Package 'tidytable'

January 13, 2023

Title Tidy Interface to 'data.table'

Version 0.9.2

Description A tidy interface to 'data.table',
giving users the speed of 'data.table' while using tidyverse-like syntax.

License MIT + file LICENSE

Encoding UTF-8

Imports data.table (>= 1.14.0), glue (>= 1.4.0), lifecycle(>= 1.0.3),
magrittr (>= 2.0.3), pillar (>= 1.8.0), rlang (>= 1.0.6),
tidyselect (>= 1.2.0), vctrs (>= 0.5.0)

RoxygenNote 7.2.2

Config/testthat/edition 3

URL https://github.com/markfairbanks/tidytable

BugReports https://github.com/markfairbanks/tidytable/issues

Suggests testthat (>= 2.1.0), bit64, knitr, rmarkdown, crayon

NeedsCompilation no

Author Mark Fairbanks [aut, cre],
Abdessabour Moutik [ctb],
Matt Carlson [ctb],
Ivan Leung [ctb],
Ross Kennedy [ctb],
Robert On [ctb],
Alexander Sevostianov [ctb],
Koen ter Berg [ctb]

Maintainer Mark Fairbanks <mark.t.fairbanks@gmail.com>

Repository CRAN

Date/Publication 2023-01-13 20:00:02 UTC

R topics documented:

`across` *Apply a function across a selection of columns*

Description

Apply a function across a selection of columns. For use in `arrange()`, `mutate()`, and `summarize()`.

Usage

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

Arguments

.cols	vector c() of unquoted column names. tidyselect compatible.
.fns	Function to apply. Can be a purrr-style lambda. Can pass also list of functions.
...	Other arguments for the passed function
.names	A glue specification that helps with renaming output columns. {.col} stands for the selected column, and {.fn} stands for the name of the function being applied. The default (NULL) is equivalent to "{.col}" for a single function case and "{.col}_{.fn}" when a list is used for .fns.

Examples

```
df <- data.table(
  x = rep(1, 3),
  y = rep(2, 3),
  z = c("a", "a", "b")
)

df %>%
  mutate(across(c(x, y), ~ .x * 2))

df %>%
  summarize(across(where(is.numeric), ~ mean(.x)),
            .by = z)

df %>%
  arrange(across(c(y, z)))
```

---

add_count *Add a count column to the data frame*

Description

Add a count column to the data frame.

df %>% add_count(a, b) is equivalent to using df %>% mutate(n = n(), .by = c(a, b))

Usage

```
add_count(.df, ..., wt = NULL, sort = FALSE, name = NULL)

add_tally(.df, wt = NULL, sort = FALSE, name = NULL)
```

Arguments

.df	A data.frame or data.table
...	Columns to group by. tidyselect compatible.
wt	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.

Examples

```
df <- data.table(
  a = c("a", "a", "b"),
  b = 1:3
)

df %>%
  add_count(a)
```

---

arrange *Arrange/reorder rows*

Description

Order rows in ascending or descending order.

Usage

```
arrange(.df, ...)
```

Arguments

.df	A data.frame or data.table
...	Variables to arrange by

Examples

```
df <- data.table(
  a = 1:3,
  b = 4:6,
  c = c("a", "a", "b")
)

df %>%
  arrange(c, -a)

df %>%
  arrange(c, desc(a))
```

arrange_across. *Arrange by a selection of variables*

Description

Deprecated

Usage

```
arrange_across.(.df, .cols = everything(), .fns = NULL)
```

Arguments

.df	A data.table or data.frame
.cols	vector c() of unquoted column names. tidyselect compatible.
.fns	Function to apply. If desc it arranges in descending order

Details

Arrange all rows in either ascending or descending order by a selection of variables.

Examples

```
## Not run:
df <- tidytable(a = c("a", "b", "a"), b = 3:1)

df %>%
  arrange_across.()

df %>%
  arrange_across.(a, desc.)

## End(Not run)
```

as_tidytable *Coerce an object to a data.table/tidytable*

Description

A tidytable object is simply a data.table with nice printing features.

