Package 'dplyr'

June 16, 2021

Type Package

Title A Grammar of Data Manipulation

Version 1.0.7

Description A fast, consistent tool for working with data frame like objects, both in memory and out of memory.

License MIT + file LICENSE

URL https://dplyr.tidyverse.org, https://github.com/tidyverse/dplyr

BugReports https://github.com/tidyverse/dplyr/issues

Depends R (>= 3.3.0)

Imports ellipsis, generics, glue (>= 1.3.2), lifecycle (>= 1.0.0), magrittr (>= 1.5), methods, R6, rlang (>= 0.4.10), tibble (>= 2.1.3), tidyselect (>= 1.1.0), utils, vctrs (>= 0.3.5), pillar (>= 1.5.1)

Suggests bench, broom, callr, covr, DBI, dbplyr (>= 1.4.3), knitr, Lahman, lobstr, microbenchmark, nycflights13, purrr, rmarkdown,

RMySQL,
RPostgreSQL,
RSQLite,
testthat (>= 3.0.2),
tidyr,
withr

VignetteBuilder knitr

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.1.1

Config/testthat/edition 3

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()`. See `vignette("colwise")` for more details.

`if_any()` and `if_all()` apply the same predicate function to a selection of columns and combine the results into a single logical vector: `if_any()` is `TRUE` when the predicate is `TRUE` for *any* of the selected columns, `if_all()` is `TRUE` when the predicate is `TRUE` for *all* selected columns.

`across()` supersedes the family of "scoped variants" like `summarise_at()`, `summarise_if()`, and `summarise_all()`.

Usage

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

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

if_all(.cols = everything(), .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 purrr-style 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	<tidy-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 tibble with one column for each column in .cols and each function in .fns.

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

Timing of evaluation

R code in dplyr verbs is generally evaluated once per group. Inside across() however, code is evaluated once for each combination of columns and groups. If the evaluation timing is important, for example if you're generating random variables, think about when it should happen and place your code in consequence.

```
gdf <-
  tibble(g = c(1, 1, 2, 3), v1 = 10:13, v2 = 20:23) %>%
  group_by(g)

set.seed(1)

# Outside: 1 normal variate
n <- rnorm(1)
gdf %>% mutate(across(v1:v2, ~ .x + n))

## # A tibble: 4 x 3
## # Groups:   g [3]
##       g    v1    v2
##   <dbl> <dbl> <dbl>
## 1     1  9.37  19.4
## 2     1 10.4   20.4
## 3     2 11.4   21.4
## 4     3 12.4   22.4

# Inside a verb: 3 normal variates (ngroup)
gdf %>% mutate(n = rnorm(1), across(v1:v2, ~ .x + n))
```

```
## # A tibble: 4 x 4
## # Groups:   g [3]
##       g    v1    v2      n
##   <dbl> <dbl> <dbl>  <dbl>
## 1     1  10.2  20.2  0.184
## 2     1  11.2  21.2  0.184
## 3     2  11.2  21.2 -0.836
## 4     3  14.6  24.6  1.60

# Inside `across()`: 6 normal variates (ncol * ngroup)
gdf %>% mutate(across(v1:v2, ~ .x + rnorm(1)))

## # A tibble: 4 x 3
## # Groups:   g [3]
##       g    v1    v2
##   <dbl> <dbl> <dbl>
## 1     1  10.3  20.7
## 2     1  11.3  21.7
## 3     2  11.2  22.6
## 4     3  13.5  22.7
```

See Also

c_across() for a function that returns a vector

Examples

```
# across() ----------------------------------------------------------------
# Different ways to select the same set of columns
# See <https://tidyselect.r-lib.org/articles/syntax.html> for details
iris %>%
  as_tibble() %>%
  mutate(across(c(Sepal.Length, Sepal.Width), round))
iris %>%
  as_tibble() %>%
  mutate(across(c(1, 2), round))
iris %>%
  as_tibble() %>%
  mutate(across(1:Sepal.Width, round))
iris %>%
  as_tibble() %>%
  mutate(across(where(is.double) & !c(Petal.Length, Petal.Width), round))

# A purrr-style formula
iris %>%
  group_by(Species) %>%
  summarise(across(starts_with("Sepal"), ~ mean(.x, 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 = "mean_{.col}"))
iris %>%
  group_by(Species) %>%
 summarise(across(starts_with("Sepal"), list(mean = mean, sd = sd), .names = "{.col}.{.fn}"))

# When the list is not named, .fn is replaced by the function's position
iris %>%
  group_by(Species) %>%
  summarise(across(starts_with("Sepal"), list(mean, sd), .names = "{.col}.fn{.fn}"))

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

`all_vars` *Apply predicate to all variables*

Description

[Superseded]

`all_vars()` and `any_vars()` were only needed for the scoped verbs, which have been superseded by the use of `across()` in an existing verb. See `vignette("colwise")` for details.

These quoting functions signal to scoped filtering verbs (e.g. `filter_if()` or `filter_all()`) that a predicate expression should be applied to all relevant variables. The `all_vars()` variant takes the intersection of the predicate expressions with `&` while the `any_vars()` variant takes the union with `|`.

Usage

```
all_vars(expr)

any_vars(expr)
```

Arguments

`expr`	<`data-masking`> An expression that returns a logical vector, using `.` to refer to the "current" variable.

See Also

`vars()` for other quoting functions that you can use with scoped verbs.

arrange *Arrange rows by column values*

Description

`arrange()` orders the rows of a data frame by the values of selected columns.

Unlike other dplyr verbs, `arrange()` largely ignores grouping; you need to explicitly mention grouping variables (or use `.by_group = TRUE`) in order to group by them, and functions of variables are evaluated once per data frame, not once per group.

Usage

```
arrange(.data, ..., .by_group = FALSE)

## S3 method for class 'data.frame'
arrange(.data, ..., .by_group = FALSE)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<data-masking> Variables, or functions of variables. Use `desc()` to sort a variable in descending order.
`.by_group`	If `TRUE`, will sort first by grouping variable. Applies to grouped data frames only.

Details

Locales:

The sort order for character vectors will depend on the collating sequence of the locale in use: see `locales()`.

Missing values:

Unlike base sorting with `sort()`, `NA` are:

- always sorted to the end for local data, even when wrapped with `desc()`.
- treated differently for remote data, depending on the backend.

Value

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

- All rows appear in the output, but (usually) in a different place.
- Columns are not modified.
- Groups are not modified.
- Data frame attributes are preserved.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

See Also

Other single table verbs: `filter()`, `mutate()`, `rename()`, `select()`, `slice()`, `summarise()`

Examples

```
arrange(mtcars, cyl, disp)
arrange(mtcars, desc(disp))

# grouped arrange ignores groups
by_cyl <- mtcars %>% group_by(cyl)
by_cyl %>% arrange(desc(wt))
# Unless you specifically ask:
by_cyl %>% arrange(desc(wt), .by_group = TRUE)

# use embracing when wrapping in a function;
# see ?dplyr_data_masking for more details
tidy_eval_arrange <- function(.data, var) {
  .data %>%
    arrange({{ var }})
}
tidy_eval_arrange(mtcars, mpg)

# use across() access select()-style semantics
iris %>% arrange(across(starts_with("Sepal")))
iris %>% arrange(across(starts_with("Sepal"), desc))
```

---

arrange_all *Arrange rows by a selection of variables*

Description

[Superseded]

Scoped verbs (_if, _at, _all) have been superseded by the use of `across()` in an existing verb. See `vignette("colwise")` for details.

These scoped variants of `arrange()` sort a data frame by a selection of variables. Like `arrange()`, you can modify the variables before ordering with the `.funs` argument.

Usage

```
arrange_all(.tbl, .funs = list(), ..., .by_group = FALSE)

arrange_at(.tbl, .vars, .funs = list(), ..., .by_group = FALSE)

arrange_if(.tbl, .predicate, .funs = list(), ..., .by_group = FALSE)
```

Arguments

.tbl	A tbl object.
.funs	A function fun, a quosure style lambda ~ fun(.) or a list of either form.
...	Additional arguments for the function calls in .funs. These are evaluated only once, with tidy dots support.
.by_group	If TRUE, will sort first by grouping variable. Applies to grouped data frames only.
.vars	A list of columns generated by vars(), a character vector of column names, a numeric vector of column positions, or NULL.
.predicate	A predicate function to be applied to the columns or a logical vector. The variables for which .predicate is or returns TRUE are selected. This argument is passed to rlang::as_function() and thus supports quosure-style lambda functions and strings representing function names.

Grouping variables

The grouping variables that are part of the selection participate in the sorting of the data frame.

Examples

```
df <- as_tibble(mtcars)
arrange_all(df)
# ->
arrange(df, across())

arrange_all(df, desc)
# ->
arrange(df, across(everything(), desc))
```

---

auto_copy *Copy tables to same source, if necessary*

Description

Copy tables to same source, if necessary

Usage

```
auto_copy(x, y, copy = FALSE, ...)
```

Arguments

x, y	y will be copied to x, if necessary.
copy	If x and y are not from the same data source, and copy is TRUE, then y will be copied into the same src as x. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
...	Other arguments passed on to methods.

band_members *Band membership*

Description

These data sets describe band members of the Beatles and Rolling Stones. They are toy data sets that can be displayed in their entirety on a slide (e.g. to demonstrate a join).

Usage

```
band_members

band_instruments

band_instruments2
```

Format

Each is a tibble with two variables and three observations

Details

band_instruments and band_instruments2 contain the same data but use different column names for the first column of the data set. band_instruments uses name, which matches the name of the key column of band_members; band_instruments2 uses artist, which does not.

Examples

```
band_members
band_instruments
band_instruments2
```

---

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

Description

This is a shortcut for x >= left & x <= right, implemented efficiently in C++ for local values, and translated to the appropriate SQL for remote tables.

Usage

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

Arguments

x	A numeric vector of values
left, right	Boundary values (must be scalars).

Examples

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

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

## Or on a tibble using filter
filter(starwars, between(height, 100, 150))
```

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

Description

This is an efficient implementation of the common pattern of `do.call(rbind,dfs)` or `do.call(cbind,dfs)` for binding many data frames into one.

Usage

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

bind_cols(
  ...,
  .name_repair = c("unique", "universal", "check_unique", "minimal")
)
```

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`	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.
`.name_repair`	One of `"unique"`, `"universal"`, or `"check_unique"`. See `vctrs::vec_as_names()` for the meaning of these options.

Details

The output of `bind_rows()` will contain a column if that column appears in any of the inputs.

Value

`bind_rows()` and `bind_cols()` return the same type as the first input, either a data frame, `tbl_df`, or `grouped_df`.

