Package 'DFplyr'

April 8, 2026

Title A `DataFrame` (`S4Vectors`) backend for `dplyr`

Version 1.5.2

Description Provides `dplyr` verbs (`mutate`, `select`, `filter`, etc...) supporting `S4Vectors::DataFrame` objects. Importantly, this is achieved without conversion to an intermediate `tibble`. Adds grouping infrastructure to `DataFrame` which is respected by the transformation verbs.

biocViews DataRepresentation, Infrastructure, Software

License GPL-3

Encoding UTF-8

RoxygenNote 7.3.3

URL https://github.com/jonocarroll/DFplyr

BugReports https://github.com/jonocarroll/DFplyr/issues

Depends dplyr

Imports BiocGenerics, methods, rlang, S4Vectors, tidyselect

Suggests BiocStyle, GenomeInfoDb, GenomicRanges, IRanges, knitr, rmarkdown, sessioninfo, testthat (>= 3.0.0), tibble

VignetteBuilder knitr

Config/testthat/edition 3

Roxygen list(markdown = TRUE)

git_url https://git.bioconductor.org/packages/DFplyr

git_branch devel

git_last_commit bc6a1bd

git_last_commit_date 2026-02-15

Repository Bioconductor 3.23

Date/Publication 2026-04-08

Author Jonathan Carroll [aut, cre] (ORCID: <https://orcid.org/0000-0002-1404-5264>), Pierre-Paul Axisa [ctb]

Maintainer Jonathan Carroll <rpkg@jcarroll.com.au>

Contents

`DFplyr-package` *Treat a* `S4Vectors::DataFrame` *as a* `dplyr` *data source*

Description

Add **dplyr** compatibility to `S4Vectors::DataFrame` for use with a selection of **dplyr** verbs.

Arguments

`x`	A `S4Vectors::DataFrame` object

Author(s)

Maintainer: Jonathan Carroll <rpkg@jcarroll.com.au> (ORCID)

Other contributors:

- Pierre-Paul Axisa [contributor]

See Also

Useful links:

- https://github.com/jonocarroll/DFplyr
- Report bugs at https://github.com/jonocarroll/DFplyr/issues

Examples

```
library(S4Vectors)
library(dplyr)

d <- as(mtcars, "DataFrame")

mutate(d, newvar = cyl + hp)

mutate_at(d, vars(starts_with("c")), ~ .^2)

group_by(d, cyl, am) %>%
    tally(gear)

count(d, gear, am, cyl)

select(d, am, cyl)

select(d, am, cyl) %>%
    rename2(foo = am)

arrange(d, desc(hp))

rbind(DataFrame(mtcars[1, ], row.names = "MyCar"), d) %>%
    distinct()

filter(d, am == 0)

slice(d, 3:6)
```

`arrange.DataFrame` *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

```
## S3 method for class 'DataFrame'
arrange(.data, ...)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Variables, or functions of variables. Use `desc()` to sort a variable in descending order.

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))
```

---

bindROWS,GroupedDataFrame-method

rbind DataFrames

Description

rbind DataFrames

Usage

```
## S4 method for signature 'GroupedDataFrame'
bindROWS(x, objects = list())
```

Arguments

x	a DataFrame
objects	a list of DataFrames

Value

a new DataFrame combining the inputs by rows

count.DataFrame *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

```
## S3 method for class 'DataFrame'
count(
  x,
  ...,
  wt = NULL,
  sort = FALSE,
  name = "n",
  .drop = group_by_drop_default(x)
)
```

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)
```

---

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

Value

the input vector in a format that will be sorted in descending order.

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.DataFrame *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

```
## S3 method for class 'DataFrame'
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()
```

`filter.DataFrame` *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

```
## S3 method for class 'DataFrame'
filter(.data, ..., .preserve = FALSE)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<`data-masking`> Expressions that return a logical 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()`).
`.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.GroupedDataFrame`

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

```
## S3 method for class 'GroupedDataFrame'
filter(.data, ..., .preserve = FALSE)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<data-masking> Expressions that return a logical 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()`).
`.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
```

`format.DataFrame` *Encode in a Common Format*

Description

Format an R object for pretty printing.

Usage

```
## S3 method for class 'DataFrame'
format(x, ...)
```

Arguments

`x`	any R object (conceptually); typically numeric.
`...`	further arguments passed to or from other methods.

Details

`format` is a generic function. Apart from the methods described here there are methods for dates (see `format.Date`), date-times (see `format.POSIXct`) and for other classes such as `format.octmode` and `format.dist`.

`format.data.frame` formats the data frame column by column, applying the appropriate method of `format` for each column. Methods for columns are often similar to `as.character` but offer more control. Matrix and data-frame columns will be converted to separate columns in the result, and character columns (normally all) will be given class `"AsIs"`.

`format.factor` converts the factor to a character vector and then calls the default method (and so `justify` applies).

