordering triad with a function that works in a similar way to `subset` and `transform` but for reordering a data frame by its columns. This saves a lot of typing!

Usage

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

Arguments

`df`	data frame to reorder
`...`	expressions evaluated in the context of `df` and then fed to `order`

Examples

```
mtcars[with(mtcars, order(cyl, disp)), ]
arrange(mtcars, cyl, disp)
arrange(mtcars, cyl, desc(disp))
```

`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

```
## S3 method for class 'function'
as.data.frame(x, row.names, optional,
  ...)
```

Arguments

x	function to make return a data frame
row.names	necessary to match the generic, but not used
optional	necessary to match the generic, but not used
...	necessary to match the generic, but not used

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

---

as.quoted *Convert input to quoted variables.*

Description

Convert characters, formulas and calls to quoted .variables

Usage

```
as.quoted(x, env = parent.frame())
```

Arguments

x	input to quote
env	environment in which unbound symbols in expression should be evaluated. Defaults to the environment in which as.quoted was executed.

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

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, ..., .expand = TRUE,
  .progress = "none", .print = FALSE)
```

Arguments

.data	matrix, array or data frame to be processed
.margins	a vector giving the subscripts to split up data by. 1 splits up by rows, 2 by columns and c(1,2) by rows and columns, and so on for higher dimensions
.fun	function to apply to each piece
...	other arguments passed on to .fun
.expand	if .data is a data frame, should output be 1d (expand = FALSE), with an element for each row; or nd (expand = TRUE), with a dimension for each variable.
.progress	name of the progress bar to use, see create_progress_bar
.print	automatically print each result? (default: FALSE)

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.

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

baseball *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.

Format

A 21699 x 22 data frame

Variables

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

References

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

Examples

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

calculate_cyear <- function(df) {
  mutate(df,
    cyear = year - min(year),
    cpercent = cyear / (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(baberuth)
models <- dlply(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 = true)
```

Arguments

.fun	function
.cols	either a 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.

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 d*ply
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))
```

`count` *Count the number of occurences.*

Description

Equivalent to `as.data.frame(table(x))`, but does not include combinations with zero counts.

Usage

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

Arguments

`df`	data frame to be processed
`vars`	variables to count unique values of
`wt_var`	optional variable to weight by - if this is non-NULL, count will sum up the value of this variable for each combination of id variables.

Details

Speed-wise count is competitive with `table` for single variables, but it really comes into its own when summarising multiple dimensions because it only counts combinations that actually occur in the data.

Compared to `table` + `as.data.frame`, `count` also preserves the type of the identifier variables, instead of converting them to characters/factors.

Value

a data frame with label and freq columns

Examples

```
count(baseball, "id")
count(baseball, "id", "g")
count(baseball, "id", "ab")
count(baseball, "lg")
count(baseball, "stint")
count(count(baseball, c("id", "year")), "id", "freq")
count(count(baseball, c("id", "year")), "freq")
```

---

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
`...`	other arguments passed onto progress bar function

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 the 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 apperance. See the examples.

See Also

progress_none, progress_text, progress_tk, progress_win

Examples

```
l_ply(1:100, identity, .progress = "none")
l_ply(1:100, identity, .progress = "tk")
l_ply(1:100, identity, .progress = "text")
l_ply(1:100, identity, .progress = progress_text(char = "-"))
```

---

daply *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. daply with a function that operates column-wise is similar to aggregate.

Usage

```
daply(.data, .variables, .fun = NULL, ...,
  .progress = "none", .drop_i = TRUE, .drop_o = TRUE,
  .parallel = FALSE)
```

Arguments

.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
.data	data frame to be processed
.variables	variables to split data frame by, as quoted variables, a formula or character vector
.drop_i	should combinations of variables that do not appear in the input data be preserved (FALSE) or dropped (TRUE, default)
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach
.drop_o	should extra dimensions of length 1 in the output be dropped, simplifying the output. Defaults to TRUE

Value

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

Input

This function splits data frames by variables.

Output

If there are no results, then this function will return a vector of length 0 (`vector()`).

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

See Also

Other array output: `aaply`, `laply`

Other data frame input: `ddply`, `dlply`

Examples

```
daply(baseball, .(year), nrow)

