Package ‘poorman’

July 1, 2020

Type Package

Title A Poor Man's Base R Copy of 'dplyr' Verbs

Version 0.2.1

Maintainer Nathan Eastwood <nathan.eastwood@icloud.com>

Description A simple replication of key 'dplyr' verbs using only base R.

URL https://github.com/nathaneastwood/poorman

BugReports https://github.com/nathaneastwood/poorman/issues

Depends R (>= 3.4)

Suggests knitr, roxygen2, tinytest

License MIT + file LICENSE

Encoding UTF-8

LazyData true

RoxygenNote 7.1.0

NeedsCompilation no

Author Nathan Eastwood [aut, cre]

Repository CRAN

Date/Publication 2020-07-01 19:30:02 UTC

R topics documented:

`arrange` *Arrange rows by variables*

Description

Order rows of a `data.frame` by an expression involving its variables.

Usage

```
arrange(.data, ...)
```

Arguments

`.data`	A `data.frame`.
`...`	A comma separated vector of unquoted name(s) to order the data by.

Value

A `data.frame`.

Examples

```
arrange(mtcars, mpg)
mtcars %>% arrange(mpg)
mtcars %>% arrange(cyl, mpg)
```

`between` *Do values in a numeric vector fall in specified range?*

Description

This is a shortcut for `x >= left & x <= right`.

Usage

```
between(x, left, right)
```

Arguments

`x`	A `numeric` vector of values.
`left, right`	Boundary values.

Value

A `logical` vector the same length as `x`.

Examples

```
between(1:12, 7, 9)

x <- rnorm(1e2)
x[between(x, -1, 1)]
```

`coalesce` *Find first non-missing element*

Description

Given a set of vectors, `coalesce()` finds the first non-missing value at each position. This is inspired by the SQL `COALESCE` function which does the same thing for `NULL`s.

Usage

```
coalesce(...)
```

Arguments

`...`	Vectors. Inputs should be recyclable (either be length `1L` or `n`) and coercible to a common type.

Details

Currently, `coalesce()` type checking does not take place.

Examples

```
# Use a single value to replace all missing vectors
x <- sample(c(1:5, NA, NA, NA))
coalesce(x, 0L)

# Or match together a complete vector from missing pieces
y <- c(1, 2, NA, NA, 5)
z <- c(NA, NA, 3, 4, 5)
coalesce(y, z)
```

`context` *Context dependent expressions*

Description

These functions return information about the "current" group or "current" variable, so only work inside specific contexts like `summarise()` and `mutate()`.

- `n()` gives the number of observations in the current group.
- `cur_data()` gives the current data for the current group (excluding grouping variables).
- `cur_group()` gives the group keys, a single row `data.frame` containing a column for each grouping variable and its value.
- `cur_group_id()` gives a unique numeric identifier for the current group.
- `cur_group_rows()` gives the rows the groups appear in the data.

Usage

```
n()

cur_data()

cur_group()

cur_group_id()

cur_group_rows()
```

`data.table`

If you're familiar with `data.table`:

- `cur_data()` <-> `.SD`
- `cur_group_id()` <-> `.GRP`
- `cur_group()` <-> `.BY`
- `cur_group_rows()` <-> `.I`

See Also

See `group_data()` for equivalent functions that return values for all groups.

Examples

```
df <- data.frame(
  g = sample(rep(letters[1:3], 1:3)),
  x = runif(6),
  y = runif(6),
  stringsAsFactors = FALSE
)
gf <- df %>% group_by(g)

gf %>% summarise(n = n())

gf %>% mutate(id = cur_group_id())
gf %>% summarise(row = cur_group_rows())
gf %>% summarise(data = list(cur_group()))
gf %>% summarise(data = list(cur_data()))
```

`count` *Count observations by group*

Description

`count()` lets you quickly count the unique values of one or more variables: `df %>% count(a,b)` is roughly equivalent to `df %>% group_by(a,b) %>% summarise(n = n())`. `count()` is paired with `tally()`, a lower-level helper that is equivalent to `df %>% summarise(n = n())`. Supply `wt` to perform weighted counts, switching the summary from from `n = n()` to `n = sum(wt)`. `add_count()` and `add_tally()` are equivalent to `count()` and `tally()` but use `mutate()` instead of `summarise()` so that they add a new column with group-wise counts.