Note that all tidytable functions automatically convert data.frames & data.tables to tidytables in the background. As such this function will rarely need to be used by the user.

Usage

```
as_tidytable(x, ..., .name_repair = "unique", .keep_rownames = NULL)
```

Arguments

x	An R object
...	Additional arguments to be passed to or from other methods.
.name_repair	Treatment of duplicate names. See ?vctrs::vec_as_names for options/details.
.keep_rownames	Default is FALSE. If TRUE, adds the input object's names as a separate column named "rn". .keep_rownames = "id" names the column "id" instead.

Examples

```
df <- data.frame(x = -2:2, y = c(rep("a", 3), rep("b", 2)))

df %>%
  as_tidytable()
```

---

between *Do the values from x fall between the left and right bounds?*

Description

between() utilizes data.table::between() in the background

Usage

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

Arguments

x	A numeric vector
left, right	Boundary values

Examples

```
df <- data.table(
  x = 1:5,
  y = 1:5
)

# Typically used in a filter()
df %>%
  filter(between(x, 2, 4))

df %>%
  filter(x %>% between(2, 4))

# Can also use the %between% operator
df %>%
  filter(x %between% c(2, 4))
```

bind_cols *Bind data.tables by row and column*

Description

Bind multiple data.tables into one row-wise or col-wise.

Usage

```
bind_cols(..., .name_repair = "unique")

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

Arguments

...	data.tables or data.frames to bind
.name_repair	Treatment of duplicate names. See ?vctrs::vec_as_names for options/details.
.id	If TRUE, an integer column is made as a group id

Examples

```
# Binding data together by row
df1 <- data.table(x = 1:3, y = 10:12)
df2 <- data.table(x = 4:6, y = 13:15)

df1 %>%
  bind_rows(df2)

# Can pass a list of data.tables
df_list <- list(df1, df2)

bind_rows(df_list)

# Binding data together by column
df1 <- data.table(a = 1:3, b = 4:6)
df2 <- data.table(c = 7:9)

df1 %>%
  bind_cols(df2)

# Can pass a list of data frames
bind_cols(list(df1, df2))
```

case *data.table::fcase() with vectorized default*

Description

This function allows you to use multiple if/else statements in one call.

It is called like `data.table::fcase()`, but allows the user to use a vector as the `default` argument.

Usage

```
case(..., default = NA, ptype = NULL, size = NULL)
```

Arguments

`...`	Sequence of condition/value designations
`default`	Default value. Set to NA by default.
`ptype`	Optional ptype to specify the output type.
`size`	Optional size to specify the output size.

Examples

```
df <- tidytable(x = 1:10)

df %>%
  mutate(case_x = case(x < 5, 1,
                       x < 7, 2,
                       default = 3))
```

case_match *Vectorized* `switch()`

Description

Allows the user to succinctly create a new vector based off conditions of a single vector.

Usage

```
case_match(.x, ..., .default = NA, .ptype = NULL)
```

Arguments

`.x`	A vector
`...`	A sequence of two-sided formulas. The left hand side gives the old values, the right hand side gives the new value.
`.default`	The default value if all conditions evaluate to `FALSE`.
`.ptype`	Optional ptype to specify the output type.

Examples

```
df <- tidytable(x = c("a", "b", "c", "d"))

df %>%
  mutate(
    case_x = case_match(x,
                        c("a", "b") ~ "new_1",
                        "c" ~ "new_2",
                        .default = x)
  )
```

`case_when` *Case when*

Description

This function allows you to use multiple if/else statements in one call.

It is called like `dplyr::case_when()`, but utilizes `data.table::fifelse()` in the background for improved performance.

Usage