Examples

```
one <- starwars[1:4, ]
two <- starwars[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(starwars, starwars$homeworld))
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),
  tibble(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")

# Columns don't need to match when row-binding
bind_rows(tibble(x = 1:3), tibble(y = 1:4))
## Not run:
# Rows do need to match when column-binding
bind_cols(tibble(x = 1:3), tibble(y = 1:2))

# even with 0 columns
bind_cols(tibble(x = 1:3), tibble())

## End(Not run)

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

---

case_when *A general vectorised 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

`...`	<dynamic-dots> 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
)

# This throws an error as NA is logical not numeric
## 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
starwars %>%
  select(name:mass, gender, species) %>%
  mutate(
    type = case_when(
      height > 200 | mass > 200 ~ "large",
      species == "Droid"        ~ "robot",
      TRUE                      ~ "other"
    )
  )
```

```
# `case_when()` is not a tidy eval function. If you'd like to reuse
# the same patterns, extract the `case_when()` call in a normal
# function:
case_character_type <- function(height, mass, species) {
  case_when(
    height > 200 | mass > 200 ~ "large",
    species == "Droid"        ~ "robot",
    TRUE                      ~ "other"
  )
}

case_character_type(150, 250, "Droid")
case_character_type(150, 150, "Droid")

# Such functions can be used inside `mutate()` as well:
starwars %>%
  mutate(type = case_character_type(height, mass, species)) %>%
  pull(type)

# `case_when()` ignores `NULL` inputs. This is useful when you'd
# like to use a pattern only under certain conditions. Here we'll
# take advantage of the fact that `if` returns `NULL` when there is
# no `else` clause:
case_character_type <- function(height, mass, species, robots = TRUE) {
  case_when(
    height > 200 | mass > 200      ~ "large",
    if (robots) species == "Droid" ~ "robot",
    TRUE                           ~ "other"
  )
}

starwars %>%
  mutate(type = case_character_type(height, mass, species, robots = FALSE)) %>%
  pull(type)
```

`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

`...`	<dynamic-dots> Vectors. Inputs should be recyclable (either be length 1 or same length as the longest vector) and coercible to a common type. If data frames, they are coalesced column by column.

Value

A vector the same length as the first ... argument with missing values replaced by the first non-missing value.

See Also

`na_if()` to replace specified values with a `NA`. `tidyr::replace_na()` to replace `NA` with a value

Examples

```
# Use a single value to replace all missing values
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)

# Supply lists by with dynamic dots
vecs <- list(
  c(1, 2, NA, NA, 5),
  c(NA, NA, 3, 4, 5)
)
coalesce(!!!vecs)
```

`compute` *Force computation of a database query*

Description

`compute()` stores results in a remote temporary table. `collect()` retrieves data into a local tibble. `collapse()` is slightly different: it doesn't force computation, but instead forces generation of the SQL query. This is sometimes needed to work around bugs in dplyr's SQL generation.

All functions preserve grouping and ordering.

Usage

```
compute(x, ...)

collect(x, ...)

collapse(x, ...)
```

Arguments

`x`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	Arguments passed on to methods

Methods

These functions are **generics**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

Methods available in currently loaded packages:

- `compute()`: no methods found
- `collect()`: no methods found
- `collapse()`: no methods found

See Also

`copy_to()`, the opposite of `collect()`: it takes a local data frame and uploads it to the remote source.

Examples

```
if (require(dbplyr)) {
  mtcars2 <- src_memdb() %>%
    copy_to(mtcars, name = "mtcars2-cc", overwrite = TRUE)

  remote <- mtcars2 %>%
    filter(cyl == 8) %>%
    select(mpg:drat)

  # Compute query and save in remote table
  compute(remote)

  # Compute query bring back to this session
  collect(remote)

  # Creates a fresh query based on the generated SQL
  collapse(remote)
}
```

`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 current group size.
- `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 tibble with one row and one column for each grouping variable.
- `cur_group_id()` gives a unique numeric identifier for the current group.
- `cur_group_rows()` gives the row indices for the current group.
- `cur_column()` gives the name of the current column (in `across()` only).

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

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`

Examples

```
df <- tibble(
  g = sample(rep(letters[1:3], 1:3)),
  x = runif(6),
  y = runif(6)
)
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))))
```

---

copy_to *Copy a local data frame to a remote src*

Description

This function uploads a local data frame into a remote data source, creating the table definition as needed. Wherever possible, the new object will be temporary, limited to the current connection to the source.

Usage

```
copy_to(dest, df, name = deparse(substitute(df)), overwrite = FALSE, ...)
```

Arguments

dest	remote data source
df	local data frame
name	name for new remote table.
overwrite	If TRUE, will overwrite an existing table with name name. If FALSE, will throw an error if name already exists.
...	other parameters passed to methods.

Value

a tbl object in the remote source

See Also

collect() for the opposite action; downloading remote data into a local dbl.

Examples

```
## Not run:
iris2 <- dbplyr::src_memdb() %>% copy_to(iris, overwrite = TRUE)
iris2

## End(Not run)
```

---

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 n = n() to n = sum(wt).

add_count() and add_tally() are equivalents 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, .drop = deprecated())

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

Arguments

x	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr).
...	<data-masking> Variables to group by.
wt	<data-masking> Frequency weights. Can be NULL or a variable: • If NULL (the default), counts the number of rows in each group. • If a variable, computes sum(wt) for each group.
sort	If TRUE, will show the largest groups at the top.
name	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.
.drop	For count(): if FALSE will include counts for empty groups (i.e. for levels of factors that don't exist in the data). Deprecated in add_count() since it didn't actually affect the output.

Value

An object of the same type as .data. count() and add_count() group transiently, so the output has the same groups as the input.

Examples

```
# count() is a convenient way to get a sense of the distribution of
# values in a dataset
starwars %>% count(species)
starwars %>% count(species, sort = TRUE)
starwars %>% count(sex, gender, sort = TRUE)
starwars %>% count(birth_decade = round(birth_year, -1))

# use the `wt` argument to perform a weighted count. This is useful
# when the data has already been aggregated once
df <- tribble(
  ~name,    ~gender,   ~runs,
  "Max",    "male",       10,
  "Sandra", "female",      1,
  "Susan",  "female",      4
)
# counts rows:
df %>% count(gender)
# counts runs:
df %>% count(gender, wt = runs)

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

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

cumall *Cumulativate versions of any, all, and mean*

Description

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

Usage

```
cumall(x)

cumany(x)

cummean(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)))
```

---

c_across *Combine values from multiple columns*

Description

`c_across()` is designed to work with `rowwise()` to make it easy to perform row-wise aggregations. It has two differences from `c()`:

- It uses tidy select semantics so you can easily select multiple variables. See `vignette("rowwise")` for more details.
- It uses `vctrs::vec_c()` in order to give safer outputs.

Usage

```
c_across(cols = everything())
```

Arguments

cols	<tidy-select> Columns to transform. Because `across()` is used within functions like `summarise()` and `mutate()`, you can't select or compute upon grouping variables.

See Also

`across()` for a function that returns a tibble.

Examples

```
df <- tibble(id = 1:4, w = runif(4), x = runif(4), y = runif(4), z = runif(4))
df %>%
  rowwise() %>%
  mutate(
    sum = sum(c_across(w:z)),
    sd = sd(c_across(w:z))
 )
```

---

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	vector to transform

Examples

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

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

starwars %>% arrange(desc(mass))
```

---

distinct *Subset distinct/unique rows*

Description

Select only unique/distinct rows from a data frame. This is similar to `unique.data.frame()` but considerably faster.

Usage

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

Arguments

.data	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
...	<data-masking> 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	If TRUE, keep all variables in .data. If a combination of ... is not distinct, this keeps the first row of values.

Value

An object of the same type as .data. The output has 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.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

Examples

```
df <- tibble(
  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))

# use across() to access select()-style semantics
distinct(starwars, across(contains("color")))

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

`distinct_all` *Select distinct rows by a selection of variables*

Description

[Superseded]

Scoped verbs (_if, _at, _all) have been superseded by the use of `across()` in an existing verb. See `vignette("colwise")` for details.

These scoped variants of `distinct()` extract distinct rows by a selection of variables. Like `distinct()`, you can modify the variables before ordering with the `.funs` argument.

Usage

```
distinct_all(.tbl, .funs = list(), ..., .keep_all = FALSE)

distinct_at(.tbl, .vars, .funs = list(), ..., .keep_all = FALSE)

distinct_if(.tbl, .predicate, .funs = list(), ..., .keep_all = FALSE)
```

Arguments

.tbl	A tbl object.
.funs	A function fun, a quosure style lambda ~ fun(.) or a list of either form.
...	Additional arguments for the function calls in .funs. These are evaluated only once, with tidy dots support.
.keep_all	If TRUE, keep all variables in .data. If a combination of ... is not distinct, this keeps the first row of values.
.vars	A list of columns generated by vars(), a character vector of column names, a numeric vector of column positions, or NULL.
.predicate	A predicate function to be applied to the columns or a logical vector. The variables for which .predicate is or returns TRUE are selected. This argument is passed to rlang::as_function() and thus supports quosure-style lambda functions and strings representing function names.

Grouping variables

The grouping variables that are part of the selection are taken into account to determine distinct rows.

Examples

```
df <- tibble(x = rep(2:5, each = 2) / 2, y = rep(2:3, each = 4) / 2)

distinct_all(df)
# ->
distinct(df, across())

distinct_at(df, vars(x,y))
# ->
distinct(df, across(c(x, y)))

distinct_if(df, is.numeric)
# ->
distinct(df, across(where(is.numeric)))

# You can supply a function that will be applied before extracting the distinct values
# The variables of the sorted tibble keep their original values.
distinct_all(df, round)
# ->
distinct(df, across(everything(), round))
```

---

explain *Explain details of a tbl*

Description

This is a generic function which gives more details about an object than print(), and is more focused on human readable output than str().

Usage

```
explain(x, ...)

show_query(x, ...)
```

Arguments

x	An object to explain
...	Other parameters possibly used by generic

Value

The first argument, invisibly.

Databases

Explaining a `tbl_sql` will run the SQL `EXPLAIN` command which will describe the query plan. This requires a little bit of knowledge about how `EXPLAIN` works for your database, but is very useful for diagnosing performance problems.

Examples

```
if (require("dbplyr")) {

lahman_s <- lahman_sqlite()
batting <- tbl(lahman_s, "Batting")
batting %>% show_query()
batting %>% explain()

# The batting database has indices on all ID variables:
# SQLite automatically picks the most restrictive index
batting %>% filter(lgID == "NL" & yearID == 2000L) %>% explain()

