Package 'plyr'

March 24, 2022

Title Tools for Splitting, Applying and Combining Data

Version 1.8.7

Description 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 piece 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 'Becton Dickinson'.

License MIT + file LICENSE

URL http://had.co.nz/plyr, https://github.com/hadley/plyr

BugReports https://github.com/hadley/plyr/issues

Depends R (>= 3.1.0)

Imports Rcpp (>= 0.11.0)

Suggests abind, covr, doParallel, foreach, iterators, itertools, tcltk, testthat

LinkingTo Rcpp

Encoding UTF-8

LazyData true

RoxygenNote 7.1.2

NeedsCompilation yes

Author Hadley Wickham [aut, cre]

Maintainer Hadley Wickham <hadley@rstudio.com>

Repository CRAN

Date/Publication 2022-03-24 21:50:02 UTC

R topics documented:

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

Description

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).

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.

Usage

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

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
.inform	produce informative error messages? This is turned off by default because it substantially slows processing speed, but is very useful for debugging
.drop	should extra dimensions of length 1 in the output be dropped, simplifying the output. Defaults to TRUE
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach
.paropts	a list of additional options passed into the foreach function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the .export and .packages arguments to supply them so that all cluster nodes have the correct environment set up for computing.

Details

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 aaply(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)

Warning

Contrary to alply and adply, passing a data frame as first argument to aaply may lead to unexpected results such as huge memory allocations.

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. https://www.jstatsoft.org/v40/i01/.

See Also

Other array input: a_ply(), adply(), alply()

Other array output: daply(), laply(), maply()

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",
  .inform = FALSE,
  .parallel = FALSE,
  .paropts = NULL,
  .id = NA
)
```

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
.inform	produce informative error messages? This is turned off by default because it substantially slows processing speed, but is very useful for debugging
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach
.paropts	a list of additional options passed into the foreach function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the .export and .packages arguments to supply them so that all cluster nodes have the correct environment set up for computing.
.id	name(s) of the index column(s). Pass NULL to avoid creation of the index column(s). Omit or pass NA to use the default names "X1", "X2", ... Otherwise, this argument must have the same length as .margins.

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. https://www.jstatsoft.org/v40/i01/.

See Also

Other array input: a_ply(), aaply(), alply()

Other data frame output: ddply(), ldply(), mdply()

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.

Usage

```
alply(
  .data,
  .margins,
  .fun = NULL,
  ...,
  .expand = TRUE,
  .progress = "none",
  .inform = FALSE,
  .parallel = FALSE,
  .paropts = NULL,
  .dims = 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
.inform	produce informative error messages? This is turned off by default because it substantially slows processing speed, but is very useful for debugging
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach
.paropts	a list of additional options passed into the foreach function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the .export and .packages arguments to supply them so that all cluster nodes have the correct environment set up for computing.
.dims	if TRUE, copy over dimensions and names from input.

Details

The list will have "dims" and "dimnames" corresponding to the margins given. For instance alply(x,c(3,2),...) where x has dims c(4,3,2) will give a result with dims c(2,3).

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

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. https://www.jstatsoft.org/v40/i01/.

See Also

Other array input: `a_ply()`, `aaply()`, `adply()`

Other list output: `dlply()`, `llply()`, `mlply()`

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`

See Also

`order` for sorting function in the base package

Examples

```
# sort mtcars data by cylinder and displacement
mtcars[with(mtcars, order(cyl, disp)), ]
# Same result using arrange: no need to use with(), as the context is implicit
# NOTE: plyr functions do NOT preserve row.names
arrange(mtcars, cyl, disp)
# Let's keep the row.names in this example
myCars = cbind(vehicle=row.names(mtcars), mtcars)
arrange(myCars, cyl, disp)
# Sort with displacement in descending order
arrange(myCars, 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 with a single column, `value`.

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. The `value` column is always created, even for empty inputs.

`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",
  .inform = FALSE,
  .print = FALSE,
  .parallel = FALSE,
  .paropts = NULL
)
```

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
.inform	produce informative error messages? This is turned off by default because it substantially slows processing speed, but is very useful for debugging
.print	automatically print each result? (default: FALSE)
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach
.paropts	a list of additional options passed into the foreach function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the .export and .packages arguments to supply them so that all cluster nodes have the correct environment set up for computing.