```
case_when(..., .default = NA, .ptype = NULL, .size = NULL)
```

Arguments

`...`	A sequence of two-sided formulas. The left hand side gives the conditions, the right hand side gives the values.
`.default`	The default value if all conditions evaluate to `FALSE`.
`.ptype`	Optional ptype to specify the output type.
`.size`	Optional size to specify the output size.

Examples

```
df <- tidytable(x = 1:10)

df %>%
  mutate(case_x = case_when(x < 5 ~ 1,
                            x < 7 ~ 2,
                            TRUE ~ 3))
```

coalesce *Coalesce missing values*

Description

Fill in missing values in a vector by pulling successively from other vectors.

Usage

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

Arguments

`...`	Input vectors. Supports dynamic dots.
`.ptype`	Optional ptype to override output type
`.size`	Optional size to override output size

Examples

```
# Use a single value to replace all missing values
x <- c(1:3, NA, NA)
coalesce(x, 0)

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

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

complete *Complete a data.table with missing combinations of data*

Description

Turns implicit missing values into explicit missing values.

Usage

```
complete(.df, ..., fill = list(), .by = NULL)
```

Arguments

.df	A data.frame or data.table
...	Columns to expand
fill	A named list of values to fill NAs with.
.by	Columns to group by

Examples

```
df <- data.table(x = 1:2, y = 1:2, z = 3:4)

df %>%
  complete(x, y)

df %>%
  complete(x, y, fill = list(z = 10))
```

---

consecutive_id *Generate a unique id for consecutive values*

Description

Generate a unique id for runs of consecutive values

Usage

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

Arguments

...	Vectors of values

Examples

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

context *Context functions*

Description

These functions give information about the "current" group.

- `cur_data()` gives the current data for the current group
- `cur_column()` gives the name of the current column (for use in `across()` only)
- `cur_group_id()` gives a group identification number
- `cur_group_rows()` gives the row indices for each group

Can be used inside `summarize()`, `mutate()`, & `filter()`

Usage

```
cur_column()

cur_data()

cur_group_id()

cur_group_rows()
```

Examples

```
df <- data.table(
  x = 1:5,
  y = c("a", "a", "a", "b", "b")
)

df %>%
  mutate(
    across(c(x, y), ~ paste(cur_column(), .x))
  )

df %>%
  summarize(data = list(cur_data()),
            .by = y)

df %>%
  mutate(group_id = cur_group_id(),
         .by = y)

df %>%
  mutate(group_rows = cur_group_rows(),
         .by = y)
```

count *Count observations by group*

Description

Returns row counts of the dataset.

`tally()` returns counts by group on a grouped tidytable.

`count()` returns counts by group on a grouped tidytable, or column names can be specified to return counts by group.

Usage

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

tally(.df, wt = NULL, sort = FALSE, name = NULL)
```

Arguments

`.df`	A data.frame or data.table
`...`	Columns to group by in `count()`. `tidyselect` compatible.
`wt`	Frequency weights. `tidyselect` compatible. 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`.

Examples

```
df <- data.table(
  x = c("a", "a", "b"),
  y = c("a", "a", "b"),
  z = 1:3
)

df %>%
  count()

df %>%
  count(x)

df %>%
  count(where(is.character))

df %>%
  count(x, wt = z, name = "x_sum")
```

```
df %>%
  count(x, sort = TRUE)

df %>%
  tally()

df %>%
  group_by(x) %>%
  tally()
```

`crossing` *Create a data.table from all unique combinations of inputs*

Description

`crossing()` is similar to `expand_grid()` but de-duplicates and sorts its inputs.

Usage

```
crossing(..., .name_repair = "check_unique")
```

Arguments

`...`	Variables to get unique combinations of
`.name_repair`	Treatment of problematic names. See `?vctrs::vec_as_names` for options/details

Examples

```
x <- 1:2
y <- 1:2

crossing(x, y)

crossing(stuff = x, y)
```

`cross_join` *Cross join*

Description

Cross join each row of `x` to every row in `y`.

Usage

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

Arguments

x	A data.frame or data.table
y	A data.frame or data.table
...	Other parameters passed on to methods
suffix	Append created for duplicated column names when using `full_join()`

Examples