# OR's will use multiple indexes
batting %>% filter(lgID == "NL" | yearID == 2000) %>% explain()

# Joins will use indexes in both tables
teams <- tbl(lahman_s, "Teams")
batting %>% left_join(teams, c("yearID", "teamID")) %>% explain()
}
```

`filter` *Subset rows using column values*

Description

The `filter()` function is used to subset a data frame, retaining all rows that satisfy your conditions. To be retained, the row must produce a value of `TRUE` for all conditions. Note that when a condition evaluates to `NA` the row will be dropped, unlike base subsetting with `[`.

Usage

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

Arguments

.data	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
...	<data-masking> Expressions that return a logical value, and are defined in terms of the variables in .data. If multiple expressions are included, they are combined with the & operator. Only rows for which all conditions evaluate to TRUE are kept.
.preserve	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.

Details

The filter() function is used to subset the rows of .data, applying the expressions in ... to the column values to determine which rows should be retained. It can be applied to both grouped and ungrouped data (see group_by() and ungroup()). However, dplyr is not yet smart enough to optimise the filtering operation on grouped datasets that do not need grouped calculations. For this reason, filtering is often considerably faster on ungrouped data.

Value

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

- Rows are a subset of the input, but appear in the same order.
- Columns are not modified.
- The number of groups may be reduced (if .preserve is not TRUE).
- Data frame attributes are preserved.

Useful filter functions

There are many functions and operators that are useful when constructing the expressions used to filter the data:

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

Grouped tibbles

Because filtering expressions are computed within groups, they may yield different results on grouped tibbles. This will be the case as soon as an aggregating, lagging, or ranking function is involved. Compare this ungrouped filtering:

```
starwars %>% filter(mass > mean(mass, na.rm = TRUE))
```

With the grouped equivalent:

```
starwars %>% group_by(gender) %>% filter(mass > mean(mass, na.rm = TRUE))
```

In the ungrouped version, `filter()` compares the value of `mass` in each row to the global average (taken over the whole data set), keeping only the rows with `mass` greater than this global average. In contrast, the grouped version calculates the average mass separately for each `gender` group, and keeps rows with `mass` greater than the relevant within-gender average.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

See Also

Other single table verbs: `arrange()`, `mutate()`, `rename()`, `select()`, `slice()`, `summarise()`

Examples

```
# Filtering by one criterion
filter(starwars, species == "Human")
filter(starwars, mass > 1000)

# Filtering by multiple criteria within a single logical expression
filter(starwars, hair_color == "none" & eye_color == "black")
filter(starwars, hair_color == "none" | eye_color == "black")

# When multiple expressions are used, they are combined using &
filter(starwars, hair_color == "none", eye_color == "black")


# The filtering operation may yield different results on grouped
# tibbles because the expressions are computed within groups.
#
# The following filters rows where `mass` is greater than the
# global average:
starwars %>% filter(mass > mean(mass, na.rm = TRUE))

# Whereas this keeps rows with `mass` greater than the gender
# average:
starwars %>% group_by(gender) %>% filter(mass > mean(mass, na.rm = TRUE))


# To refer to column names that are stored as strings, use the `.data` pronoun:
vars <- c("mass", "height")
cond <- c(80, 150)
starwars %>%
  filter(
    .data[[vars[[1]]]] > cond[[1]],
    .data[[vars[[2]]]] > cond[[2]]
  )
# Learn more in ?dplyr_data_masking
```

filter-joins *Filtering joins*

Description

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

- 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

```
semi_join(x, y, by = NULL, copy = FALSE, ...)

## S3 method for class 'data.frame'
semi_join(x, y, by = NULL, copy = FALSE, ..., na_matches = c("na", "never"))

anti_join(x, y, by = NULL, copy = FALSE, ...)

## S3 method for class 'data.frame'
anti_join(x, y, by = NULL, copy = FALSE, ..., na_matches = c("na", "never"))
```

Arguments

x, y	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
by	A character vector of variables to join by. If NULL, the default, *_join() will perform a natural join, using all variables in common across x and y. A message lists the variables so that you can check they're correct; suppress the message by supplying by explicitly. 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().
copy	If x and y are not from the same data source, and copy is TRUE, then y will be copied into the same src as x. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
...	Other parameters passed onto methods.
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).

Value

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

- Rows are a subset of the input, but appear in the same order.
- Columns are not modified.
- Data frame attributes are preserved.
- Groups are taken from `x`. The number of groups may be reduced.

Methods

These function are **generic**s, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

Methods available in currently loaded packages:

- `semi_join()`: no methods found.
- `anti_join()`: no methods found.

See Also

Other joins: `mutate-joins`, `nest_join()`

Examples

```
# "Filtering" joins keep cases from the LHS
band_members %>% semi_join(band_instruments)
band_members %>% anti_join(band_instruments)

# To suppress the message about joining variables, supply `by`
band_members %>% semi_join(band_instruments, by = "name")
# This is good practice in production code
```

`filter_all` *Filter within a selection of variables*

Description

[Superseded]

Scoped verbs (_if, _at, _all) have been superseded by the use of `across()` in an existing verb. See `vignette("colwise")` for details.

These scoped filtering verbs apply a predicate expression to a selection of variables. The predicate expression should be quoted with `all_vars()` or `any_vars()` and should mention the pronoun `.` to refer to variables.

Usage

```
filter_all(.tbl, .vars_predicate, .preserve = FALSE)

filter_if(.tbl, .predicate, .vars_predicate, .preserve = FALSE)

filter_at(.tbl, .vars, .vars_predicate, .preserve = FALSE)
```

Arguments

.tbl	A tbl object.
.vars_predicate	A quoted predicate expression as returned by all_vars() or any_vars(). Can also be a function or purrr-like formula. In this case, the intersection of the results is taken by default and there's currently no way to request the union.
.preserve	when FALSE (the default), the grouping structure is recalculated based on the resulting data, otherwise it is kept as is.
.predicate	A predicate function to be applied to the columns or a logical vector. The variables for which .predicate is or returns TRUE are selected. This argument is passed to rlang::as_function() and thus supports quosure-style lambda functions and strings representing function names.
.vars	A list of columns generated by vars(), a character vector of column names, a numeric vector of column positions, or NULL.

Grouping variables

The grouping variables that are part of the selection are taken into account to determine filtered rows.

Examples

```
# While filter() accepts expressions with specific variables, the
# scoped filter verbs take an expression with the pronoun `.` and
# replicate it over all variables. This expression should be quoted
# with all_vars() or any_vars():
all_vars(is.na(.))
any_vars(is.na(.))


# You can take the intersection of the replicated expressions:
filter_all(mtcars, all_vars(. > 150))
# ->
filter(mtcars, if_all(everything(), ~ .x > 150))

# Or the union:
filter_all(mtcars, any_vars(. > 150))
# ->
filter(mtcars, if_any(everything(), ~ . > 150))


# You can vary the selection of columns on which to apply the
# predicate. filter_at() takes a vars() specification:
filter_at(mtcars, vars(starts_with("d")), any_vars((. %% 2) == 0))
# ->
filter(mtcars, if_any(starts_with("d"), ~ (.x %% 2) == 0))

# And filter_if() selects variables with a predicate function:
filter_if(mtcars, ~ all(floor(.) == .), all_vars(. != 0))
# ->
is_int <- function(x) all(floor(x) == x)
filter(mtcars, if_all(where(is_int), ~ .x != 0))
```

group_by *Group by one or more variables*

Description

Most data operations are done on groups defined by variables. `group_by()` takes an existing tbl and converts it into a grouped tbl where operations are performed "by group". `ungroup()` removes grouping.

Usage

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

ungroup(x, ...)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	In `group_by()`, variables or computations to group by. In `ungroup()`, variables to remove from the grouping.
`.add`	When `FALSE`, the default, `group_by()` will override existing groups. To add to the existing groups, use `.add = TRUE`. This argument was previously called `add`, but that prevented creating a new grouping variable called `add`, and conflicts with our naming conventions.
`.drop`	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 `tbl()`

Value

A grouped data frame with class `grouped_df`, unless the combination of `...` and `add` yields a empty set of grouping columns, in which case a tibble will be returned.

Methods

These function are **generic**s, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

Methods available in currently loaded packages:

- `group_by()`: no methods found.
- `ungroup()`: no methods found.

See Also

Other grouping functions: `group_map()`, `group_nest()`, `group_split()`, `group_trim()`

Examples

```
by_cyl <- mtcars %>% group_by(cyl)

# grouping doesn't change how the data looks (apart from listing
# how it's grouped):
by_cyl

# It changes how it acts with the other dplyr verbs:
by_cyl %>% summarise(
  disp = mean(disp),
  hp = mean(hp)
)
by_cyl %>% filter(disp == max(disp))

# Each call to summarise() removes a layer of grouping
by_vs_am <- mtcars %>% group_by(vs, am)
by_vs <- by_vs_am %>% summarise(n = n())
by_vs
by_vs %>% summarise(n = sum(n))

# To removing grouping, use ungroup
by_vs %>%
  ungroup() %>%
  summarise(n = sum(n))

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

# By default, group_by() overrides existing grouping
by_cyl %>%
  group_by(vs, am) %>%
  group_vars()

# Use add = TRUE to instead append
by_cyl %>%
  group_by(vs, am, .add = TRUE) %>%
  group_vars()


# when factors are involved and .drop = FALSE, groups can be empty
tbl <- tibble(
  x = 1:10,
  y = factor(rep(c("a", "c"), each  = 5), levels = c("a", "b", "c"))
)
tbl %>%
  group_by(y, .drop = FALSE) %>%
  group_rows()
```

`group_by_all` *Group by a selection of variables*

Description

[Superseded]

Scoped verbs (_if, _at, _all) have been superseded by the use of `across()` in an existing verb. See `vignette("colwise")` for details.

These scoped variants of `group_by()` group a data frame by a selection of variables. Like `group_by()`, they have optional mutate semantics.

Usage

```
group_by_all(
  .tbl,
  .funs = list(),
  ...,
  .add = FALSE,
  .drop = group_by_drop_default(.tbl)
)

group_by_at(
  .tbl,
  .vars,
  .funs = list(),
  ...,
  .add = FALSE,
  .drop = group_by_drop_default(.tbl)
)

group_by_if(
  .tbl,
  .predicate,
  .funs = list(),
  ...,
  .add = FALSE,
  .drop = group_by_drop_default(.tbl)
)
```

Arguments

`.tbl`	A `tbl` object.
`.funs`	A function `fun`, a quosure style lambda `~ fun(.)` or a list of either form.
`...`	Additional arguments for the function calls in `.funs`. These are evaluated only once, with tidy dots support.
`.add`	See `group_by()`
`.drop`	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.
`.vars`	A list of columns generated by `vars()`, a character vector of column names, a numeric vector of column positions, or `NULL`.
`.predicate`	A predicate function to be applied to the columns or a logical vector. The variables for which `.predicate` is or returns `TRUE` are selected. This argument is passed to `rlang::as_function()` and thus supports quosure-style lambda functions and strings representing function names.

Grouping variables

Existing grouping variables are maintained, even if not included in the selection.

Examples

```
# Group a data frame by all variables:
group_by_all(mtcars)
# ->
mtcars %>% group_by(across())

# Group by variables selected with a predicate:
group_by_if(iris, is.factor)
# ->
iris %>% group_by(across(where(is.factor)))

