Package ‘dplyr’

February 3, 2026

Type Package

Title A Grammar of Data Manipulation

Version 1.2.0

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 (>= 4.1.0)

Imports cli (>= 3.6.2), generics, glue (>= 1.3.2), lifecycle (>= 1.0.5), magrittr (>= 1.5), methods, pillar (>= 1.9.0), R6, rlang (>= 1.1.7), tibble (>= 3.2.0), tidyselect (>= 1.2.0), utils, vctrs (>= 0.7.1)

Suggests broom, covr, DBI, dbplyr (>= 2.2.1), ggplot2, knitr, Lahman, lobstr, nycflights13, purrr, rmarkdown, RSQLite, stringi (>= 1.7.6), testthat (>= 3.1.5), tidyr (>= 1.3.0), withr

VignetteBuilder knitr

Config/build/compilation-database true

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

LazyData true

RoxygenNote 7.3.3

NeedsCompilation yes

Author Hadley Wickham [aut, cre] (ORCID:
<https://orcid.org/0000-0003-4757-117X>),
Romain François [aut] (ORCID: <https://orcid.org/0000-0002-2444-4226>),
Lionel Henry [aut],
Kirill Müller [aut] (ORCID: <https://orcid.org/0000-0002-1416-3412>),
Davis Vaughan [aut] (ORCID: <https://orcid.org/0000-0003-4777-038X>),
Posit Software, PBC [cph, fnd]

Maintainer Hadley Wickham <hadley@posit.co>

Repository CRAN

Date/Publication 2026-02-03 08:50:47 UTC

Contents

across 3
all_vars 8
arrange 9
auto_copy 11
band_members 11
between 12
bind_cols 13
bind_rows 14
case-and-replace-when 15
coalesce 20
compute 21
consecutive_id 23
context 23
copy_to 25
count 26
cross_join 28
cumall 29
c_across 30
desc 31
distinct 31
dplyr_by 33
explain 37
filter 38
filter-joins 43
glimpse 45
group_by 45
group_cols 48
group_map 49
group_trim 51
ident 52
if_else 53
join_by 54
lead-lag 58
mutate 59
mutate-joins 63
na_if 69
near 71
nest_join 71
nth 74
ntile 76
n_distinct 77

order_by 78
percent_rank 79
pick 80
pull 81
recode 82
recode-and-replace-values 86
reframe 91
relocate 94
rename 95
rows 97
rowwise 100
row_number 102
scoped 103
select 105
setops 109
slice 111
sql 114
starwars 115
storms 116
summarise 117
tbl 119
vars 120
when-any-all 120

Index **124**

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.

If you just need to select columns without applying a transformation to each of them, then you probably want to use `pick()` instead.

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

Usage

```
across(.cols, .fns, ..., .names = NULL, .unpack = FALSE)

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

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

Arguments

`.cols` <`tidy-select`> Columns to transform. You can't select grouping columns because they are already automatically handled by the verb (i.e. `summarise()` or `mutate()`).

`.fns` Functions to apply to each of the selected columns. Possible values are:

- A function, e.g. `mean`.
- A purrr-style lambda, e.g. `~ mean(.x, na.rm = TRUE)`
- A named list of functions or lambdas, e.g. `list(mean = mean, n_miss = ~ sum(is.na(.x))`. Each function is applied to each column, and the output is named by combining the function name and the column name using the glue specification in `.names`.

Within these functions you can use `cur_column()` and `cur_group()` to access the current column and grouping keys respectively.

`...` **[Deprecated]**

Additional arguments for the function calls in `.fns` are no longer accepted in `...` because it's not clear when they should be evaluated: once per `across()` or once per group? Instead supply additional arguments directly in `.fns` by using a lambda. For example, instead of `across(a:b, mean, na.rm = TRUE)` write `across(a:b, ~ mean(.x, na.rm = TRUE))`.

`.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`.

`.unpack` **[Experimental]**

Optionally unpack data frames returned by functions in `.fns`, which expands the df-columns out into individual columns, retaining the number of rows in the data frame.

- If `FALSE`, the default, no unpacking is done.
- If `TRUE`, unpacking is done with a default glue specification of `"{outer}_{inner}"`.
- Otherwise, a single glue specification can be supplied to describe how to name the unpacked columns. This can use `{outer}` to refer to the name originally generated by `.names`, and `{inner}` to refer to the names of the data frame you are unpacking.

Details

When there are no selected columns:

- if_any() will return FALSE, consistent with the behavior of any() when called without inputs.
- if_all() will return TRUE, consistent with the behavior of all() when called without inputs.

Value

across() typically returns a tibble with one column for each column in .cols and each function in .fns. If .unpack is used, more columns may be returned depending on how the results of .fns are unpacked.

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

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

# across() -----------------------------------------------------------------
# Using everything() to apply the same function to all columns
iris |>
  mutate(across(everything(), as.character))

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

# Using an external vector of names
cols <- c("Sepal.Length", "Petal.Width")
iris |>
  mutate(across(all_of(cols), round))

# If the external vector is named, the output columns will be named according
# to those names
names(cols) <- tolower(cols)
iris |>
  mutate(across(all_of(cols), 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}"
    )
  )

# If a named external vector is used for column selection, .names will use
# those names when constructing the output names
iris |>
  group_by(Species) |>
  summarise(across(all_of(cols), mean, .names = "mean_{.col}"))

# 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}")
  )

# When the functions in .fns return a data frame, you typically get a
# "packed" data frame back
quantile_df <- function(x, probs = c(0.25, 0.5, 0.75)) {
  tibble(quantile = probs, value = quantile(x, probs))
}

iris |>
  reframe(across(starts_with("Sepal"), quantile_df))

# Use .unpack to automatically expand these packed data frames into their
# individual columns
iris |>
  reframe(across(starts_with("Sepal"), quantile_df, .unpack = TRUE))

# .unpack can utilize a glue specification if you don't like the defaults
iris |>
  reframe(
    across(starts_with("Sepal"), quantile_df, .unpack = "{outer}.{inner}")
  )
```

```
# This is also useful inside mutate(), for example, with a multi-lag helper
multilag <- function(x, lags = 1:3) {
  names(lags) <- as.character(lags)
  purrr::map_dfr(lags, lag, x = x)
}

iris |>
  group_by(Species) |>
  mutate(across(starts_with("Sepal"), multilag, .unpack = TRUE)) |>
  select(Species, starts_with("Sepal"))

# if_any() and if_all() ----------------------------------------------------
iris |>
  filter(if_any(ends_with("Width"), ~ . > 4))
iris |>
  filter_out(if_any(ends_with("Width"), ~ . > 4))

iris |>
  filter(if_all(ends_with("Width"), ~ . > 2))
iris |>
  filter_out(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 *Order rows using 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, .locale = 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> 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.

.locale The locale to sort character vectors in.

- If NULL, the default, uses the "C" locale unless the deprecated dplyr.legacy_locale global option escape hatch is active. See the dplyr-locale help page for more details.
- If a single string from stringi::stri_locale_list() is supplied, then this will be used as the locale to sort with. For example, "en" will sort with the American English locale. This requires the stringi package.
- If "C" is supplied, then character vectors will always be sorted in the C locale. This does not require stringi and is often much faster than supplying a locale identifier.

The C locale is not the same as English locales, such as "en", particularly when it comes to data containing a mix of upper and lower case letters. This is explained in more detail on the locale help page under the Default locale section.

Details

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()`, `reframe()`, `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 ?rlang::args_data_masking for more details
tidy_eval_arrange <- function(.data, var) {
  .data |>
    arrange({{ var }})
}
tidy_eval_arrange(mtcars, mpg)

# Use `across()` or `pick()` to select columns with tidy-select
iris |> arrange(pick(starts_with("Sepal")))
iris |> arrange(across(starts_with("Sepal"), 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 *Detect where values fall in a specified range*

Description

This is a shortcut for `x >= left & x <= right`, implemented for local vectors and translated to the appropriate SQL for remote tables.

Usage

```
between(x, left, right, ..., ptype = NULL)
```

Arguments

`x`	A vector
`left, right`	Boundary values. Both `left` and `right` are recycled to the size of `x`.
`...`	These dots are for future extensions and must be empty.
`ptype`	An optional prototype giving the desired output type. The default is to compute the common type of `x`, `left`, and `right` using `vctrs::vec_cast_common()`.

Details

`x`, `left`, and `right` are all cast to their common type before the comparison is made. Use the `ptype` argument to specify the type manually.

Value

A logical vector the same size as `x` with a type determined by `ptype`.

See Also

`join_by()` if you are looking for documentation for the `between()` overlap join helper.

Examples

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

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

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

```
# Using the `ptype` argument with ordered factors, where otherwise everything
# is cast to the common type of character before the comparison
x <- ordered(
  c("low", "medium", "high", "medium"),
  levels = c("low", "medium", "high")
)
between(x, "medium", "high")
between(x, "medium", "high", ptype = x)
```

bind_cols *Bind multiple data frames by column*

Description

Bind any number of data frames by column, making a wider result. This is similar to `do.call(cbind, dfs)`.

Where possible prefer using a join to combine multiple data frames. `bind_cols()` binds the rows in order in which they appear so it is easy to create meaningless results without realising it.

Usage

```
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. Inputs are recycled to the same length, then matched by position.
`.name_repair`	One of `"unique"`, `"universal"`, or `"check_unique"`. See `vctrs::vec_as_names()` for the meaning of these options.

Value

A data frame the same type as the first element of `...`.

Examples

```
df1 <- tibble(x = 1:3)
df2 <- tibble(y = 3:1)
bind_cols(df1, df2)

# Row sizes must be compatible when column-binding
try(bind_cols(tibble(x = 1:3), tibble(y = 1:2)))
```

bind_rows *Bind multiple data frames by row*

Description

Bind any number of data frames by row, making a longer result. This is similar to `do.call(rbind, dfs)`, but the output will contain all columns that appear in any of the inputs.

Usage

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

Arguments

`...`	Data frames to combine. Each argument can either be a data frame, a list that could be a data frame, or a list of data frames. Columns are matched by name, and any missing columns will be filled with `NA`.
`.id`	The name of an optional identifier column. Provide a string to create an output column that identifies each input. The column will use names if available, otherwise it will use positions.

Value

A data frame the same type as the first element of `...`.

Examples

```
df1 <- tibble(x = 1:2, y = letters[1:2])
df2 <- tibble(x = 4:5, z = 1:2)

# You can supply individual data frames as arguments:
bind_rows(df1, df2)

# Or a list of data frames:
bind_rows(list(df1, df2))

# 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(df1, df2), .id = "id")
bind_rows(list(a = df1, b = df2), .id = "id")
```

case-and-replace-when *A general vectorised if-else*

Description

`case_when()` and `replace_when()` are two forms of vectorized `if_else()`. They work by evaluating each case sequentially and using the first match for each element to determine the corresponding value in the output vector.

- Use `case_when()` when creating an entirely new vector.
- Use `replace_when()` when partially updating an existing vector.

If you are just replacing a few values within an existing vector, then `replace_when()` is always a better choice because it is type stable, size stable, pipes better, and better expresses intent.

A major difference between the two functions is what happens when no cases match:

- `case_when()` falls through to a `.default` as a final "else" statement.
- `replace_when()` retains the original values from `x`.

See `vignette("recoding-replacing")` for more examples.

Usage

```
case_when(
  ...,
  .default = NULL,
  .unmatched = "default",
  .ptype = NULL,
  .size = NULL
)

replace_when(x, ...)
```

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.

For `case_when()`:

- The LHS inputs must be logical vectors. For backwards compatibility, scalars are recycled, but we no longer recommend supplying scalars.
- The RHS inputs will be cast to their common type, and will be recycled to the common size of the LHS inputs.

For `replace_when()`:

- The LHS inputs must be logical vectors the same size as `x`.
- The RHS inputs will be cast to the type of `x` and recycled to the size of `x`.

NULL inputs are ignored.

.default — The value used when all of the LHS inputs return either FALSE or NA.

- If NULL, the default, a missing value will be used.
- If provided, .default will follow the same type and size rules as the RHS inputs.

NA values in the LHS conditions are treated like FALSE, meaning that the result at those locations will be assigned the .default value. To handle missing values in the conditions differently, you must explicitly catch them with another condition before they fall through to the .default. This typically involves some variation of is.na(x) ~ value tailored to your usage of case_when().

.unmatched — Handling of unmatched locations.

One of:

- "default" to use .default in unmatched locations.
- "error" to error when there are unmatched locations.

.ptype — An optional prototype declaring the desired output type. If supplied, this overrides the common type of the RHS inputs.

.size — An optional size declaring the desired output size. If supplied, this overrides the common size computed from the LHS inputs.

x — A vector.

Value

For case_when(), a new vector where the size is the common size of the LHS inputs, the type is the common type of the RHS inputs, and the names correspond to the names of the RHS elements used in the result.

For replace_when(), an updated version of x, with the same size, type, and names as x.

