Package ‘poorman’

October 30, 2023

Type Package

Title A Poor Man's Dependency Free Recreation of 'dplyr'

Version 0.2.7

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

Description A replication of key functionality from 'dplyr' and the wider 'tidyverse' using only 'base'.

URL `https://nathaneastwood.github.io/poorman/`,
`https://github.com/nathaneastwood/poorman`

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

Depends R (>= 3.3)

Suggests knitr, rmarkdown, roxygen2, tinytest

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.2.1

VignetteBuilder knitr

Language en-GB

NeedsCompilation no

Author Nathan Eastwood [aut, cre],
Etienne Bacher [ctb] (<`https://orcid.org/0000-0002-9271-5075`>)

Repository CRAN

Date/Publication 2023-10-30 21:40:02 UTC

R topics documented:

`across` *Apply a function (or functions) across multiple columns*

Description

`across()` makes it easy to apply the same transformation to multiple columns, allowing you to use `select()` semantics inside in "data-masking" functions like `summarise()` and `mutate()`.

`if_any()` and `if_all()` are used to apply the same predicate function to a selection of columns and combine the results into a single logical vector.

`across()` supersedes the family of `dplyr` "scoped variants" like `summarise_at()`, `summarise_if()`, and `summarise_all()` and therefore these functions will not be implemented in `poorman`.

Usage

```
across(.cols = everything(), .fns = NULL, ..., .names = NULL)

if_any(.cols, .fns = NULL, ..., .names = NULL)

if_all(.cols, .fns = NULL, ..., .names = NULL)
```

Arguments

`.fns`	Functions to apply to each of the selected columns. Possible values are: • `NULL`, to returns the columns untransformed. • A function, e.g. `mean`. • A lambda, e.g. `~ mean(.x, na.rm = TRUE)` • A list of functions/lambdas, e.g. `list(mean = mean, n_miss = ~ sum(is.na(.x))` Within these functions you can use `cur_column()` and `cur_group()` to access the current column and grouping keys respectively.
`...`	Additional arguments for the function calls in `.fns`.
`.names`	A glue specification that describes how to name the output columns. This can use `{.col}` to stand for the selected column name, and `{.fn}` to stand for the name of the function being applied. The default (`NULL`) is equivalent to "`{.col}`" for the single function case and "`{.col}_{.fn}`" for the case where a list is used for `.fns`.
`cols, .cols`	<`poor-select`> Columns to transform. Because `across()` is used within functions like `summarise()` and `mutate()`, you can't select or compute upon grouping variables.

Value

`across()` returns a `data.frame` with one column for each column in `.cols` and each function in `.fns`.

`if_any()` and `if_all()` return a logical vector.

Examples

```
# across() ---------------------------------------------------------------
iris %>%
  group_by(Species) %>%
  summarise(across(starts_with("Sepal"), mean))
iris %>%
  mutate(across(where(is.factor), as.character))

# Additional parameters can be passed to functions
iris %>%
  group_by(Species) %>%
  summarise(across(starts_with("Sepal"), mean, na.rm = TRUE))

# A named list of functions
iris %>%
  group_by(Species) %>%
  summarise(across(starts_with("Sepal"), list(mean = mean, sd = sd)))

# Use the .names argument to control the output names
iris %>%
  group_by(Species) %>%
  summarise(
    across(starts_with("Sepal"),
    mean,
    .names = c("mean_sepal_length", "mean_sepal_width"))
  )

# if_any() and if_all() ----------------------------------------------------
iris %>%
  filter(if_any(ends_with("Width"), ~ . > 4))
iris %>%
  filter(if_all(ends_with("Width"), ~ . > 2))
```

`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)]
```

`bind` *Efficiently bind multiple* `data.frame`*s by row and column*

Description

Efficiently bind multiple `data.frames` by row and column

Usage

```
bind_cols(...)

bind_rows(..., .id = NULL)
```

Arguments

`...`	`data.frames` to combine. Each argument can either be a `data.frame`, a `list` that could be a `data.frame`, or a `list` of `data.frames`. When row-binding, columns are matched by name, and any missing columns will be filled with `NA`. When column-binding, rows are matched by position, so all `data.frames` must have the same number of rows. To match by value, not position, see mutate_joins.
`.id`	`character(1)`. `data.frame` identifier. When `.id` is supplied, a new column of identifiers is created to link each row to its original `data.frame`. The labels are taken from the named arguments to `bind_rows()`. When a `list` of `data.frames` is supplied, the labels are taken from the names of the `list`. If no names are found a numeric sequence is used instead.

Examples

```
one <- mtcars[1:4, ]
two <- mtcars[9:12, ]

# You can supply data frames as arguments:
bind_rows(one, two)

# The contents of lists are spliced automatically:
bind_rows(list(one, two))
bind_rows(split(mtcars, mtcars$cyl))
bind_rows(list(one, two), list(two, one))

# In addition to data frames, you can supply vectors. In the rows
# direction, the vectors represent rows and should have inner
# names:
bind_rows(
```