`format.AsIs` deals with columns of complicated objects that have been extracted from a data frame. Character objects and (atomic) matrices are passed to the default method (and so `width` does not apply). Otherwise it calls `toString` to convert the object to character (if a vector or list, element by element) and then right-justifies the result.

Justification for character vectors (and objects converted to character vectors by their methods) is done on display width (see `nchar`), taking double-width characters and the rendering of special characters (as escape sequences, including escaping backslash but not double quote: see `print.default`) into account. Thus the width is as displayed by `print(quote = FALSE)` and not as displayed by `cat`. Character strings are padded with blanks to the display width of the widest. (If `na.encode = FALSE` missing character strings are not included in the width computations and are not encoded.)

Numeric vectors are encoded with the minimum number of decimal places needed to display all the elements to at least the `digits` significant digits. However, if all the elements then have trailing zeroes, the number of decimal places is reduced until at least one element has a non-zero final digit; see also the argument documentation for `big.*`, `small.*` etc, above. See the note in `print.default` about `digits >= 16`.

Raw vectors are converted to their 2-digit hexadecimal representation by `as.character`.

`format.default(x)` now provides a "minimal" string when `isS4(x)` is true.

While the internal code respects the option `getOption("OutDec")` for the 'decimal mark' in general, `decimal.mark` takes precedence over that option. Similarly, `scientific` takes precedence over `getOption("scipen")`.

Value

An object of similar structure to `x` containing character representations of the elements of the first argument `x` in a common format, and in the current locale's encoding.

For character, numeric, complex or factor `x`, dims and dimnames are preserved on matrices/arrays and names on vectors: no other attributes are copied.

If `x` is a list, the result is a character vector obtained by applying `format.default(x, ...)` to each element of the list (after `unlist`ing elements which are themselves lists), and then collapsing the result for each element with `paste(collapse = ", ")`. The defaults in this case are `trim = TRUE`, `justify = "none"` since one does not usually want alignment in the collapsed strings.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`format.info` indicates how an atomic vector would be formatted.

`formatC`, `paste`, `as.character`, `sprintf`, `print`, `prettyNum`, `toString`, `encodeString`.

Examples

```
format(1:10)
format(1:10, trim = TRUE)

zz <- data.frame("(row names)"= c("aaaaa", "b"), check.names = FALSE)
format(zz)
format(zz, justify = "left")

## use of nsmall
format(13.7)
format(13.7, nsmall = 3)
format(c(6.0, 13.1), digits = 2)
format(c(6.0, 13.1), digits = 2, nsmall = 1)

## use of scientific
format(2^31-1)
```

```
format(2^31-1, scientific = TRUE)
## scientific = numeric scipen (= {sci}entific notation {pen}alty) :
x <- c(1e5, 1000, 10, 0.1, .001, .123)
t(sapply(setNames(,-4:1),
         \(sci) sapply(x, format, scientific=sci)))


## a list
z <- list(a = letters[1:3], b = (-pi+0i)^((-2:2)/2), c = c(1,10,100,1000),
          d = c("a", "longer", "character", "string"),
          q = quote( a + b ), e = expression(1+x))
## can you find the "2" small differences?
(f1 <- format(z, digits = 2))
(f2 <- format(z, digits = 2, justify = "left", trim = FALSE))
f1 == f2 ## 2 FALSE, 4 TRUE

## A "minimal" format() for S4 objects without their own format() method:
cc <- methods::getClassDef("standardGeneric")
format(cc) ## "<S4 class ......>"
```

`group_by.DataFrame` *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

```
## S3 method for class 'DataFrame'
group_by(.data, ..., add = FALSE, .drop = group_by_drop_default(.data))
```

Arguments

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

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_by.GroupedDataFrame` *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

```
## S3 method for class 'GroupedDataFrame'
group_by(.data, ..., add = FALSE, .drop = group_by_drop_default(.data))
```

Arguments

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

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_by_drop_default.DataFrame` *Default value for .drop argument of group_by*

Description

Default value for .drop argument of group_by

Usage

```
## S3 method for class 'DataFrame'
group_by_drop_default(.tbl)
```

Arguments

`.tbl`	A data frame

Value

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

Examples

```
group_by_drop_default(iris)

iris |>
  group_by(Species) |>
  group_by_drop_default()

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

`group_data` *Set and Get Group Data on a DataFrame*

Description

The location of group data is an internal implemnetation detail, so these get and set methods enable interfacing with that data.

Usage

```
set_group_data(x, g, .drop = group_by_drop_default(x))

get_group_data(x)
```

Arguments

x	A S4Vectors::DataFrame on which to set group data.
g	Group data (a data.frame).
.drop	Drop groups formed by factor levels that don't appear in the data?

Value

For set_group_data, the input x with group data set as metadata. For get_group_data, the group data that is set on x.

---

group_data.DataFrame *Grouping metadata*

Description

This collection of functions accesses data about grouped data frames in various ways:

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

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

Usage