Usage

```
count(x, ..., wt = NULL, sort = FALSE, name = NULL)

tally(x, wt = NULL, sort = FALSE, name = NULL)

add_count(x, ..., wt = NULL, sort = FALSE, name = NULL)

add_tally(x, wt = NULL, sort = FALSE, name = NULL)
```

Arguments

`x`	A `data.frame`.
`...`	Variables to group by.
`wt`	If omitted, will count the number of rows. If specified, will perform a "weighted" count by summing the (non-missing) values of variable `wt`. If omitted, and column `n` exists, it will automatically be used as a weighting variable, although you will have to specify `name` to provide a new name for the output.
`sort`	`logical(1)`. If `TRUE`, will show the largest groups at the top.
`name`	`character(1)`. The name of the new column in the output. If omitted, it will default to `n`. If there's already a column called `n`, it will error, and require you to specify the name.

Value

A `data.frame`. `count()` and `add_count()` have the same groups as the input.

Examples

```
# count() is a convenient way to get a sense of the distribution of
# values in a dataset
mtcars %>% count(cyl)
mtcars %>% count(cyl, sort = TRUE)
mtcars %>% count(cyl, am, sort = TRUE)
# Note that if the data are already grouped, count() adds an additional grouping variable
# which is removed afterwards
mtcars %>% group_by(gear) %>% count(cyl)

# tally() is a lower-level function that assumes you've done the grouping
mtcars %>% tally()
mtcars %>% group_by(cyl) %>% tally()

# both count() and tally() have add_ variants that work like mutate() instead of summarise
mtcars %>% add_count(cyl, wt = am)
mtcars %>% add_tally(wt = am)
```

`desc` *Descending order*

Description

Transform a vector into a format that will be sorted in descending order. This is useful within `arrange()`.

Usage

```
desc(x)
```

Arguments

`x`	A vector to transform.

Value

A vector of the same length as `x`.

Examples

```
desc(1:10)
desc(factor(letters))

first_day <- seq(as.Date("1910/1/1"), as.Date("1920/1/1"), "years")
desc(first_day)

mtcars %>% arrange(desc(mpg))
```

`distinct` *Subset distinct/unique rows*

Description

Select only distinct/unique rows from a `data.frame`.

Usage

```
distinct(.data, ..., .keep_all = FALSE)
```

Arguments

`.data`	A `data.frame`.
`...`	Optional variables to use when determining uniqueness. If there are multiple rows for a given combination of inputs, only the first row will be preserved. If omitted, will use all variables.
`.keep_all`	`logical(1)`. If `TRUE`, keep all variables in `.data`. If a combination of `...` is not distinct, this keeps the first row of values.

Value

A `data.frame` with the following properties:

- Rows are a subset of the input but appear in the same order.
- Columns are not modified if `...` is empty or `.keep_all` is `TRUE`. Otherwise, `distinct()` first calls `mutate()` to create new columns.
- Groups are not modified.
- `data.frame` attributes are preserved.

Examples

```
df <- data.frame(
  x = sample(10, 100, rep = TRUE),
  y = sample(10, 100, rep = TRUE)
)
nrow(df)
nrow(distinct(df))
nrow(distinct(df, x, y))

distinct(df, x)
distinct(df, y)

# You can choose to keep all other variables as well
distinct(df, x, .keep_all = TRUE)
distinct(df, y, .keep_all = TRUE)

# You can also use distinct on computed variables
distinct(df, diff = abs(x - y))

# The same behaviour applies for grouped data frames,
# except that the grouping variables are always included
df <- data.frame(
  g = c(1, 1, 2, 2),
  x = c(1, 1, 2, 1)
) %>% group_by(g)
df %>% distinct(x)
```

`filter` *Return rows with matching conditions*

Description

Use `filter()` to choose rows/cases where conditions are `TRUE`.

Usage

```
filter(.data, ...)
```

Arguments

`.data`	A `data.frame`.
`...`	Logical predicated defined in terms of the variables in `.data`. Multiple conditions are combined with `&`. Arguments within `...` are automatically quoted and evaluated within the context of the `data.frame`.

Value

A `data.frame`.

Useful filter functions

- ==, >, >=, etc.
- &, |, !, `xor()`
- `is.na()`

Examples

```
filter(mtcars, am == 1)
mtcars %>% filter(cyl == 4)
mtcars %>% filter(cyl <= 5 & am > 0)
mtcars %>% filter(cyl == 4 | cyl == 8)
mtcars %>% filter(!(cyl %in% c(4, 6)), am != 0)
```

filter_joins — *Filtering joins filter rows from* x *based on the presence or absence of matches in* y*:*

Description

- semi_join() return all rows from x with a match in y.
- anti_join() return all rows from x with*out* a match in y.