```
  c(a = 1, b = 2),
  c(a = 3, b = 4)
)

# You can mix vectors and data frames:
bind_rows(
  c(a = 1, b = 2),
  data.frame(a = 3:4, b = 5:6),
  c(a = 7, b = 8)
)

# When you supply a column name with the `.id` argument, a new
# column is created to link each row to its original data frame
bind_rows(list(one, two), .id = "id")
bind_rows(list(a = one, b = two), .id = "id")
bind_rows("group 1" = one, "group 2" = two, .id = "groups")

## Not run:
# Rows need to match when column-binding
bind_cols(data.frame(x = 1:3), data.frame(y = 1:2))

# even with 0 columns
bind_cols(data.frame(x = 1:3), data.frame())

## End(Not run)

bind_cols(one, two)
bind_cols(list(one, two))
```

`case_when` *A General Vetorised* `if()`

Description

This function allows you to vectorise multiple `if_else()` statements. It is an R equivalent of the SQL `CASE WHEN` statement. If no cases match, `NA` is returned.

Usage

```
case_when(...)
```

Arguments

`...`	A sequence of two-sided formulas. The left hand side (LHS) determines which values match this case. The right hand side (RHS) provides the replacement value. The LHS must evaluate to a logical vector. The RHS does not need to be logical, but all RHSs must evaluate to the same type of vector.

Both LHS and RHS may have the same length of either 1 or n. The value of n must be consistent across all cases. The case of `n == 0` is treated as a variant of `n != 1`.

`NULL` inputs are ignored.

Value

A vector of length 1 or n, matching the length of the logical input or output vectors, with the type (and attributes) of the first RHS. Inconsistent lengths or types will generate an error.

Examples

```
x <- 1:50
case_when(
  x %% 35 == 0 ~ "fizz buzz",
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  TRUE ~ as.character(x)
)

# Like an if statement, the arguments are evaluated in order, so you must
# proceed from the most specific to the most general. This won't work:
case_when(
  TRUE ~ as.character(x),
  x %%  5 == 0 ~ "fizz",
  x %%  7 == 0 ~ "buzz",
  x %% 35 == 0 ~ "fizz buzz"
)

# If none of the cases match, NA is used:
case_when(
  x %%  5 == 0 ~ "fizz",
  x %%  7 == 0 ~ "buzz",
  x %% 35 == 0 ~ "fizz buzz"
)

# Note that NA values in the vector x do not get special treatment. If you want
# to explicitly handle NA values you can use the `is.na` function:
x[2:4] <- NA_real_
case_when(
  x %% 35 == 0 ~ "fizz buzz",
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  is.na(x) ~ "nope",
  TRUE ~ as.character(x)
)

# All RHS values need to be of the same type. Inconsistent types will throw an error.
# This applies also to NA values used in RHS: NA is logical, use
# typed values like NA_real_, NA_complex, NA_character_, NA_integer_ as appropriate.
case_when(
  x %% 35 == 0 ~ NA_character_,
```

```
    x %% 5 == 0 ~ "fizz",
    x %% 7 == 0 ~ "buzz",
    TRUE ~ as.character(x)
  )
  case_when(
    x %% 35 == 0 ~ 35,
    x %% 5 == 0 ~ 5,
    x %% 7 == 0 ~ 7,
    TRUE ~ NA_real_
  )

  # case_when() evaluates all RHS expressions, and then constructs its
  # result by extracting the selected (via the LHS expressions) parts.
  # In particular NaN are produced in this case:
  y <- seq(-2, 2, by = .5)
  case_when(
    y >= 0 ~ sqrt(y),
    TRUE   ~ y
  )

  ## Not run:
  case_when(
    x %% 35 == 0 ~ 35,
    x %% 5 == 0 ~ 5,
    x %% 7 == 0 ~ 7,
    TRUE ~ NA
  )

  ## End(Not run)

  # case_when is particularly useful inside mutate when you want to
  # create a new variable that relies on a complex combination of existing
  # variables
  mtcars %>%
    mutate(
      efficient = case_when(
        mpg > 25 ~ TRUE,
        TRUE ~ FALSE
      )
    )
```

`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.

See Also

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

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

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_data_all()` gives the current data for the current group (including 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.
- `cur_column()` gives the name of the current column (in `across()` only).

Usage

```
n()

cur_data()

cur_data_all()

cur_group()

cur_group_id()

cur_group_rows()

cur_column()
```

`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()))
gf %>% summarise(data = list(cur_data_all()))

gf %>% mutate(across(everything(), ~ paste(cur_column(), round(.x, 2))))
```

`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)
```

`cummean` *Cumulative versions of any, all, and mean*

Description

poorman provides `cumall()`, `cumany()`, and `cummean()` to complete R's set of cumulative functions.

Usage