# Group by variables selected by name:
group_by_at(mtcars, vars(vs, am))
# ->
mtcars %>% group_by(across(c(vs, am)))

# Like group_by(), the scoped variants have optional mutate
# semantics. This provide a shortcut for group_by() + mutate():
d <- tibble(x=c(1,1,2,2), y=c(1,2,1,2))
group_by_all(d, as.factor)
# ->
d %>% group_by(across(everything(), as.factor))

group_by_if(iris, is.factor, as.character)
# ->
iris %>% group_by(across(where(is.factor), as.character))
```

---

group_cols *Select grouping variables*

Description

This selection helpers matches grouping variables. It can be used in `select()` or `vars()` selections.

Usage

```
group_cols(vars = NULL, data = NULL)
```

Arguments

vars	Deprecated; please use data instead.
data	For advanced use only. The default NULL automatically finds the "current" data frames.

See Also

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

Examples

```
gdf <- iris %>% group_by(Species)
gdf %>% select(group_cols())

# Remove the grouping variables from mutate selections:
gdf %>% mutate_at(vars(-group_cols()), `/`, 100)
# -> No longer necessary with across()
gdf %>% mutate(across(everything(), ~ . / 100))
```

`group_map` *Apply a function to each group*

Description

[Experimental]

`group_map()`, `group_modify()` and `group_walk()` are purrr-style functions that can be used to iterate on grouped tibbles.

Usage

```
group_map(.data, .f, ..., .keep = FALSE)

group_modify(.data, .f, ..., .keep = FALSE)

group_walk(.data, .f, ...)
```

Arguments

`.data`	A grouped tibble
`.f`	A function or formula to apply to each group. If a **function**, it is used as is. It should have at least 2 formal arguments. If a **formula**, e.g. `~ head(.x)`, it is converted to a function. In the formula, you can use • `.` or `.x` to refer to the subset of rows of `.tbl` for the given group • `.y` to refer to the key, a one row tibble with one column per grouping variable that identifies the group
`...`	Additional arguments passed on to `.f`
`.keep`	are the grouping variables kept in `.x`

Details

Use `group_modify()` when `summarize()` is too limited, in terms of what you need to do and return for each group. `group_modify()` is good for "data frame in, data frame out". If that is too limited, you need to use a nested or split workflow. `group_modify()` is an evolution of `do()`, if you have used that before.

Each conceptual group of the data frame is exposed to the function `.f` with two pieces of information:

- The subset of the data for the group, exposed as `.x`.

- The key, a tibble with exactly one row and columns for each grouping variable, exposed as `.y`.

For completeness, `group_modify()`, `group_map` and `group_walk()` also work on ungrouped data frames, in that case the function is applied to the entire data frame (exposed as `.x`), and `.y` is a one row tibble with no column, consistently with `group_keys()`.

Value

- `group_modify()` returns a grouped tibble. In that case `.f` must return a data frame.
- `group_map()` returns a list of results from calling `.f` on each group.
- `group_walk()` calls `.f` for side effects and returns the input `.tbl`, invisibly.

See Also

Other grouping functions: `group_by()`, `group_nest()`, `group_split()`, `group_trim()`

Examples

```
# return a list
mtcars %>%
  group_by(cyl) %>%
  group_map(~ head(.x, 2L))

# return a tibble grouped by `cyl` with 2 rows per group
# the grouping data is recalculated
mtcars %>%
  group_by(cyl) %>%
  group_modify(~ head(.x, 2L))

if (requireNamespace("broom", quietly = TRUE)) {
  # a list of tibbles
  iris %>%
    group_by(Species) %>%
    group_map(~ broom::tidy(lm(Petal.Length ~ Sepal.Length, data = .x)))

  # a restructured grouped tibble
  iris %>%
    group_by(Species) %>%
    group_modify(~ broom::tidy(lm(Petal.Length ~ Sepal.Length, data = .x)))
}

# a list of vectors
iris %>%
  group_by(Species) %>%
  group_map(~ quantile(.x$Petal.Length, probs = c(0.25, 0.5, 0.75)))

# to use group_modify() the lambda must return a data frame
iris %>%
  group_by(Species) %>%
  group_modify(~ {
     quantile(.x$Petal.Length, probs = c(0.25, 0.5, 0.75)) %>%
     tibble::enframe(name = "prob", value = "quantile")
  })

iris %>%
  group_by(Species) %>%
```

```
  group_modify(~ {
    .x %>%
      purrr::map_dfc(fivenum) %>%
      mutate(nms = c("min", "Q1", "median", "Q3", "max"))
  })

# group_walk() is for side effects
dir.create(temp <- tempfile())
iris %>%
  group_by(Species) %>%
  group_walk(~ write.csv(.x, file = file.path(temp, paste0(.y$Species, ".csv"))))
list.files(temp, pattern = "csv$")
unlink(temp, recursive = TRUE)

# group_modify() and ungrouped data frames
mtcars %>%
  group_modify(~ head(.x, 2L))
```

`group_split` *Split data frame by groups*

Description

[Experimental] `group_split()` works like `base::split()` but

- it uses the grouping structure from `group_by()` and therefore is 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.

`group_keys()` explains the grouping structure, by returning a data frame that has one row per group and one column per grouping variable.

Usage

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

Arguments

`.tbl`	A tbl
`...`	Grouping specification, forwarded to `group_by()`
`.keep`	Should the grouping columns be kept

Value

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

Grouped data frames

The primary use case for group_split() is with already grouped data frames, typically a result of group_by(). In this case group_split() only uses the first argument, the grouped tibble, 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 frames

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

Using these functions on an ungrouped data frame only makes sense if you need only one or the other, because otherwise the grouping algorithm is performed each time.

Rowwise data frames

group_split() returns a list of one-row tibbles is returned, and the ... are ignored and warned against

See Also

Other grouping functions: group_by(), group_map(), group_nest(), group_trim()

Examples

```
# ----- use case 1 : on an already grouped tibble
ir <- iris %>%
  group_by(Species)

group_split(ir)
group_keys(ir)

# this can be useful if the grouped data has been altered before the split
ir <- iris %>%
  group_by(Species) %>%
  filter(Sepal.Length > mean(Sepal.Length))

group_split(ir)
group_keys(ir)

# ----- use case 2: using a group_by() grouping specification

# both group_split() and group_keys() have to perform the grouping
# so it only makes sense to do this if you only need one or the other
iris %>%
  group_split(Species)

iris %>%
  group_keys(Species)
```

group_trim *Trim grouping structure*

Description

[Experimental] Drop unused levels of all factors that are used as grouping variables, then recalculates the grouping structure.

`group_trim()` is particularly useful after a `filter()` that is intended to select a subset of groups.

Usage

```
group_trim(.tbl, .drop = group_by_drop_default(.tbl))
```

Arguments

`.tbl`	A grouped data frame
`.drop`	See `group_by()`

Value

A grouped data frame

See Also

Other grouping functions: `group_by()`, `group_map()`, `group_nest()`, `group_split()`

Examples

```
iris %>%
  group_by(Species) %>%
  filter(Species == "setosa", .preserve = TRUE) %>%
  group_trim()
```

ident *Flag a character vector as SQL identifiers*

Description

`ident()` takes unquoted strings and flags them as identifiers. `ident_q()` assumes its input has already been quoted, and ensures it does not get quoted again. This is currently used only for for `schema.table`.

Usage

```
ident(...)
```

Arguments

`...`	A character vector, or name-value pairs

Examples

```
# Identifiers are escaped with "
if (requireNamespace("dbplyr", quietly = TRUE)) {
  ident("x")
}
```

`if_else` *Vectorised if*

Description

Compared to the base `ifelse()`, this function is more strict. It checks that `true` and `false` are the same type. This strictness makes the output type more predictable, and makes it somewhat faster.

Usage

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

Arguments

`condition`	Logical vector
`true, false`	Values to use for `TRUE` and `FALSE` values of `condition`. They must be either the same length as `condition`, or length 1. They must also be the same type: `if_else()` checks that they have the same type and same class. All other attributes are taken from `true`.
`missing`	If not `NULL`, will be used to replace missing values.

Value

Where `condition` is `TRUE`, the matching value from `true`, where it's `FALSE`, the matching value from `false`, otherwise `NA`.

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))
if_else(x %in% c("a", "b", "c"), x, factor(NA))
# Attributes are taken from the `true` vector,
```

lead-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, order_by = NULL, ...)

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

Arguments

`x`	Vector of values
`n`	Positive integer of length 1, giving the number of positions to lead or lag by
`default`	Value used for non-existent rows. Defaults to `NA`.
`order_by`	Override the default ordering to use another vector or column
`...`	Needed for compatibility with lag generic.

Examples

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

x <- 1:5
tibble(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)

# If data are not already ordered, use `order_by`
scrambled <- slice_sample(tibble(year = 2000:2005, value = (0:5) ^ 2), prop = 1)

wrong <- mutate(scrambled, previous_year_value = lag(value))
arrange(wrong, year)

right <- mutate(scrambled, previous_year_value = lag(value, order_by = year))
arrange(right, year)
```

mutate *Create, modify, and delete columns*

Description

`mutate()` adds new variables and preserves existing ones; `transmute()` adds new variables and drops existing ones. 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, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output.
`.keep`	**[Experimental]** This is an experimental argument that 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`	**[Experimental]** <`tidy-select`> Optionally, control where new columns should appear (the default is to add to the right hand side). See `relocate()` for more details.

Value

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

- Rows are not affected.
- Existing columns will be preserved according to the `.keep` argument. New columns will be placed according to the `.before` and `.after` arguments. If `.keep = "none"` (as in `transmute()`), the output order is determined only by `...`, not the order of existing columns.
- Columns given value `NULL` will be removed
- Groups will be recomputed if a grouping variable is mutated.
- Data frame attributes are preserved.

Useful mutate functions

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

Grouped tibbles

Because mutating expressions are computed within groups, they may yield different results on grouped tibbles. This will be the case as soon as an aggregating, lagging, or ranking function is involved. Compare this ungrouped mutate:

```
starwars %>%
  select(name, mass, species) %>%
  mutate(mass_norm = mass / mean(mass, na.rm = TRUE))
```

With the grouped equivalent:

```
starwars %>%
  select(name, mass, species) %>%
  group_by(species) %>%
  mutate(mass_norm = mass / mean(mass, na.rm = TRUE))
```

The former normalises `mass` by the global average whereas the latter normalises by the averages within species levels.

Methods

These function are **generic**s, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

Methods available in currently loaded packages:

- `mutate()`: no methods found.
- `transmute()`: no methods found.

See Also

Other single table verbs: arrange(), filter(), rename(), select(), slice(), summarise()

Examples

```
# Newly created variables are available immediately
starwars %>%
 select(name, mass) %>%
 mutate(
  mass2 = mass * 2,
  mass2_squared = mass2 * mass2
)

# As well as adding new variables, you can use mutate() to
# remove variables and modify existing variables.
starwars %>%
 select(name, height, mass, homeworld) %>%
 mutate(
  mass = NULL,
  height = height * 0.0328084 # convert to feet
)