Usage

```
anti_join(x, y, by = NULL)

semi_join(x, y, by = NULL)
```

Arguments

x, y	The data.frames to join.
by	A character vector of variables to join by. If NULL, the default, *_join() will do a natural join, using all variables with common names across the two tables. A message lists the variables so that you can check they're right (to suppress the message, simply explicitly list the variables that you want to join).

Examples

```
table1 <- data.frame(
  pupil = rep(1:3, each = 2),
  test = rep(c("A", "B"), 3),
  score = c(60, 70, 65, 80, 85, 70),
  stringsAsFactors = FALSE
)
table2 <- table1[c(1, 3, 4), ]

table1 %>% anti_join(table2, by = c("pupil", "test"))
table1 %>% semi_join(table2, by = c("pupil", "test"))
```

glimpse — *Get a glimpse of your data*

Description

glimpse() is like a transposed version of print(): columns run down the page, and data runs across. This makes it possible to see every column in a data.frame. It is no more than a wrapper around utils::str() only it returns the input (invisibly) meaning it can be used within a data pipeline.

Usage

```
glimpse(x, width = getOption("width"), ...)
```

Arguments

`x`	An object to glimpse at.
`width`	`integer(1)`. Width of the output.
`...`	Additional parameters to pass to `utils::str()`.

Value

`x`, invisibly.

Examples

```
glimpse(mtcars)
```

`group_by` *Group by one or more variables*

Description

Determine the groups within a `data.frame` to perform operations on. `ungroup()` removes the grouping levels.

Usage

```
group_by(.data, ..., .add = FALSE)

ungroup(x, ...)
```

Arguments

`.data`	`data.frame`. The data to group.
`...`	One or more unquoted column names to group/ungroup the data by.
`.add`	`logical(1)`. When `FALSE` (the default) `group_by()` will override existing groups. To add to existing groups, use `.add = TRUE`.
`x`	A `data.frame`.

Value

When using `group_by()`, a `data.frame`, grouped by the grouping variables.

When using `ungroup()`, a `data.frame`.

Examples

```
group_by(mtcars, am, cyl)
ungroup(mutate(group_by(mtcars, am, cyl), sumMpg = sum(mpg)))
mtcars %>%
  group_by(am, cyl) %>%
  mutate(sumMpg = sum(mpg)) %>%
  ungroup()
mtcars %>%
  group_by(carb) %>%
  filter(any(gear == 5))
```

`group_metadata` *Grouping metadata*

Description

- `group_data()` returns a data frame that defines the grouping structure. The columns give the values of the grouping variables. The last column, always called `.rows`, is a list of integer vectors that gives the location of the rows in each group.
- `group_rows()` returns the rows which each group contains.
- `group_indices()` returns an integer vector the same length as `.data` that gives the group that each row belongs to.
- `group_vars()` gives names of grouping variables as character vector.
- `groups()` gives the names as a list of symbols.
- `group_size()` gives the size of each group.
- `n_groups()` gives the total number of groups.

Usage

```
group_data(.data)

group_rows(.data)

group_indices(.data)

group_vars(x)

groups(x)

group_size(x)

n_groups(x)
```

Arguments

`.data, x`	A `data.frame`.

See Also

See context for equivalent functions that return values for the current group.

Examples

```
df <- data.frame(x = c(1,1,2,2))
group_vars(df)
group_rows(df)
group_data(df)

gf <- group_by(df, x)
group_vars(gf)
group_rows(gf)
group_data(gf)
```

`group_split` *Split data.frame by groups*

Description

`group_split()` works like `base::split()` but

- it uses the grouping structure from `group_by()` and is therefore subject to the data mask
- it does not name the elements of the list based on the grouping as this typically loses information and is confusing

Usage

```
group_split(.data, ..., .keep = TRUE)

group_keys(.data)
```

Arguments

`.data`	A `data.frame`.
`...`	Grouping specification, forwarded to `group_by()`.
`.keep`	`logical(1)`. Should the grouping columns be kept (default: TRUE)?

Details

Grouped `data.frame`**s:**

The primary use case for `group_split()` is with already groups `data.frames`, typically a result of `group_by()`. In this case, `group_split()` only uses the first argument, the grouped `data.frame`, and warns when `...` is used.

Because some of these groups may be empty, it is best paired with `group_keys()` which identifies the representatives of each grouping variable for the group.

Ungrouped `data.frame`**s:**

When used on ungrouped `data.frames`, `group_split()` forwards the `...` to `group_by()` before the split, therefore the `...` are subject to the data mask.