See Also

recode_values(), vctrs::vec_case_when()

Examples

```
x <- 1:70
case_when(
  x %% 35 == 0 ~ "fizz buzz",
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  .default = 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(
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  x %% 35 == 0 ~ "fizz buzz",
  .default = as.character(x)
)

# If none of the cases match and no `.default` is supplied, NA is used:
case_when(
  x %% 35 == 0 ~ "fizz buzz",
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz"
)

# Note that `NA` values on the LHS are treated like `FALSE` and will be
# assigned the `.default` value. You must handle them explicitly if you
# want to use a different value. The exact way to handle missing values is
# dependent on the set of LHS conditions you use.
x[2:4] <- NA_real_
case_when(
  x %% 35 == 0 ~ "fizz buzz",
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  is.na(x) ~ "nope",
  .default = as.character(x)
)

# `case_when()` is not a replacement for basic if/else control flow. When
# you have a single scalar condition, using if/else is faster, simpler to
# reason about, and is lazy on the branch that isn't run. For example, this
# seems to work:
x <- "value"
case_when(is.character(x) ~ x, .default = "not-a-character")

# Until `x` is a non-character type
x <- 1
try(case_when(is.character(x) ~ x, .default = "not-a-character"))

# Instead, you should use if/else
if (is.character(x)) {
  y <- x
} else {
  y <- "not-a-character"
}
y

# If you believe that you've covered every possible case, then set
# `.unmatched = "error"` rather than supplying a `.default`. This adds an
# extra layer of safety to `case_when()` and is particularly useful when you
# have a series of complex expressions!
set.seed(123)
x <- sample(50)

# Oops, we forgot to handle `50`
try(case_when(
  x < 10 ~ "ten",
  x < 20 ~ "twenty",
  x < 30 ~ "thirty",
  x < 40 ~ "forty",
  x < 50 ~ "fifty",
  .unmatched = "error"
))

case_when(
  x < 10 ~ "ten",
  x < 20 ~ "twenty",
  x < 30 ~ "thirty",
  x < 40 ~ "forty",
  x <= 50 ~ "fifty",
  .unmatched = "error"
)

# Note that `NA` is considered unmatched and must be handled with its own
# explicit case, even if that case just propagates the missing value!
x[c(2, 5)] <- NA

case_when(
  x < 10 ~ "ten",
  x < 20 ~ "twenty",
  x < 30 ~ "thirty",
  x < 40 ~ "forty",
  x <= 50 ~ "fifty",
  is.na(x) ~ NA,
  .unmatched = "error"
)

# `replace_when()` is useful when you're updating an existing vector,
# rather than creating an entirely new one. Note the so-far unused "puppy"
# factor level:
pets <- tibble(
  name = c("Max", "Bella", "Chuck", "Luna", "Cooper"),
  type = factor(
    c("dog", "dog", "cat", "dog", "cat"),
    levels = c("dog", "cat", "puppy")
  ),
  age = c(1, 3, 5, 2, 4)
)

# We can replace some values with `"puppy"` based on arbitrary conditions.
# Even though we are using a character `"puppy"` value, `replace_when()` will
# automatically cast it to the factor type of `type` for us.
pets |>
  mutate(
    type = replace_when(type, type == "dog" & age <= 2 ~ "puppy")
  )

# Compare that with this `case_when()` call, which loses the factor class.
# It's always better to use `replace_when()` when updating a few values in
# an existing vector!
pets |>
```

```
  mutate(
    type = case_when(type == "dog" & age <= 2 ~ "puppy", .default = type)
  )

# `case_when()` and `replace_when()` evaluate all RHS expressions, and then
# construct their result by extracting the selected (via the LHS expressions)
# parts. For example, `NaN`s are produced here because `sqrt(y)` is evaluated
# on all of `y`, not just where `y >= 0`.
y <- seq(-2, 2, by = .5)
replace_when(y, y >= 0 ~ sqrt(y))

# These functions are 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",
      .default = "other"
    )
  )

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

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

# `replace_when()` can also be used in combination with `pick()` to
# conditionally mutate rows within multiple columns using a single condition.
# Here `replace_when()` returns a data frame with new `species` and `name`
# columns, which `mutate()` then automatically unpacks.
starwars |>
  select(homeworld, species, name) |>
  mutate(replace_when(
    pick(species, name),
    homeworld == "Tatooine" ~ tibble(
      species = "Tatooinese",
      name = paste(name, "(Tatooine)")
    )
  ))
```

coalesce *Find the first non-missing element*

Description

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

Usage

```
coalesce(..., .ptype = NULL, .size = NULL)
```

Arguments

...	<dynamic-dots> One or more vectors. These will be recycled against each other, and will be cast to their common type.
.ptype	An optional prototype declaring the desired output type. If supplied, this overrides the common type of the vectors in ....
.size	An optional size declaring the desired output size. If supplied, this overrides the common size of the vectors in ....

Value

A vector with the same type and size as the common type and common size of the vectors in ....

See Also

- na_if() to replace a specified value with NA.
- replace_values() for making arbitrary replacements by value.
- replace_when() for making arbitrary replacements using logical conditions.

Examples

```
# Replace missing values with a single value
x <- sample(c(1:5, NA, NA, NA))
coalesce(x, 0L)

# Or replace missing values with the corresponding non-missing value in
# another vector
x <- c(1, 2, NA, NA, 5, NA)
y <- c(NA, NA, 3, 4, 5, NA)
coalesce(x, y)

# For cases like these where your replacement is a single value or a single
# vector, `replace_values()` works just as well
replace_values(x, NA ~ 0)
coalesce(x, 0)

replace_values(x, NA ~ y)
coalesce(x, y)

# `coalesce()` really shines when you have >2 vectors to coalesce with
z <- c(NA, 2, 3, 4, 5, 6)
coalesce(x, y, z)

# If you're looking to replace values with `NA`, rather than replacing `NA`
# with a value, then use `replace_values()`
x <- c(0, -1, 5, -99, 8)
replace_values(x, c(-1, -99) ~ NA)

# The equivalent to a missing value in a list is `NULL`
coalesce(list(1, 2, NULL, NA), list(0))

# Supply lists of vectors by splicing them into 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

```
mtcars2 <- dbplyr::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)
```

consecutive_id *Generate a unique identifier for consecutive combinations*

Description

consecutive_id() generates a unique identifier that increments every time a variable (or combination of variables) changes. Inspired by data.table::rleid().

Usage

```
consecutive_id(...)
```

Arguments

...	Unnamed vectors. If multiple vectors are supplied, then they should have the same length.

Value

A numeric vector the same length as the longest element of ....

Examples

```
consecutive_id(c(TRUE, TRUE, FALSE, FALSE, TRUE, FALSE, NA, NA))
consecutive_id(c(1, 1, 1, 2, 1, 1, 2, 2))

df <- data.frame(x = c(0, 0, 1, 0), y = c(2, 2, 2, 2))
df |> group_by(x, y) |> summarise(n = n())
df |> group_by(id = consecutive_id(x, y), x, y) |> summarise(n = n())
```

context *Information about the "current" group or variable*

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_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.

See `pick()` for a way to select a subset of columns using tidyselect syntax while inside `summarise()` or `mutate()`.

Usage

```
n()

cur_group()

cur_group_id()

cur_group_rows()

cur_column()
```

data.table

If you're familiar with data.table:

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

See `pick()` for an equivalent to `.SD`.

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 |> reframe(row = cur_group_rows())
gf |> summarise(data = list(cur_group()))

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

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

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 the observations in each 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)

## S3 method for class 'data.frame'
count(
  x,
  ...,
  wt = NULL,
  sort = FALSE,
  name = NULL,
  .drop = group_by_drop_default(x)
)

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 use nn. If there's a column called n and nn, it'll use nnn, and so on, adding ns until it gets a new name.

`.drop` Handling of factor levels that don't appear in the data, passed on to `group_by()`.
For `count()`: if `FALSE` will include counts for empty groups (i.e. for levels of factors that don't exist in the data).
[Defunct] For `add_count()`: defunct since it can'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)

# When factors are involved, `.drop = FALSE` can be used to retain factor
# levels that don't appear in the data
df2 <- tibble(
  id = 1:5,
  type = factor(c("a", "c", "a", NA, "a"), levels = c("a", "b", "c"))
)
df2 |> count(type)
df2 |> count(type, .drop = FALSE)

# Or, using `group_by()`:
df2 |> group_by(type, .drop = FALSE) |> count()

# 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)
```

cross_join *Cross join*

Description

Cross joins match each row in x to every row in y, resulting in a data frame with nrow(x) * nrow(y) rows.

Since cross joins result in all possible matches between x and y, they technically serve as the basis for all mutating joins, which can generally be thought of as cross joins followed by a filter. In practice, a more specialized procedure is used for better performance.

Usage

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

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.
...	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.
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.

Value

An object of the same type as x (including the same groups). The output has the following properties:

- There are nrow(x) * nrow(y) rows returned.
- Output columns include all columns from both x and y. Column name collisions are resolved using suffix.
- The order of the rows and columns of x is preserved as much as possible.

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, nest_join()

Examples

```
# Cross joins match each row in `x` to every row in `y`.
# Data within the columns is not used in the matching process.
cross_join(band_instruments, band_members)

# Control the suffix added to variables duplicated in
# `x` and `y` with `suffix`.
cross_join(band_instruments, band_members, suffix = c("", "_y"))
```

`cumall` *Cumulative 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)
```

Arguments

cols	<tidy-select> Columns to transform. You can't select grouping columns because they are already automatically handled by the verb (i.e. summarise() or mutate()).

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 *Keep distinct/unique rows*

Description

Keep 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 in the data frame.

`.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 `pick()` to select columns with tidy-select
distinct(starwars, pick(contains("color")))
```

```
# Grouping -------------------------------------------------

df <- tibble(
  g = c(1, 1, 2, 2, 2),
  x = c(1, 1, 2, 1, 2),
  y = c(3, 2, 1, 3, 1)
)
df <- df |> group_by(g)

# With grouped data frames, distinctness is computed within each group
df |> distinct(x)

# When `...` are omitted, `distinct()` still computes distinctness using
# all variables in the data frame
df |> distinct()
```

---

dplyr_by *Per-operation grouping with* .by/by

Description

There are two ways to group in dplyr:

- Persistent grouping with group_by()
- Per-operation grouping with .by/by

This help page is dedicated to explaining where and why you might want to use the latter.

Depending on the dplyr verb, the per-operation grouping argument may be named .by or by. The *Supported verbs* section below outlines this on a case-by-case basis. The remainder of this page will refer to .by for simplicity.

Grouping radically affects the computation of the dplyr verb you use it with, and one of the goals of .by is to allow you to place that grouping specification alongside the code that actually uses it. As an added benefit, with .by you no longer need to remember to ungroup() after summarise(), and summarise() won't ever message you about how it's handling the groups!

This idea comes from data.table, which allows you to specify by alongside modifications in j, like: dt[, .(x = mean(x)), by = g].

Supported verbs:

- mutate(.by =)
- summarise(.by =)
- reframe(.by =)
- filter(.by =)
- filter_out(.by =)
- slice(.by =)
- slice_head(by =) and slice_tail(by =)
- slice_min(by =) and slice_max(by =)

- `slice_sample(by = )`

Note that some dplyr verbs use `by` while others use `.by`. This is a purely technical difference.

Differences between `.by` and `group_by()`:

`.by`	`group_by()`
Grouping only affects a single verb	Grouping is persistent across multiple verbs
Selects variables with tidy-select	Computes expressions with data-masking
Summaries use existing order of group keys	Summaries sort group keys in ascending order

Using `.by`:

Let's take a look at the two grouping approaches using this `expenses` data set, which tracks costs accumulated across various `id`s and `region`s:

```
expenses <- tibble(
  id = c(1, 2, 1, 3, 1, 2, 3),
  region = c("A", "A", "A", "B", "B", "A", "A"),
  cost = c(25, 20, 19, 12, 9, 6, 6)
)
expenses
#> # A tibble: 7 x 3
#>      id region  cost
#>   <dbl> <chr>  <dbl>
#> 1     1 A         25
#> 2     2 A         20
#> 3     1 A         19
#> 4     3 B         12
#> 5     1 B          9
#> 6     2 A          6
#> 7     3 A          6
```

Imagine that you wanted to compute the average cost per region. You'd probably write something like this:

```
expenses |>
  group_by(region) |>
  summarise(cost = mean(cost))
#> # A tibble: 2 x 2
#>   region  cost
#>   <chr>  <dbl>
#> 1 A       15.2
#> 2 B       10.5
```

Instead, you can now specify the grouping *inline* within the verb:

```
expenses |>
  summarise(cost = mean(cost), .by = region)
#> # A tibble: 2 x 2
#>   region  cost
#>   <chr>  <dbl>
#> 1 A       15.2
#> 2 B       10.5
```

.by applies to a single operation, meaning that since expenses was an ungrouped data frame, the result after applying .by will also always be an ungrouped data frame, regardless of the number of grouping columns.

```
expenses |>
  summarise(cost = mean(cost), .by = c(id, region))
#> # A tibble: 5 x 3
#>      id region  cost
#>   <dbl> <chr>  <dbl>
#> 1     1 A         22
#> 2     2 A         13
#> 3     3 B         12
#> 4     1 B          9
#> 5     3 A          6
```

Compare that with group_by() |> summarise(), where summarise() generally peels off 1 layer of grouping by default, typically with a message that it is doing so:

```
expenses |>
  group_by(id, region) |>
  summarise(cost = mean(cost))
#> `summarise()` has regrouped the output.
#> i Summaries were computed grouped by id and region.
#> i Output is grouped by id.
#> i Use `summarise(.groups = "drop_last")` to silence this message.
#> i Use `summarise(.by = c(id, region))` for per-operation grouping
#>   (`?dplyr::dplyr_by`) instead.
#> # A tibble: 5 x 3
#> # Groups:   id [3]
#>      id region  cost
#>   <dbl> <chr>  <dbl>
#> 1     1 A         22
#> 2     1 B          9
#> 3     2 A         13
#> 4     3 A          6
#> 5     3 B         12
```

Because .by grouping applies to a single operation, you don't need to worry about ungrouping, and it never needs to emit a message to remind you what it is doing with the groups.

Note that with .by we specified multiple columns to group by using the tidy-select syntax c(id, region). If you have a character vector of column names you'd like to group by, you can do so with .by = all_of(my_cols). It will group by the columns in the order they were provided.