# Use across() with mutate() to apply a transformation
# to multiple columns in a tibble.
starwars %>%
 select(name, homeworld, species) %>%
 mutate(across(!name, as.factor))
# see more in ?across

# Window functions are useful for grouped mutates:
starwars %>%
 select(name, mass, homeworld) %>%
 group_by(homeworld) %>%
 mutate(rank = min_rank(desc(mass)))
# see `vignette("window-functions")` for more details

# By default, new columns are placed on the far right.
# Experimental: you can override with `.before` or `.after`
df <- tibble(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.
# Experimental: You can override with `.keep`
df <- tibble(x = 1, y = 2, a = "a", b = "b")
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()

# Grouping --------------------------------------
# The mutate operation may yield different results on grouped
# tibbles because the expressions are computed within groups.
# The following normalises `mass` by the global average:
starwars %>%
```

```
  select(name, mass, species) %>%
  mutate(mass_norm = mass / mean(mass, na.rm = TRUE))

# Whereas this normalises `mass` by the averages within species
# levels:
starwars %>%
  select(name, mass, species) %>%
  group_by(species) %>%
  mutate(mass_norm = mass / mean(mass, na.rm = TRUE))

# Indirection ----------------------------------------
# Refer to column names stored as strings with the `.data` pronoun:
vars <- c("mass", "height")
mutate(starwars, prod = .data[[vars[[1]]]] * .data[[vars[[2]]]])
# Learn more in ?dplyr_data_masking
```

`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,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = FALSE
)

## S3 method for class 'data.frame'
inner_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = FALSE,
```

```
  na_matches = c("na", "never")
)

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

## S3 method for class 'data.frame'
left_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = FALSE,
  na_matches = c("na", "never")
)

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

## S3 method for class 'data.frame'
right_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = FALSE,
  na_matches = c("na", "never")
)

full_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
```

```
  suffix = c(".x", ".y"),
  ...,
  keep = FALSE
)

## S3 method for class 'data.frame'
full_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = FALSE,
  na_matches = c("na", "never")
)
```

Arguments

x, y	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
by	A character vector of variables to join by. If NULL, the default, *_join() will perform a natural join, using all variables in common across x and y. A message lists the variables so that you can check they're correct; suppress the message by supplying by explicitly. 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().
copy	If x and y are not from the same data source, and copy is TRUE, then y will be copied into the same src as x. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
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.
...	Other parameters passed onto methods.
keep	Should the join keys from both x and y be preserved in the output?
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).

Value

An object of the same type as x. 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`), `suffixes` are added to disambiguate.
- Output columns included in `by` are coerced to common type across `x` and `y`.
- Groups are taken from `x`.

Methods

These functions are **generic**s, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

Methods available in currently loaded packages:

- `inner_join()`: no methods found.
- `left_join()`: no methods found.
- `right_join()`: no methods found.
- `full_join()`: no methods found.

See Also

Other joins: `filter-joins`, `nest_join()`

Examples

```
band_members %>% inner_join(band_instruments)
band_members %>% left_join(band_instruments)
band_members %>% right_join(band_instruments)
band_members %>% full_join(band_instruments)

# To suppress the message about joining variables, supply `by`
band_members %>% inner_join(band_instruments, by = "name")
# This is good practice in production code

# Use a named `by` if the join variables have different names
band_members %>% full_join(band_instruments2, by = c("name" = "artist"))
# By default, the join keys from `x` and `y` are coalesced in the output; use
# `keep = TRUE` to keep the join keys from both `x` and `y`
band_members %>%
  full_join(band_instruments2, by = c("name" = "artist"), keep = TRUE)

# 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 <- tibble(x = 1:3)
df2 <- tibble(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")
```

`mutate_all` *Mutate multiple columns*

Description

[Superseded]

Scoped verbs (_if, _at, _all) have been superseded by the use of `across()` in an existing verb. See `vignette("colwise")` for details.

The scoped variants of `mutate()` and `transmute()` make it easy to apply the same transformation to multiple variables. There are three variants:

- _all affects every variable
- _at affects variables selected with a character vector or vars()
- _if affects variables selected with a predicate function:

Usage

```
mutate_all(.tbl, .funs, ...)

mutate_if(.tbl, .predicate, .funs, ...)

mutate_at(.tbl, .vars, .funs, ..., .cols = NULL)

transmute_all(.tbl, .funs, ...)

transmute_if(.tbl, .predicate, .funs, ...)

transmute_at(.tbl, .vars, .funs, ..., .cols = NULL)
```

Arguments

`.tbl`	A `tbl` object.
`.funs`	A function `fun`, a quosure style lambda `~ fun(.)` or a list of either form.
`...`	Additional arguments for the function calls in `.funs`. These are evaluated only once, with tidy dots support.
`.predicate`	A predicate function to be applied to the columns or a logical vector. The variables for which `.predicate` is or returns `TRUE` are selected. This argument is passed to `rlang::as_function()` and thus supports quosure-style lambda functions and strings representing function names.
`.vars`	A list of columns generated by `vars()`, a character vector of column names, a numeric vector of column positions, or `NULL`.
`.cols`	This argument has been renamed to `.vars` to fit dplyr's terminology and is deprecated.

Value

A data frame. By default, the newly created columns have the shortest names needed to uniquely identify the output. To force inclusion of a name, even when not needed, name the input (see examples for details).

Grouping variables

If applied on a grouped tibble, these operations are *not* applied to the grouping variables. The behaviour depends on whether the selection is **implicit** (`all` and `if` selections) or **explicit** (`at` selections).

- Grouping variables covered by explicit selections in `mutate_at()` and `transmute_at()` are always an error. Add `-group_cols()` to the `vars()` selection to avoid this:

  ```
  data %>% mutate_at(vars(-group_cols(), ...), myoperation)
  ```

  Or remove `group_vars()` from the character vector of column names:

  ```
  nms <- setdiff(nms, group_vars(data))
  data %>% mutate_at(vars, myoperation)
  ```

- Grouping variables covered by implicit selections are ignored by `mutate_all()`, `transmute_all()`, `mutate_if()`, and `transmute_if()`.

Naming

The names of the new columns are derived from the names of the input variables and the names of the functions.

- if there is only one unnamed function (i.e. if `.funs` is an unnamed list of length one), the names of the input variables are used to name the new columns;
- for _at functions, if there is only one unnamed variable (i.e., if `.vars` is of the form `vars(a_single_column)`) and `.funs` has length greater than one, the names of the functions are used to name the new columns;
- otherwise, the new names are created by concatenating the names of the input variables and the names of the functions, separated with an underscore `"_"`.

The `.funs` argument can be a named or unnamed list. If a function is unnamed and the name cannot be derived automatically, a name of the form "fn#" is used. Similarly, `vars()` accepts named and unnamed arguments. If a variable in `.vars` is named, a new column by that name will be created.

Name collisions in the new columns are disambiguated using a unique suffix.

Life cycle

The functions are maturing, because the naming scheme and the disambiguation algorithm are subject to change in dplyr 0.9.0.

See Also

The other scoped verbs, `vars()`

Examples

```
iris <- as_tibble(iris)

# All variants can be passed functions and additional arguments,
# purrr-style. The _at() variants directly support strings. Here
# we'll scale the variables `height` and `mass`:
scale2 <- function(x, na.rm = FALSE) (x - mean(x, na.rm = na.rm)) / sd(x, na.rm)
starwars %>% mutate_at(c("height", "mass"), scale2)
# ->
starwars %>% mutate(across(c("height", "mass"), scale2))

# You can pass additional arguments to the function:
starwars %>% mutate_at(c("height", "mass"), scale2, na.rm = TRUE)
starwars %>% mutate_at(c("height", "mass"), ~scale2(., na.rm = TRUE))
# ->
starwars %>% mutate(across(c("height", "mass"), ~ scale2(.x, na.rm = TRUE)))

# You can also supply selection helpers to _at() functions but you have
# to quote them with vars():
iris %>% mutate_at(vars(matches("Sepal")), log)
iris %>% mutate(across(matches("Sepal"), log))

# The _if() variants apply a predicate function (a function that
# returns TRUE or FALSE) to determine the relevant subset of
# columns. Here we divide all the numeric columns by 100:
starwars %>% mutate_if(is.numeric, scale2, na.rm = TRUE)
starwars %>% mutate(across(where(is.numeric), ~ scale2(.x, na.rm = TRUE)))

# mutate_if() is particularly useful for transforming variables from
# one type to another
iris %>% mutate_if(is.factor, as.character)
iris %>% mutate_if(is.double, as.integer)
# ->
iris %>% mutate(across(where(is.factor), as.character))
iris %>% mutate(across(where(is.double), as.integer))

# Multiple transformations ----------------------------------------

# If you want to apply multiple transformations, pass a list of
# functions. When there are multiple functions, they create new
# variables instead of modifying the variables in place:
iris %>% mutate_if(is.numeric, list(scale2, log))
iris %>% mutate_if(is.numeric, list(~scale2(.), ~log(.)))
iris %>% mutate_if(is.numeric, list(scale = scale2, log = log))
# ->
iris %>%
  as_tibble() %>%
  mutate(across(where(is.numeric), list(scale = scale2, log = log)))

# When there's only one function in the list, it modifies existing
# variables in place. Give it a name to instead create new variables:
iris %>% mutate_if(is.numeric, list(scale2))
iris %>% mutate_if(is.numeric, list(scale = scale2))
```

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	Vector to modify
y	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 with a specified value.

tidyr::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
starwars %>%
  select(name, eye_color) %>%
  mutate(eye_color = na_if(eye_color, "unknown"))

# na_if() can also be used with mutate() and across()
# to mutate multiple columns
starwars %>%
   mutate(across(where(is.character), ~na_if(., "unknown")))
```

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_join *Nest join*

Description

nest_join() returns all rows and columns in x with a new nested-df column that contains all matches from y. When there is no match, the list column is a 0-row tibble.

Usage

```
nest_join(x, y, by = NULL, copy = FALSE, keep = FALSE, name = NULL, ...)