Value

Nothing

Input

This function splits matrices, arrays and data frames by dimensions

Output

All output is discarded. This is useful for functions that you are calling purely for their side effects like displaying plots or saving output.

References

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

See Also

Other array input: `aaply()`, `adply()`, `alply()`

Other no output: `d_ply()`, `l_ply()`, `m_ply()`

`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.

Usage

```
baseball
```

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, ...)

catcolwise(.fun, ...)

numcolwise(.fun, ...)
```

Arguments

.fun	function
.cols	either a function that tests columns for inclusion, or a quoted object giving which columns to process
...	other arguments passed on to .fun

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))

# You can supply additional arguments to either colwise, or the function
# it generates:
numcolwise(mean)(baseball, na.rm = TRUE)
numcolwise(mean, na.rm = TRUE)(baseball)
```

---

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

See Also

`table` for related functionality in the base package

Examples

```
# Count of each value of "id" in the first 100 cases
count(baseball[1:100,], vars = "id")
# Count of ids, weighted by their "g" loading
count(baseball[1:100,], vars = "id", wt_var = "g")
count(baseball, "id", "ab")
count(baseball, "lg")
# How many stints do players do?
count(baseball, "stint")
# Count of times each player appeared in each of the years they played
count(baseball[1:100,], c("id", "year"))
# Count of counts
count(count(baseball[1:100,], 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

```
# No progress bar
l_ply(1:100, identity, .progress = "none")
## Not run:
# Use the Tcl/Tk interface
l_ply(1:100, identity, .progress = "tk")

## End(Not run)
# Text-based progress (|======|)
l_ply(1:100, identity, .progress = "text")
# Choose a progress character, run a length of time you can see
l_ply(1:10000, 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. To apply a function for each row, use aaply with .margins set to 1.

Usage

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

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
.inform	produce informative error messages? This is turned off by default because it substantially slows processing speed, but is very useful for debugging

.drop_i	should combinations of variables that do not appear in the input data be preserved (FALSE) or dropped (TRUE, default)
.drop_o	should extra dimensions of length 1 in the output be dropped, simplifying the output. Defaults to TRUE
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach
.paropts	a list of additional options passed into the foreach function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the .export and .packages arguments to supply them so that all cluster nodes have the correct environment set up for computing.

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. https://www.jstatsoft.org/v40/i01/.

See Also

Other array output: aaply(), laply(), maply()

Other data frame input: d_ply(), 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. To apply a function for each row, use adply with .margins set to 1.

Usage

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

Arguments

.data	data frame to be processed
.variables	variables to split data frame by, as 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
.inform	produce informative error messages? This is turned off by default because it substantially slows processing speed, but is very useful for debugging
.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
.paropts	a list of additional options passed into the foreach function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the .export and .packages arguments to supply them so that all cluster nodes have the correct environment set up for computing.

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 `rbind`ed 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. https://www.jstatsoft.org/v40/i01/.