```
cummean(x)

cumany(x)

cumall(x)
```

Arguments

`x`	For `cumall()` and `cumany()`, a logical vector; for `cummean()` an integer or numeric vector.

Value

A vector the same length as `x`.

Cumulative logical functions

These are particularly useful in conjunction with `filter()`:

- `cumall(x)`: all cases until the first `FALSE`.
- `cumall(!x)`: all cases until the first `TRUE`.
- `cumany(x)`: all cases after the first `TRUE`.
- `cumany(!x)`: all cases after the first `FALSE`.

Examples

```
# `cummean()` returns a numeric/integer vector of the same length
# as the input vector.
x <- c(1, 3, 5, 2, 2)
cummean(x)
cumsum(x) / seq_along(x)

# `cumall()` and `cumany()` return logicals
cumall(x < 5)
cumany(x == 3)

# `cumall()` vs. `cumany()`
df <- data.frame(
  date = as.Date("2020-01-01") + 0:6,
  balance = c(100, 50, 25, -25, -50, 30, 120)
)
# all rows after first overdraft
df %>% filter(cumany(balance < 0))
# all rows until first overdraft
df %>% filter(cumall(!(balance < 0)))
```

`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)
```

`fill` *Fill in missing values with previous or next value*

Description

Fills missing values in selected columns using the next or previous entry. This is useful in the common output format where values are not repeated, and are only recorded when they change.

Usage

```
fill(data, ..., .direction = c("down", "up", "downup", "updown"))
```

Arguments

`data`	A `data.frame`.
`...`	Columns to fill.
`.direction`	Direction in which to fill missing values. Currently either `"down"` (the default), `"up"`, `"downup"` (i.e. first down and then up) or `"updown"` (first up and then down).

Details

Missing values are replaced in atomic vectors; `NULL`s are replaced in lists.

Examples

```
# Value (year) is recorded only when it changes
sales <- data.frame(
  quarter = c(
    "Q1", "Q2", "Q3", "Q4", "Q1", "Q2", "Q3", "Q4", "Q1", "Q2",
    "Q3", "Q4", "Q1", "Q2", "Q3", "Q4"
  ),
  year = c(2000, NA, NA, NA, 2001, NA, NA, NA, 2002, NA, NA, NA, 2004, NA, NA, NA),
  sales = c(
```

```
    66013, 69182, 53175, 21001, 46036, 58842, 44568, 50197, 39113, 41668, 30144,
    52897, 32129, 67686, 31768, 49094
  )
)

# `fill()` defaults to replacing missing data from top to bottom
sales %>% fill(year)

# Value (pet_type) is missing above
tidy_pets <- data.frame(
  rank = c(1L, 2L, 3L, 4L, 5L, 6L, 1L, 2L, 3L, 4L, 5L, 6L),
  pet_type = c(NA, NA, NA, NA, NA, "Dog", NA, NA, NA, NA, NA, "Cat"),
  breed = c(
    "Boston Terrier", "Retrievers (Labrador)", "Retrievers (Golden)",
    "French Bulldogs", "Bulldogs", "Beagles", "Persian", "Maine Coon",
    "Ragdoll", "Exotic", "Siamese", "American Short"
  )
)

# For values that are missing above you can use `.direction = "up"`
tidy_pets %>%
  fill(pet_type, .direction = "up")

# Value (n_squirrels) is missing above and below within a group
squirrels <- data.frame(
  group = c(1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3),
  name = c(
    "Sam", "Mara", "Jesse", "Tom", "Mike", "Rachael", "Sydekea",
    "Gabriela", "Derrick", "Kara", "Emily", "Danielle"
  ),
  role = c(
    "Observer", "Scorekeeper", "Observer", "Observer", "Observer",
    "Observer", "Scorekeeper", "Observer", "Observer", "Scorekeeper",
    "Observer", "Observer"
  ),
  n_squirrels = c(NA, 8, NA, NA, NA, NA, 14, NA, NA, 9, NA, NA)
)

# The values are inconsistently missing by position within the group
# Use .direction = "downup" to fill missing values in both directions
squirrels %>%
  group_by(group) %>%
  fill(n_squirrels, .direction = "downup") %>%
  ungroup()

# Using `.direction = "updown"` accomplishes the same goal in this example
```

`filter` *Return rows with matching conditions*

Description

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

Usage

```
filter(.data, ..., .preserve = FALSE)
```

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`.
`.preserve`	`logical(1)`. Relevant when the .data input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is.