## S3 method for class 'data.frame'
nest_join(x, y, by = NULL, copy = FALSE, keep = FALSE, name = NULL, ...)
```

Arguments

x, y	A pair of data frames, data frame extensions (e.g. a tibble), or lazy data frames (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
by	A character vector of variables to join by. If NULL, the default, *_join() will perform a natural join, using all variables in common across x and y. A message lists the variables so that you can check they're correct; suppress the message by supplying by explicitly. 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().

copy	If x and y are not from the same data source, and copy is TRUE, then y will be copied into the same src as x. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
keep	Should the join keys from both x and y be preserved in the output?
name	The name of the list column nesting joins create. If NULL the name of y is used.
...	Other parameters passed onto methods.

Details

In some sense, a nest_join() is the most fundamental join since you can recreate the other joins from it:

- inner_join() is a nest_join() plus tidyr::unnest()
- left_join() nest_join() plus unnest(.drop = FALSE).
- semi_join() is a nest_join() plus a filter() where you check that every element of data has at least one row,
- anti_join() is a nest_join() plus a filter() where you check every element has zero rows.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

See Also

Other joins: filter-joins, mutate-joins

Examples

```
band_members %>% nest_join(band_instruments)
```

---

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 *Efficiently count the number of unique values in a set of vectors*

Description

This is a faster and more concise equivalent of length(unique(x))

Usage

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

Arguments

...	vectors of values
na.rm	if TRUE missing values don't count

Examples

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

---

order_by *A helper function for ordering window function output*

Description

This function makes it possible to control the ordering of window functions in R that don't have a specific ordering parameter. When translated to SQL it will modify the order clause of the OVER function.

Usage

```
order_by(order_by, call)
```

Arguments

order_by	a vector to order_by
call	a function call to a window function, where the first argument is the vector being operated on

Details

This function works by changing the call to instead call with_order() with the appropriate arguments.

Examples

```
order_by(10:1, cumsum(1:10))
x <- 10:1
y <- 1:10
order_by(x, cumsum(y))

df <- data.frame(year = 2000:2005, value = (0:5) ^ 2)
scrambled <- df[sample(nrow(df)), ]

wrong <- mutate(scrambled, running = cumsum(value))
arrange(wrong, year)

right <- mutate(scrambled, running = order_by(year, cumsum(value)))
arrange(right, year)
```

`pull` *Extract a single column*

Description

`pull()` is similar to `$`. It's mostly useful because it looks a little nicer in pipes, it also works with remote data frames, and it can optionally name the output.

Usage

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

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`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). This argument is taken by expression and supports quasiquotation (you can unquote column names and column locations).
`name`	An optional parameter that specifies the column to be used as names for a named vector. Specified in a similar manner as `var`.
`...`	For use by methods.

Value

A vector the same size as `.data`.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

Examples

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

# Also works for remote sources
if (requireNamespace("dbplyr", quietly = TRUE)) {
df <- dbplyr::memdb_frame(x = 1:10, y = 10:1, .name = "pull-ex")
df %>%
  mutate(z = x * y) %>%
```

```
  pull()
}

# Pull a named vector
starwars %>% pull(height, name)
```

`ranking` *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, and are provided mainly as a convenience when converting between R and SQL. All ranking functions map smallest inputs to smallest outputs. Use `desc()` to reverse the direction.

Usage

```
row_number(x)

ntile(x = row_number(), n)

min_rank(x)

dense_rank(x)

percent_rank(x)

cume_dist(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`	number of groups to split up into.

Details

- `row_number()`: equivalent to `rank(ties.method = "first")`
- `min_rank()`: equivalent to `rank(ties.method = "min")`
- `dense_rank()`: like `min_rank()`, but with no gaps between ranks
- `percent_rank()`: a number between 0 and 1 computed by rescaling `min_rank` to [0, 1]
- `cume_dist()`: a cumulative distribution function. Proportion of all values less than or equal to the current rank.
- `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.

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))
```

`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: dplyr 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. See the forcats package for more tools for working with factors and their levels.

[Questioning] `recode()` is questioning because the arguments are in the wrong order. We have `new <-old`, `mutate(df,new = old)`, and `rename(df,new = old)` but `recode(x,old = new)`. We don't yet know how to fix this problem, but it's likely to involve creating a new function then retiring or deprecating `recode()`.

Usage

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

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

Arguments

`.x`	A vector to modify
`...`	<dynamic-dots> 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	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.

tidyr::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. For more information, see
# https://adv-r.hadley.nz/vectors-chap.html#missing-values
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)

# Use a named character vector for unquote splicing with !!!
level_key <- c(a = "apple", b = "banana", c = "carrot")
recode(char_vec, !!!level_key)

# 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")

# Use a named character vector to recode factors with unquote splicing.
level_key <- c(a = "apple", b = "banana", c = "carrot")
recode_factor(char_vec, !!!level_key)
```

`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, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<tidy-select> Columns to move.
`.before, .after`	<tidy-select> Destination of columns selected by `...`. Supplying neither will move columns to the left-hand side; specifying both is 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.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

Examples

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

# relocated columns can change name
df %>% relocate(ff = f)

# 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 <- tibble(a = 1, b = "a", c = 1, d = "a")
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, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
...	For rename(): <tidy-select> Use 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	<tidy-select> Columns to rename; defaults to all columns.

Value

An object of the same type as .data. The output has 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.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

See Also

Other single table verbs: arrange(), filter(), mutate(), select(), slice(), summarise()

Examples

```
iris <- as_tibble(iris) # so it prints a little nicer
rename(iris, petal_length = Petal.Length)

rename_with(iris, toupper)
rename_with(iris, toupper, starts_with("Petal"))
rename_with(iris, ~ tolower(gsub(".", "_", .x, fixed = TRUE)))
```

---

rows *Manipulate individual rows*

Description

[Experimental]

These functions provide a framework for modifying rows in a table using a second table of data. The two tables are matched by a set of key variables whose values must uniquely identify each row. The functions are inspired by SQL's `INSERT`, `UPDATE`, and `DELETE`, and can optionally modify `in_place` for selected backends.

- `rows_insert()` adds new rows (like `INSERT`); key values in `y` must not occur in `x`.
- `rows_update()` modifies existing rows (like `UPDATE`); key values in `y` must occur in `x`.
- `rows_patch()` works like `rows_update()` but only overwrites `NA` values.
- `rows_upsert()` inserts or updates depending on whether or not the key value in `y` already exists in `x`.
- `rows_delete()` deletes rows (like `DELETE`); key values in `y` must exist in `x`.

Usage

```
rows_insert(x, y, by = NULL, ..., copy = FALSE, in_place = FALSE)

rows_update(x, y, by = NULL, ..., copy = FALSE, in_place = FALSE)

rows_patch(x, y, by = NULL, ..., copy = FALSE, in_place = FALSE)

rows_upsert(x, y, by = NULL, ..., copy = FALSE, in_place = FALSE)

rows_delete(x, y, by = NULL, ..., copy = FALSE, in_place = FALSE)
```

Arguments

`x, y`	A pair of data frames or data frame extensions (e.g. a tibble). `y` must have the same columns of `x` or a subset.
`by`	An unnamed character vector giving the key columns. The key values must uniquely identify each row (i.e. each combination of key values occurs at most once), and the key columns must exist in both `x` and `y`. By default, we use the first column in `y`, since the first column is a reasonable place to put an identifier variable.
`...`	Other parameters passed onto methods.
`copy`	If `x` and `y` are not from the same data source, and `copy` is `TRUE`, then `y` will be copied into the same src as `x`. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
`in_place`	Should `x` be modified in place? This argument is only relevant for mutable backends (e.g. databases, data.tables). When `TRUE`, a modified version of `x` is returned invisibly; when `FALSE`, a new object representing the resulting changes is returned.

Value

An object of the same type as `x`. The order of the rows and columns of `x` is preserved as much as possible. The output has the following properties:

- `rows_update()` preserves rows as is; `rows_insert()` and `rows_upsert()` return all existing rows and potentially new rows; `rows_delete()` returns a subset of the rows.

- Columns are not added, removed, or relocated, though the data may be updated.
- Groups are taken from `x`.
- Data frame attributes are taken from `x`.

If `in_place = TRUE`, the result will be returned invisibly.

Examples

```
data <- tibble(a = 1:3, b = letters[c(1:2, NA)], c = 0.5 + 0:2)
data

# Insert
rows_insert(data, tibble(a = 4, b = "z"))
try(rows_insert(data, tibble(a = 3, b = "z")))

# Update
rows_update(data, tibble(a = 2:3, b = "z"))
rows_update(data, tibble(b = "z", a = 2:3), by = "a")

# Variants: patch and upsert
rows_patch(data, tibble(a = 2:3, b = "z"))
rows_upsert(data, tibble(a = 2:4, b = "z"))

# Delete and truncate
rows_delete(data, tibble(a = 2:3))
rows_delete(data, tibble(a = 2:3, b = "b"))
try(rows_delete(data, tibble(a = 2:3, b = "b"), by = c("a", "b")))
```

---

rowwise *Group input by rows*

Description

`rowwise()` allows you to compute on a data frame a row-at-a-time. This is most useful when a vectorised function doesn't exist.

Most dplyr verbs preserve row-wise grouping. The exception is `summarise()`, which return a grouped_df. You can explicitly ungroup with `ungroup()` or `as_tibble()`, or convert to a grouped_df with `group_by()`.

Usage

```
rowwise(data, ...)
```

Arguments

`data`	Input data frame.
`...`	<`tidy-select`> Variables to be preserved when calling `summarise()`. This is typically a set of variables whose combination uniquely identify each row. **NB**: unlike `group_by()` you can not create new variables here but instead you can select multiple variables with (e.g.) `everything()`.

Value

A row-wise data frame with class `rowwise_df`. Note that a `rowwise_df` is implicitly grouped by row, but is not a `grouped_df`.

List-columns

Because a rowwise has exactly one row per group it offers a small convenience for working with list-columns. Normally, `summarise()` and `mutate()` extract a groups worth of data with `[`. But when you index a list in this way, you get back another list. When you're working with a `rowwise` tibble, then dplyr will use `[[` instead of `[` to make your life a little easier.

See Also

`nest_by()` for a convenient way of creating rowwise data frames with nested data.

Examples

```
df <- tibble(x = runif(6), y = runif(6), z = runif(6))
# Compute the mean of x, y, z in each row
df %>% rowwise() %>% mutate(m = mean(c(x, y, z)))
# use c_across() to more easily select many variables
df %>% rowwise() %>% mutate(m = mean(c_across(x:z)))

# Compute the minimum of x and y in each row
df %>% rowwise() %>% mutate(m = min(c(x, y, z)))
# In this case you can use an existing vectorised function:
df %>% mutate(m = pmin(x, y, z))
# Where these functions exist they'll be much faster than rowwise
# so be on the lookout for them.