See Also

`tapply` for similar functionality in the base package

Other data frame input: `d_ply()`, `daply()`, `dlply()`

Other data frame output: `adply()`, `ldply()`, `mdply()`

Examples

```
# Summarize a dataset by two variables
dfx <- data.frame(
  group = c(rep('A', 8), rep('B', 15), rep('C', 6)),
  sex = sample(c("M", "F"), size = 29, replace = TRUE),
  age = runif(n = 29, min = 18, max = 54)
)

# Note the use of the '.' function to allow
# group and sex to be used without quoting
ddply(dfx, .(group, sex), summarize,
 mean = round(mean(age), 2),
 sd = round(sd(age), 2))

# An example using a formula for .variables
ddply(baseball[1:100,], ~ year, nrow)
# Applying two functions; nrow and ncol
ddply(baseball, .(lg), c("nrow", "ncol"))

# Calculate mean runs batted in for each year
rbi <- ddply(baseball, .(year), summarise,
  mean_rbi = mean(rbi, na.rm = TRUE))
# Plot a line chart of the result
plot(mean_rbi ~ year, type = "l", data = rbi)

# make new variable career_year based on the
# start year for each player (id)
```

```
base2 <- ddply(baseball, .(id), mutate,
 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. To apply a function for each row, use `alply` with `.margins` set to `1`.

Usage

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

Arguments

`.data`	data frame to be processed
`.variables`	variables to split data frame by, as `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`
`.inform`	produce informative error messages? This is turned off by default because it substantially slows processing speed, but is very useful for debugging
`.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
`.paropts`	a list of additional options passed into the `foreach` function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the `.export` and `.packages` arguments to supply them so that all cluster nodes have the correct environment set up for computing.

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. https://www.jstatsoft.org/v40/i01/.

See Also

Other data frame input: `d_ply()`, `daply()`, `ddply()`

Other list output: `alply()`, `llply()`, `mlply()`

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. To apply a function for each row, use `a_ply` with `.margins` set to 1.

Usage

```
d_ply(
  .data,
  .variables,
  .fun = NULL,
  ...,
  .progress = "none",
  .inform = FALSE,
  .drop = TRUE,
```

```
  .print = FALSE,
  .parallel = FALSE,
  .paropts = NULL
)
```

Arguments

.data	data frame to be processed
.variables	variables to split data frame by, as 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
.inform	produce informative error messages? This is turned off by default because it substantially slows processing speed, but is very useful for debugging
.drop	should combinations of variables that do not appear in the input data be preserved (FALSE) or dropped (TRUE, default)
.print	automatically print each result? (default: FALSE)
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach
.paropts	a list of additional options passed into the foreach function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the .export and .packages arguments to supply them so that all cluster nodes have the correct environment set up for computing.

Value

Nothing

Input

This function splits data frames by variables.

Output

All output is discarded. This is useful for functions that you are calling purely for their side effects like displaying plots or saving output.

References

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

See Also

Other data frame input: daply(), ddply(), dlply()

Other no output: a_ply(), l_ply(), m_ply()

each *Aggregate multiple functions into a single function.*

Description

Combine multiple functions into a single function returning a named vector of outputs. Note: you cannot supply additional parameters for the summary functions

Usage

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

Arguments

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

See Also

summarise for applying summary functions to data

Examples

```
# Call min() and max() on the vector 1:10
each(min, max)(1:10)
# This syntax looks a little different.  It is shorthand for the
# the following:
f<- each(min, max)
f(1:10)
# Three equivalent ways to call min() and max() on the vector 1:10
each("min", "max")(1:10)
each(c("min", "max"))(1:10)
each(c(min, max))(1:10)
# Call length(), min() and max() on a random normal vector
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)
```

---

here *Capture current evaluation context.*

Description

This function captures the current context, making it easier to use **ply with functions that do special evaluation and need access to the environment where ddply was called from.

Usage

```
here(f)
```

Arguments

f	a function that does non-standard evaluation

Author(s)

Peter Meilstrup, https://github.com/crowding

Examples

```
df <- data.frame(a = rep(c("a","b"), each = 10), b = 1:20)
f1 <- function(label) {
   ddply(df, "a", mutate, label = paste(label, b))
}
## Not run: f1("name:")
# Doesn't work because mutate can't find label in the current scope

f2 <- function(label) {
   ddply(df, "a", here(mutate), label = paste(label, b))
}
f2("name:")
# Works :)
```

`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 = NULL, type = "left", match = "all")
```

Arguments

`x`	data frame
`y`	data frame
`by`	character vector of variable names to join by. If omitted, will match on all common variables.
`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. Defaults to `"all"` for compatibility with merge, but `"first"` is significantly faster.