```
df1 <- tidytable(x = 1:3)
df2 <- tidytable(y = 4:6)

cross_join(df1, df2)
```

---

c_across *Combine values from multiple columns*

Description

`c_across()` works inside of `mutate_rowwise()`. It uses tidyselect so you can easily select multiple variables.

Usage

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

Arguments

cols	Columns to transform.

Examples

```
df <- data.table(x = runif(6), y = runif(6), z = runif(6))

df %>%
  mutate_rowwise(row_mean = mean(c_across(x:z)))
```

desc *Descending order*

Description

Arrange in descending order. Can be used inside of `arrange()`

Usage

```
desc(x)
```

Arguments

`x`	Variable to arrange in descending order

Examples

```
df <- data.table(
  a = 1:3,
  b = 4:6,
  c = c("a", "a", "b")
)

df %>%
  arrange(c, desc(a))
```

distinct *Select distinct/unique rows*

Description

Retain only unique/distinct rows from an input df.

Usage

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

Arguments

`.df`	A data.frame or data.table
`...`	Columns to select before determining uniqueness. If omitted, will use all columns. `tidyselect` compatible.
`.keep_all`	Only relevant if columns are provided to ... arg. This keeps all columns, but only keeps the first row of each distinct values of columns provided to ... arg.

Examples

```
df <- tidytable(
  x = 1:3,
  y = 4:6,
  z = c("a", "a", "b")
)

df %>%
  distinct()

df %>%
  distinct(z)
```

---

drop_na *Drop rows containing missing values*

Description

Drop rows containing missing values

Usage

```
drop_na(.df, ...)
```

Arguments

.df	A data.frame or data.table
...	Optional: A selection of columns. If empty, all variables are selected. `tidyselect` compatible.

Examples

```
df <- data.table(
  x = c(1, 2, NA),
  y = c("a", NA, "b")
)

df %>%
  drop_na()

df %>%
  drop_na(x)

df %>%
  drop_na(where(is.numeric))
```

dt *Pipeable data.table call*

Description

Pipeable data.table call.

Has *experimental* support for tidy evaluation.

Note: This function does not use data.table's modify-by-reference

Usage

```
dt(.df, ...)
```

Arguments

.df	A data.frame or data.table
...	Arguments passed to data.table call. See ?data.table::[.data.table

Examples

```
df <- tidytable(
  x = 1:3,
  y = 4:6,
  z = c("a", "a", "b")
)

df %>%
  dt(, double_x := x * 2) %>%
  dt(order(-double_x))

# Experimental support for tidy evaluation
add_one <- function(data, col) {
  data %>%
    dt(, {{ col }} := {{ col }} + 1)
}

df %>%
  add_one(x)
```

enframe *Convert a vector to a data.table/tidytable*

Description

Converts named and unnamed vectors to a data.table/tidytable.

Usage

```
enframe(x, name = "name", value = "value")
```

Arguments

x	A vector
name	Name of the column that stores the names. If name = NULL, a one-column tidytable will be returned.
value	Name of the column that stores the values.

Examples

```
vec <- 1:3
names(vec) <- letters[1:3]

enframe(vec)
```

expand *Expand a data.table to use all combinations of values*

Description

Generates all combinations of variables found in a dataset.

expand() is useful in conjunction with joins:

- use with right_join() to convert implicit missing values to explicit missing values
- use with anti_join() to find out which combinations are missing

nesting() is a helper that only finds combinations already present in the dataset.

Usage

```
expand(.df, ..., .name_repair = "check_unique", .by = NULL)

nesting(..., .name_repair = "check_unique")
```

Arguments

.df	A data.frame or data.table
...	Columns to get combinations of
.name_repair	Treatment of duplicate names. See ?vctrs::vec_as_names for options/details
.by	Columns to group by

Examples