# rowwise() is also useful when doing simulations
params <- tribble(
 ~sim, ~n, ~mean, ~sd,
    1,  1,     1,   1,
    2,  2,     2,   4,
    3,  3,    -1,   2
)
# Here I supply variables to preserve after the summary
params %>%
  rowwise(sim) %>%
  summarise(z = rnorm(n, mean, sd))

# If you want one row per simulation, put the results in a list()
params %>%
  rowwise(sim) %>%
  summarise(z = list(rnorm(n, mean, sd)))
```

---

scoped *Operate on a selection of variables*

Description

[Superseded]

Scoped verbs (_if, _at, _all) have been superseded by the use of `across()` in an existing verb. See `vignette("colwise")` for details.

The variants suffixed with _if, _at or _all apply an expression (sometimes several) to all variables within a specified subset. This subset can contain all variables (_all variants), a `vars()` selection (_at variants), or variables selected with a predicate (_if variants).

The verbs with scoped variants are:

- `mutate()`, `transmute()` and `summarise()`. See `summarise_all()`.
- `filter()`. See `filter_all()`.
- `group_by()`. See `group_by_all()`.
- `rename()` and `select()`. See `select_all()`.
- `arrange()`. See `arrange_all()`

There are three kinds of scoped variants. They differ in the scope of the variable selection on which operations are applied:

- Verbs suffixed with _all() apply an operation on all variables.
- Verbs suffixed with _at() apply an operation on a subset of variables specified with the quoting function `vars()`. This quoting function accepts `tidyselect::vars_select()` helpers like `starts_with()`. Instead of a `vars()` selection, you can also supply an integerish vector of column positions or a character vector of column names.
- Verbs suffixed with _if() apply an operation on the subset of variables for which a predicate function returns `TRUE`. Instead of a predicate function, you can also supply a logical vector.

Arguments

`.tbl`	A `tbl` object.
`.funs`	A function `fun`, a quosure style lambda `~ fun(.)` or a list of either form.
`.vars`	A list of columns generated by `vars()`, a character vector of column names, a numeric vector of column positions, or `NULL`.
`.predicate`	A predicate function to be applied to the columns or a logical vector. The variables for which `.predicate` is or returns `TRUE` are selected. This argument is passed to `rlang::as_function()` and thus supports quosure-style lambda functions and strings representing function names.
`...`	Additional arguments for the function calls in `.funs`. These are evaluated only once, with tidy dots support.

Grouping variables

Most of these operations also apply on the grouping variables when they are part of the selection. This includes:

- `arrange_all()`, `arrange_at()`, and `arrange_if()`
- `distinct_all()`, `distinct_at()`, and `distinct_if()`
- `filter_all()`, `filter_at()`, and `filter_if()`
- `group_by_all()`, `group_by_at()`, and `group_by_if()`

- `select_all()`, `select_at()`, and `select_if()`

This is not the case for summarising and mutating variants where operations are *not* applied on grouping variables. The behaviour depends on whether the selection is **implicit** (`all` and `if` selections) or **explicit** (`at` selections). Grouping variables covered by explicit selections (with `summarise_at()`, `mutate_at()`, and `transmute_at()`) are always an error. For implicit selections, the grouping variables are always ignored. In this case, the level of verbosity depends on the kind of operation:

- Summarising operations (`summarise_all()` and `summarise_if()`) ignore grouping variables silently because it is obvious that operations are not applied on grouping variables.
- On the other hand it isn't as obvious in the case of mutating operations (`mutate_all()`, `mutate_if()`, `transmute_all()`, and `transmute_if()`). For this reason, they issue a message indicating which grouping variables are ignored.

`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.

Overview of selection features:

Tidyverse 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`.

Usage

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

Arguments

.data	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
...	<tidy-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.

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.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

Examples

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

The selection language can be used in functions like dplyr::select() or tidyr::pivot_longer(). Let's first attach the tidyverse:

```
library(tidyverse)

# For better printing
iris <- as_tibble(iris)
```

Select variables by name:

```
starwars %>% select(height)
#> # A tibble: 87 x 1
#>   height
#>    <int>
#> 1    172
#> 2    167
#> 3     96
#> 4    202
#> # ... with 83 more rows
```

```
iris %>% pivot_longer(Sepal.Length)
#> # A tibble: 150 x 6
#>   Sepal.Width Petal.Length Petal.Width Species name         value
#>         <dbl>        <dbl>       <dbl> <fct>   <chr>        <dbl>
#> 1         3.5          1.4         0.2 setosa  Sepal.Length   5.1
#> 2         3            1.4         0.2 setosa  Sepal.Length   4.9
#> 3         3.2          1.3         0.2 setosa  Sepal.Length   4.7
#> 4         3.1          1.5         0.2 setosa  Sepal.Length   4.6
#> # ... with 146 more rows
```

Select multiple variables by separating them with commas. Note how the order of columns is determined by the order of inputs:

```
starwars %>% select(homeworld, height, mass)
#> # A tibble: 87 x 3
#>   homeworld height  mass
#>   <chr>      <int> <dbl>
#> 1 Tatooine     172    77
#> 2 Tatooine     167    75
#> 3 Naboo         96    32
#> 4 Tatooine     202   136
#> # ... with 83 more rows
```

Functions like `tidyr::pivot_longer()` don't take variables with dots. In this case use `c()` to select multiple variables:

```
iris %>% pivot_longer(c(Sepal.Length, Petal.Length))
#> # A tibble: 300 x 5
#>   Sepal.Width Petal.Width Species name         value
#>         <dbl>       <dbl> <fct>   <chr>        <dbl>
#> 1         3.5         0.2 setosa  Sepal.Length   5.1
#> 2         3.5         0.2 setosa  Petal.Length   1.4
#> 3         3           0.2 setosa  Sepal.Length   4.9
#> 4         3           0.2 setosa  Petal.Length   1.4
#> # ... with 296 more rows
```

Operators::

The `:` operator selects a range of consecutive variables:

```
starwars %>% select(name:mass)
#> # A tibble: 87 x 3
#>   name           height  mass
#>   <chr>           <int> <dbl>
#> 1 Luke Skywalker    172    77
#> 2 C-3PO             167    75
#> 3 R2-D2              96    32
#> 4 Darth Vader       202   136
#> # ... with 83 more rows
```

The `!` operator negates a selection:

```
starwars %>% select(!(name:mass))
#> # A tibble: 87 x 11
```

```
#>  hair_color skin_color  eye_color birth_year sex  gender   homeworld species
#>   <chr>      <chr>       <chr>          <dbl> <chr> <chr>    <chr>     <chr>
#> 1 blond      fair        blue            19   male  masculine Tatooine  Human
#> 2 <NA>       gold        yellow         112   none  masculine Tatooine  Droid
#> 3 <NA>       white, blue red             33   none  masculine Naboo     Droid
#> 4 none       white       yellow          41.9 male  masculine Tatooine  Human
#> # ... with 83 more rows, and 3 more variables: films <list>, vehicles <list>,
#> #   starships <list>

iris %>% select(!c(Sepal.Length, Petal.Length))
#> # A tibble: 150 x 3
#>   Sepal.Width Petal.Width Species
#>         <dbl>       <dbl> <fct>
#> 1         3.5         0.2 setosa
#> 2         3           0.2 setosa
#> 3         3.2         0.2 setosa
#> 4         3.1         0.2 setosa
#> # ... with 146 more rows

iris %>% select(!ends_with("Width"))
#> # A tibble: 150 x 3
#>   Sepal.Length Petal.Length Species
#>          <dbl>        <dbl> <fct>
#> 1          5.1          1.4 setosa
#> 2          4.9          1.4 setosa
#> 3          4.7          1.3 setosa
#> 4          4.6          1.5 setosa
#> # ... with 146 more rows
```

& and | take the intersection or the union of two selections:

```
iris %>% select(starts_with("Petal") & ends_with("Width"))
#> # A tibble: 150 x 1
#>   Petal.Width
#>         <dbl>
#> 1         0.2
#> 2         0.2
#> 3         0.2
#> 4         0.2
#> # ... with 146 more rows

iris %>% select(starts_with("Petal") | ends_with("Width"))
#> # A tibble: 150 x 3
#>   Petal.Length Petal.Width Sepal.Width
#>          <dbl>       <dbl>       <dbl>
#> 1          1.4         0.2         3.5
#> 2          1.4         0.2         3
#> 3          1.3         0.2         3.2
#> 4          1.5         0.2         3.1
#> # ... with 146 more rows
```

To take the difference between two selections, combine the & and ! operators:

```
iris %>% select(starts_with("Petal") & !ends_with("Width"))
#> # A tibble: 150 x 1
```

```
#>   Petal.Length
#>          <dbl>
#> 1          1.4
#> 2          1.4
#> 3          1.3
#> 4          1.5
#> # ... with 146 more rows
```

See Also

Other single table verbs: `arrange()`, `filter()`, `mutate()`, `rename()`, `slice()`, `summarise()`

`setops` *Set operations*

Description

These functions override the set functions provided in base to make them generic so that efficient versions for data frames and other tables can be provided. The default methods call the base versions. Beware that `intersect()`, `union()` and `setdiff()` remove duplicates.

Usage

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

Arguments

`x, y`	objects to perform set function on (ignoring order)
`...`	other arguments passed on to methods

Examples

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

intersect(first, second)
union(first, second)
setdiff(first, second)
setdiff(second, first)

union_all(first, second)
setequal(mtcars, mtcars[32:1, ])

# Handling of duplicates:
a <- data.frame(column = c(1:10, 10))
b <- data.frame(column = c(1:5, 5))

# intersection is 1 to 5, duplicates removed (5)
intersect(a, b)

# union is 1 to 10, duplicates removed (5 and 10)
union(a, b)
```

```
# set difference, duplicates removed (10)
setdiff(a, b)

# union all does not remove duplicates
union_all(a, b)
```

`slice` *Subset rows using their positions*

Description

`slice()` lets you index rows by their (integer) locations. It allows you to select, remove, and duplicate rows. It is accompanied by a number of helpers for common use cases:

- `slice_head()` and `slice_tail()` select the first or last rows.
- `slice_sample()` randomly selects rows.
- `slice_min()` and `slice_max()` select rows with highest or lowest values of a variable.

If `.data` is a grouped_df, the operation will be performed on each group, so that (e.g.) `slice_head(df,n = 5)` will select the first five rows in each group.

Usage

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

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, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	For `slice()`: <data-masking> Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. For `slice_helpers()`, these arguments are passed on to methods.
`.preserve`	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.

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	Variable or function of variables to order by.
with_ties	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	Should sampling be performed with (TRUE) or without (FALSE, the default) replacement.

Details

Slice does not work with relational databases because they have no intrinsic notion of row order. If you want to perform the equivalent operation, use filter() and row_number().

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.

Methods

These function are **generic**s, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

Methods available in currently loaded packages:

- slice(): no methods found.
- slice_head(): no methods found.
- slice_tail(): no methods found.
- slice_min(): no methods found.
- slice_max(): no methods found.
- slice_sample(): no methods found.

See Also

Other single table verbs: arrange(), filter(), mutate(), rename(), select(), summarise()

Examples

```
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)

# Rows with minimum and maximum values of a variable
mtcars %>% slice_min(mpg, n = 5)
mtcars %>% slice_max(mpg, n = 5)

# slice_min() and slice_max() may return more rows than requested
# in the presence of ties. Use with_ties = FALSE to suppress
mtcars %>% slice_min(cyl, n = 1)
mtcars %>% slice_min(cyl, n = 1, with_ties = FALSE)

# slice_sample() allows you to random select with or without replacement
mtcars %>% slice_sample(n = 5)
mtcars %>% slice_sample(n = 5, replace = TRUE)

# you can optionally weight by a variable - this code weights by the
# physical weight of the cars, so heavy cars are more likely to get
# selected
mtcars %>% slice_sample(weight_by = wt, n = 5)

# Group wise operation --------------------------------------
df <- tibble(
  group = rep(c("a", "b", "c"), c(1, 2, 4)),
  x = runif(7)
)

# All slice helpers operate per group, silently truncating to the group
# size, so the following code works without error
df %>% group_by(group) %>% slice_head(n = 2)

# When specifying the proportion of rows to include non-integer sizes
# are rounded down, so group a gets 0 rows
df %>% group_by(group) %>% slice_head(prop = 0.5)

# Filter equivalents ----------------------------------------------
# slice() expressions can often be written to use `filter()` and
# `row_number()`, which can also be translated to SQL. For many databases,
# you'll need to supply an explicit variable to use to compute the row number.
filter(mtcars, row_number() == 1L)
filter(mtcars, row_number() == n())
filter(mtcars, between(row_number(), 5, n()))
```

`sql` *SQL escaping.*

Description

These functions are critical when writing functions that translate R functions to sql functions. Typically a conversion function should escape all its inputs and return an sql object.