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))
```

join_all *Recursively join a list of data frames.*

Description

Recursively join a list of data frames.

Usage

```
join_all(dfs, by = NULL, type = "left", match = "all")
```

Arguments

dfs	A list of data frames.
by	character vector of variable names to join by. If omitted, will match on all common variables.
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. Defaults to "all" for compatibility with merge, but "first" is significantly faster.

Examples

```
dfs <- list(
  a = data.frame(x = 1:10, a = runif(10)),
  b = data.frame(x = 1:10, b = runif(10)),
  c = data.frame(x = 1:10, c = runif(10))
)
join_all(dfs)
join_all(dfs, "x")
```

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.

Usage

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

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
.inform	produce informative error messages? This is turned off by default because it substantially slows processing speed, but is very useful for debugging
.drop	should extra dimensions of length 1 in the output be dropped, simplifying the output. Defaults to TRUE
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach
.paropts	a list of additional options passed into the foreach function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the .export and .packages arguments to supply them so that all cluster nodes have the correct environment set up for computing.

Details

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 row, not a column.

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.

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. https://www.jstatsoft.org/v40/i01/.

See Also

Other list input: `l_ply()`, `ldply()`, `llply()`

Other array output: `aaply()`, `daply()`, `maply()`

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",
  .inform = FALSE,
  .parallel = FALSE,
  .paropts = NULL,
  .id = NA
)
```

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`

.inform	produce informative error messages? This is turned off by default because it substantially slows processing speed, but is very useful for debugging
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach
.paropts	a list of additional options passed into the foreach function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the .export and .packages arguments to supply them so that all cluster nodes have the correct environment set up for computing.
.id	name of the index column (used if .data is a named list). Pass NULL to avoid creation of the index column. For compatibility, omit this argument or pass NA to avoid converting the index column to a factor; in this case, ".id" is used as colum name.

Value

A data frame, as described in the output section.

Input

This function splits lists by elements.

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. https://www.jstatsoft.org/v40/i01/.

See Also

Other list input: l_ply(), laply(), llply()

Other data frame output: adply(), ddply(), mdply()

---

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

Warning

Deprecated, do not use in new code.

See Also

plyr-deprecated

---

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.

Usage

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

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

.paropts — a list of additional options passed into the foreach function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the .export and .packages arguments to supply them so that all cluster nodes have the correct environment set up for computing.

Details

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

Value

list of results

Input

This function splits lists by elements.

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. https://www.jstatsoft.org/v40/i01/.

See Also

Other list input: l_ply(), laply(), ldply()

Other list output: alply(), dlply(), mlply()

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",
  .inform = FALSE,
  .print = FALSE,
  .parallel = FALSE,
  .paropts = NULL
)
```

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`
`.inform`	produce informative error messages? This is turned off by default because it substantially slows processing speed, but is very useful for debugging
`.print`	automatically print each result? (default: `FALSE`)
`.parallel`	if `TRUE`, apply function in parallel, using parallel backend provided by foreach
`.paropts`	a list of additional options passed into the `foreach` function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the `.export` and `.packages` arguments to supply them so that all cluster nodes have the correct environment set up for computing.

Value

Nothing

Input

This function splits lists by elements.

Output

All output is discarded. This is useful for functions that you are calling purely for their side effects like displaying plots or saving output.

References

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

See Also

Other list input: laply(), ldply(), llply()

Other no output: a_ply(), d_ply(), m_ply()

Examples

```
l_ply(llply(mtcars, round), table, .print = TRUE)
l_ply(baseball, function(x) print(summary(x)))
```

`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",
  .inform = FALSE,
  .drop = TRUE,
  .parallel = FALSE,
  .paropts = NULL
)
```

Arguments

.data	matrix or data frame to use as source of arguments
.fun	function to apply to each piece
...	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

.inform	produce informative error messages? This is turned off by default because it substantially slows processing speed, but is very useful for debugging
.drop	should extra dimensions of length 1 in the output be dropped, simplifying the output. Defaults to TRUE
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach
.paropts	a list of additional options passed into the foreach function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the .export and .packages arguments to supply them so that all cluster nodes have the correct environment set up for computing.

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.