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

daply(baseball[, c(2, 6:9)], .(year), colwise(mean))
daply(baseball[, 6:9], .(baseball$year), colwise(mean))
daply(baseball, .(year), function(df) colwise(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, .parallel = FALSE)
```

Arguments

`.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`
`.data`	data frame to be processed
`.variables`	variables to split data frame by, as quoted variables, a formula or character vector

.drop	should combinations of variables that do not appear in the input data be preserved (FALSE) or dropped (TRUE, default)
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach

Value

A data frame, as described in the output section.

Input

This function splits data frames by variables.

Output

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.

If there are no results, then this function will return a data frame with zero rows and columns (data.frame()).

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

See Also

Other data frame input: daply, dlply

Other data frame output: adply, ldply

Examples

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

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

base2 <- ddply(baseball, .(id), transform,
 career_year = year - min(year) + 1
)
```

defaults *Set defaults.*

Description

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

Usage

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

Arguments

x	list of values
y	defaults

desc *Descending order.*

Description

Transform a vector into a format that will be sorted in descending order.

Usage

```
desc(x)
```

Arguments

x	vector to transform

Examples

```
desc(1:10)
desc(factor(letters))
first_day <- seq(as.Date("1910/1/1"), as.Date("1920/1/1"), "years")
desc(first_day)
```

dlply *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. dlply is similar to by except that the results are returned in a different format.

Usage

```
dlply(.data, .variables, .fun = NULL, ...,
  .progress = "none", .drop = TRUE, .parallel = FALSE)
```

Arguments

.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
.data	data frame to be processed
.variables	variables to split data frame by, as quoted variables, a formula or character vector
.drop	should combinations of variables that do not appear in the input data be preserved (FALSE) or dropped (TRUE, default)
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach

Value

list of results

Input

This function splits data frames by variables.

Output

If there are no results, then this function will return a list of length 0 (list()).

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

See Also

Other data frame input: daply, ddply

Other list output: alply, llply

Examples

```
linmod <- function(df) {
  lm(rbi ~ year, data = mutate(df, year = year - min(year)))
}
models <- dlply(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 *Split data frame, apply function, and discard results.*

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	automatically print each result? (default: FALSE)

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.

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

each *Aggregate multiple functions into a single function.*

Description

Combine multiple functions into a single function returning a named vector of outputs.

Usage

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

Arguments

...	functions to combine. each function should produce a single number as output

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 specified value.*

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	all error messages be suppressed?

Value

a function

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)
```

---

idata.frame *Construct an immutable data frame.*

Description

An immutable data frame works like an ordinary data frame, except that when you subset it, it returns a reference to the original data frame, not a a copy. This makes subsetting substantially faster and has a big impact when you are working with large datasets with many groups.

Usage

```
idata.frame(df)
```

Arguments

df	a data frame

Details

This method is still a little experimental, so please let me know if you run into any problems.

Value

an immutable data frame

Examples

```
system.time(dlply(baseball, "id", nrow))
system.time(dlply(idata.frame(baseball), "id", nrow))
```

`join` *Join two data frames together.*

Description

Join, like merge, is designed for the types of problems where you would use a sql join.

Usage

```
join(x, y, by = intersect(names(x), names(y)),
  type = "left", match = "all")
```

Arguments

`x`	data frame
`y`	data frame
`by`	character vector of variable names to join by
`type`	type of join: left (default), right, inner or full. See details for more information.
`match`	how should duplicate ids be matched? Either match just the `"first"` matching row, or match `"all"` matching rows.

Details

The four join types return:

- `inner`: only rows with matching keys in both x and y
- `left`: all rows in x, adding matching columns from y
- `right`: all rows in y, adding matching columns from x
- `full`: all rows in x with matching columns in y, then the rows of y that don't match x.

Note that from plyr 1.5, `join` will (by default) return all matches, not just the first match, as it did previously.

Unlike merge, preserves the order of x no matter what join type is used. If needed, rows from y will be added to the bottom. Join is often faster than merge, although it is somewhat less featureful - it currently offers no way to rename output or merge on different variables in the x and y data frames.