```
## S3 method for class 'DataFrame'
group_data(.data)
```

Arguments

.data	a S4Vectors::DataFrame()

Value

a `data.frame` of group data

`group_vars.DataFrame` *Grouping metadata*

Description

This collection of functions accesses data about grouped data frames in various ways:

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

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

Usage

```
## S3 method for class 'DataFrame'
group_vars(x)
```

Arguments

x	a `S4Vectors::DataFrame()`, not already grouped

Value

the grouping variables as a character vector

`group_vars.GroupedDataFrame` *Grouping metadata*

Description

This collection of functions accesses data about grouped data frames in various ways:

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

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

Usage

```
## S3 method for class 'GroupedDataFrame'
group_vars(x)
```

Arguments

`x`	a `GroupedDataFrame`, likely already grouped

Value

the grouping variables as a character vector

`inner_join.DataFrame` *Mutating joins*

Description

Mutating joins

Usage

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

Arguments

x	a DataFrame
y	a DataFrame or data.frame
by	columns to use for joining the objects. If NULL, the function will look for common columns.

Value

a DataFrame

Examples

```
library(dplyr)
library(S4Vectors)
da <- starwars[, c("name", "mass", "species")][1:10, ]
db <- starwars[, c("name", "homeworld")]

Da <- as(da, "DataFrame")
Db <- as(db, "DataFrame")

Res_inner <- inner_join(Da, Db[1:3, ])
```

`mutate.DataFrame` *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

```
## S3 method for class 'DataFrame'
mutate(.data, ...)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<data-masking> Name-value pairs. The name gives the name of the column in the output. The value can be: • A vector of length 1, which will be recycled to the correct length. • A vector the same length as the current group (or the whole data frame if ungrouped). • `NULL`, to remove the column. • A data frame or tibble, to create multiple columns in the output.

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
```

`pull.DataFrame` *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

```
## S3 method for class 'DataFrame'
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

```
d <- S4Vectors::DataFrame(mtcars)
pull(d, cyl)
```

`rename,DataFrame-method` *Rename Columns of a DataFrame*

Description

Rename Columns of a DataFrame

Usage

```
## S4 method for signature 'DataFrame'
rename(x, ...)
```

Arguments

`x`	a `DataFrame`
`...`	NSE syntax; `new_name = old_name` to rename selected variables

Value

a `DataFrame` with columns renamed

`rename2` *Rename Columns of a DataFrame (deprecated)*

Description

Rename Columns of a DataFrame (deprecated)

Usage

```
rename2(.data, ...)
```

Arguments

`.data`	a DataFrame
`...`	columns to be renamed with syntax `new = old`

Value

Deprecated - use `rename`

Examples

```
# see rename
```

`select.DataFrame` *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

```
## S3 method for class 'DataFrame'
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()`

`slice.DataFrame` *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

```
## S3 method for class 'DataFrame'
slice(.data, ..., .preserve = FALSE)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	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.
`.preserve`	Relevant when the `.data` input is grouped. If `.preserve = FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise the grouping is kept as is.

Details

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()))
```

`summarise.DataFrame` *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

```
## S3 method for class 'DataFrame'
summarise(.data, ...)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<data-masking> Name-value pairs 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.

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
```

`summarize.DataFrame` *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

```
## S3 method for class 'DataFrame'
summarize(.data, ...)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	<data-masking> Name-value pairs 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.

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
```

`tally.DataFrame` *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

```
## S3 method for class 'DataFrame'
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).
`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 `n`s until it gets a new name.

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)
```

`tbl_vars.DataFrame` *List variables provided by a tbl.*

Description

`tbl_vars()` returns all variables while `tbl_nongroup_vars()` returns only non-grouping variables. The `groups` attribute of the object returned by `tbl_vars()` is a character vector of the grouping columns.

Usage

```
## S3 method for class 'DataFrame'
tbl_vars(x)
```

Arguments

x	A tbl object

Value

all variables, with a groups attribute when grouped.

See Also

group_vars() for a function that returns grouping variables.

---

ungroup.DataFrame *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

```
## S3 method for class 'DataFrame'
ungroup(x, ...)
```

Arguments

x	A tbl()
...	<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.

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()

---

ungroup.GroupedDataFrame

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

```
## S3 method for class 'GroupedDataFrame'
ungroup(x, ...)
```

Arguments

x	A tbl()
...	<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.

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()
```

`[,GroupedDataFrame-method` *Subset a DataFrame*

Description

Subset a DataFrame

Usage

```
## S4 method for signature 'GroupedDataFrame'
x[i, j, ..., drop = FALSE]
```

Arguments

x	a DataFrame to be subset
i	rows to subset
j	columns to subset
...	other params, passed to regular S4Vectors subsetting
drop	drop dimensions?

Value

a DataFrame subset by rows and/or columns

Index