Value

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

Input

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

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. https://www.jstatsoft.org/v40/i01/.

See Also

Other multiple arguments input: m_ply(), mdply(), mlply()

Other array output: aaply(), daply(), laply()

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)
```

`mapvalues` *Replace specified values with new values, in a vector or factor.*

Description

Item in `x` that match items `from` will be replaced by items in `to`, matched by position. For example, items in `x` that match the first element in `from` will be replaced by the first element of `to`.

Usage

```
mapvalues(x, from, to, warn_missing = TRUE)
```

Arguments

`x`	the factor or vector to modify
`from`	a vector of the items to replace
`to`	a vector of replacement values
`warn_missing`	print a message if any of the old values are not actually present in `x`

Details

If `x` is a factor, the matching levels of the factor will be replaced with the new values.

The related `revalue` function works only on character vectors and factors, but this function works on vectors of any type and factors.

See Also

`revalue` to do the same thing but with a single named vector instead of two separate vectors.

Examples

```
x <- c("a", "b", "c")
mapvalues(x, c("a", "c"), c("A", "C"))

# Works on factors
y <- factor(c("a", "b", "c", "a"))
mapvalues(y, c("a", "c"), c("A", "C"))

# Works on numeric vectors
z <- c(1, 4, 5, 9)
mapvalues(z, from = c(1, 5, 9), to = c(10, 50, 90))
```

match_df *Extract matching rows of a data frame.*

Description

Match works in the same way as join, but instead of return the combined dataset, it only returns the matching rows from the first dataset. 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.

Details

match_df shares the same semantics as join, not match:

- the match criterion is ==, not identical).
- it doesn't work for columns that are not atomic vectors
- if there are no matches, the row will be omitted'

Value

a data frame

See Also

join to combine the columns from both x and y and match for the base function selecting matching items

Examples

```
# count the occurrences of each id in the baseball dataframe, then get the subset with a freq >25
longterm <- subset(count(baseball, "id"), freq > 25)
# longterm
#             id freq
# 30   ansonca01   27
# 48   baineha01   27
# ...
# Select only rows from these longterm players from the baseball dataframe
```

```
# (match would default to match on shared column names, but here was explicitly set "id")
bb_longterm <- match_df(baseball, longterm, on="id")
bb_longterm[1:5,]
```

`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",
  .inform = FALSE,
  .parallel = FALSE,
  .paropts = NULL
)
```

Arguments

`.data`	matrix or data frame to use as source of arguments
`.fun`	function to apply to each piece
`...`	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`
`.inform`	produce informative error messages? This is turned off by default because it substantially slows processing speed, but is very useful for debugging
`.parallel`	if `TRUE`, apply function in parallel, using parallel backend provided by foreach
`.paropts`	a list of additional options passed into the `foreach` function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the `.export` and `.packages` arguments to supply them so that all cluster nodes have the correct environment set up for computing.

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`.

Value

A data frame, as described in the output section.

Input

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

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 `rbind`ed 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. https://www.jstatsoft.org/v40/i01/.

See Also

Other multiple arguments input: `m_ply()`, `maply()`, `mlply()`

Other data frame output: `adply()`, `ddply()`, `ldply()`

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",
  .inform = FALSE,
  .parallel = FALSE,
  .paropts = NULL
)
```

Arguments

.data	matrix or data frame to use as source of arguments
.fun	function to apply to each piece
...	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
.inform	produce informative error messages? This is turned off by default because it substantially slows processing speed, but is very useful for debugging
.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach
.paropts	a list of additional options passed into the foreach function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the .export and .packages arguments to supply them so that all cluster nodes have the correct environment set up for computing.

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.

Value

list of results

Input

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

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. https://www.jstatsoft.org/v40/i01/.

See Also

Other multiple arguments input: m_ply(), maply(), mdply()

Other list output: alply(), dlply(), llply()

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 into a list.

Usage