Value

- `group_split()` returns a list of `data.frames`. Each `data.frame` contains the rows of `.data` with the associated group and all the columns, including the grouping variables.
- `group_keys()` returns a `data.frame` with one row per group, and one column per grouping variable

See Also

`group_by()`

Examples

```
# Grouped data.frames:
mtcars %>% group_by(cyl, am) %>% group_split()
mtcars %>% group_by(cyl, am) %>% group_split(.keep = FALSE)
mtcars %>% group_by(cyl, am) %>% group_keys()

# Ungrouped data.frames:
mtcars %>% group_split(am, cyl)
```

`if_else` *Vectorised if*

Description

This is a wrapper around `ifelse()` which checks that `true` and `false` are of the same type, making the output more predictable.

Usage

```
if_else(condition, true, false, missing = NULL)
```

Arguments

`condition`	A `logical(n)` vector.
`true, false`	Values to use for `TRUE` and `FALSE` in `condition`. They must either be the same length as `condition` or be length 1. They must also be the same type.
`missing`	If not `NULL` (the default), this will replace any missing values.

Value

A vector the same length as `condition` with values for `TRUE` and `FALSE` replaced by those specified in `true` and `false`, respectively.

Examples

```
x <- c(-5:5, NA)
if_else(x < 0, NA_integer_, x)
if_else(x < 0, "negative", "positive", "missing")

# Unlike ifelse, if_else preserves types
x <- factor(sample(letters[1:5], 10, replace = TRUE))
ifelse(x %in% c("a", "b", "c"), x, factor(NA))
# Attributes are taken from the `true` vector
if_else(x %in% c("a", "b", "c"), x, factor(NA))
```

`joins` *Join two data.frames together*

Description

Join two data.frames together

Usage

```
inner_join(x, y, by = NULL, suffix = c(".x", ".y"))

left_join(x, y, by = NULL, suffix = c(".x", ".y"))

right_join(x, y, by = NULL, suffix = c(".x", ".y"))

full_join(x, y, by = NULL, suffix = c(".x", ".y"))
```

Arguments

`x, y`	The `data.frames` to join.

`by`	A character vector of variables to join by. If `NULL`, the default, *_join() will do a natural join, using all variables with common names across the two tables. A message lists the variables so that you can check they're right (to suppress the message, simply explicitly list the variables that you want to join). To join by different variables on x and y use a named vector. For example, `by = c("a" = "b")` will match `x.a` to `y.b`.
`suffix`	If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.

`lag` *Compute lagged or leading values*

Description

Find the "previous" (`lag()`) or "next" (`lead()`) values in a vector. Useful for comparing values behind of or ahead of the current values.

Usage

```
lag(x, n = 1L, default = NA)

lead(x, n = 1L, default = NA)
```

Arguments

`x`	A `vector` of values
`n`	A positive `integer(1)`, giving the number of positions to lead or lag by.
`default`	The value used for non-existent rows (default: `NA`).

Examples

```
lag(1:5)
lead(1:5)

x <- 1:5
data.frame(behind = lag(x), x, ahead = lead(x))

# If you want to look more rows behind or ahead, use `n`
lag(1:5, n = 1)
lag(1:5, n = 2)

lead(1:5, n = 1)
lead(1:5, n = 2)

# If you want to define a value for non-existing rows, use `default`
lag(1:5)
lag(1:5, default = 0)
```

```
lead(1:5)
lead(1:5, default = 6)
```

---

mutate *Create or transform variables*

Description

`mutate()` adds new variables and preserves existing ones; `transmute()` adds new variables and drops existing ones. Both functions preserve the number of rows of the input. New variables overwrite existing variables of the same name.

Usage

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

transmute(.data, ...)
```

Arguments

`.data`	A `data.frame`.
`...`	Name-value pairs of expressions, each with length `1L`. The name of each argument will be the name of a new column and the value will be its corresponding value. Use a `NULL` value in `mutate` to drop a variable. New variables overwrite existing variables of the same name.

Examples

```
mutate(mtcars, mpg2 = mpg * 2)
mtcars %>% mutate(mpg2 = mpg * 2)
mtcars %>% mutate(mpg2 = mpg * 2, cyl2 = cyl * 2)

# Newly created variables are available immediately
mtcars %>% mutate(mpg2 = mpg * 2, mpg4 = mpg2 * 2)

# You can also use mutate() to remove variables and modify existing variables
mtcars %>% mutate(
  mpg = NULL,
  disp = disp * 0.0163871 # convert to litres
)

# mutate() vs transmute --------------------------
# mutate() keeps all existing variables
mtcars %>%
  mutate(displ_l = disp / 61.0237)