To prevent surprising results, you can't use .by on an existing grouped data frame:

```
expenses |>
  group_by(id) |>
  summarise(cost = mean(cost), .by = c(id, region))
#> Error in `summarise()`:
#> ! Can't supply `.by` when `.data` is a grouped data frame.
```

So far we've focused on the usage of .by with summarise(), but .by works with a number of other dplyr verbs. For example, you could append the mean cost per region onto the original data frame as a new column rather than computing a summary:

```
expenses |>
  mutate(cost_by_region = mean(cost), .by = region)
#> # A tibble: 7 x 4
#>      id region  cost cost_by_region
#>   <dbl> <chr>  <dbl>          <dbl>
#> 1     1 A         25           15.2
#> 2     2 A         20           15.2
#> 3     1 A         19           15.2
#> 4     3 B         12           10.5
#> 5     1 B          9           10.5
#> 6     2 A          6           15.2
#> 7     3 A          6           15.2
```

Or you could slice out the maximum cost per combination of id and region:

```
# Note that the argument is named `by` in `slice_max()`
expenses |>
  slice_max(cost, n = 1, by = c(id, region))
#> # A tibble: 5 x 3
#>      id region  cost
#>   <dbl> <chr>  <dbl>
#> 1     1 A         25
#> 2     2 A         20
#> 3     3 B         12
#> 4     1 B          9
#> 5     3 A          6
```

Result ordering:

When used with .by, summarise(), reframe(), and slice() all maintain the ordering of the existing data. This is different from group_by(), which has always sorted the group keys in ascending order.

```
df <- tibble(
  month = c("jan", "jan", "feb", "feb", "mar"),
  temp = c(20, 25, 18, 20, 40)
)

# Uses ordering by "first appearance" in the original data
df |>
  summarise(average_temp = mean(temp), .by = month)
#> # A tibble: 3 x 2
#>   month average_temp
#>   <chr>        <dbl>
#> 1 jan           22.5
#> 2 feb           19
#> 3 mar           40

# Sorts in ascending order
df |>
  group_by(month) |>
  summarise(average_temp = mean(temp))
#> # A tibble: 3 x 2
#>   month average_temp
#>   <chr>        <dbl>
#> 1 feb           19
#> 2 jan           22.5
#> 3 mar           40
```

If you need sorted group keys, we recommend that you explicitly use `arrange()` either before or after the call to `summarise()`, `reframe()`, or `slice()`. This also gives you full access to all of `arrange()`'s features, such as `desc()` and the `.locale` argument.

Verbs without `.by` support:

If a dplyr verb doesn't support `.by`, then that typically means that the verb isn't inherently affected by grouping. For example, `pull()` and `rename()` don't support `.by`, because specifying columns to group by would not affect their implementations.

That said, there are a few exceptions to this where sometimes a dplyr verb doesn't support `.by`, but *does* have special support for grouped data frames created by `group_by()`. This is typically because the verbs are required to retain the grouping columns, for example:

- `select()` always retains grouping columns, with a message if any aren't specified in the `select()` call.
- `distinct()` and `count()` place unspecified grouping columns at the front of the data frame before computing their results.
- `arrange()` has a `.by_group` argument to optionally order by grouping columns first.

If `group_by()` didn't exist, then these verbs would not have special support for grouped data frames.

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

```
lahman_s <- dbplyr::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` *Keep or drop rows that match a condition*

Description

These functions are used to subset a data frame, applying the expressions in `...` to determine which rows should be kept (for `filter()`) or dropped (for `filter_out()`).

Multiple conditions can be supplied separated by a comma. These will be combined with the `&` operator. To combine comma separated conditions using `|` instead, wrap them in `when_any()`.

Both `filter()` and `filter_out()` treat `NA` like `FALSE`. This subtle behavior can impact how you write your conditions when missing values are involved. See the section on `Missing values` for important details and examples.

Usage

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

filter_out(.data, ..., .by = NULL, .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 vector, defined in terms of the variables in `.data`. If multiple expressions are included, they are combined with the `&` operator. To combine expressions using `|` instead, wrap them in `when_any()`. Only rows for which all expressions evaluate to `TRUE` are kept (for `filter()`) or dropped (for `filter_out()`).

`.by` <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.

`.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.

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.

Missing values

Both `filter()` and `filter_out()` treat `NA` like `FALSE`. This results in the following behavior:

- `filter()` *drops* both `NA` and `FALSE`.
- `filter_out()` *keeps* both `NA` and `FALSE`.

This means that `filter(data, <conditions>) + filter_out(data, <conditions>)` captures every row within `data` exactly once.

The `NA` handling of these functions has been designed to match your *intent*. When your intent is to keep rows, use `filter()`. When your intent is to drop rows, use `filter_out()`.

For example, if your goal with this `cars` data is to "drop rows where the `class` is suv", then you might write this in one of two ways:

```
cars <- tibble(class = c("suv", NA, "coupe"))
cars
#> # A tibble: 3 x 1
#>   class
#>   <chr>
#> 1 suv
#> 2 <NA>
#> 3 coupe

cars |> filter(class != "suv")
#> # A tibble: 1 x 1
#>   class
#>   <chr>
#> 1 coupe

cars |> filter_out(class == "suv")
#> # A tibble: 2 x 1
#>   class
#>   <chr>
#> 1 <NA>
#> 2 coupe
```

Note how `filter()` drops the `NA` rows even though our goal was only to drop `"suv"` rows, but `filter_out()` matches our intuition.

To generate the correct result with `filter()`, you'd need to use:

```
cars |> filter(class != "suv" | is.na(class))
#> # A tibble: 2 x 1
#>   class
#>   <chr>
#> 1 <NA>
#> 2 coupe
```

This quickly gets unwieldy when multiple conditions are involved.

In general, if you find yourself:

- Using "negative" operators like `!=` or `!`
- Adding in `NA` handling like `| is.na(col)` or `& !is.na(col)`

then you should consider if swapping to the other filtering variant would make your conditions simpler.

Comparison to base subsetting:

Base subsetting with `[` doesn't treat `NA` like `TRUE` or `FALSE`. Instead, it generates a fully missing row, which is different from how both `filter()` and `filter_out()` work.

```
cars <- tibble(class = c("suv", NA, "coupe"), mpg = c(10, 12, 14))
cars
#> # A tibble: 3 x 2
#>   class   mpg
#>   <chr> <dbl>
#> 1 suv      10
#> 2 <NA>     12
#> 3 coupe    14
```

```
cars[cars$class == "suv",]
#> # A tibble: 2 x 2
#>   class   mpg
#>   <chr> <dbl>
#> 1 suv      10
#> 2 <NA>     NA

cars |> filter(class == "suv")
#> # A tibble: 1 x 2
#>   class   mpg
#>   <chr> <dbl>
#> 1 suv      10
```

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()
- when_any(), when_all()

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 |> filter(mass > mean(mass, na.rm = TRUE), .by = gender)
```

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()`, `reframe()`, `rename()`, `select()`, `slice()`, `summarise()`

Examples

```
# Filtering for one criterion
filter(starwars, species == "Human")

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

# Multiple comma separated expressions are combined using `&`
starwars |> filter(hair_color == "none", eye_color == "black")

# To combine comma separated expressions using `|` instead, use `when_any()`
starwars |> filter(when_any(hair_color == "none", eye_color == "black"))

# Filtering out to drop rows
filter_out(starwars, hair_color == "none")

# When filtering out, it can be useful to first interactively filter for the
# rows you want to drop, just to double check that you've written the
# conditions correctly. Then, just change `filter()` to `filter_out()`.
filter(starwars, mass > 1000, eye_color == "orange")
filter_out(starwars, mass > 1000, eye_color == "orange")

# The filtering operation may yield different results on grouped
# tibbles because the expressions are computed within groups.
#
# The following keeps 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 per `gender`
# average:
starwars |> filter(mass > mean(mass, na.rm = TRUE), .by = gender)

# If you find yourself trying to use a `filter()` to drop rows, then
# you should consider if switching to `filter_out()` can simplify your
# conditions. For example, to drop blond individuals, you might try:
starwars |> filter(hair_color != "blond")

# But this also drops rows with an `NA` hair color! To retain those:
starwars |> filter(hair_color != "blond" | is.na(hair_color))

# But explicit `NA` handling like this can quickly get unwieldy, especially
# with multiple conditions. Since your intent was to specify rows to drop
# rather than rows to keep, use `filter_out()`. This also removes the need
# for any explicit `NA` handling.
starwars |> filter_out(hair_color == "blond")
```

```
# 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 ?rlang::args_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() returns all rows from x with a match in y.
- anti_join() returns 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 join specification created with join_by(), or 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 on different variables between x and y, use a join_by() specification. For example, join_by(a == b) will match x$a to y$b.

To join by multiple variables, use a join_by() specification with multiple expressions. For example, join_by(a == b, c == d) will match x$a to y$b and x$c to y$d. If the column names are the same between x and y, you can shorten this by listing only the variable names, like join_by(a, c).

join_by() can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, by = c("a", "b") joins x$a to y$a and x$b to y$b. If variable names differ between x and y, use a named character vector like by = c("x_a" = "y_a", "x_b" = "y_b").

To perform a cross-join, generating all combinations of x and y, see cross_join().

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 two NA or two NaN values match?

- "na", the default, treats two NA or two NaN values as equal, like %in%, match(), and merge().
- "never" treats two NA or two NaN values as different, and will never match them together or to any other values. This is similar to joins for database sources and to base::merge(incomparables = NA).

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: cross_join(), 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 = join_by(name))
# This is good practice in production code
```

glimpse *Get a glimpse of your data*

Description

glimpse() is like a transposed version of print(): columns run down the page, and data runs across. This makes it possible to see every column in a data frame. It's a little like str() applied to a data frame but it tries to show you as much data as possible. (And it always shows the underlying data, even when applied to a remote data source.)

glimpse() is provided by the pillar package, and re-exported by dplyr. See pillar::glimpse() for more details.

Value

x original x is (invisibly) returned, allowing glimpse() to be used within a data pipeline.

Examples

```
glimpse(mtcars)

# Note that original x is (invisibly) returned, allowing `glimpse()` to be
# used within a pipeline.
mtcars |>
  glimpse() |>
  select(1:3)

glimpse(starwars)
```

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.

`...` <`data-masking`> In `group_by()`, variables or computations to group by. Computations are always done on the ungrouped data frame. To perform computations on the grouped data, you need to use a separate `mutate()` step before the `group_by()`. Computations are not allowed in `nest_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`.

`.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.

Ordering

Currently, `group_by()` internally orders the groups in ascending order. This results in ordered output from functions that aggregate groups, such as `summarise()`.

When used as grouping columns, character vectors are ordered in the C locale for performance and reproducibility across R sessions. If the resulting ordering of your grouped operation matters and is dependent on the locale, you should follow up the grouped operation with an explicit call to `arrange()` and set the `.locale` argument. For example:

```
data |>
  group_by(chr) |>
  summarise(avg = mean(x)) |>
  arrange(chr, .locale = "en")
```

This is often useful as a preliminary step before generating content intended for humans, such as an HTML table.

Legacy behavior:

[Deprecated]

Prior to dplyr 1.1.0, character vector grouping columns were ordered in the system locale. Setting the global option `dplyr.legacy_locale` to `TRUE` retains this legacy behavior, but this has been deprecated. Update existing code to explicitly call `arrange(.locale = )` instead. Run `Sys.getlocale("LC_COLLATE")` to determine your system locale, and compare that against the list in `stringi::stri_locale_list()` to find an appropriate value for `.locale`, i.e. for American English, `"en_US"`.

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))

# 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()

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

# The implicit mutate() step is always performed on the
# ungrouped data. Here we get 3 groups:
mtcars |>
  group_by(vs) |>
  group_by(hp_cut = cut(hp, 3))

# If you want it to be performed by groups,
# you have to use an explicit mutate() call.
# Here we get 3 groups per value of vs
mtcars |>
  group_by(vs) |>
  mutate(hp_cut = cut(hp, 3)) |>
  group_by(hp_cut)

# 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_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	**[Defunct]**
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, ..., .keep = FALSE)
```

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))


# 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_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 strings and turns them as database identifiers (e.g. table or column names) quoting them using the identifer rules for your database. `ident_q()` does the same, but assumes the names have already been quoted, preventing them from being quoted again.

These are generally for internal use only; if you need to supply an table name that is qualified with schema or catalog, or has already been quoted for some other reason, use `I()`.

Usage

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

Arguments

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

Examples

```
# Identifiers are escaped with "

ident("x")
```

if_else *Vectorised if-else*

Description

if_else() is a vectorized if-else. Compared to the base R equivalent, ifelse(), this function allows you to handle missing values in the condition with missing and always takes true, false, and missing into account when determining what the output type should be.

Usage

```
if_else(
  condition,
  true,
  false,
  missing = NULL,
  ...,
  ptype = NULL,
  size = deprecated()
)
```

Arguments

condition	A logical vector
true, false	Vectors to use for TRUE and FALSE values of condition. Both true and false will be recycled to the size of condition. true, false, and missing (if used) will be cast to their common type.
missing	If not NULL, will be used as the value for NA values of condition. Follows the same size and type rules as true and false.
...	These dots are for future extensions and must be empty.
ptype	An optional prototype declaring the desired output type. If supplied, this overrides the common type of true, false, and missing.
size	**[Deprecated]** Output size is always taken from condition.

Value

A vector with the same size as condition and the same type as the common type of true, false, and missing.

Where condition is TRUE, the matching values from true, where it is FALSE, the matching values from false, and where it is NA, the matching values from missing, if provided, otherwise a missing value will be used.

See Also

vctrs::vec_if_else()

Examples

```
x <- c(-5:5, NA)
if_else(x < 0, NA, x)

# Explicitly handle `NA` values in the `condition` with `missing`
if_else(x < 0, "negative", "positive", missing = "missing")

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

# `if_else()` is often useful for creating new columns inside of `mutate()`
starwars |>
  mutate(category = if_else(height < 100, "short", "tall"), .keep = "used")
```

join_by *Join specifications*

Description

join_by() constructs a specification that describes how to join two tables using a small domain specific language. The result can be supplied as the by argument to any of the join functions (such as left_join()).

Usage