```
m_ply(
  .data,
  .fun = NULL,
  ...,
  .expand = TRUE,
  .progress = "none",
  .inform = FALSE,
  .print = FALSE,
  .parallel = FALSE,
  .paropts = NULL
)
```

Arguments

`.data`	matrix or data frame to use as source of arguments
`.fun`	function to apply to each piece
`...`	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`
`.inform`	produce informative error messages? This is turned off by default because it substantially slows processing speed, but is very useful for debugging
`.print`	automatically print each result? (default: `FALSE`)

.parallel	if TRUE, apply function in parallel, using parallel backend provided by foreach
.paropts	a list of additional options passed into the foreach function when parallel computation is enabled. This is important if (for example) your code relies on external data or packages: use the .export and .packages arguments to supply them so that all cluster nodes have the correct environment set up for computing.

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.

Value

Nothing

Input

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

Output

All output is discarded. This is useful for functions that you are calling purely for their side effects like displaying plots or saving output.

References

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

See Also

Other multiple arguments input: maply(), mdply(), mlply()

Other no output: a_ply(), d_ply(), l_ply()

---

name_rows *Toggle row names between explicit and implicit.*

Description

Plyr functions ignore row names, so this function provides a way to preserve them by converting them to an explicit column in the data frame. After the plyr operation, you can then apply name_rows again to convert back from the explicit column to the implicit rownames.

Usage

```
name_rows(df)
```

Arguments

df	a data.frame, with either rownames, or a column called .rownames.

Examples

```
name_rows(mtcars)
name_rows(name_rows(mtcars))

df <- data.frame(a = sample(10))
arrange(df, a)
arrange(name_rows(df), a)
name_rows(arrange(name_rows(df), a))
```

---

ozone *Monthly ozone measurements over Central America.*

Description

This data set is a subset of the data from the 2006 ASA Data expo challenge, https://community.amstat.org/jointscsg-section/dataexpo/dataexpo2006. 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.

Usage

```
ozone
```

Format

A 24 x 24 x 72 numeric array

References

https://community.amstat.org/jointscsg-section/dataexpo/dataexpo2006

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)
```

---

plyr *plyr: the split-apply-combine paradigm for R.*

Description

The plyr package is a set of clean and consistent tools that implement the split-apply-combine pattern in R. This is an extremely common pattern in data analysis: you solve a complex problem by breaking it down into small pieces, doing something to each piece and then combining the results back together again.

Details

The plyr functions are named according to what sort of data structure they split up and what sort of data structure they return:

a array

l list

d data.frame

m multiple inputs

r repeat multiple times

_ nothing

So `ddply` takes a data frame as input and returns a data frame as output, and `l_ply` takes a list as input and returns nothing as output.

Row names

By design, no plyr function will preserve row names - in general it is too hard to know what should be done with them for many of the operations supported by plyr. If you want to preserve row names, use `name_rows` to convert them into an explicit column in your data frame, perform the plyr operations, and then use `name_rows` again to convert the column back into row names.

Helpers

Plyr also provides a set of helper functions for common data analysis problems:

- arrange: re-order the rows of a data frame by specifying the columns to order by
- mutate: add new columns or modifying existing columns, like transform, but new columns can refer to other columns that you just created.
- summarise: like mutate but create a new data frame, not preserving any columns in the old data frame.
- join: an adapation of merge which is more similar to SQL, and has a much faster implementation if you only want to find the first match.
- match_df: a version of join that instead of returning the two tables combined together, only returns the rows in the first table that match the second.
- colwise: make any function work colwise on a dataframe
- rename: easily rename columns in a data frame
- round_any: round a number to any degree of precision
- count: quickly count unique combinations and return return as a data frame.

---

plyr-deprecated *Deprecated Functions in Package plyr*

Description

These functions are provided for compatibility with older versions of plyr only, and may be defunct as soon as the next release.

Details

- liply
- isplit2

---

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_time(), progress_tk(), progress_win()

Examples

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

---

progress_time *Text progress bar with time.*

Description

A textual progress bar that estimates time remaining. It displays the estimated time remaining and, when finished, total duration.