Examples

```
first <- ddply(baseball, "id", summarise, first = min(year))
system.time(b2 <- merge(baseball, first, by = "id", all.x = TRUE))
system.time(b3 <- join(baseball, first, by = "id"))

b2 <- arrange(b2, id, year, stint)
b3 <- arrange(b3, id, year, stint)
stopifnot(all.equal(b2, b3))
```

laply *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. laply is 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.

Usage

```
laply(.data, .fun = NULL, ..., .progress = "none",
  .drop = TRUE, .parallel = FALSE)
```

Arguments

.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
.data	list to be processed
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach
.drop	should extra dimensions of length 1 in the output be dropped, simplifying the output. Defaults to TRUE

Value

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

Input

This function splits lists by elements and combines the result into a data frame.

Output

If there are no results, then this function will return a vector of length 0 (vector()).

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

See Also

Other array output: aaply, daply

Other list input: ldply, llply

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",
  .parallel = FALSE)
```

Arguments

.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
.data	list to be processed
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach

Value

A data frame, as described in the output section.

Input

This function splits lists by elements and combines the result into a data frame.

Output

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.

If there are no results, then this function will return a data frame with zero rows and columns (data.frame()).

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

See Also

Other data frame output: adply, ddply

Other list input: laply, llply

`liply` *Experimental iterator based version of llply.*

Description

Because iterators do not have known length, `liply` starts by allocating an output list of length 50, and then doubles that length whenever it runs out of space. This gives O(n ln n) performance rather than the O(n ^ 2) performance from the naive strategy of growing the list each time.

Usage

```
liply(.iterator, .fun = NULL, ...)
```

Arguments

`.iterator`	iterator object
`.fun`	function to apply to each piece
`...`	other arguments passed on to `.fun`

Examples

```
if(require("iterators")) {
  system.time(dlply(baseball, "id", summarise, mean_rbi = mean(rbi)))
  system.time({
    baseball_id <- isplit2(baseball, baseball$id)
    liply(baseball_id, summarise, mean_rbi = mean(rbi, na.rm = TRUE))
  })
  # Iterators get used up:
  liply(baseball_id, summarise, mean_rbi = mean(rbi, na.rm = TRUE))
}
```

`llply` *Split list, apply function, and return results in a list.*

Description

For each element of a list, apply function, keeping results as a list. `llply` is equivalent to `lapply` except that it will preserve labels and can display a progress bar.

Usage

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

Arguments

`.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`
`.data`	list to be processed
`.parallel`	if `TRUE`, apply function in parallel, using parallel backend provided by foreach
`.inform`	produce informative error messages? This is turned off by by default because it substantially slows processing speed, but is very useful for debugging

Value

list of results

Input

This function splits lists by elements and combines the result into a data frame.

Output

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

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

See Also

Other list input: `laply`, `ldply`

Other list output: `alply`, `dlply`

Examples

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

llply(x, mean)
llply(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

.data	list to be processed
.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	automatically print each result? (default: FALSE)

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.

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

`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, ..., .expand = TRUE,
  .progress = "none", .parallel = FALSE)
```

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`
`.expand`	should output be 1d (expand = FALSE), with an element for each row; or nd (expand = TRUE), with a dimension for each variable.
`.progress`	name of the progress bar to use, see `create_progress_bar`
`.parallel`	if `TRUE`, apply function in parallel, using parallel backend provided by foreach

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)

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

Examples

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

match_df — *Extract matching rows of a data frame.*

Description

This is particularly useful when you've summarised the data in some way and want to subset the original data by a characteristic of the subset.

Usage

```
match_df(x, y, on = NULL)
```

Arguments

x	data frame to subset.
y	data frame defining matching rows.
on	variables to match on - by default will use all variables common to both data frames.

Value

a data frame

See Also

join to combine the columns from both x and y

Examples

```
longterm <- subset(count(baseball, "id"), freq > 25)
bb_longterm <- match_df(baseball, longterm)
```

mdply — *Call function with arguments in array or data frame, returning a data frame.*

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, ..., .expand = TRUE,
  .progress = "none", .parallel = FALSE)
```

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
.expand	should output be 1d (expand = FALSE), with an element for each row; or nd (expand = TRUE), with a dimension for each variable.
.progress	name of the progress bar to use, see create_progress_bar
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach

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

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

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

---

mlply *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