```
join_by(...)
```

Arguments

... Expressions specifying the join.

Each expression should consist of one of the following:

- Equality condition: ==
- Inequality conditions: >=, >, <=, or <
- Rolling helper: closest()
- Overlap helpers: between(), within(), or overlaps()

Other expressions are not supported. If you need to perform a join on a computed variable, e.g. join_by(sales_date - 40 >= promo_date), you'll need to precompute and store it in a separate column.

Column names should be specified as quoted or unquoted names. By default, the name on the left-hand side of a join condition refers to the left-hand table, unless overridden by explicitly prefixing the column name with either x$ or y$.

If a single column name is provided without any join conditions, it is interpreted as if that column name was duplicated on each side of ==, i.e. x is interpreted as x == x.

Join types

The following types of joins are supported by dplyr:

- Equality joins
- Inequality joins
- Rolling joins
- Overlap joins
- Cross joins

Equality, inequality, rolling, and overlap joins are discussed in more detail below. Cross joins are implemented through `cross_join()`.

Equality joins:

Equality joins require keys to be equal between one or more pairs of columns, and are the most common type of join. To construct an equality join using `join_by()`, supply two column names to join with separated by `==`. Alternatively, supplying a single name will be interpreted as an equality join between two columns of the same name. For example, `join_by(x)` is equivalent to `join_by(x == x)`.

Inequality joins:

Inequality joins match on an inequality, such as `>`, `>=`, `<`, or `<=`, and are common in time series analysis and genomics. To construct an inequality join using `join_by()`, supply two column names separated by one of the above mentioned inequalities.

Note that inequality joins will match a single row in `x` to a potentially large number of rows in `y`. Be extra careful when constructing inequality join specifications!

Rolling joins:

Rolling joins are a variant of inequality joins that limit the results returned from an inequality join condition. They are useful for "rolling" the closest match forward/backwards when there isn't an exact match. To construct a rolling join, wrap an inequality with `closest()`.

- `closest(expr)`
  `expr` must be an inequality involving one of: `>`, `>=`, `<`, or `<=`.
  For example, `closest(x >= y)` is interpreted as: For each value in `x`, find the closest value in `y` that is less than or equal to that `x` value.

`closest()` will always use the left-hand table (`x`) as the primary table, and the right-hand table (`y`) as the one to find the closest match in, regardless of how the inequality is specified. For example, `closest(y$a >= x$b)` will always be interpreted as `closest(x$b <= y$a)`.

Overlap joins:

Overlap joins are a special case of inequality joins involving one or two columns from the left-hand table *overlapping* a range defined by two columns from the right-hand table. There are three helpers that `join_by()` recognizes to assist with constructing overlap joins, all of which can be constructed from simpler inequalities.

- `between(x, y_lower, y_upper, ..., bounds = "[]")`
  For each value in `x`, this finds everywhere that value falls between `[y_lower, y_upper]`. Equivalent to `x >= y_lower, x <= y_upper` by default.

bounds can be one of "[]", "[)", "(]", or "()" to alter the inclusiveness of the lower and upper bounds. This changes whether >= or > and <= or < are used to build the inequalities shown above.
Dots are for future extensions and must be empty.

- within(x_lower, x_upper, y_lower, y_upper)
For each range in [x_lower, x_upper], this finds everywhere that range falls completely within [y_lower, y_upper]. Equivalent to x_lower >= y_lower, x_upper <= y_upper.
The inequalities used to build within() are the same regardless of the inclusiveness of the supplied ranges.
- overlaps(x_lower, x_upper, y_lower, y_upper, ..., bounds = "[]")
For each range in [x_lower, x_upper], this finds everywhere that range overlaps [y_lower, y_upper] in any capacity. Equivalent to x_lower <= y_upper, x_upper >= y_lower by default.
bounds can be one of "[]", "[)", "(]", or "()" to alter the inclusiveness of the lower and upper bounds. "[]" uses <= and >=, but the 3 other options use < and > and generate the exact same inequalities.
Dots are for future extensions and must be empty.

These conditions assume that the ranges are well-formed and non-empty, i.e. x_lower <= x_upper when bounds are treated as "[]", and x_lower < x_upper otherwise.

Column referencing

When specifying join conditions, join_by() assumes that column names on the left-hand side of the condition refer to the left-hand table (x), and names on the right-hand side of the condition refer to the right-hand table (y). Occasionally, it is clearer to be able to specify a right-hand table name on the left-hand side of the condition, and vice versa. To support this, column names can be prefixed by x$ or y$ to explicitly specify which table they come from.

Examples

```
sales <- tibble(
  id = c(1L, 1L, 1L, 2L, 2L),
 sale_date = as.Date(c("2018-12-31", "2019-01-02", "2019-01-05", "2019-01-04", "2019-01-01"))
)
sales

promos <- tibble(
  id = c(1L, 1L, 2L),
  promo_date = as.Date(c("2019-01-01", "2019-01-05", "2019-01-02"))
)
promos

# Match `id` to `id`, and `sale_date` to `promo_date`
by <- join_by(id, sale_date == promo_date)
left_join(sales, promos, by)

# For each `sale_date` within a particular `id`,
# find all `promo_date`s that occurred before that particular sale
by <- join_by(id, sale_date >= promo_date)
left_join(sales, promos, by)
```

```
# For each `sale_date` within a particular `id`,
# find only the closest `promo_date` that occurred before that sale
by <- join_by(id, closest(sale_date >= promo_date))
left_join(sales, promos, by)

# If you want to disallow exact matching in rolling joins, use `>` rather
# than `>=`. Note that the promo on `2019-01-05` is no longer considered the
# closest match for the sale on the same date.
by <- join_by(id, closest(sale_date > promo_date))
left_join(sales, promos, by)

# Same as before, but also require that the promo had to occur at most 1
# day before the sale was made. We'll use a full join to see that id 2's
# promo on `2019-01-02` is no longer matched to the sale on `2019-01-04`.
sales <- mutate(sales, sale_date_lower = sale_date - 1)
by <- join_by(id, closest(sale_date >= promo_date), sale_date_lower <= promo_date)
full_join(sales, promos, by)

# ---------------------------------------------------------------------------

segments <- tibble(
  segment_id = 1:4,
  chromosome = c("chr1", "chr2", "chr2", "chr1"),
  start = c(140, 210, 380, 230),
  end = c(150, 240, 415, 280)
)
segments

reference <- tibble(
  reference_id = 1:4,
  chromosome = c("chr1", "chr1", "chr2", "chr2"),
  start = c(100, 200, 300, 415),
  end = c(150, 250, 399, 450)
)
reference

# Find every time a segment `start` falls between the reference
# `[start, end]` range.
by <- join_by(chromosome, between(start, start, end))
full_join(segments, reference, by)

# If you wanted the reference columns first, supply `reference` as `x`
# and `segments` as `y`, then explicitly refer to their columns using `x$`
# and `y$`.
by <- join_by(chromosome, between(y$start, x$start, x$end))
full_join(reference, segments, by)

# Find every time a segment falls completely within a reference.
# Sometimes using `x$` and `y$` makes your intentions clearer, even if they
# match the default behavior.
by <- join_by(chromosome, within(x$start, x$end, y$start, y$end))
inner_join(segments, reference, by)
```

```
# Find every time a segment overlaps a reference in any way.
by <- join_by(chromosome, overlaps(x$start, x$end, y$start, y$end))
full_join(segments, reference, by)

# It is common to have right-open ranges with bounds like `[)`, which would
# mean an end value of `415` would no longer overlap a start value of `415`.
# Setting `bounds` allows you to compute overlaps with those kinds of ranges.
by <- join_by(chromosome, overlaps(x$start, x$end, y$start, y$end, bounds = "[)"))
full_join(segments, reference, by)
```

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 = NULL, order_by = NULL, ...)

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

Arguments

x	A vector
n	Positive integer of length 1, giving the number of positions to lag or lead by
default	The value used to pad x back to its original size after the lag or lead has been applied. The default, NULL, pads with a missing value. If supplied, this must be a vector with size 1, which will be cast to the type of x.
order_by	An optional secondary vector that defines the ordering to use when applying the lag or lead to x. If supplied, this must be the same size as x.
...	Not used.

Value

A vector with the same type and size as x.

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 to pad with, use `default`
lag(1:5)
lag(1:5, default = 0)

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

# If the 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() creates new columns that are functions of existing variables. It can also modify (if the name is the same as an existing column) and delete columns (by setting their value to NULL).

Usage

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

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

`...` <`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.

`.by` <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.

`.keep` Control which columns from `.data` are retained in the output. Grouping columns and columns created by `...` are always kept.

- `"all"` retains all columns from `.data`. This is the default.
- `"used"` retains only the columns used in `...` to create new columns. This is useful for checking your work, as it displays inputs and outputs side-by-side.
- `"unused"` retains only the columns *not* used in `...` to create new columns. This is useful if you generate new columns, but no longer need the columns used to generate them.
- `"none"` doesn't retain any extra columns from `.data`. Only the grouping variables and columns created by `...` are kept.

`.before, .after` <`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:

- Columns from `.data` will be preserved according to the `.keep` argument.
- Existing columns that are modified by `...` will always be returned in their original location.
- New columns created through `...` will be placed according to the `.before` and `.after` arguments.
- The number of rows is not affected.
- Columns given the 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

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.

Methods available in currently loaded packages: no methods found.

See Also

Other single table verbs: arrange(), filter(), reframe(), 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.
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.
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")

# 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 ?rlang::args_data_masking
```

mutate-joins *Mutating joins*

Description

Mutating joins add columns from y to x, matching observations based on the keys. There are four mutating joins: the inner join, and the three outer joins.

Inner join:

An inner_join() only keeps observations from x that have a matching key in y.

The most important property of an inner join is that unmatched rows in either input are not included in the result. This means that generally inner joins are not appropriate in most analyses, because it is too easy to lose observations.

Outer joins:

The three outer joins keep observations that appear in at least one of the data frames:

- A left_join() keeps all observations in x.
- A right_join() keeps all observations in y.
- A full_join() keeps all observations in x and y.

Usage

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

## S3 method for class 'data.frame'
inner_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(".x", ".y"),
  ...,
  keep = NULL,
  na_matches = c("na", "never"),
  multiple = "all",
  unmatched = "drop",
  relationship = NULL
)

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

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

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

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

full_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(”.x”, ”.y”),
  ...,
  keep = NULL
)

## S3 method for class 'data.frame'
full_join(
  x,
  y,
  by = NULL,
  copy = FALSE,
  suffix = c(”.x”, ”.y”),
  ...,
  keep = NULL,
  na_matches = c(”na”, ”never”),
  multiple = ”all”,
  relationship = 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 join specification created with `join_by()`, or 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 on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`.

To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`.

`join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`.

To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.

`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?

- If `NULL`, the default, joins on equality retain only the keys from `x`, while joins on inequality retain the keys from both inputs.
- If `TRUE`, all keys from both inputs are retained.
- If `FALSE`, only keys from `x` are retained. For right and full joins, the data in key columns corresponding to rows that only exist in `y` are merged into the key columns from `x`. Can't be used when joining on inequality conditions.

`na_matches` Should two `NA` or two `NaN` values match?

- `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`.
- `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.

`multiple` Handling of rows in `x` with multiple matches in `y`. For each row of `x`:

- `"all"`, the default, returns every match detected in `y`. This is the same behavior as SQL.
- `"any"` returns one match detected in `y`, with no guarantees on which match will be returned. It is often faster than `"first"` and `"last"` if you just need to detect if there is at least one match.
- `"first"` returns the first match detected in `y`.
- `"last"` returns the last match detected in `y`.

`unmatched` How should unmatched keys that would result in dropped rows be handled?

- `"drop"` drops unmatched keys from the result.
- `"error"` throws an error if unmatched keys are detected.

`unmatched` is intended to protect you from accidentally dropping rows during a join. It only checks for unmatched keys in the input that could potentially drop rows.

- For left joins, it checks `y`.
- For right joins, it checks `x`.

- For inner joins, it checks both `x` and `y`. In this case, `unmatched` is also allowed to be a character vector of length 2 to specify the behavior for `x` and `y` independently.

`relationship` Handling of the expected relationship between the keys of `x` and `y`. If the expectations chosen from the list below are invalidated, an error is thrown.

- `NULL`, the default, doesn't expect there to be any relationship between `x` and `y`. However, for equality joins it will check for a many-to-many relationship (which is typically unexpected) and will warn if one occurs, encouraging you to either take a closer look at your inputs or make this relationship explicit by specifying `"many-to-many"`.
  See the *Many-to-many relationships* section for more details.
- `"one-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
  - Each row in `y` matches at most 1 row in `x`.
- `"one-to-many"` expects:
  - Each row in `y` matches at most 1 row in `x`.
- `"many-to-one"` expects:
  - Each row in `x` matches at most 1 row in `y`.
- `"many-to-many"` doesn't perform any relationship checks, but is provided to allow you to be explicit about this relationship if you know it exists.

`relationship` doesn't handle cases where there are zero matches. For that, see `unmatched`.

Value

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

- The rows are affect by the join type.
  - `inner_join()` returns matched `x` rows.
  - `left_join()` returns all `x` rows.
  - `right_join()` returns matched of `x` rows, followed by unmatched `y` rows.
  - `full_join()` returns all `x` rows, followed by unmatched `y` rows.
- Output columns include all columns from `x` and all non-key columns from `y`. If `keep = TRUE`, the key columns from `y` are included as well.
- If non-key columns in `x` and `y` have the same name, `suffix`es are added to disambiguate. If `keep = TRUE` and key columns in `x` and `y` have the same name, `suffix`es are added to disambiguate these as well.
- If `keep = FALSE`, output columns included in `by` are coerced to their common type between `x` and `y`.

Many-to-many relationships

By default, dplyr guards against many-to-many relationships in equality joins by throwing a warning. These occur when both of the following are true:

- A row in x matches multiple rows in y.
- A row in y matches multiple rows in x.

This is typically surprising, as most joins involve a relationship of one-to-one, one-to-many, or many-to-one, and is often the result of an improperly specified join. Many-to-many relationships are particularly problematic because they can result in a Cartesian explosion of the number of rows returned from the join.

If a many-to-many relationship is expected, silence this warning by explicitly setting `relationship = "many-to-many"`.

In production code, it is best to preemptively set `relationship` to whatever relationship you expect to exist between the keys of x and y, as this forces an error to occur immediately if the data doesn't align with your expectations.

Inequality joins typically result in many-to-many relationships by nature, so they don't warn on them by default, but you should still take extra care when specifying an inequality join, because they also have the capability to return a large number of rows.

Rolling joins don't warn on many-to-many relationships either, but many rolling joins follow a many-to-one relationship, so it is often useful to set `relationship = "many-to-one"` to enforce this.

Note that in SQL, most database providers won't let you specify a many-to-many relationship between two tables, instead requiring that you create a third *junction table* that results in two one-to-many relationships instead.

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: `cross_join()`, `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 = join_by(name))
```

```
# This is good practice in production code

# Use an equality expression if the join variables have different names
band_members |> full_join(band_instruments2, by = join_by(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 = join_by(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)

# If a row in `y` also matches multiple rows in `x`, this is known as a
# many-to-many relationship, which is typically a result of an improperly
# specified join or some kind of messy data. In this case, a warning is
# thrown by default:
df3 <- tibble(x = c(1, 1, 1, 3))
df3 |> left_join(df2)

# In the rare case where a many-to-many relationship is expected, set
# `relationship = "many-to-many"` to silence this warning
df3 |> left_join(df2, relationship = "many-to-many")

# Use `join_by()` with a condition other than `==` to perform an inequality
# join. Here we match on every instance where `df1$x > df2$x`.
df1 |> left_join(df2, join_by(x > x))

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

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

na_if *Convert values to* NA

Description

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

Usage

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

Arguments

x Vector to modify

y Value or vector to compare against. When x and y are equal, the value in x will be replaced with NA.

y is cast to the type of x before comparison.

y is recycled to the size of x before comparison. This means that y can be a vector with the same size as x, but most of the time this will be a single value.