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, .drop = group_by_drop_default(.data))

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`.
`.drop`	`logical(1)`. Drop groups formed by factor levels that don't appear in the data? The default is `TRUE` except when `.data` has been previously grouped with `.drop` = `FALSE`. See `group_by_drop_default()` for details.
`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))

# You can group by expressions: this is just short-hand for
# a mutate() followed by a group_by()
mtcars %>% group_by(vsam = vs + am)
```

`group_by_drop_default` *Default value for .drop argument of group_by*

Description

Default value for .drop argument of group_by

Usage

```
group_by_drop_default(.tbl)
```

Arguments

`.tbl`	A `data.frame`.

Value

`TRUE` unless `.tbl` is a grouped `data.frame` that was previously obtained by `group_by(.drop = FALSE)`

Examples

```
group_by_drop_default(iris)

iris %>%
  group_by(Species) %>%
  group_by_drop_default()

iris %>%
  group_by(Species, .drop = FALSE) %>%
  group_by_drop_default()
```

`group_cols` *Select Grouping Variables*

Description

This selection helper matches grouping variables. It can be used within `select()` and `relocate()` selections.

Usage

```
group_cols()
```

See Also

`groups()` and `group_vars()` for retrieving the grouping variables outside selection contexts.

Examples

```
mtcars %>% group_by(am, cyl) %>% select(group_cols())
```

`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))
```

`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)
```

`lst` *Build a list*

Description

`lst()` constructs a list, similar to `base::list()`, but where components are built sequentially. When defining a component, you can refer to components created earlier in the call. `lst()` also generates missing names automatically.

Usage

```
lst(...)
```

Arguments

`...`	Named or unnamed elements of a list. If the element is unnamed, its expression will be used as its name.

Value

A named list.

Examples

```
# the value of n can be used immediately in the definition of x
lst(n = 5, x = runif(n))

# missing names are constructed from user's input
lst(1:3, z = letters[4:6], runif(3))

a <- 1:3
b <- letters[4:6]
lst(a, b)
```

`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. Variables can be removed by setting their value to `NULL`.

Usage

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

## S3 method for class 'data.frame'
mutate(
  .data,
  ...,
  .keep = c("all", "used", "unused", "none"),
  .before = NULL,
  .after = NULL
)

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.
`.keep`	This argument allows you to control which columns from `.data` are retained in the output: • `"all"`, the default, retains all variables. • `"used"` keeps any variables used to make new variables; it's useful for checking your work as it displays inputs and outputs side-by-side. • `"unused"` keeps only existing variables **not** used to make new variables. • `"none"`, only keeps grouping keys (like `transmute()`). Grouping variables are always kept, unconditional to `.keep`.
`.before, .after`	<`poor-select`> Optionally, control where new columns should appear (the default is to add to the right hand side). See `relocate()` for more details.

Useful mutate functions

- `+`, `-`, `log()`, etc., for their usual mathematical meanings
- `lead()`, `lag()`
- `dense_rank()`, `min_rank()`, `percent_rank()`, `row_number()`, `cume_dist()`, `ntile()`
- `cumsum()`, `cummin()`, `cummax()`
- `na_if()`, `coalesce()`
- `if_else()`, `recode()`, `case_when()`

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
)

# By default, new columns are placed on the far right.
# You can override this with `.before` or `.after`.
df <- data.frame(x = 1, y = 2)
df %>% mutate(z = x + y)
```

```
df %>% mutate(z = x + y, .before = 1)
df %>% mutate(z = x + y, .after = x)

# By default, mutate() keeps all columns from the input data.
# You can override with `.keep`
df <- data.frame(
  x = 1, y = 2, a = "a", b = "b",
  stringsAsFactors = FALSE
)
df %>% mutate(z = x + y, .keep = "all") # the default
df %>% mutate(z = x + y, .keep = "used")
df %>% mutate(z = x + y, .keep = "unused")
df %>% mutate(z = x + y, .keep = "none") # same as transmute()

# 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)
```

`mutate_joins` *Mutating Joins*

Description

The mutating joins add columns from `y` to `x`, matching rows based on the keys:

- `inner_join()`: includes all rows in `x` and `y`.
- `left_join()`: includes all rows in `x`.
- `right_join()`: includes all rows in `y`.
- `full_join()`: includes all rows in `x` or `y`.

If a row in `x` matches multiple rows in `y`, all the rows in `y` will be returned once for each matching row in `x`.

Usage

```
inner_join(
  x,
  y,
  by = NULL,
  suffix = c(".x", ".y"),
  ...,
  na_matches = c("na", "never")
)
```

```
left_join(
  x,
  y,
  by = NULL,
  suffix = c(".x", ".y"),
  ...,
  keep = FALSE,
  na_matches = c("na", "never")
)

right_join(
  x,
  y,
  by = NULL,
  suffix = c(".x", ".y"),
  ...,
  keep = FALSE,
  na_matches = c("na", "never")
)

full_join(
  x,
  y,
  by = NULL,
  suffix = c(".x", ".y"),
  ...,
  keep = FALSE,
  na_matches = c("na", "never")
)
```

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`. To join by multiple variables, use a vector with length > 1. For example, `by = c("a", "b")` will match `x$a` to `y$a` and `x$b` to `y$b`. Use a named vector to match different variables in `x` and `y`. For example, `by = c("a" = "b", "c" = "d")` will match `x$a` to `y$b` and `x$c` to `y$d`. To perform a cross-join, generating all combinations of `x` and `y`, use `by = character()`.
`suffix`	`character(2)`. If there are non-joined duplicate variables in `x` and `y`, these suffixes will be added to the output to disambiguate them.
`...`	Additional arguments to pass to `merge()`

`na_matches`	Should `NA` and `NaN` values match one another? The default, `"na"`, treats two `NA` or `NaN` values as equal, like `%in%`, `match()`, `merge()`. Use `"never"` to always treat two `NA` or `NaN` values as different, like joins for database sources, similarly to `merge(incomparables = FALSE)`.
`keep`	`logical(1)`. Should the join keys from both `x` and `y` be preserved in the output? Only applies to `left_join()`, `right_join()`, and `full_join()`.