# transmute keeps only the variables you create
mtcars %>%
  transmute(displ_l = disp / 61.0237)
```

na_if — *Convert values to NA*

Description

This is a translation of the SQL command `NULLIF`. It is useful if you want to convert an annoying value to `NA`.

Usage

```
na_if(x, y)
```

Arguments

`x`	The vector to modify.
`y`	The value to replace with `NA`.

Value

A modified version of `x` that replaces any values that are equal to `y` with `NA`.

See Also

`coalesce()` to replace missing values within subsequent `vector`(s) of value(s). `replace_na()` to replace `NA` with a value.

Examples

```
na_if(1:5, 5:1)

x <- c(1, -1, 0, 10)
100 / x
100 / na_if(x, 0)

y <- c("abc", "def", "", "ghi")
na_if(y, "")

# na_if() is particularly useful inside mutate(),
# and is meant for use with vectors rather than entire data.frames
mtcars %>%
  mutate(cyl = na_if(cyl, 6))
```

near — *Compare two numeric vectors*

Description

This is a safe way of comparing if two vectors of floating point numbers are (pairwise) equal. This is safer than using ==, because it has a built in tolerance.

Usage

```
near(x, y, tol = .Machine$double.eps^0.5)
```

Arguments

`x, y`	Numeric vectors to compare
`tol`	Tolerance of comparison.

Examples

```
sqrt(2) ^ 2 == 2
near(sqrt(2) ^ 2, 2)
```

n_distinct — *Count the number of unique values in a set of vectors*

Description

This is the equivalent of `length(unique(x))` for multiple vectors.

Usage

```
n_distinct(..., na.rm = FALSE)
```

Arguments

`...`	A vectors of values.
`na.rm`	`logical(1)`. If `TRUE` missing values don't count.

Examples

```
x <- sample(1:10, 1e5, rep = TRUE)
length(unique(x))
n_distinct(x)
```

`peek_vars` *Peek at variables in the selection context*

Description

Return the vector of column names of the data currently available for selection.

Usage

```
peek_vars()
```

Value

A vector of column names.

`pipe` *Forward-pipe operator*

Description

Pipe an object forward into a function or call expression.

Usage

```
lhs %>% rhs
```

Arguments

`lhs`	The result you are piping.
`rhs`	Where you are piping the result to.

Details

Unlike the `magrittr` pipe, you must supply an actual function instead of just a function name. For example `mtcars %>% head` will not work, but `mtcars %>% head()` will.

Examples

```
mtcars %>% head()
mtcars %>% select(mpg)
```

`pull` *Pull out a single variable*

Description

This is a direct replacement for [[.data.frame.

Usage

```
pull(.data, var = -1)
```

Arguments

`.data`	A `data.frame`.
`var`	A variable specified as: • a literal variable name • a positive integer, giving the position counting from the left • a negative integer, giving the position counting from the right The default returns the last column (on the assumption that's the column you've created most recently).

Examples

```
mtcars %>% pull(-1)
mtcars %>% pull(1)
mtcars %>% pull(cyl)
mtcars %>% pull("cyl")
```

`recode` *Recode values*

Description

This is a vectorised version of `switch()`: you can replace `numeric` values based on their position or their name, and `character` or `factor` values only by their name. This is an S3 generic: `{poorman}` provides methods for `numeric`, `character`, and `factors`. For `logical` vectors, use `if_else()`.

You can use `recode()` directly with `factors`; it will preserve the existing order of levels while changing the values. Alternatively, you can use `recode_factor()`, which will change the order of levels to match the order of replacements.

This is a direct port of the `dplyr::recode()` function.

Usage

```
recode(.x, ..., .default = NULL, .missing = NULL)

recode_factor(.x, ..., .default = NULL, .missing = NULL, .ordered = FALSE)
```

Arguments

`.x`	A vector to modify
`...`	Replacements. For `character` and `factor` `.x`, these should be named and replacement is based only on their name. For `numeric` `.x`, these can be named or not. If not named, the replacement is done based on position i.e. `.x` represents positions to look for in replacements. See examples. When named, the argument names should be the current values to be replaced, and the argument values should be the new (replacement) values. All replacements must be the same type, and must have either length one or the same length as `.x`.
`.default`	If supplied, all values not otherwise matched will be given this value. If not supplied and if the replacements are the same type as the original values in `.x`, unmatched values are not changed. If not supplied and if the replacements are not compatible, unmatched values are replaced with `NA`. `.default` must be either length 1 or the same length as `.x`.
`.missing`	If supplied, any missing values in `.x` will be replaced by this value. Must be either length 1 or the same length as `.x`.
`.ordered`	`logical(1)`. If `TRUE`, `recode_factor()` creates an ordered `factor`.

Value

A vector the same length as `.x`, and the same type as the first of `...`, `.default`, or `.missing`. `recode_factor()` returns a factor whose levels are in the same order as in `...`. The levels in `.default` and `.missing` come last.

See Also

`na_if()` to replace specified values with a `NA`.

`coalesce()` to replace missing values with a specified value.

`replace_na()` to replace `NA` with a value.

Examples