```
df <- tidytable(x = c(1, 1, 2), y = c(1, 1, 2))

df %>%
  expand(x, y)

df %>%
  expand(nesting(x, y))
```

---

expand_grid *Create a data.table from all combinations of inputs*

Description

Create a data.table from all combinations of inputs

Usage

```
expand_grid(..., .name_repair = "check_unique")
```

Arguments

...	Variables to get combinations of
.name_repair	Treatment of problematic names. See ?vctrs::vec_as_names for options/details

Examples

```
x <- 1:2
y <- 1:2

expand_grid(x, y)

expand_grid(stuff = x, y)
```

extract *Extract a character column into multiple columns using regex*

Description

Superseded

`extract()` has been superseded by `separate_wider_regex()`.

Given a regular expression with capturing groups, `extract()` turns each group into a new column. If the groups don't match, or the input is `NA`, the output will be `NA`. When you pass same name in the `into` argument it will merge the groups together. Whilst passing `NA` in the `into` arg will drop the group from the resulting `tidytable`

Usage

```
extract(
  .df,
  col,
  into,
  regex = "([[:alnum:]]+)",
  remove = TRUE,
  convert = FALSE,
  ...
)
```

Arguments

`.df`	A data.table or data.frame
`col`	Column to extract from
`into`	New column names to split into. A character vector.
`regex`	A regular expression to extract the desired values. There should be one group (defined by `()`) for each element of `into`
`remove`	If TRUE, remove the input column from the output data.table
`convert`	If TRUE, runs `type.convert()` on the resulting column. Useful if the resulting column should be type integer/double.
`...`	Additional arguments passed on to methods.

Examples

```
df <- data.table(x = c(NA, "a-b-1", "a-d-3", "b-c-2", "d-e-7"))
df %>% extract(x, "A")
df %>% extract(x, c("A", "B"), "([[:alnum:]]+)-([[:alnum:]]+)")

# If no match, NA:
df %>% extract(x, c("A", "B"), "([a-d]+)-([a-d]+)")
# drop columns by passing NA
```

```
df %>% extract(x, c("A", NA, "B"), "([a-d]+)-([a-d]+)-(\\d+)")
# merge groups by passing same name
df %>% extract(x, c("A", "B", "A"), "([a-d]+)-([a-d]+)-(\\d+)")
```

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

Description

Fills missing values in the selected columns using the next or previous entry. Can be done by group.

Supports tidyselect

Usage

```
fill(.df, ..., .direction = c("down", "up", "downup", "updown"), .by = NULL)
```

Arguments

`.df`	A data.frame or data.table
`...`	A selection of columns. `tidyselect` compatible.
`.direction`	Direction in which to fill missing values. Currently "down" (the default), "up", "downup" (first down then up), or "updown" (first up and then down)
`.by`	Columns to group by when filling should be done by group

Examples

```
df <- data.table(
  a = c(1, NA, 3, 4, 5),
  b = c(NA, 2, NA, NA, 5),
  groups = c("a", "a", "a", "b", "b")
)

df %>%
  fill(a, b)

df %>%
  fill(a, b, .by = groups)

df %>%
  fill(a, b, .direction = "downup", .by = groups)
```

filter *Filter rows on one or more conditions*

Description

Filters a dataset to choose rows where conditions are true.

Usage

```
filter(.df, ..., .by = NULL)
```

Arguments

.df	A data.frame or data.table
...	Conditions to filter by
.by	Columns to group by if filtering with a summary function

Examples

```
df <- tidytable(
  a = 1:3,
  b = 4:6,
  c = c("a", "a", "b")
)

df %>%
  filter(a >= 2, b >= 4)

df %>%
  filter(b <= mean(b), .by = c)
```

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

Description

Extract the first, last, or nth value from a vector.

Note: These are simple wrappers around `vctrs::vec_slice()`.

Usage

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

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

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

Arguments

x	A vector
default	The default value if the value doesn't exist.
na_rm	If TRUE ignores missing values.
n	For nth(), a number specifying the position to grab.

Examples