Value

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

See Also

- coalesce() to replace NAs with the first non-missing value.
- replace_values() for making arbitrary replacements by value.
- replace_when() for making arbitrary replacements using logical conditions.

Examples

```
# `na_if()` is useful for replacing a single problematic value with `NA`
na_if(c(-99, 1, 4, 3, -99, 5), -99)
na_if(c("abc", "def", "", "ghi"), "")

# You can use it to standardize `NaN`s to `NA`
na_if(c(1, NaN, NA, 2, NaN), NaN)

# Because `na_if()` is an R translation of SQL's `NULLIF` command,
# it compares `x` and `y` element by element. Where `x` and `y` are
# equal, the value in `x` is replaced with an `NA`.
na_if(
  x = c(1, 2, 5, 5, 6),
  y = c(0, 2, 3, 5, 4)
)

# If you have multiple problematic values that you'd like to replace with
# `NA`, then `replace_values()` is a better choice than `na_if()`
x <- c(-99, 1, 4, 0, -99, 5, -1, 0, 5)
replace_values(x, c(0, -1, -99) ~ NA)

# You'd have to nest `na_if()`s to achieve this
try(na_if(x, c(0, -1, -99)))
na_if(na_if(na_if(x, 0), -1), -99)

# If you'd like to replace values that match a logical condition with `NA`,
# use `replace_when()`
replace_when(x, x < 0 ~ NA)

# If you'd like to replace `NA` with some other value, use `replace_values()`
x <- c(NA, 5, 2, NA, 0, 3)
```

```
replace_values(x, NA ~ 0)

# `na_if()` is particularly useful inside `mutate()`
starwars |>
  select(name, eye_color) |>
  mutate(eye_color = na_if(eye_color, "unknown"))

# `na_if()` can also be used with `mutate()` and `across()`
# to alter 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

A nest join leaves x almost unchanged, except that it adds a new list-column, where each element contains the rows from y that match the corresponding row in x.

Usage

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

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

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 join specification created with `join_by()`, or 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 on different variables between `x` and `y`, use a `join_by()` specification. For example, `join_by(a == b)` will match `x$a` to `y$b`.

To join by multiple variables, use a `join_by()` specification with multiple expressions. For example, `join_by(a == b, c == d)` will match `x$a` to `y$b` and `x$c` to `y$d`. If the column names are the same between `x` and `y`, you can shorten this by listing only the variable names, like `join_by(a, c)`.

`join_by()` can also be used to perform inequality, rolling, and overlap joins. See the documentation at ?join_by for details on these types of joins.

For simple equality joins, you can alternatively specify a character vector of variable names to join by. For example, `by = c("a", "b")` joins `x$a` to `y$a` and `x$b` to `y$b`. If variable names differ between `x` and `y`, use a named character vector like `by = c("x_a" = "y_a", "x_b" = "y_b")`.

To perform a cross-join, generating all combinations of `x` and `y`, see `cross_join()`.

`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 new list-column contain join keys? The default will preserve the join keys for inequality joins.

`name` The name of the list-column created by the join. If `NULL`, the default, the name of `y` is used.

`...` Other parameters passed onto methods.

`na_matches` Should two `NA` or two `NaN` values match?

- `"na"`, the default, treats two `NA` or two `NaN` values as equal, like `%in%`, `match()`, and `merge()`.
- `"never"` treats two `NA` or two `NaN` values as different, and will never match them together or to any other values. This is similar to joins for database sources and to `base::merge(incomparables = NA)`.

`unmatched` How should unmatched keys that would result in dropped rows be handled?

- `"drop"` drops unmatched keys from the result.
- `"error"` throws an error if unmatched keys are detected.

`unmatched` is intended to protect you from accidentally dropping rows during a join. It only checks for unmatched keys in the input that could potentially drop rows.

- For left joins, it checks `y`.
- For right joins, it checks `x`.
- For inner joins, it checks both `x` and `y`. In this case, `unmatched` is also allowed to be a character vector of length 2 to specify the behavior for `x` and `y` independently.

Value

The output:

- Is same type as `x` (including having the same groups).
- Has exactly the same number of rows as `x`.
- Contains all the columns of `x` in the same order with the same values. They are only modified (slightly) if `keep = FALSE`, when columns listed in `by` will be coerced to their common type across `x` and `y`.
- Gains one new column called `{name}` on the far right, a list column containing data frames the same type as `y`.

Relationship to other joins

You can recreate many other joins from the result of a nest join:

- `inner_join()` is a `nest_join()` plus `tidyr::unnest()`.
- `left_join()` is a `nest_join()` plus `tidyr::unnest(keep_empty = TRUE)`.
- `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 that 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: `cross_join()`, `filter-joins`, `mutate-joins`

Examples

```
df1 <- tibble(x = 1:3)
df2 <- tibble(x = c(2, 3, 3), y = c(”a”, ”b”, ”c”))

out <- nest_join(df1, df2)
out
out$df2
```

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

Description

These are useful helpers for extracting a single value from a vector. They are guaranteed to return a meaningful value, even when the input is shorter than expected. You can also provide an optional secondary vector that defines the ordering.

Usage

```
nth(x, n, order_by = NULL, default = NULL, na_rm = FALSE)

first(x, order_by = NULL, default = NULL, na_rm = FALSE)

last(x, order_by = NULL, default = NULL, na_rm = FALSE)
```

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).

`order_by` An optional vector the same size as `x` used to determine the order.

`default` A default value to use if the position does not exist in `x`.

If `NULL`, the default, a missing value is used.

If supplied, this must be a single value, which will be cast to the type of `x`.

When `x` is a list , `default` is allowed to be any value. There are no type or size restrictions in this case.

`na_rm` Should missing values in `x` be removed before extracting the value?

Details

For most vector types, `first(x)`, `last(x)`, and `nth(x, n)` work like `x[[1]]`, `x[[length(x)]`, and `x[[n]]`, respectively. The primary exception is data frames, where they instead retrieve rows, i.e. `x[1, ]`, `x[nrow(x), ]`, and `x[n, ]`. This is consistent with the tidyverse/vctrs principle which treats data frames as a vector of rows, rather than a vector of columns.

Value

If x is a list, a single element from that list. Otherwise, a vector the same type as x with size 1.

Examples

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

first(x)
last(y)

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

# `first()` and `last()` are often useful in `summarise()`
df <- tibble(x = x, y = y)
df |>
  summarise(
    across(x:y, first, .names = "{col}_first"),
    y_last = last(y)
  )

# Selecting a position that is out of bounds returns a default value
nth(x, 11)
nth(x, 0)

# This out of bounds behavior also applies to empty vectors
first(integer())

# You can customize the default value with `default`
nth(x, 11, default = -1L)
first(integer(), default = 0L)

# `order_by` provides optional ordering
last(x)
last(x, order_by = y)

# `na_rm` removes missing values before extracting the value
z <- c(NA, NA, 1, 3, NA, 5, NA)
first(z)
first(z, na_rm = TRUE)
last(z, na_rm = TRUE)
nth(z, 3, na_rm = TRUE)

# For data frames, these select entire rows
df <- tibble(a = 1:5, b = 6:10)
first(df)
nth(df, 4)
```

ntile — *Bucket a numeric vector into* n *groups*

Description

ntile() is a sort of very rough rank, which breaks the input vector into n buckets. If length(x) is not an integer multiple of n, the size of the buckets will differ by up to one, with larger buckets coming first.

Unlike other ranking functions, ntile() ignores ties: it will create evenly sized buckets even if the same value of x ends up in different buckets.

Usage

```
ntile(x = row_number(), n)
```

Arguments

x — A vector to rank

By default, the smallest values will get the smallest ranks. Use desc() to reverse the direction so the largest values get the smallest ranks.

Missing values will be given rank NA. Use coalesce(x, Inf) or coalesce(x, -Inf) if you want to treat them as the largest or smallest values respectively.

To rank by multiple columns at once, supply a data frame.

n — Number of groups to bucket into

See Also

Other ranking functions: percent_rank(), row_number()

Examples

```
x <- c(5, 1, 3, 2, 2, NA)
ntile(x, 2)
ntile(x, 4)

# If the bucket sizes are uneven, the larger buckets come first
ntile(1:8, 3)

# Ties are ignored
ntile(rep(1, 8), 3)
```

n_distinct *Count unique combinations*

Description

`n_distinct()` counts the number of unique/distinct combinations in a set of one or more vectors. It's a faster and more concise equivalent to `nrow(unique(data.frame(...)))`.

Usage

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

Arguments

`...` Unnamed vectors. If multiple vectors are supplied, then they should have the same length.

`na.rm` If `TRUE`, exclude missing observations from the count. If there are multiple vectors in `...`, an observation will be excluded if *any* of the values are missing.

Value

A single number.

Examples

```
x <- c(1, 1, 2, 2, 2)
n_distinct(x)

y <- c(3, 3, NA, 3, 3)
n_distinct(y)
n_distinct(y, na.rm = TRUE)

# Pairs (1, 3), (2, 3), and (2, NA) are distinct
n_distinct(x, y)

# (2, NA) is dropped, leaving 2 distinct combinations
n_distinct(x, y, na.rm = TRUE)

# Also works with data frames
n_distinct(data.frame(x, y))
```

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)
```

percent_rank *Proportional ranking functions*

Description

These two ranking functions implement two slightly different ways to compute a percentile. For each `x_i` in `x`:

- `cume_dist(x)` counts the total number of values less than or equal to `x_i`, and divides it by the number of observations.
- `percent_rank(x)` counts the total number of values less than `x_i`, and divides it by the number of observations minus 1.

In both cases, missing values are ignored when counting the number of observations.

Usage

```
percent_rank(x)

cume_dist(x)
```

Arguments

`x` A vector to rank

By default, the smallest values will get the smallest ranks. Use `desc()` to reverse the direction so the largest values get the smallest ranks.

Missing values will be given rank `NA`. Use `coalesce(x, Inf)` or `coalesce(x, -Inf)` if you want to treat them as the largest or smallest values respectively.

To rank by multiple columns at once, supply a data frame.

Value

A numeric vector containing a proportion.

See Also

Other ranking functions: `ntile()`, `row_number()`

Examples

```
x <- c(5, 1, 3, 2, 2)

cume_dist(x)
percent_rank(x)

# You can understand what's going on by computing it by hand
sapply(x, function(xi) sum(x <= xi) / length(x))
sapply(x, function(xi) sum(x < xi)  / (length(x) - 1))
```

```
# The real computations are a little more complex in order to
# correctly deal with missing values
```

pick   *Select a subset of columns*

Description

pick() provides a way to easily select a subset of columns from your data using select() semantics while inside a "data-masking" function like mutate() or summarise(). pick() returns a data frame containing the selected columns for the current group.

pick() is complementary to across():

- With pick(), you typically apply a function to the full data frame.
- With across(), you typically apply a function to each column.

Usage

```
pick(...)
```

Arguments

...	<tidy-select> Columns to pick. You can't pick grouping columns because they are already automatically handled by the verb (i.e. summarise() or mutate()).

Details

Theoretically, pick() is intended to be replaceable with an equivalent call to tibble(). For example, pick(a, c) could be replaced with tibble(a = a, c = c), and pick(everything()) on a data frame with cols a, b, and c could be replaced with tibble(a = a, b = b, c = c). pick() specially handles the case of an empty selection by returning a 1 row, 0 column tibble, so an exact replacement is more like:

```
size <- vctrs::vec_size_common(..., .absent = 1L)
out <- vctrs::vec_recycle_common(..., .size = size)
tibble::new_tibble(out, nrow = size)
```

Value

A tibble containing the selected columns for the current group.

See Also

across()

Examples

```
df <- tibble(
  x = c(3, 2, 2, 2, 1),
  y = c(0, 2, 1, 1, 4),
  z1 = c("a", "a", "a", "b", "a"),
  z2 = c("c", "d", "d", "a", "c")
)
df

# `pick()` provides a way to select a subset of your columns using
# tidyselect. It returns a data frame.
df |> mutate(cols = pick(x, y))

# This is useful for functions that take data frames as inputs.
# For example, you can compute a joint rank between `x` and `y`.
df |> mutate(rank = dense_rank(pick(x, y)))

# `pick()` is also useful as a bridge between data-masking functions (like
# `mutate()` or `group_by()`) and functions with tidy-select behavior (like
# `select()`). For example, you can use `pick()` to create a wrapper around
# `group_by()` that takes a tidy-selection of columns to group on. For more
# bridge patterns, see
# https://rlang.r-lib.org/reference/topic-data-mask-programming.html#bridge-patterns.
my_group_by <- function(data, cols) {
  group_by(data, pick({{ cols }}))
}

df |> my_group_by(c(x, starts_with("z")))

# Or you can use it to dynamically select columns to `count()` by
df |> count(pick(starts_with("z")))
```

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
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)
```

`recode` *Recode values*

Description

[Superseded]

`recode()` is superseded in favor of `recode_values()` and `replace_values()`, which are more general and have a much better interface. `recode_factor()` is also superseded, however, its direct replacement is not currently available but will eventually live in forcats. For creating new variables based on logical vectors, use `if_else()`. For even more complicated criteria, use `case_when()`.

`recode()` 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. 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.