```
mlply(.data, .fun = NULL, ..., .expand = TRUE,
  .progress = "none", .parallel = FALSE)
```

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
.expand	should output be 1d (expand = FALSE), with an element for each row; or nd (expand = TRUE), with a dimension for each variable.
.progress	name of the progress bar to use, see create_progress_bar
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach

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

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

Examples

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

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

---

mutate *Mutate a data frame by adding new or replacing existing columns.*

Description

This function is very similar to transform but it executes the transformations iteratively so that later transformations can use the columns created by earlier transformations. Like transform, unnamed components are silently dropped.

Usage

```
mutate(.data, ...)
```

Arguments

.data	the data frame to transform
...	named parameters giving definitions of new columns.

Details

Mutate seems to be considerably faster than transform for large data frames.

See Also

subset, summarise, arrange. For another somewhat different approach to solving the same problem, see within.

Examples

```
# Examples from transform
mutate(airquality, Ozone = -Ozone)
mutate(airquality, new = -Ozone, Temp = (Temp - 32) / 1.8)

# Things transform can't do
mutate(airquality, Temp = (Temp - 32) / 1.8, OzT = Ozone / Temp)

# mutate is rather faster than transform
system.time(transform(baseball, avg_ab = ab / g))
system.time(mutate(baseball, avg_ab = ab / g))
```

---

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, ..., .expand = TRUE,
  .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
.expand	should output be 1d (expand = FALSE), with an element for each row; or nd (expand = TRUE), with a dimension for each variable.
.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 a list. If there are no results, then this function will return a list of length 0 (list()).

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

---

ozone *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 area with the first two dimensions representing latitude and longitude, and the third representing time.

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` *Text progress bar.*

Description

A textual progress bar

Usage

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

Arguments

`style`	style of text bar, see Details section of `txtProgressBar`
`...`	other arugments passed on to `txtProgressBar`

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.

See Also

Other progress bars: `progress_none`, `progress_tk`, `progress_win`

Examples

```
l_ply(1:100, identity, .progress = "text")
l_ply(1:100, 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.

See Also

tkProgressBar for the function that powers this progress bar

Other progress bars: progress_none, progress_text, progress_win

Examples

```
l_ply(1:100, identity, .progress = "tk")
l_ply(1:100, identity, .progress = progress_tk(width=400))
l_ply(1:100, 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.

See Also

winProgressBar for the function that powers this progress bar

Other progress bars: progress_none, progress_text, progress_tk

Examples

```
if(exists("winProgressBar")) {
l_ply(1:100, identity, .progress = "win")
l_ply(1:100, identity, .progress = progress_win(title="Working..."))
}
```

---

raply *Replicate expression and return results in a array.*

Description

Evalulate expression n times then combine results into an array

Usage

```
raply(.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	should extra dimensions of length 1 be dropped, simplifying the output. Defaults to TRUE

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.

Value

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

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

Examples

```
raply(100, mean(runif(100)))
raply(100, each(mean, var)(runif(100)))

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

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

---

rbind.fill *Combine data.frames by row, filling in missing columns.*

Description

rbinds a list of data frames filling missing columns with NA.

Usage

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

Arguments

...	input data frames to row bind together

Details

This is an enhancement to rbind that adds in columns that are not present in all inputs, accepts a list of data frames, and operates substantially faster.

Column names and types in the output will appear in the order in which they were encountered. No checking is performed to ensure that each column is of consistent type in the inputs.

Value

a single data frame

See Also

Other binding functions: rbind.fill.matrix

Examples

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

`rbind.fill.matrix` *Bind matrices by row, and fill missing columns with NA.*

Description

The matrices are bound together using their column names or the column indices (in that order of precedence.) Numeric columns may be converted to character beforehand, e.g. using format. If a matrix doesn't have colnames, the column number is used. Note that this means that a column with name `"1"` is merged with the first column of a matrix without name and so on. The returned matrix will always have column names.

Usage

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

Arguments

...	the matrices to rbind

Details

Vectors are converted to 1-column matrices.

Matrices of factors are not supported. (They are anyways quite inconvenient.) You may convert them first to either numeric or character matrices. If a matrices of different types are merged, then normal covnersion precendence will apply.

Row names are ignored.

Value

a matrix with column names

Author(s)

C. Beleites

See Also

rbind, cbind, rbind.fill

Other binding functions: rbind.fill

Examples