Value

A `data.frame`. The order of the rows and columns of `x` is preserved as much as possible. The output has the following properties:

- For `inner_join()`, a subset of `x` rows. For `left_join()`, all `x` rows. For `right_join()`, a subset of `x` rows, followed by unmatched `y` rows. For `full_join()`, all `x` rows, followed by unmatched `y` rows.
- For all joins, rows will be duplicated if one or more rows in `x` matches multiple rows in `y`.
- Output columns include all `x` columns and all `y` columns. If columns in `x` and `y` have the same name (and aren't included in `by`), `suffix`es are added to disambiguate.
- Output columns included in `by` are coerced to common type across `x` and `y`.
- Groups are taken from `x`.

Examples

```
# If a row in `x` matches multiple rows in `y`, all the rows in `y` will be
# returned once for each matching row in `x`
df1 <- data.frame(x = 1:3)
df2 <- data.frame(x = c(1, 1, 2), y = c("first", "second", "third"))
df1 %>% left_join(df2)

# By default, NAs match other NAs so that there are two
# rows in the output of this join:
df1 <- data.frame(x = c(1, NA), y = 2)
df2 <- data.frame(x = c(1, NA), z = 3)
left_join(df1, df2)

# You can optionally request that NAs don't match, giving a
# a result that more closely resembles SQL joins
left_join(df1, df2, na_matches = "never")
```

`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.

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

`recode()` to more generally replace values.

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)
```

`nest_by` *Nest By*

Description

`nest_by()` is similar to `group_by()` however instead of storing the group structure in the metadata, it is made explicit in the data. Each group key is given a single row within the `data.frame` and the group's data is stored within a list-column of the `data.frame`.

Usage

```
nest_by(.data, ..., .key = "data", .keep = FALSE)
```

Arguments

`.data`	A `data.frame`.
`...`	Grouping specification, forwarded to `group_by()`.
`.key`	`character(1)`. The name of the column in which to nest the data (default: "data").
`.keep`	`logical(1)`. Should the grouping columns be kept (default: TRUE)?

Details

Currently there is no pretty-printing provided for the results of `nest_by()` and they are not useable with other functions such as `mutate()`.

Examples

```
mtcars %>% nest_by(am, cyl)
# Or equivalently
mtcars %>% group_by(am, cyl) %>% nest_by()
```

`nth` *Extract the first, last or nth value from a vector*

Description

These are straightforward wrappers around `[[`. The main advantage is that you can provide an optional secondary vector that defines the ordering, and provide a default value to use when the input is shorter than expected.

Usage

```
nth(x, n, order_by = NULL, default = default_missing(x))

first(x, order_by = NULL, default = default_missing(x))

last(x, order_by = NULL, default = default_missing(x))
```

Arguments

`x`	A vector
`n`	For `nth()`, a single integer specifying the position. Negative integers index from the end (i.e. `-1L` will return the last value in the vector). If a double is supplied, it will be silently truncated.
`order_by`	An optional vector used to determine the order
`default`	A default value to use if the position does not exist in the input. This is guessed by default for base vectors, where a missing value of the appropriate type is returned, and for lists, where a `NULL` is return. For more complicated objects, you'll need to supply this value. Make sure it is the same type as `x`.

Value

A single value. `[[` is used to do the subsetting.

Examples

```
x <- 1:10
y <- 10:1

first(x)
last(y)

nth(x, 1)
nth(x, 5)
nth(x, -2)
nth(x, 11)
```

```
last(x)
# Second argument provides optional ordering
last(x, y)

# These functions always return a single value
first(integer())
```

`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

`...`	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.

Author(s)

Nathan Eastwood and Antoine Fabri <`antoine.fabri@gmail.com`>.

Examples

```
# Basic use:
iris %>% head

# Use with lhs as first argument
iris %>% head(10)

# Using the dot place-holder
"Ceci n'est pas une pipe" %>% gsub("une", "un", .)

# When dot is nested, lhs is still placed first:
sample(1:10) %>% paste0(LETTERS[.])

# This can be avoided:
rnorm(100) %>% {c(min(.), mean(.), max(.))} %>% floor

# Lambda expressions:
iris %>%
  {
    size <- sample(1:10, size = 1)
    rbind(head(., size), tail(., size))
  }

# renaming in lambdas:
iris %>%
  {
    my_data <- .
    size <- sample(1:10, size = 1)
```

```
  rbind(head(my_data, size), tail(my_data, size))
}
```

---

pivot_longer *Pivot data from wide to long*

Description

`pivot_longer()` "lengthens" data, increasing the number of rows and decreasing the number of columns. The inverse transformation is `pivot_wider()`.

Usage