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

`recode_values()`

Examples

```
set.seed(1234)

x <- sample(c("a", "b", "c"), 10, replace = TRUE)

# `recode()` is superseded by `recode_values()` and `replace_values()`

# If you are fully recoding a vector use `recode_values()`
recode(x, a = "Apple", b = "Banana", .default = NA_character_)
recode_values(x, "a" ~ "Apple", "b" ~ "Banana")

# With a default
recode(x, a = "Apple", b = "Banana", .default = "unknown")
recode_values(x, "a" ~ "Apple", "b" ~ "Banana", default = "unknown")

# If you are partially updating a vector and want to retain the original
# vector's values in locations you don't make a replacement, use
# `replace_values()`
recode(x, a = "Apple", b = "Banana")
replace_values(x, "a" ~ "Apple", "b" ~ "Banana")

# `replace_values()` is easier to use with numeric vectors, because you don't
# need to turn the numeric values into names
y <- c(1:4, NA)
recode(y, `2` = 20L, `4` = 40L)
replace_values(y, 2 ~ 20L, 4 ~ 40L)

# `recode()` is particularly confusing because it tries to handle both
# full recodings to new vector types and partial updating of an existing
# vector. With the above example, using doubles (20) rather than integers
# (20L) results in a warning from `recode()`, because it thinks you are
# doing a full recode and missed a case. `replace_values()` is type stable
# on `y` and will instead coerce the double values to integer.
recode(y, `2` = 20, `4` = 40)
replace_values(y, 2 ~ 20, 4 ~ 40)

# This also makes `replace_values()` much safer. If you provide
# incompatible types, it will error.
recode(y, `2` = "20", `4` = "40")
try(replace_values(y, 2 ~ "20", 4 ~ "40"))

# If you were trying to fully recode the vector and want a different output
# type, use `recode_values()`
recode_values(y, 2 ~ "20", 4 ~ "40")

# And if you want to ensure you don't miss a case, use `unmatched`, which
# errors rather than warns
try(recode_values(y, 2 ~ "20", 4 ~ "40", unmatched = "error"))

# ----------------------------------------------------------------------------
# Lookup tables
# If you were splicing an external lookup vector into `recode()`, you can
# instead use the `from` and `to` arguments of `recode_values()`
x <- c("a", "b", "a", "c", "d", "c")

lookup <- c(
  "a" = "A",
  "b" = "B",
  "c" = "C",
  "d" = "D"
)

recode(x, !!!lookup)
recode_values(x, from = names(lookup), to = unname(lookup))

# `recode_values()` is much more flexible here because the lookup table
# isn't restricted to just character values. We recommend using `tribble()`
# to build your lookup tables.
lookup <- tribble(
  ~from, ~to,
  "a", 1,
  "b", 2,
  "c", 3,
  "d", 4
)

recode_values(x, from = lookup$from, to = lookup$to)

# ---------------------------------------------------------------------------
# Factors

# The factor method of `recode()` can generally be replaced with
# `forcats::fct_recode()`
x <- factor(c("a", "b", "c"))
recode(x, a = "Apple")
# forcats::fct_recode(x, "Apple" = "a")

# `recode_factor()` does not currently have a direct replacement, but we
# plan to add one to forcats. In the meantime, use a lookup table that
# recodes every case, and then convert the `to` column to a factor. If you
# define your lookup table in your preferred level order, then the conversion
# to factor is straightforward!
y <- c(3, 4, 1, 2, 4, NA)

recode_factor(
  y,
  `1` = "a",
  `2` = "b",
  `3` = "c",
  `4` = "d",
  .missing = "M"
)

lookup <- tribble(
  ~from, ~to,
  1, "a",
  2, "b",
  3, "c",
  4, "d",
  NA, "M"
)
# `factor()` generates levels by sorting the unique values of `to`, which we
# don't want, so we supply `levels = to` directly. Alternatively, use
# `forcats::fct(to)`, which generates levels in order of appearance.
lookup <- mutate(lookup, to = factor(to, levels = to))

recode_values(y, from = lookup$from, to = lookup$to)
```

recode-and-replace-values

Recode and replace values

Description

recode_values() and replace_values() provide two ways to map old values to new values. They work by matching values against x and using the first match to determine the corresponding value in the output vector. You can also think of these functions as a way to use a lookup table to recode a vector.

- Use recode_values() when creating an entirely new vector.
- Use replace_values() when partially updating an existing vector.

If you are just replacing a few values within an existing vector, then replace_values() is always a better choice because it is type stable and better expresses intent.

A major difference between the two functions is what happens when no cases match:

- recode_values() falls through to a default.
- replace_values() retains the original values from x.

These functions have two mutually exclusive ways to use them:

- A formula-based approach, i.e. recode_values(x, from1 ~ to1, from2 ~ to2), similar to case_when(), which is useful when you have a small number of cases.
- A vector-based approach, i.e. recode_values(x, from = from, to = to), which is useful when you have a pre-built lookup table (which may come from an external source, like a CSV file).

See vignette("recoding-replacing") for more examples.

Usage

```
recode_values(
  x,
  ...,
  from = NULL,
  to = NULL,
  default = NULL,
  unmatched = "default",
  ptype = NULL
)

replace_values(x, ..., from = NULL, to = NULL)
```

Arguments

`x` A vector.

`...` <`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 inputs can be any size, but will be cast to the type of `x`.
- The RHS inputs will be recycled to the same size as `x`. For `recode_values()` they will be cast to their common type, and for `replace_values()` they will be cast to the type of `x`.

`NULL` inputs are ignored.

Mutually exclusive with `from` and `to`.

`from` Values to look up in `x` and map to values in `to`.

Typically this is a single vector of any size that is cast to the type of `x`. For more advanced usage, this can be a list of vectors of any size each of which are cast to the type of `x`.

Mutually exclusive with `...`.

`to` Values that `from` map to.

Typically this is a single vector that is recycled to the size of `from`. For more advanced usage, this can be a list of vectors each of which are recycled to the size of `x`.

Mutually exclusive with `...`.

`default` Default value to use when there is a value present in `x` that is unmatched by a value in `from`.

By default, a missing value is used as the default value.

If supplied, will be recycled to the size of `x`.

Can only be set when `unmatched = "default"`.

`unmatched` Handling of unmatched locations.

One of:

- `"default"` to use `default` in unmatched locations.
- `"error"` to error when there are unmatched locations.

`ptype` An optional override for the output type, which is usually computed as the common type of `to` and `default`.

Value

A vector the same size as x.

- For recode_values(), the type of the output is computed as the common type of to and default, unless overridden by ptype. The names of the output come from the names of to and default.
- For replace_values(), the type of the output will have the same type as x. The names of the output will be the same as the names of x.

See Also

case_when(), vctrs::vec_recode_values()

Examples

```
x <- c("NC", "NYC", "CA", NA, "NYC", "Unknown")

# `recode_values()` is useful for fully recoding from one set of values to
# another, creating an entirely new vector in the process. Note that any
# unmatched values result in `NA`, or a `default` value.
recode_values(
  x,
  "NC" ~ "North Carolina",
  "NYC" ~ "New York",
  "CA" ~ "California"
)

recode_values(
  x,
  "NC" ~ "North Carolina",
  "NYC" ~ "New York",
  "CA" ~ "California",
  default = "<not recorded>"
)

# `replace_values()` is useful for updating an existing vector, tweaking a
# few values along the way
replace_values(x, "NYC" ~ "NY")

# `replace_values()` is particularly nice for replacing `NA`s with values...
replace_values(x, NA ~ "Unknown (NA)")
# ...or values with `NA`s
replace_values(x, "Unknown" ~ NA)

# Multiple values can be grouped within a single left-hand side to normalize
# all problematic values at once
replace_values(x, c(NA, "Unknown") ~ "<not recorded>")

# ---------------------------------------------------------------------------
# Lookup tables
# `recode_values()` works with more than just character vectors. Imagine you
# have this series of Likert Scale scores, which is a scoring system that is
# ordered from 1-5.
data <- tibble(
  score = c(1, 2, 3, 4, 5, 2, 3, 1, 4)
)

# To recode each `score` to its corresponding Likert Score label, you may
# initially be inclined to reach for `case_when()`
data |>
  mutate(
    score = case_when(
      score == 1 ~ "Strongly disagree",
      score == 2 ~ "Disagree",
      score == 3 ~ "Neutral",
      score == 4 ~ "Agree",
      score == 5 ~ "Strongly agree"
    )
  )

# While this works, it can be written more efficiently using
# `recode_values()`
data |>
  mutate(
    score = score |>
      recode_values(
        1 ~ "Strongly disagree",
        2 ~ "Disagree",
        3 ~ "Neutral",
        4 ~ "Agree",
        5 ~ "Strongly agree"
      )
  )

# `recode_values()` actually has two mutually exclusive APIs. The formula API
# used above, which is like `case_when()`, and a lookup style API that uses
# `from` and `to` arguments. The lookup API is even better suited for this
# problem, because we can move the mapping outside of the `mutate()` call
# into a standalone lookup table. You could even imagine reading this
# `likert` lookup table in from a separate CSV file.
likert <- tribble(
  ~from, ~to,
  1, "Strongly disagree",
  2, "Disagree",
  3, "Neutral",
  4, "Agree",
  5, "Strongly agree"
)

data |>
  mutate(score = recode_values(score, from = likert$from, to = likert$to))

# You can utilize the same lookup table across multiple columns by using
# `across()`
data_months <- tibble(
  score_january = c(1, 2, 3, 4, 5, 2, 3, 1, 4),
  score_february = c(4, 2, 1, 2, 1, 5, 2, 4, 4)
)

data_months |>
  mutate(across(
    starts_with("score"),
    ~ recode_values(.x, from = likert$from, to = likert$to)
  ))

# The `unmatched` argument allows you to assert that you believe that you've
# recoded all of the cases and will error if you've missed one, adding an
# extra layer of safety
data_with_zero <- add_row(data, score = 0)

try({
  recode_values(
    data_with_zero$score,
    from = likert$from,
    to = likert$to,
    unmatched = "error"
  )
})

# Note that missing values are considered unmatched. If you expect missing
# values, you'll need to handle them explicitly in your lookup table.
data_with_missing <- add_row(data, score = NA)

try({
  recode_values(
    data_with_missing$score,
    from = likert$from,
    to = likert$to,
    unmatched = "error"
  )
})

likert <- add_row(likert, from = NA, to = NA)

recode_values(
  data_with_missing$score,
  from = likert$from,
  to = likert$to,
  unmatched = "error"
)

# ---------------------------------------------------------------------------
# Lists of vectors

# In some cases, your mapping may collapse multiple groups together into a
# single value. For example, here we'd like to standardize the school names.
```

```
schools <- c(
  "UNC",
  "Chapel Hill",
  NA,
  "Duke",
  "Duke University",
  "UNC",
  "NC State",
  "ECU",
  "East Carolina"
)

# This `tribble()` is more complex than it may appear, it actually
# creates a list column!
standardized <- tribble(
  ~from,                        ~to,
  c("UNC", "Chapel Hill"),      "UNC",
  c("Duke", "Duke University"), "Duke",
  c("NC State"),                "NC State",
  c("ECU", "East Carolina"),    "ECU",
  NA,                           NA
)

standardized
standardized$from

# `recode_values()` treats a list `from` value as a list of vectors, where
# any match within one of the vectors is mapped to its corresponding `to`
# value
recode_values(
  schools,
  from = standardized$from,
  to = standardized$to,
  unmatched = "error"
)

# This formula based approach is equivalent, but the lookup based approach is
# nicer because the lookup table can be defined separately
recode_values(
  schools,
  c("UNC", "Chapel Hill") ~ "UNC",
  c("Duke", "Duke University") ~ "Duke",
  c("NC State") ~ "NC State",
  c("ECU", "East Carolina") ~ "ECU",
  NA ~ NA,
  unmatched = "error"
)
```

reframe *Transform each group to an arbitrary number of rows*

Description

While `summarise()` requires that each argument returns a single value, and `mutate()` requires that each argument returns the same number of rows as the input, `reframe()` is a more general workhorse with no requirements on the number of rows returned per group.

`reframe()` creates a new data frame by applying functions to columns of an existing data frame. It is most similar to `summarise()`, with two big differences:

- `reframe()` can return an arbitrary number of rows per group, while `summarise()` reduces each group down to a single row.
- `reframe()` always returns an ungrouped data frame, while `summarise()` might return a grouped or rowwise data frame, depending on the scenario.

We expect that you'll use `summarise()` much more often than `reframe()`, but `reframe()` can be particularly helpful when you need to apply a complex function that doesn't return a single summary value.

Usage

```
reframe(.data, ..., .by = 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 functions. The name will be the name of the variable in the result. The value can be a vector of any length.

Unnamed data frame values add multiple columns from a single expression.

`.by` <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.

Value

If `.data` is a tibble, a tibble. Otherwise, a data.frame.

- The rows originate from the underlying grouping keys.
- The columns are a combination of the grouping keys and the expressions that you provide.
- The output is always ungrouped.
- Data frame attributes are **not** preserved, because `reframe()` fundamentally creates a new data frame.

Connection to tibble

`reframe()` is theoretically connected to two functions in tibble, `tibble::enframe()` and `tibble::deframe()`:

- `enframe()`: vector -> data frame
- `deframe()`: data frame -> vector
- `reframe()`: data frame -> data frame

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()`, `summarise()`

Examples

```
table <- c("a", "b", "d", "f")

df <- tibble(
  g = c(1, 1, 1, 2, 2, 2, 2),
  x = c("e", "a", "b", "c", "f", "d", "a")
)

# `reframe()` allows you to apply functions that return
# an arbitrary number of rows
df |>
  reframe(x = intersect(x, table))

# Functions are applied per group, and each group can return a
# different number of rows.
df |>
  reframe(x = intersect(x, table), .by = g)

# The output is always ungrouped, even when using `group_by()`
df |>
  group_by(g) |>
  reframe(x = intersect(x, table))

# You can add multiple columns at once using a single expression by returning
# a data frame.
quantile_df <- function(x, probs = c(0.25, 0.5, 0.75)) {
  tibble(
    val = quantile(x, probs, na.rm = TRUE),
    quant = probs
  )
}

x <- c(10, 15, 18, 12)
quantile_df(x)

starwars |>
  reframe(quantile_df(height))

starwars |>
  reframe(quantile_df(height), .by = homeworld)
```

```
starwars |>
  reframe(
    across(c(height, mass), quantile_df, .unpack = TRUE),
    .by = homeworld
  )
```

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 and possibly renamed.
- 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()`, `reframe()`, `select()`, `slice()`, `summarise()`

Examples

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

# Rename using a named vector and `all_of()`
lookup <- c(pl = "Petal.Length", sl = "Sepal.Length")
rename(iris, all_of(lookup))

# If your named vector might contain names that don't exist in the data,
# use `any_of()` instead
lookup <- c(lookup, new = "unknown")
try(rename(iris, all_of(lookup)))
rename(iris, any_of(lookup))

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


# If your renaming function uses `paste0()`, make sure to set
# `recycle0 = TRUE` to ensure that empty selections are recycled correctly
try(rename_with(
  iris,
  ~ paste0("prefix_", .x),
  starts_with("nonexistent")
))

rename_with(
  iris,
  ~ paste0("prefix_", .x, recycle0 = TRUE),
  starts_with("nonexistent")
)
```

rows *Manipulate individual rows*

Description

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 typically 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`). By default, key values in `y` must not exist in `x`.
- `rows_append()` works like `rows_insert()` but ignores keys.
- `rows_update()` modifies existing rows (like `UPDATE`). Key values in `y` must be unique, and, by default, key values in `y` must exist 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`. Key values in `y` must be unique.
- `rows_delete()` deletes rows (like `DELETE`). By default, key values in `y` must exist in `x`.