```
# For character values, recode values with named arguments only. Unmatched
# values are unchanged.
char_vec <- sample(c("a", "b", "c"), 10, replace = TRUE)
recode(char_vec, a = "Apple")
recode(char_vec, a = "Apple", b = "Banana")

# Use .default as replacement for unmatched values. Note that NA and
# replacement values need to be of the same type.
```

```
recode(char_vec, a = "Apple", b = "Banana", .default = NA_character_)

# Throws an error as NA is logical, not character.
## Not run:
recode(char_vec, a = "Apple", b = "Banana", .default = NA)

## End(Not run)

# For numeric values, named arguments can also be used
num_vec <- c(1:4, NA)
recode(num_vec, `2` = 20L, `4` = 40L)

# Or if you don't name the arguments, recode() matches by position.
# (Only works for numeric vector)
recode(num_vec, "a", "b", "c", "d")
# .x (position given) looks in (...), then grabs (... value at position)
# so if nothing at position (here 5), it uses .default or NA.
recode(c(1, 5, 3), "a", "b", "c", "d", .default = "nothing")

# Note that if the replacements are not compatible with .x,
# unmatched values are replaced by NA and a warning is issued.
recode(num_vec, `2` = "b", `4` = "d")
# use .default to change the replacement value
recode(num_vec, "a", "b", "c", .default = "other")
# use .missing to replace missing values in .x
recode(num_vec, "a", "b", "c", .default = "other", .missing = "missing")

# For factor values, use only named replacements
# and supply default with levels()
factor_vec <- factor(c("a", "b", "c"))
recode(factor_vec, a = "Apple", .default = levels(factor_vec))

# Use recode_factor() to create factors with levels ordered as they
# appear in the recode call. The levels in .default and .missing
# come last.
recode_factor(num_vec, `1` = "z", `2` = "y", `3` = "x")
recode_factor(num_vec, `1` = "z", `2` = "y", `3` = "x", .default = "D")
recode_factor(num_vec, `1` = "z", `2` = "y", `3` = "x", .default = "D", .missing = "M")

# When the input vector is a compatible vector (character vector or
# factor), it is reused as default.
recode_factor(letters[1:3], b = "z", c = "y")
recode_factor(factor(letters[1:3]), b = "z", c = "y")
```

`relocate` *Select/relocate variables by name*

Description

Choose or relocate variables from a `data.frame`. `select()` keeps only the variables you mention; `relocate()` keeps all the variables.

Usage

```
relocate(.data, ..., .before = NULL, .after = NULL)

select(.data, ...)
```

Arguments

`.data`	A `data.frame`.
`...`	The name(s) of the column(s) to select.
`.before, .after`	Destination of the columns selected by `...`. Supplying neither will move the columns to the left-hand side whereas supplying both will result in an error.

Value

A `data.frame`.

Useful functions

There are a number of special functions which are designed to work in `select()` and `relocate()`:

- `starts_with()`, `ends_with()`, `contains()`
- `matches()`
- `num_range()`
- `everything()`

Examples

```
select(mtcars, mpg:cyl)
select(mtcars, MilesPerGallon = mpg, Cylinders = cyl)
mtcars %>% select(mpg)
mtcars %>% select(!mpg, !cyl)
iris %>% select(contains("Petal"))

df <- as.data.frame(matrix(runif(100), nrow = 10))
df <- as.data.frame(df[c(3, 4, 7, 1, 9, 8, 5, 2, 6, 10)])
df %>% select(num_range("V", 4:6))

mtcars %>% relocate(ends_with("p"), .before = mpg)
```

rename — *Rename columns*

Description

`rename()` changes the names of individual variables using `new_name = old_name` syntax.

Usage

```
rename(.data, ...)
```

Arguments

`.data`	A `data.frame`
`...`	Comma separated key-value pairs in the form of `new_name = old_name` to rename selected variables.

Value

A `data.frame`

Examples

```
rename(mtcars, MilesPerGallon = mpg)
rename(mtcars, Cylinders = cyl, Gears = gear)
mtcars %>% rename(MilesPerGallon = mpg)
```

replace_na — *Replace missing values*

Description

Replace missing values in a `data.frame` or `vector`.

Usage