```
pivot_longer(
  data,
  cols,
  names_to = "name",
  names_prefix = NULL,
  names_sep = NULL,
  names_pattern = NULL,
  values_to = "value",
  values_drop_na = FALSE,
  ...
)
```

Arguments

`data`	`data.frame`. The data to pivot.
`cols`	<`poor-select`>. Columns to pivot into longer format.
`names_to`	`character(n)`. The name of the new column(s) that will contain the column names.
`names_prefix`	`character(1)`. A regular expression used to remove matching text from the start of each variable name.
`names_sep, names_pattern`	`character(1)`. If `names_to` contains multiple values, this argument controls how the column name is broken up. `names_pattern` takes a regular expression containing matching groups (`()`).
`values_to`	`character(n)`. The name of the new column(s) that will contain the values of the pivoted variables.
`values_drop_na`	`logical(1)`. If `TRUE`, will drop rows that contain only `NA` in the `values_to` column. This effectively converts explicit missing values to implicit missing values, and should generally be used only when missing values in data were created by its structure.
`...`	Additional arguments passed on to methods.

Value

A `data.frame`.

Examples

```
wide_data <- data.frame(replicate(5, rnorm(10)))
# Customizing the names
pivot_longer(
  data = wide_data,
  cols = c(1, 2),
  names_to = "Column",
  values_to = "Numbers"
)
```

`pivot_wider` *Pivot data from long to wide*

Description

`pivot_wider()` "widens" data, increasing the number of columns and decreasing the number of rows. The inverse transformation is `pivot_longer()`.

Usage

```
pivot_wider(
  data,
  id_cols = NULL,
  values_from = "Value",
  names_from = "Name",
  names_sep = "_",
  names_prefix = "",
  names_glue = NULL,
  values_fill = NULL,
  ...
)
```

Arguments

`data`	`data.frame`. The data to pivot.
`id_cols`	`character(1)`. The name of the column that identifies the rows. If `NULL`, it will use all the unique rows.
`values_from`	`character(n)`. The name of the column that contains the values to be used as future variable values.
`names_from`	`character(n)`. The name of the column(s) that contains the levels to be used as future column names.

`names_sep`	`character(1)`. If `names_from` or `values_from` contains multiple variables, this will be used to join their values together into a single string to use as a column name.
`names_prefix`	`character(1)`. String added to the start of every variable name. This is particularly useful if `names_from` is a numeric vector and you want to create syntactic variable names.
`names_glue`	`character(1)`. Instead of `names_sep` and `names_prefix`, you can supply a glue specification that uses the `names_from` columns to create custom column names. Note that the only delimiters supported by `names_glue` are curly brackets, { and }.
`values_fill`	`numeric(n)`. Optionally, a (scalar) value that will be used to replace missing values in the new columns created.
`...`	Not used for now.

Value

If a tibble was provided as input, `pivot_wider()` also returns a tibble. Otherwise, it returns a data frame.

Examples

```
data_long <- read.table(header = TRUE, text = "
 subject sex condition measurement
       1   M   control         7.9
       1   M     cond1        12.3
       1   M     cond2        10.7
       2   F   control         6.3
       2   F     cond1        10.6
       2   F     cond2        11.1
       3   F   control         9.5
       3   F     cond1        13.1
       3   F     cond2        13.8
       4   M   control        11.5
       4   M     cond1        13.4
       4   M     cond2        12.9")


pivot_wider(
  data_long,
  id_cols = "subject",
  names_from = "condition",
  values_from = "measurement"
)

pivot_wider(
  data_long,
  id_cols = "subject",
  names_from = "condition",
  values_from = "measurement",
  names_prefix = "Var.",
```

```
  names_sep = "."
)

production <- expand.grid(
  product = c("A", "B"),
  country = c("AI", "EI"),
  year = 2000:2014
) %>%
  filter((product == "A" & country == "AI") | product == "B") %>%
  mutate(production = rnorm(nrow(.)))

pivot_wider(
  production,
  names_from = c("product", "country"),
  values_from = "production",
  names_glue = "prod_{product}_{country}"
)
```

`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()`. For more complicated criteria, use `case_when()`.

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` *Change column order*

Description

Use `relocate()` to change column positions, using the same syntax as `select()` to make it easy to move blocks of columns at once.

Usage

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

Arguments

`.data`	A `data.frame`.
`...`	<`poor-select`> Columns to move.
`.before, .after`	<`poor-select`> Destination of columns selected by `...`. Supplying neither will move columns to the left-hand side; specifying both will result in an error.

Value

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

- Rows are not affected.
- The same columns appear in the output, but (usually) in a different place.
- Data frame attributes are preserved.
- Groups are not affected.

Examples

```
df <- data.frame(
  a = 1, b = 1, c = 1, d = "a", e = "a", f = "a",
  stringsAsFactors = FALSE
)
df %>% relocate(f)
df %>% relocate(a, .after = c)
df %>% relocate(f, .before = b)
df %>% relocate(a, .after = last_col())
```