Usage

```
rows_insert(
  x,
  y,
  by = NULL,
  ...,
  conflict = c("error", "ignore"),
  copy = FALSE,
  in_place = FALSE
)

rows_append(x, y, ..., copy = FALSE, in_place = FALSE)

rows_update(
  x,
  y,
  by = NULL,
  ...,
  unmatched = c("error", "ignore"),
  copy = FALSE,
  in_place = FALSE
)
```

```
rows_patch(
  x,
  y,
  by = NULL,
  ...,
  unmatched = c("error", "ignore"),
  copy = FALSE,
  in_place = FALSE
)

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

rows_delete(
  x,
  y,
  by = NULL,
  ...,
  unmatched = c("error", "ignore"),
  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 columns must exist in both `x` and `y`. Keys typically uniquely identify each row, but this is only enforced for the key values of `y` when `rows_update()`, `rows_patch()`, or `rows_upsert()` are used.

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.

`conflict` For `rows_insert()`, how should keys in `y` that conflict with keys in `x` be handled? A conflict arises if there is a key in `y` that already exists in `x`.

One of:

- `"error"`, the default, will error if there are any keys in `y` that conflict with keys in `x`.
- `"ignore"` will ignore rows in `y` with keys that conflict with keys in `x`.

`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.

`unmatched` For `rows_update()`, `rows_patch()`, and `rows_delete()`, how should keys in `y` that are unmatched by the keys in `x` be handled?

One of:

- `"error"`, the default, will error if there are any keys in `y` that are unmatched by the keys in `x`.
- `"ignore"` will ignore rows in `y` with keys that are unmatched by the keys in `x`.

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()` and `rows_patch()` preserve the number of rows; `rows_insert()`, `rows_append()`, 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.

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:

- `rows_insert()`: no methods found.
- `rows_append()`: no methods found.
- `rows_update()`: no methods found.
- `rows_patch()`: no methods found.
- `rows_upsert()`: no methods found.
- `rows_delete()`: no methods found.

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"))

# By default, if a key in `y` matches a key in `x`, then it can't be inserted
# and will throw an error. Alternatively, you can ignore rows in `y`
# containing keys that conflict with keys in `x` with `conflict = "ignore"`,
# or you can use `rows_append()` to ignore keys entirely.
```

```
try(rows_insert(data, tibble(a = 3, b = "z")))
rows_insert(data, tibble(a = 3, b = "z"), conflict = "ignore")
rows_append(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"))

# By default, for update, patch, and delete it is an error if a key in `y`
# doesn't exist in `x`. You can ignore rows in `y` that have unmatched keys
# with `unmatched = "ignore"`.
y <- tibble(a = 3:4, b = "z")
try(rows_update(data, y, by = "a"))
rows_update(data, y, by = "a", unmatched = "ignore")
rows_patch(data, y, by = "a", unmatched = "ignore")
rows_delete(data, y, by = "a", unmatched = "ignore")
```

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 computation
params |>
  rowwise(sim) |>
  reframe(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)), .groups = "keep")
```

row_number *Integer ranking functions*

Description

Three ranking functions inspired by SQL2003. They differ primarily in how they handle ties:

- `row_number()` gives every input a unique rank, so that `c(10, 20, 20, 30)` would get ranks `c(1, 2, 3, 4)`. It's equivalent to `rank(ties.method = "first")`.
- `min_rank()` gives every tie the same (smallest) value so that `c(10, 20, 20, 30)` gets ranks `c(1, 2, 2, 4)`. It's the way that ranks are usually computed in sports and is equivalent to `rank(ties.method = "min")`.
- `dense_rank()` works like `min_rank()`, but doesn't leave any gaps, so that `c(10, 20, 20, 30)` gets ranks `c(1, 2, 2, 3)`.

Usage

```
row_number(x)

min_rank(x)

dense_rank(x)
```

Arguments

x A vector to rank

By default, the smallest values will get the smallest ranks. Use `desc()` to reverse the direction so the largest values get the smallest ranks.

Missing values will be given rank `NA`. Use `coalesce(x, Inf)` or `coalesce(x, -Inf)` if you want to treat them as the largest or smallest values respectively.

To rank by multiple columns at once, supply a data frame.

Value

An integer vector.

See Also

Other ranking functions: `ntile()`, `percent_rank()`

Examples

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

```
# Ranking functions can be used in `filter()` to select top/bottom rows
df <- data.frame(
  grp = c(1, 1, 1, 2, 2, 2, 3, 3, 3),
  x = c(3, 2, 1, 1, 2, 2, 1, 1, 1),
  y = c(1, 3, 2, 3, 2, 2, 4, 1, 2),
  id = 1:9
)
# Always gives exactly 1 row per group
df |> group_by(grp) |> filter(row_number(x) == 1)
# May give more than 1 row if ties
df |> group_by(grp) |> filter(min_rank(x) == 1)
# Rank by multiple columns (to break ties) by selecting them with `pick()`
df |> group_by(grp) |> filter(min_rank(pick(x, y)) == 1)
# See slice_min() and slice_max() for another way to tackle the same problem

# You can use row_number() without an argument to refer to the "current"
# row number.
df |> group_by(grp) |> filter(row_number() == 1)

# It's easiest to see what this does with mutate():
df |> group_by(grp) |> mutate(grp_id = row_number())
```

`scoped` *Operate on a selection of variables*

Description

[Superseded]

Scoped verbs (`_if`, `_at`, `_all`) have been superseded by the use of `pick()` or `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 — *Keep or drop 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) or type (e.g. `where(is.numeric)` selects all numeric columns).

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.
- `group_cols()`: Select all grouping columns.

Other 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.

Or from variables stored in 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.

Or using a predicate 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()`. 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
#> # i 83 more rows

iris |> select(Sepal.Length)
#> # A tibble: 150 x 1
#>   Sepal.Length
#>          <dbl>
#> 1          5.1
#> 2          4.9
#> 3          4.7
#> 4          4.6
#> # i 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
#> # i 83 more rows

iris |> select(Sepal.Length, Petal.Length)
#> # A tibble: 150 x 2
#>   Sepal.Length Petal.Length
#>          <dbl>        <dbl>
#> 1          5.1          1.4
#> 2          4.9          1.4
#> 3          4.7          1.3
#> 4          4.6          1.5
#> # i 146 more rows
```

If you use a named vector to select columns, the output will have its columns renamed:

```
selection <- c(
  new_homeworld = "homeworld",
  new_height = "height",
  new_mass = "mass"
)
starwars |> select(all_of(selection))
#> # A tibble: 87 x 3
#>   new_homeworld new_height new_mass
#>   <chr>              <int>    <dbl>
#> 1 Tatooine             172       77
#> 2 Tatooine             167       75
#> 3 Naboo                 96       32
#> 4 Tatooine             202      136
#> # i 83 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
#> # i 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
#> # i 83 more rows
#> # i 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
#> # i 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
#> # i 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
#> # i 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
#> # i 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
#> # i 146 more rows
```

See Also

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

---

setops *Set operations*

Description

Perform set operations using the rows of a data frame.

- intersect(x, y) finds all rows in both x and y.
- union(x, y) finds all rows in either x or y, excluding duplicates.
- union_all(x, y) finds all rows in either x or y, including duplicates.
- setdiff(x, y) finds all rows in x that aren't in y.
- symdiff(x, y) computes the symmetric difference, i.e. all rows in x that aren't in y and all rows in y that aren't in x.
- setequal(x, y) returns TRUE if x and y contain the same rows (ignoring order).

Note that intersect(), union(), setdiff(), and symdiff() remove duplicates in x and y.

Usage

```
intersect(x, y, ...)

union(x, y, ...)

union_all(x, y, ...)

setdiff(x, y, ...)

setequal(x, y, ...)

symdiff(x, y, ...)
```

Arguments

x, y	Pair of compatible data frames. A pair of data frames is compatible if they have the same column names (possibly in different orders) and compatible types.
...	These dots are for future extensions and must be empty.

Base functions

`intersect()`, `union()`, `setdiff()`, and `setequal()` override the base functions of the same name in order to make them generic. The existing behaviour for vectors is preserved by providing default methods that call the base functions.

Examples

```
df1 <- tibble(x = 1:3)
df2 <- tibble(x = 3:5)

intersect(df1, df2)
union(df1, df2)
union_all(df1, df2)
setdiff(df1, df2)
setdiff(df2, df1)
symdiff(df1, df2)

setequal(df1, df2)
setequal(df1, df1[3:1, ])

# Note that the following functions remove pre-existing duplicates:
df1 <- tibble(x = c(1:3, 3, 3))
df2 <- tibble(x = c(3:5, 5))

intersect(df1, df2)
union(df1, df2)
setdiff(df1, df2)
symdiff(df1, df2)
```

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 the smallest or largest 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, ..., .by = NULL, .preserve = FALSE)

slice_head(.data, ..., n, prop, by = NULL)

slice_tail(.data, ..., n, prop, by = NULL)

slice_min(
  .data,
  order_by,
  ...,
  n,
  prop,
  by = NULL,
  with_ties = TRUE,
  na_rm = FALSE
)

slice_max(
  .data,
  order_by,
  ...,
  n,
  prop,
  by = NULL,
  with_ties = TRUE,
  na_rm = FALSE
)

slice_sample(.data, ..., n, prop, by = NULL, 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_*()`, these arguments are passed on to methods.

`.by, by` <`tidy-select`> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to `group_by()`. For details and examples, see ?dplyr_by.

`.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. `prop` will be rounded towards zero to generate an integer number of rows.

A negative value of `n` or `prop` will be subtracted from the group size. For example, `n = -2` with a group of 5 rows will select 5 - 2 = 3 rows; `prop = -0.25` with 8 rows will select 8 * (1 - 0.25) = 6 rows.

`order_by` <`data-masking`> Variable or function of variables to order by. To order by multiple variables, wrap them in a data frame or tibble.

`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.

`na_rm` Should missing values in `order_by` be removed from the result? If `FALSE`, `NA` values are sorted to the end (like in `arrange()`), so they will only be included if there are insufficient non-missing values to reach `n`/`prop`.

`weight_by` <`data-masking`> 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. See the `Details` section for more technical details regarding these weights.

`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()`.

For `slice_sample()`, note that the weights provided in `weight_by` are passed through to the `prob` argument of `base::sample.int()`. This means they cannot be used to reconstruct summary statistics from the underlying population. See this discussion for more details.

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()`, `reframe()`, `rename()`, `select()`, `summarise()`

Examples

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

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

# 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.
mtcars |> slice_min(cyl, n = 1)
```

```
# Use with_ties = FALSE to return exactly n matches
mtcars |> slice_min(cyl, n = 1, with_ties = FALSE)
# Or use additional variables to break the tie:
mtcars |> slice_min(tibble(cyl, mpg), n = 1)

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

# slice_sample() can be used to shuffle rows with `prop = 1`
mtcars |> slice_sample(prop = 1)

# 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.py4e.com/`, 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 dataset is the NOAA Atlantic hurricane database best track data, https://www.nhc.noaa.gov/data/#hurdat. The data includes the positions and attributes of storms from 1975-2024. Storms from 1979 onward are measured every six hours during the lifetime of the storm. Storms in earlier years have some missing data.

Usage

```
storms
```

Format

A tibble with 20,778 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 hurricane category calculated from wind speed.

- `NA`: Not a hurricane
- 1: 64+ knots
- 2: 83+ knots
- 3: 96+ knots
- 4: 113+ knots
- 5: 137+ knots

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

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

tropicalstorm_force_diameter Diameter (in nautical miles) of the area experiencing tropical storm strength winds (34 knots or above). Only available starting in 2004.

hurricane_force_diameter Diameter (in nautical miles) of the area experiencing hurricane strength winds (64 knots or above). Only available starting in 2004.

See Also

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

Examples

```
storms

# Show a few recent storm paths
if (requireNamespace("ggplot2", quietly = TRUE)) {
  library(ggplot2)
  storms |>
    filter(year >= 2000) |>
    ggplot(aes(long, lat, color = paste(year, name))) +
    geom_path(show.legend = FALSE) +
    facet_wrap(~year)
}

storms
```

summarise *Summarise each group down to one row*

Description

summarise() creates a new data frame. It returns one row 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, ..., .by = NULL, .groups = NULL)

summarize(.data, ..., .by = NULL, .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 data frame with 1 row, to add multiple columns from a single expression.

.by <tidy-select> Optionally, a selection of columns to group by for just this operation, functioning as an alternative to group_by(). For details and examples, see ?dplyr_by.

.groups **[Experimental]** Grouping structure of the result.