```
replace_na(data, replace, ...)
```

Arguments

`data`	A `data.frame` or `vector`.
`replace`	If `data` is a `data.frame`, a named `list` giving the value to replace `NA` with for each column. If `data` is a `vector`, a single value used for replacement.
`...`	Additional arguments passed onto methods; not currently used.

Value

If `data` is a `data.frame`, `replace_na()` returns a `data.frame`. If `data` is a `vector`, `replace_na()` returns a `vector` of class determined by the union of `data` and `replace`.

See Also

`na_if()` to replace specified values with a `NA`. `coalesce()` to replace missing values within subsequent `vector(s)` of value(s).

Examples

```
df <- data.frame(x = c(1, 2, NA), y = c("a", NA, "b"), stringsAsFactors = FALSE)
df %>% replace_na(list(x = 0, y = "unknown"))
df %>% mutate(x = replace_na(x, 0))

df$x %>% replace_na(0)
df$y %>% replace_na("unknown")
```

`rownames` *Tools for working with row names*

Description

Tools for working with row names

Usage

```
rownames_to_column(.data, var = "rowname")
```

Arguments

`.data`	A `data.frame`.
`var`	`character(1)`. The name of the column to use for row names.

Value

A `data.frame`

Examples

```
mtcars %>% rownames_to_column()
```

`select_helpers` *Select Helpers*

Description

These functions allow you to select variables based on their names.

- `starts_with()`: Starts with a prefix.
- `ends_with()`: Ends with a prefix.
- `contains()`: Contains a literal string.
- `matches()`: Matches a regular expression.
- `all_of()`: Matches variable names in a character vector. All names must be present, otherwise an error is thrown.
- `any_of()`: The same as `all_of()` except it doesn't throw an error.
- `everything()`: Matches all variables.
- `last_col()`: Select the last variable, possibly with an offset.

Usage

```
starts_with(match, ignore.case = TRUE, vars = peek_vars())

ends_with(match, ignore.case = TRUE, vars = peek_vars())

contains(match, ignore.case = TRUE, vars = peek_vars())

matches(match, ignore.case = TRUE, perl = FALSE, vars = peek_vars())

num_range(prefix, range, width = NULL, vars = peek_vars())

all_of(x, vars = peek_vars())

any_of(x, vars = peek_vars())

everything(vars = peek_vars())

last_col(offset = 0L, vars = peek_vars())
```

Arguments

`match`	`character(n)`. If length > 1, the union of the matches is taken.
`ignore.case`	`logical(1)`. If `TRUE`, the default, ignores case when matching names.
`vars`	`character(n)`. A character vector of variable names. When called from inside selecting functions such as `select()`, these are automatically set to the names of the table.

`perl`	`logical(1)`. Should Perl-compatible regexps be used?
`prefix`	A prefix which starts the numeric range.
`range`	`integer(n)`. A sequence of integers, e.g. `1:5`.
`width`	`numeric(1)`. Optionally, the "width" of the numeric range. For example, a range of 2 gives "01", a range of three "001", etc.
`x`	`character(n)`. A vector of column names.
`offset`	`integer(1)`. Select the nth variable from the end of the `data.frame`.

Value

An integer vector giving the position of the matched variables.

See Also

`select()`, `relocate()`, `where()`

Examples

```
mtcars %>% select(starts_with("c"))
mtcars %>% select(starts_with(c("c", "h")))
mtcars %>% select(ends_with("b"))
mtcars %>% relocate(contains("a"), .before = mpg)
iris %>% select(matches(".t."))
mtcars %>% select(last_col())

# `all_of()` selects the variables in a character vector:
iris %>% select(all_of(c("Petal.Length", "Petal.Width")))
# `all_of()` is strict and will throw an error if the column name isn't found
try({iris %>% select(all_of(c("Species", "Genres")))})
# However `any_of()` allows missing variables
iris %>% select(any_of(c("Species", "Genres")))
```

`slice` *Subset rows by position*

Description

Subset rows by their original position in the `data.frame`. Grouped `data.frames` use the position within each group.

Usage

```
slice(.data, ...)