```
# Can also select variables based on their type
df %>% relocate(where(is.character))
df %>% relocate(where(is.numeric), .after = last_col())
# Or with any other select helper
df %>% relocate(any_of(c("a", "e", "i", "o", "u")))

# When .before or .after refers to multiple variables they will be
# moved to be immediately before/after the selected variables.
df2 <- data.frame(
  a = 1, b = "a", c = 1, d = "a",
  stringsAsFactors = FALSE
)
df2 %>% relocate(where(is.numeric), .after = where(is.character))
df2 %>% relocate(where(is.numeric), .before = where(is.character))
```

`rename` *Rename columns*

Description

`rename()` changes the names of individual variables using `new_name = old_name` syntax. `rename_with()` renames columns using a function.

Usage

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

rename_with(.data, .fn, .cols = everything(), ...)
```

Arguments

`.data`	A `data.frame`
`...`	For `rename()`: comma separated key-value pairs in the form of `new_name = old_name` to rename selected variables. For `rename_with()`: additional arguments passed onto `.fn`.
`.fn`	A `function()` used to transform the selected `.cols`. Should return a character vector the same length as the input.
`.cols`	Columns to rename; defaults to all columns.

Value

A `data.frame` with the following properties:

- Rows are not affected.
- Column names are changed; column order is preserved.
- `data.frame` attributes are preserved.
- Groups are updated to reflect new names.

Examples

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

rename_with(mtcars, toupper)
rename_with(mtcars, toupper, starts_with("c"))
```

`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

In some quarters, it is considered best to avoid row names, because they are effectively a character column with different semantics than every other column. These functions allow to you detect if a `data.frame` has row names (`has_rownames()`), remove them (`remove_rownames()`), or convert them back-and-forth between an explicit column (`rownames_to_column()` and `column_to_rownames()`). Also included is `rowid_to_column()`, which adds a column at the start of the dataframe of ascending sequential row ids starting at 1. Note that this will remove any existing row names.

Usage

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

rowid_to_column(.data, var = "rowid")

column_to_rownames(.data, var = "rowname")

remove_rownames(.data)

has_rownames(.data)
```

Arguments

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

Value

- `column_to_rownames()` always returns a `data.frame`.
- `has_rownames()` returns a `logical(1)`.
- All other functions return an object of the same class as the input.

Examples

```
# Detect row names
has_rownames(mtcars)
has_rownames(iris)

# Remove row names
remove_rownames(mtcars) %>% has_rownames()

# Convert between row names and column
mtcars <- rownames_to_column(mtcars, var = "car")
column_to_rownames(mtcars, var = "car") %>% head()
```

```
# Adding rowid as a column
rowid_to_column(iris) %>% head()
```

`select` *Subset columns using their names and types*

Description

Select (and optionally rename) variables in a `data.frame`, using a concise mini-language that makes it easy to refer to variables based on their name (e.g. `a:f` selects all columns from `a` on the left to `f` on the right). You can also use predicate functions like `is.numeric()` to select variables based on their properties.

Usage

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

Arguments

`.data`	A `data.frame`.
`...`	<`poor-select`> One or more unquoted expressions separated by commas. Variable names can be used as if they were positions in the data frame, so expressions like `x:y` can be used to select a range of variables.

Details

Overview of selection features:

poorman selections implement a dialect of R where operators make it easy to select variables:

- `:` for selecting a range of consecutive variables.
- `!` for taking the complement of a set of variables.
- `&` and `|` for selecting the intersection or the union of two sets of variables.
- `c()` for combining selections.

In addition, you can use **selection helpers**. Some helpers select specific columns:

- `everything()`: Matches all variables.
- `last_col()`: Select last variable, possibly with an offset.

These helpers select variables by matching patterns in their names:

- `starts_with()`: Starts with a prefix.
- `ends_with()`: Ends with a suffix.
- `contains()`: Contains a literal string.
- `matches()`: Matches a regular expression.
- `num_range()`: Matches a numerical range like `x01`, `x02`, `x03`.

These helpers select variables from a character vector:

- `all_of()`: Matches variable names in a character vector. All names must be present, otherwise an out-of-bounds error is thrown.
- `any_of()`: Same as `all_of()`, except that no error is thrown for names that don't exist.

This helper selects variables with a function:

- `where()`: Applies a function to all variables and selects those for which the function returns `TRUE`.

Value

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

- Rows are not affected.
- Output columns are a subset of input columns, potentially with a different order. Columns will be renamed if `new_name = old_name` form is used.
- Data frame attributes are preserved.
- Groups are maintained; you can't select off grouping variables.