```
A <- matrix (1:4, 2)
B <- matrix (6:11, 2)
A
B
rbind.fill.matrix (A, B)

colnames (A) <- c (3, 1)
A
rbind.fill.matrix (A, B)

rbind.fill.matrix (A, 99)
```

`rdply` *Replicate expression and return results in a data frame.*

Description

Evalulate expression n times then combine results into a data frame

Usage

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

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`

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

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

Examples

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

---

rename *Modify names by name, not position.*

Description

Modify names by name, not position.

Usage

```
rename(x, replace)
```

Arguments

x	named object to modify
replace	named character vector, with new names as values, and old names as names.

Examples

```
x <- c("a" = 1, "b" = 2, d = 3, 4)
rename(x, c("d" = "c"))
rename(mtcars, c("disp" = "displ"))
```

---

rlply *Replicate expression and return results in a list.*

Description

Evalulate expression n times then combine results into a list

Usage

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

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

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.

Value

list of results

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

Examples

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

---

round_any *Round to multiple of any number.*

Description

Round to multiple of any number.

Usage

```
round_any(x, accuracy, f = round)
```

Arguments

x	numeric vector to round
accuracy	number to round to
f	rounding function: `floor`, `ceiling` or `round`

Examples

```
round_any(135, 10)
round_any(135, 100)
round_any(135, 25)
round_any(135, 10, floor)
round_any(135, 100, floor)
round_any(135, 25, floor)
round_any(135, 10, ceiling)
round_any(135, 100, ceiling)
round_any(135, 25, ceiling)
```

r_ply *Replicate expression and discard results.*

Description

Evalulate 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	automatically print each result? (default: FALSE)

Details

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

References

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. http://www.jstatsoft.org/v40/i01/.

Examples

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

splat *'Splat' arguments to a function.*

Description

Wraps a function in do.call, so instead of taking multiple arguments, it takes a single named list which will be interpreted as its arguments.

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.

Value

a function

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))
```

---

strip_splits *Remove splitting variables from a data frame.*

Description

This is useful when you want to perform some operation to every column in the data frame, except the variables that you have used to split it. These variables will be automatically added back on to the result when combining all results together.

Usage

```
strip_splits(df)
```

Arguments

df	data frame produced by d*ply.

Examples

```
dlply(mtcars, c("vs", "am"))
dlply(mtcars, c("vs", "am"), strip_splits)
```

summarise *Summarise a data frame.*

Description

Summarise works in an analagous 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.

Usage

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

Arguments

`.data`	the data frame to be summarised
`...`	further arguments of the form var = value

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)))
```

vaggregate *Vector aggregate.*

Description

This function is somewhat similar to `tapply`, but is designed for use in conjunction with `id`. It is simpler in that it only accepts a single grouping vector (use `id` if you have more) and uses `vapply` internally, using the `.default` value as the template.

Usage

```
vaggregate(.value, .group, .fun, ..., .default = NULL,
  .n = nlevels(.group))
```

Arguments

.value	vector of values to aggregate
.group	grouping vector
.fun	aggregation function
...	other arguments passed on to .fun
.default	default value used for missing groups. This argument is also used as the template for function output.
.n	total number of groups

Details

vaggregate should be faster than tapply in most situations because it avoids making a copy of the data.

Examples

```
# Some examples of use borrowed from ?tapply
n <- 17; fac <- factor(rep(1:3, length = n), levels = 1:5)
table(fac)
vaggregate(1:n, fac, sum)
vaggregate(1:n, fac, sum, .default = NA_integer_)
vaggregate(1:n, fac, range)
vaggregate(1:n, fac, range, .default = c(NA, NA) + 0)
vaggregate(1:n, fac, quantile)
# Unlike tapply, vaggregate does not support multi-d output:
tapply(warpbreaks$breaks, warpbreaks[,-1], sum)
vaggregate(warpbreaks$breaks, id(warpbreaks[,-1]), sum)

# But it is about 10x faster
x <- rnorm(1e6)
y1 <- sample.int(10, 1e6, replace = TRUE)
system.time(tapply(x, y1, mean))
system.time(vaggregate(x, y1, mean))
```

Index