- "drop_last": drops 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 set to "drop_last" for a grouped data frame, and "keep" for a rowwise data frame. In addition, a message informs you of how the result will be grouped 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(),
- 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(), reframe(), 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())

# 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 ?rlang::args_data_masking
```

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

[Superseded]

`vars()` is superseded because it is only needed for the scoped verbs (i.e. `mutate_at()`, `summarise_at()`, and friends), which have been been superseded in favour of `across()`. See `vignette("colwise")` for details.

This helper is intended to provide tidy-select semantics for scoped verbs like `mutate_at()` and `summarise_at()`. Note that anywhere you can supply `vars()` specification, you can also supply a numeric vector of column positions or a character vector of column names.

Usage

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

Arguments

`...`	<`tidy-select`> Variables to operate on.

See Also

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

`when-any-all` *Elementwise* `any()` *and* `all()`

Description

These functions are variants of `any()` and `all()` that work elementwise across multiple inputs. You can also think of these functions as generalizing `|` and `&` to any number of inputs, rather than just two, for example:

- `when_any(x, y, z)` is equivalent to `x | y | z`.
- `when_all(x, y, z)` is equivalent to `x & y & z`.

`when_any()` is particularly useful within `filter()` and `filter_out()` to specify comma separated conditions combined with `|` rather than `&`.

Usage

```
when_any(..., na_rm = FALSE, size = NULL)

when_all(..., na_rm = FALSE, size = NULL)
```

Arguments

`...`	Logical vectors of equal size.
`na_rm`	Missing value handling: • If `FALSE`, missing values are propagated according to the same rules as `|` and `&`. • If `TRUE`, missing values are removed from the elementwise computation.
`size`	An optional output size. Only useful to specify if it is possible for `...` to be empty, with no inputs provided.

Details

`when_any()` and `when_all()` are "parallel" versions of `any()` and `all()` in the same way that `pmin()` and `pmax()` are "parallel" versions of `min()` and `max()`.

See Also

`base::any()`, `base::all()`, `cumany()`, `cumall()`, `base::pmin()`, `base::pmax()`

Examples

```
x <- c(TRUE, TRUE, TRUE, FALSE, FALSE, FALSE, NA, NA, NA)
y <- c(TRUE, FALSE, NA, TRUE, FALSE, NA, TRUE, FALSE, NA)

# `any()` and `all()` summarise down to 1 value
any(x, y)
all(x, y)

# `when_any()` and `when_all()` work element by element across all inputs
# at the same time. Their defaults are equivalent to calling `|` or `&`.
when_any(x, y)
x | y

when_all(x, y)
x & y

# `na_rm = TRUE` is useful when you'd like to force these functions to
# return only `TRUE` or `FALSE`. This argument does so by removing any `NA`
# from the elementwise computation entirely.
tibble(
  x = x,
  y = y,
  any_propagate = when_any(x, y),
  any_remove = when_any(x, y, na_rm = TRUE),
  all_propagate = when_all(x, y),
  all_remove = when_all(x, y, na_rm = TRUE)
)

# ---------------------------------------------------------------------------
# With `filter()` and `filter_out()`

# `when_any()` is particularly useful inside of `filter()` and
# `filter_out()` as a way to combine comma separated conditions with `|`
# instead of with `&`.

countries <- tibble(
  name = c("US", "CA", "PR", "RU", "US", NA, "CA", "PR", "RU"),
  score = c(200, 100, 150, NA, 50, 100, 300, 250, 120)
)
countries

# Find rows where any of the following are true:
# - "US" and "CA" have a score between 200-300
# - "PR" and "RU" have a score between 100-200
countries |>
  filter(
    (name %in% c("US", "CA") & between(score, 200, 300)) |
      (name %in% c("PR", "RU") & between(score, 100, 200))
  )

# With `when_any()`, you drop the explicit `|`, the extra `()`, and your
# conditions are all indented to the same level
countries |>
  filter(when_any(
    name %in% c("US", "CA") & between(score, 200, 300),
    name %in% c("PR", "RU") & between(score, 100, 200)
  ))

# To drop these rows instead, use `filter_out()`
countries |>
  filter_out(when_any(
    name %in% c("US", "CA") & between(score, 200, 300),
    name %in% c("PR", "RU") & between(score, 100, 200)
  ))

# ---------------------------------------------------------------------------
# Programming with `when_any()` and `when_all()`

# The `size` argument is useful for making these functions size stable when
# you aren't sure how many inputs you're going to receive
size <- length(x)

# Two inputs
inputs <- list(x, y)
when_all(!!!inputs, size = size)

# One input
inputs <- list(x)
```

```
when_all(!!!inputs, size = size)

# Zero inputs (without `size`, this would return `logical()`)
inputs <- list()
when_all(!!!inputs, size = size)

# When no inputs are provided, these functions are consistent with `any()`
# and `all()`
any()
when_any(size = 1)

all()
when_all(size = 1)
```

Index

- ∗ **datasets**
  - band_members, 11
  - starwars, 115
  - storms, 116
- ∗ **grouping functions**
  - group_by, 45
  - group_map, 49
  - group_trim, 51
- ∗ **joins**
  - cross_join, 28
  - filter-joins, 43
  - mutate-joins, 63
  - nest_join, 71
- ∗ **ranking functions**
  - ntile, 76
  - percent_rank, 79
  - row_number, 102
- ∗ **single table verbs**
  - arrange, 9
  - filter, 38
  - mutate, 59
  - reframe, 91
  - rename, 95
  - select, 105
  - slice, 111
  - summarise, 117
- +, *61*
- ==, *41*
- >, *41*
- >=, *41*
- ?dplyr_by, *39*, *60*, *92*, *112*, *117*
- ?join_by, *44*, *66*, *72*
- &, *41*, *120*

- across, 3
- across(), *8*, *24*, *30*, *80*, *103*, *120*
- add_count *(*count*)*, 26
- add_tally *(*count*)*, 26
- all(), *118*, *120*, *121*
- all_of(), *105*
- all_vars, 8
- all_vars(), *120*
- anti_join *(*filter-joins*)*, 43
- anti_join(), *73*
- any(), *118*, *120*, *121*
- any_of(), *105*
- any_vars *(*all_vars*)*, 8
- any_vars(), *120*
- arrange, 9, *42*, *61*, *93*, *96*, *109*, *113*, *119*
- arrange(), *31*, *37*, *46*, *103*, *112*
- arrange_all(), *103*, *104*
- arrange_at(), *104*
- arrange_if(), *104*
- as_tibble(), *100*
- auto_copy, 11

- band_instruments *(*band_members*)*, 11
- band_instruments2 *(*band_members*)*, 11
- band_members, 11
- base::all(), *121*
- base::any(), *121*
- base::pmax(), *121*
- base::pmin(), *121*
- base::sample.int(), *112*
- between, 12
- between(), *41*
- bind *(*bind_rows*)*, 14
- bind_cols, 13
- bind_rows, 14

- c_across, 30
- c_across(), *6*
- case-and-replace-when, 15
- case_when *(*case-and-replace-when*)*, 15
- case_when(), *61*, *83*, *86*, *88*
- cast, *15*, *70*, *87*
- closest *(*join_by*)*, 54
- coalesce, 20
- coalesce(), *61*, *70*
- collapse *(*compute*)*, 21

collect (compute), 21
collect(), *25*
compute, 21
consecutive_id, 23
contains(), *105*
context, 23
copy_to, 25
copy_to(), *22*
count, 26
count(), *37*
cross_join, 28, *44*, *68*, *74*
cross_join(), *44*, *55*, *66*, *72*
cumall, 29
cumall(), *61*, *121*
cumany (cumall), 29
cumany(), *61*, *121*
cume_dist (percent_rank), 79
cume_dist(), *61*
cummax(), *61*
cummean (cumall), 29
cummean(), *61*
cummin(), *61*
cumsum(), *61*
cur_column (context), 23
cur_column(), *4*
cur_group (context), 23
cur_group(), *4*
cur_group_id (context), 23
cur_group_rows (context), 23

data-masking, *34*
dense_rank (row_number), 102
dense_rank(), *61*
desc, 31
desc(), *9*, *76*, *79*, *102*
distinct, 31
distinct(), *37*
distinct_all(), *104*
distinct_at(), *104*
distinct_if(), *104*
do(), *50*
dplyr-locale, *9*
dplyr_by, 33

ends_with(), *105*
everything(), *105*
explain, 37

filter, *10*, 38, *61*, *93*, *96*, *109*, *113*, *119*
filter(), *51*, *103*, *112*, *120*
filter-joins, 43
filter_all(), *8*, *103, 104*
filter_at(), *104*
filter_if(), *8*, *104*
filter_out (filter), 38
filter_out(), *120*
first (nth), 74
first(), *118*
full_join (mutate-joins), 63

glimpse, 45
group_by, 45, *50*, *52*
group_by(), *27*, *33*, *37*, *39*, *51*, *60*, *92*, *100*, *103*, *112*, *117*
group_by_all(), *103, 104*
group_by_at(), *104*
group_by_drop_default(), *46*
group_by_if(), *104*
group_cols, 48
group_cols(), *105*
group_data(), *24*
group_keys(), *50*, *118*
group_map, *47*, 49, *52*
group_modify (group_map), 49
group_nest, *47*, *50*, *52*
group_split, *47*, *50*, *52*
group_trim, *47*, *50*, 51
group_vars(), *49*
group_walk (group_map), 49
grouped data frame, *51, 52*
grouped_df, *46*, *100*, *111*, *118*
groups(), *49*

ident, 52
if-else, *53*
if_all (across), 3
if_any (across), 3
if_else, 53
if_else(), *15*, *61*, *83*
ifelse(), *53*
inner_join (mutate-joins), 63
inner_join(), *73*
integerish, *104*
intersect (setops), 109
IQR(), *118*
is.na(), *41*
is.tbl (tbl), 119

join, *13*
join *(*mutate-joins*)*, 63
join_by, 54
join_by(), *12*, *43*, *44*, *65*, *66*, *72*

lag *(*lead-lag*)*, 58
lag(), *61*
last *(*nth*)*, 74
last(), *118*
last_col(), *105*
lead *(*lead-lag*)*, 58
lead(), *61*
lead-lag, 58
left_join *(*mutate-joins*)*, 63
left_join(), *54*, *73*
locale, *9*
log(), *61*

mad(), *118*
match(), *44*, *66*, *73*
matches(), *105*
max(), *118*, *121*
mean(), *118*
median(), *118*
merge(), *44*, *66*, *73*
min(), *118*, *121*
min_rank *(*row_number*)*, 102
min_rank(), *61*
mutate, *10*, *42*, 59, *93*, *96*, *109*, *113*, *119*
mutate(), *3*, *4*, *24*, *30*, *80*, *92*, *103*, *118*
mutate-joins, 63
mutate_all(), *104*
mutate_at(), *104*, *120*
mutate_if(), *104*
mutating joins, *28*

n *(*context*)*, 23
n(), *118*
n_distinct, 77
n_distinct(), *118*
na_if, 69
na_if(), *21*, *61*
near, 71
near(), *41*
nest_by(), *101*
nest_join, *28*, *44*, *68*, 71
nested, *50*
nth, 74
nth(), *118*

ntile, 76, *79*, *102*
ntile(), *61*
num_range(), *105*

order_by, 78
overlaps *(*join_by*)*, 54

percent_rank, *76*, 79, *102*
percent_rank(), *61*
pick, 80
pick(), *3*, *24*, *103*
pillar::glimpse(), *45*
pmax(), *121*
pmin(), *121*
print(), *37*
pull, 81
pull(), *37*

quasiquotation, *82*

recode, 82
recode(), *61*
recode-and-replace-values, 86
recode_factor *(*recode*)*, 82
recode_values
*(*recode-and-replace-values*)*, 86
recode_values(), *16*, *83*
recycled, *13*, *15*, *20*, *53*, *70*, *87*
reframe, *10*, *42*, *61*, 91, *96*, *109*, *113*, *119*
relocate, 94
relocate(), *60*
rename, *10*, *42*, *61*, *93*, 95, *109*, *113*, *119*
rename(), *37*, *103*
rename_with *(*rename*)*, 95
replace_values
*(*recode-and-replace-values*)*, 86
replace_values(), *21*, *70*, *83*
replace_when *(*case-and-replace-when*)*, 15
replace_when(), *21*, *70*
right_join *(*mutate-joins*)*, 63
rlang::as_function(), *104*
row_number, *76*, *79*, 102
row_number(), *61*, *112*
rows, 97
rows_append *(*rows*)*, 97
rows_delete *(*rows*)*, 97
rows_insert *(*rows*)*, 97
rows_patch *(*rows*)*, 97
rows_update *(*rows*)*, 97

rows_upsert (rows), 97
rowwise, 100, *118*
rowwise(), *30*

scoped, 103
sd(), *118*
select, *10*, *42*, *61*, *93*, *96*, 105, *113*, *119*
select(), *3*, *37*, *48*, *80*, *103*
select_all(), *103, 104*
select_at(), *104*
select_if(), *104*
semi_join (filter-joins), 43
semi_join(), *73*
setdiff (setops), 109
setequal (setops), 109
setops, 109
show_query (explain), 37
slice, *10*, *42*, *61*, *93*, *96*, *109*, 111, *119*
slice_head (slice), 111
slice_max (slice), 111
slice_min (slice), 111
slice_sample (slice), 111
slice_tail (slice), 111
split, *50*
sql, 114
starts_with(), *104–106*
starwars, 115
storms, 116
str(), *37*, *45*
stringi::stri_locale_list(), *9*, *47*
summarise, *10*, *42*, *61*, *93*, *96*, *109*, *113*, 117
summarise(), *3, 4*, *24*, *30*, *33*, *46*, *80*, *92*, *100*, *103*
summarise_all(), *103, 104*
summarise_at(), *104*, *120*
summarise_if(), *104*
summarize (summarise), 117
switch(), *83*
symdiff (setops), 109

tally (count), 26
tbl, 119
tbl(), *46*
tibble, *118*
tibble::deframe(), *92*
tibble::enframe(), *92*
tidy dots, *104*
tidy-select, *34, 35*
tidyr::unnest(), *73*
tidyselect::vars_select(), *104*
transmute(), *103*
transmute_all(), *104*
transmute_at(), *104*
transmute_if(), *104*

ungroup (group_by), 45
ungroup(), *33*, *100*
union (setops), 109
union_all (setops), 109
unique.data.frame(), *31*
unpack, *4*

vars, 120
vars(), *9*, *48*, *103, 104*
vctrs::vec_as_names(), *13*
vctrs::vec_c(), *30*
vctrs::vec_case_when(), *16*
vctrs::vec_cast_common(), *12*
vctrs::vec_if_else(), *53*
vctrs::vec_recode_values(), *88*

when-any-all, 120
when_all (when-any-all), 120
when_all(), *41*
when_any (when-any-all), 120
when_any(), *38, 39*, *41*
where(), *105*
with_order(), *78*
within (join_by), 54

xor(), *41*