Usage

```
sql(...)
```

Arguments

...	Character vectors that will be combined into a single SQL expression.

---

starwars *Starwars characters*

Description

The original data, from SWAPI, the Star Wars API, https://swapi.dev/, has been revised to reflect additional research into gender and sex determinations of characters.

Usage

```
starwars
```

Format

A tibble with 87 rows and 14 variables:

name Name of the character

height Height (cm)

mass Weight (kg)

hair_color,skin_color,eye_color Hair, skin, and eye colors

birth_year Year born (BBY = Before Battle of Yavin)

sex The biological sex of the character, namely male, female, hermaphroditic, or none (as in the case for Droids).

gender The gender role or gender identity of the character as determined by their personality or the way they were programmed (as in the case for Droids).

homeworld Name of homeworld

species Name of species

films List of films the character appeared in

vehicles List of vehicles the character has piloted

starships List of starships the character has piloted

Examples

```
starwars
```

`storms` *Storm tracks data*

Description

This data is a subset of the NOAA Atlantic hurricane database best track data, https://www.nhc.noaa.gov/data/#hurdat. The data includes the positions and attributes of 198 tropical storms, measured every six hours during the lifetime of a storm.

Usage

```
storms
```

Format

A tibble with 10,010 observations and 13 variables:

name Storm Name

year,month,day Date of report

hour Hour of report (in UTC)

lat,long Location of storm center

status Storm classification (Tropical Depression, Tropical Storm, or Hurricane)

category Saffir-Simpson storm category (estimated from wind speed. -1 = Tropical Depression, 0 = Tropical Storm)

wind storm's maximum sustained wind speed (in knots)

pressure Air pressure at the storm's center (in millibars)

ts_diameter Diameter of the area experiencing tropical storm strength winds (34 knots or above)

hu_diameter Diameter of the area experiencing hurricane strength winds (64 knots or above)

See Also

The script to create the storms data set: https://github.com/tidyverse/dplyr/blob/master/data-raw/storms.R

Examples

```
storms
```

summarise *Summarise each group to fewer rows*

Description

`summarise()` creates a new data frame. It will have one (or more) rows for each combination of grouping variables; if there are no grouping variables, the output will have a single row summarising all observations in the input. It will contain one column for each grouping variable and one column for each of the summary statistics that you have specified.

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

Usage

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

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

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<data-masking> 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()`. • A data frame, to add multiple columns from a single expression.
`.groups`	**[Experimental]** Grouping structure of the result. • "drop_last": dropping the last level of grouping. This was the only supported option before version 1.0.0. • "drop": All levels of grouping are dropped. • "keep": Same grouping structure as `.data`. • "rowwise": Each row is its own group. 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 "dplyr.summarise.inform" is set to `FALSE`, or when `summarise()` is called from a function in a package.

Value

An object *usually* of the same type as `.data`.

- The rows come from the underlying `group_keys()`.
- The columns are a combination of the grouping keys and the summary expressions that you provide.

- The grouping structure is controlled by the .groups= argument, the output may be another grouped_df, a tibble or a rowwise data frame.
- Data frame attributes are **not** preserved, because `summarise()` fundamentally creates a new data frame.

Useful functions

- Center: `mean()`, `median()`
- Spread: `sd()`, `IQR()`, `mad()`
- Range: `min()`, `max()`, `quantile()`
- Position: `first()`, `last()`, `nth()`,
- Count: `n()`, `n_distinct()`
- Logical: `any()`, `all()`

Backend variations

The data frame backend supports creating a variable and using it in the same summary. This means that previously created summary variables can be further transformed or combined within the summary, as in `mutate()`. However, it also means that summary variables with the same names as previous variables overwrite them, making those variables unavailable to later summary variables.

This behaviour may not be supported in other backends. To avoid unexpected results, consider using new names for your summary variables, especially when creating multiple summaries.

Methods

This function is a **generic**, which means that packages can provide implementations (methods) for other classes. See the documentation of individual methods for extra arguments and differences in behaviour.

The following methods are currently available in loaded packages: no methods found.

See Also

Other single table verbs: `arrange()`, `filter()`, `mutate()`, `rename()`, `select()`, `slice()`

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())

# dplyr 1.0.0 allows to 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) {
  tibble(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()

# BEWARE: reusing variables may lead to unexpected results
mtcars %>%
  group_by(cyl) %>%
  summarise(disp = mean(disp), sd = sd(disp))

# Refer to column names stored as strings with the `.data` pronoun:
var <- "mass"
summarise(starwars, avg = mean(.data[[var]], na.rm = TRUE))
# Learn more in ?dplyr_data_masking
```

`summarise_all` *Summarise multiple columns*

Description

[Superseded]

Scoped verbs (_if, _at, _all) have been superseded by the use of `across()` in an existing verb. See `vignette("colwise")` for details.

The scoped variants of `summarise()` make it easy to apply the same transformation to multiple variables. There are three variants.

- `summarise_all()` affects every variable
- `summarise_at()` affects variables selected with a character vector or vars()
- `summarise_if()` affects variables selected with a predicate function

Usage

```
summarise_all(.tbl, .funs, ...)

summarise_if(.tbl, .predicate, .funs, ...)

summarise_at(.tbl, .vars, .funs, ..., .cols = NULL)

summarize_all(.tbl, .funs, ...)

summarize_if(.tbl, .predicate, .funs, ...)

summarize_at(.tbl, .vars, .funs, ..., .cols = NULL)
```

Arguments

.tbl	A tbl object.
.funs	A function fun, a quosure style lambda ~ fun(.) or a list of either form.
...	Additional arguments for the function calls in .funs. These are evaluated only once, with tidy dots support.
.predicate	A predicate function to be applied to the columns or a logical vector. The variables for which .predicate is or returns TRUE are selected. This argument is passed to rlang::as_function() and thus supports quosure-style lambda functions and strings representing function names.
.vars	A list of columns generated by vars(), a character vector of column names, a numeric vector of column positions, or NULL.
.cols	This argument has been renamed to .vars to fit dplyr's terminology and is deprecated.

Value

A data frame. By default, the newly created columns have the shortest names needed to uniquely identify the output. To force inclusion of a name, even when not needed, name the input (see examples for details).

Grouping variables

If applied on a grouped tibble, these operations are *not* applied to the grouping variables. The behaviour depends on whether the selection is **implicit** (all and if selections) or **explicit** (at selections).

- Grouping variables covered by explicit selections in summarise_at() are always an error. Add -group_cols() to the vars() selection to avoid this:

  ```
  data %>%
    summarise_at(vars(-group_cols(), ...), myoperation)
  ```

  Or remove group_vars() from the character vector of column names:

  ```
  nms <- setdiff(nms, group_vars(data))
  data %>% summarise_at(nms, myoperation)
  ```

- Grouping variables covered by implicit selections are silently ignored by summarise_all() and summarise_if().

Naming

The names of the new columns are derived from the names of the input variables and the names of the functions.

- if there is only one unnamed function (i.e. if .funs is an unnamed list of length one), the names of the input variables are used to name the new columns;
- for _at functions, if there is only one unnamed variable (i.e., if .vars is of the form vars(a_single_column)) and .funs has length greater than one, the names of the functions are used to name the new columns;
- otherwise, the new names are created by concatenating the names of the input variables and the names of the functions, separated with an underscore "_".

The .funs argument can be a named or unnamed list. If a function is unnamed and the name cannot be derived automatically, a name of the form "fn#" is used. Similarly, vars() accepts named and unnamed arguments. If a variable in .vars is named, a new column by that name will be created.

Name collisions in the new columns are disambiguated using a unique suffix.

Life cycle

The functions are maturing, because the naming scheme and the disambiguation algorithm are subject to change in dplyr 0.9.0.

See Also

The other scoped verbs, vars()

Examples

```
# The _at() variants directly support strings:
starwars %>%
  summarise_at(c("height", "mass"), mean, na.rm = TRUE)
# ->
starwars %>% summarise(across(c("height", "mass"), ~ mean(.x, na.rm = TRUE)))

# You can also supply selection helpers to _at() functions but you have
# to quote them with vars():
starwars %>%
  summarise_at(vars(height:mass), mean, na.rm = TRUE)
# ->
starwars %>%
  summarise(across(height:mass, ~ mean(.x, na.rm = TRUE)))

# The _if() variants apply a predicate function (a function that
# returns TRUE or FALSE) to determine the relevant subset of
# columns. Here we apply mean() to the numeric columns:
starwars %>%
  summarise_if(is.numeric, mean, na.rm = TRUE)
starwars %>%
  summarise(across(where(is.numeric), ~ mean(.x, na.rm = TRUE)))

by_species <- iris %>%
  group_by(Species)

# If you want to apply multiple transformations, pass a list of
# functions. When there are multiple functions, they create new
# variables instead of modifying the variables in place:
by_species %>%
  summarise_all(list(min, max))
# ->
by_species %>%
  summarise(across(everything(), list(min = min, max = max)))
```

tbl *Create a table from a data source*

Description

This is a generic method that dispatches based on the first argument.

Usage

```
tbl(src, ...)

is.tbl(x)
```

Arguments

src	A data source
...	Other arguments passed on to the individual methods
x	Any object

vars *Select variables*

Description

vars() was only needed for the scoped verbs, which have been superseded by the use of across() in an existing verb. See vignette("colwise") for details.

This helper is intended to provide equivalent semantics to select(). It is used for instance in scoped summarising and mutating verbs (mutate_at() and summarise_at()).

Note that verbs accepting a vars() specification also accept a numeric vector of positions or a character vector of column names.

Usage

```
vars(...)
```

Arguments

...	<tidy-select> Variables to include/exclude in mutate/summarise. You can use same specifications as in select(). If missing, defaults to all non-grouping variables.

See Also

all_vars() and any_vars() for other quoting functions that you can use with scoped verbs.

`with_groups` *Perform an operation with temporary groups*

Description

[Experimental]

This is an experimental new function that allows you to modify the grouping variables for a single operation.

Usage

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

Arguments

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

Examples

```
df <- tibble(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, x1 = first(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)
```