Examples

```
# Here we show the usage for the basic selection operators. See the
# specific help pages to learn about helpers like [starts_with()].

# Select variables by name:
mtcars %>% select(mpg)

# Select multiple variables by separating them with commas. Note
# how the order of columns is determined by the order of inputs:
mtcars %>% select(disp, gear, am)

# Rename variables:
mtcars %>% select(MilesPerGallon = mpg, everything())

# The `:` operator selects a range of consecutive variables:
select(mtcars, mpg:cyl)

# The `!` operator negates a selection:
mtcars %>% select(!(mpg:qsec))
mtcars %>% select(!ends_with("p"))

# `&` and `|` take the intersection or the union of two selections:
iris %>% select(starts_with("Petal") & ends_with("Width"))
iris %>% select(starts_with("Petal") | ends_with("Width"))

# To take the difference between two selections, combine the `&` and
# `!` operators:
iris %>% select(starts_with("Petal") & !ends_with("Width"))
```

`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()`, `group_cols()`

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, ..., .groups = NULL)

summarize(.data, ..., .groups = NULL)
```

Arguments

`.data`	A `data.frame`.
`...`	Name-value pairs of summary functions. The name will be the name of the variable in the result.
`.groups`	`character(1)`. Grouping structure of the result. • `"drop_last"`: drops the last level of grouping. • `"drop"`: all levels of grouping are dropped. • `"keep"`: keeps the same grouping structure as `.data`. When `.groups` is not specified, it is chosen based on the number of rows of the results:

- If all the results have 1 row, you get `"drop_last"`.
- If the number of rows varies, you get `"keep"`.

In addition, a message informs you of that choice, unless the result is ungrouped, the option `"poorman.summarise.inform"` is set to `FALSE`.

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

```
# A summary applied to ungrouped tbl returns a single row
mtcars %>%
  summarise(mean = mean(disp), n = n())

# Usually, you'll want to group first
mtcars %>%
  group_by(cyl) %>%
  summarise(mean = mean(disp), n = n())

# You can summarise to more than one value:
mtcars %>%
   group_by(cyl) %>%
   summarise(qs = quantile(disp, c(0.25, 0.75)), prob = c(0.25, 0.75))

# You use a data frame to create multiple columns so you can wrap
# this up into a function:
my_quantile <- function(x, probs) {
  data.frame(x = quantile(x, probs), probs = probs)
}
mtcars %>%
  group_by(cyl) %>%
  summarise(my_quantile(disp, c(0.25, 0.75)))

# Each summary call removes one grouping level (since that group
# is now just a single row)
mtcars %>%
  group_by(cyl, vs) %>%
  summarise(cyl_n = n()) %>%
  group_vars()
```

`union_all` *Union All*

Description

Union all elements of R objects together.

Usage

```
union_all(x, y, ...)
```

Arguments

`x, y`	objects to union all elements of (ignoring order)
`...`	other arguments passed on to methods

Examples

```
first <- mtcars[1:20, ]
second <- mtcars[10:32, ]
union_all(first, second)

# union_all does not remove duplicates
a <- data.frame(column = c(1:10, 10))
b <- data.frame(column = c(1:5, 5))
union_all(a, b)
```

`unite` *Unite Multiple Columns Into One*

Description

Convenience function to paste together multiple columns.

Usage

```
unite(data, col, ..., sep = "_", remove = TRUE, na.rm = FALSE)
```

Arguments

`data`	A `data.frame`.
`col`	`character(1)` or `symbol(1)`. The name of the new column.
`...`	The columns to unite.
`sep`	`character(1)`. Separator to use between the values.
`remove`	`logical(1)`. If `TRUE`, remove the input columns from the output `data.frame`.
`na.rm`	`logical(1)`. If `TRUE`, missing values will be remove prior to uniting each value.

Value

A `data.frame` with the columns passed via `...` pasted together in a new column.

Examples

```
df <- data.frame(x = c("a", "a", NA, NA), y = c("b", NA, "b", NA))
df

df %>% unite("z", x:y, remove = FALSE)
# To remove missing values:
df %>% unite("z", x:y, na.rm = TRUE, remove = FALSE)
```

`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))
```

`with_groups` *Perform an operation with temporary groups*

Description

This function allows you to modify the grouping variables for a single operation.

Usage

```
with_groups(.data, .groups, .f, ...)
```

Arguments

`.data`	A `data.frame`.
`.groups`	<`poor-select`> One or more variables to group by. Unlike `group_by()`, you can only group by existing variables, and you can use `poor-select` syntax like `c(x, y, z)` to select multiple variables. Use `NULL` to temporarily **un**group.
`.f`	A `function` to apply to regrouped data. Supports lambda-style ~ syntax.
`...`	Additional arguments passed on to `.f`.

Examples

```
df <- data.frame(g = c(1, 1, 2, 2, 3), x = runif(5))
df %>% with_groups(g, mutate, x_mean = mean(x))
df %>% with_groups(g, ~ mutate(.x, x_mean = mean(x)))

df %>%
  group_by(g) %>%
  with_groups(NULL, mutate, x_mean = mean(x))

# NB: grouping can't be restored if you remove the grouping variables
df %>%
  group_by(g) %>%
  with_groups(NULL, mutate, g = NULL)
```

Index