slice_head(.data, ..., n, prop)

slice_tail(.data, ..., n, prop)

slice_min(.data, order_by, ..., n, prop, with_ties = TRUE)

slice_max(.data, order_by, ..., n, prop, with_ties = TRUE)

slice_sample(.data, ..., n, prop, weight_by = NULL, replace = FALSE)
```

Arguments

`.data`	A `data.frame`.
`...`	For `slice()`: integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or negative. Indices beyond the number of rows in the input are silently ignored.
`n, prop`	Provide either `n`, the number of rows, or `prop`, the proportion of rows to select. If neither are supplied, `n = 1` will be used. If `n` is greater than the number of rows in the group (or `prop > 1`), the result will be silently truncated to the group size. If the `proportion` of a group size is not an integer, it is rounded down.
`order_by`	The variable to order by.
`with_ties`	`logical(1)`. Should ties be kept together? The default, `TRUE`, may return more rows than you request. Use `FALSE` to ignore ties, and return the first `n` rows.
`weight_by`	Sampling weights. This must evaluate to a vector of non-negative numbers the same length as the input. Weights are automatically standardised to sum to 1.
`replace`	`logical(1)`. Should sampling be performed with (`TRUE`) or without (`FALSE`, the default) replacement.

Value

An object of the same type as `.data`. The output has the following properties:

- Each row may appear 0, 1, or many times in the output.
- Columns are not modified.
- Groups are not modified.
- Data frame attributes are preserved.

Examples

```
slice(mtcars, c(1, 2, 3))
mtcars %>% slice(1:3)

# Similar to head(mtcars, 1)
mtcars %>% slice(1L)

# Similar to tail(mtcars, 1):
mtcars %>% slice(n())
mtcars %>% slice(5:n())
# Rows can be dropped with negative indices:
slice(mtcars, -(1:4))

# First and last rows based on existing order
mtcars %>% slice_head(n = 5)
mtcars %>% slice_tail(n = 5)

# Grouped operations:
mtcars %>% group_by(am, cyl, gear) %>% slice_head(n = 2)
```

`summarise` *Reduce multiple values down to a single value*

Description

Create one or more scalar variables summarising the variables of an existing `data.frame`. Grouped `data.frames` will result in one row in the output for each group.

Usage

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

summarize(.data, ...)
```

Arguments

`.data`	A `data.frame`.
`...`	Name-value pairs of summary functions. The name will be the name of the variable in the result. The value can be: • A vector of length `1`, e.g. `min(x)`, `n()`, or `sum(is.na(y))`. • A vector of length `n`, e.g. `quantile()`.

Details

`summarise()` and `summarize()` are synonyms.

Examples

```
summarise(mtcars, mean(mpg))
summarise(mtcars, meanMpg = mean(mpg), sumMpg = sum(mpg))
mtcars %>% summarise(mean(mpg))
```

`where` *Select variables with a function*

Description

This selection helper selects the variables for which a function returns `TRUE`.

Usage

```
where(fn)
```

Arguments

`fn`	A function that returns `TRUE` or `FALSE`.

Value

A vector of `integer` column positions which are the result of the `fn` evaluation.

See Also

select_helpers

Examples

```
iris %>% select(where(is.numeric))
iris %>% select(where(function(x) is.numeric(x)))
iris %>% select(where(function(x) is.numeric(x) && mean(x) > 3.5))
```

window_rank *Windowed Rank Functions*

Description

Six variations on ranking functions, mimicking the ranking functions described in SQL2003. They are currently implemented using the built in `rank()` function. All ranking functions map smallest inputs to smallest outputs. Use `desc()` to reverse the direction.

Usage

```
cume_dist(x)

dense_rank(x)

min_rank(x)

ntile(x = row_number(), n)

percent_rank(x)

row_number(x)
```

Arguments

x	A vector of values to rank. Missing values are left as is. If you want to treat them as the smallest or largest values, replace with `Inf` or `-Inf` before ranking.
n	`integer(1)`. The number of groups to split up into.

Details

- `cume_dist()`: a cumulative distribution function. Proportion of all values less than or equal to the current rank.
- `dense_rank()`: like `min_rank()`, but with no gaps between ranks
- `min_rank()`: equivalent to `rank(ties.method = "min")`
- `ntile()`: a rough rank, which breaks the input vector into `n` buckets. The size of the buckets may differ by up to one, larger buckets have lower rank.
- `percent_rank()`: a number between `0` and `1` computed by rescaling `min_rank` to [0, 1]
- `row_number()`: equivalent to `rank(ties.method = "first")`

Examples

```
x <- c(5, 1, 3, 2, 2, NA)
row_number(x)
min_rank(x)
dense_rank(x)
```

```
percent_rank(x)
cume_dist(x)

ntile(x, 2)
ntile(1:8, 3)

# row_number can be used with single table verbs without specifying x
# (for data frames and databases that support windowing)
mutate(mtcars, row_number() == 1L)
mtcars %>% filter(between(row_number(), 1, 10))
```

Index