Usage

```
progress_time()
```

See Also

Other progress bars: progress_none(), progress_text(), progress_tk(), progress_win()

Examples

```
l_ply(1:100, function(x) Sys.sleep(.01), .progress = "time")
```

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_time(), progress_win()

Examples

```
## Not run:
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=""))

## End(Not run)
```

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_time(), 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. https://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. The first argument can be a list of data frames, in which case all other arguments are ignored. Any NULL inputs are silently dropped. If all inputs are NULL, the output is NULL.

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.

Unordered factor columns will have their levels unified and character data bound with factors will be converted to character. POSIXct data will be converted to be in the same time zone. Array and matrix columns must have identical dimensions after the row count. Aside from these there are no general checks that each column is of consistent data type.

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. The first argument can be a list of matrices, in which case all other arguments are ignored.

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

Evaluate expression n times then combine results into a data frame

Usage

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

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
.id	name of the index column. Pass NULL to avoid creation of the index column. For compatibility, omit this argument or pass NA to use ".n" as column name.

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. https://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, warn_missing = TRUE, warn_duplicated = TRUE)
```

Arguments

x	named object to modify
replace	named character vector, with new names as values, and old names as names.
warn_missing	print a message if any of the old names are not actually present in x.
warn_duplicated	print a message if any name appears more than once in x after the operation. Note: x is not altered: To save the result, you need to copy the returned data into a variable.

Examples

```
x <- c("a" = 1, "b" = 2, d = 3, 4)
# Rename column d to "c", updating the variable "x" with the result
x <- rename(x, replace = c("d" = "c"))
x
# Rename column "disp" to "displacement"
rename(mtcars, c("disp" = "displacement"))
```

`revalue` *Replace specified values with new values, in a factor or character vector.*

Description

If `x` is a factor, the named levels of the factor will be replaced with the new values.

Usage

```
revalue(x, replace = NULL, warn_missing = TRUE)
```

Arguments

`x`	factor or character vector to modify
`replace`	named character vector, with new values as values, and old values as names.
`warn_missing`	print a message if any of the old values are not actually present in `x`

Details

This function works only on character vectors and factors, but the related `mapvalues` function works on vectors of any type and factors, and instead of a named vector specifying the original and replacement values, it takes two separate vectors

See Also

`mapvalues` to replace values with vectors of any type

Examples

```
x <- c("a", "b", "c")
revalue(x, c(a = "A", c = "C"))
revalue(x, c("a" = "A", "c" = "C"))

y <- factor(c("a", "b", "c", "a"))
revalue(y, c(a = "A", c = "C"))
```

`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. https://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 or date-time (POSIXct) vector to round
accuracy	number to round to; for POSIXct objects, a number of seconds
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)

round_any(Sys.time() + 1:10, 5)
round_any(Sys.time() + 1:10, 5, floor)
round_any(Sys.time(), 3600)
```

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. https://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 analogous way to mutate, except instead of adding columns to an existing data frame, it creates a new data frame. 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

Note

Be careful when using existing variable names; the corresponding columns will be immediately updated with the new data and this can affect subsequent operations referring to those variables.

Examples

```
# Let's extract the number of teams and total period of time
# covered by the baseball dataframe
summarise(baseball,
 duration = max(year) - min(year),
 nteams = length(unique(team)))
# Combine with ddply to do that for each separate id
ddply(baseball, "id", summarise,
 duration = max(year) - min(year),
 nteams = length(unique(team)))
```

---

take *Take a subset along an arbitrary dimension*

Description

Take a subset along an arbitrary dimension

Usage

```
take(x, along, indices, drop = FALSE)
```

Arguments

x	matrix or array to subset
along	dimension to subset along
indices	the indices to select
drop	should the dimensions of the array be simplified? Defaults to FALSE which is the opposite of the useful R default.

Examples

```
x <- array(seq_len(3 * 4 * 5), c(3, 4, 5))
take(x, 3, 1)
take(x, 2, 1)
take(x, 1, 1)
take(x, 3, 1, drop = TRUE)
take(x, 2, 1, drop = TRUE)
take(x, 1, 1, drop = TRUE)
```

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.out = 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