```
vec <- letters

first(vec)
last(vec)
nth(vec, 4)
```

---

fread *Read/write files*

Description

fread() is a simple wrapper around data.table::fread() that returns a tidytable instead of a data.table.

Usage

```
fread(...)
```

Arguments

...	Arguments passed on to data.table::fread

Examples

```
fake_csv <- "A,B
             1,2
             3,4"

fread(fake_csv)
```

get_dummies *Convert character and factor columns to dummy variables*

Description

Convert character and factor columns to dummy variables

Usage

```
get_dummies(
  .df,
  cols = where(~is.character(.x) | is.factor(.x)),
  prefix = TRUE,
  prefix_sep = "_",
  drop_first = FALSE,
  dummify_na = TRUE
)
```

Arguments

.df	A data.frame or data.table
cols	A single column or a vector of unquoted columns to dummify. Defaults to all character & factor columns using `c(where(is.character), where(is.factor))`. `tidyselect` compatible.
prefix	TRUE/FALSE - If TRUE, a prefix will be added to new column names
prefix_sep	Separator for new column names
drop_first	TRUE/FALSE - If TRUE, the first dummy column will be dropped
dummify_na	TRUE/FALSE - If TRUE, NAs will also get dummy columns

Examples

```
df <- tidytable(
  chr = c("a", "b", NA),
  fct = as.factor(c("a", NA, "c")),
  num = 1:3
)

# Automatically does all character/factor columns
df %>%
  get_dummies()

df %>%
  get_dummies(cols = chr)

df %>%
  get_dummies(cols = c(chr, fct), drop_first = TRUE)
```

```
df %>%
  get_dummies(prefix_sep = ".", dummify_na = FALSE)
```

---

group_by *Grouping*

Description

- group_by() adds a grouping structure to a tidytable. Can use tidyselect syntax.
- ungroup() removes grouping.

Usage

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

ungroup(.df, ...)
```

Arguments

.df	A data.frame or data.table
...	Columns to group by
.add	Should grouping cols specified be added to the current grouping

Examples

```
df <- data.table(
  a = 1:3,
  b = 4:6,
  c = c("a", "a", "b"),
  d = c("a", "a", "b")
)

df %>%
  group_by(c, d) %>%
  summarize(mean_a = mean(a)) %>%
  ungroup()

# Can also use tidyselect
df %>%
  group_by(where(is.character)) %>%
  summarize(mean_a = mean(a)) %>%
  ungroup()
```

group_cols *Selection helper for grouping columns*

Description

Selection helper for grouping columns

Usage

```
group_cols()
```

Examples

```
df <- tidytable(
  x = c("a", "b", "c"),
  y = 1:3,
  z = 1:3
)

df %>%
  group_by(x) %>%
  select(group_cols(), y)
```

group_split *Split data frame by groups*

Description

Split data frame by groups. Returns a list.

Usage

```
group_split(.df, ..., .keep = TRUE, .named = FALSE)
```

Arguments

.df	A data.frame or data.table
...	Columns to group and split by. `tidyselect` compatible.
.keep	Should the grouping columns be kept
.named	*experimental*: Should the list be named with labels that identify the group

Examples

```
df <- tidytable(
  a = 1:3,
  b = 1:3,
  c = c("a", "a", "b"),
  d = c("a", "a", "b")
)

df %>%
  group_split(c, d)

df %>%
  group_split(c, d, .keep = FALSE)

df %>%
  group_split(c, d, .named = TRUE)
```

`group_vars` *Get the grouping variables*

Description

Get the grouping variables

Usage

```
group_vars(x)
```

Arguments

`x`	A grouped tidytable

Examples

```
df <- data.table(
  a = 1:3,
  b = 4:6,
  c = c("a", "a", "b"),
  d = c("a", "a", "b")
)

df %>%
  group_by(c, d) %>%
  group_vars()
```

if_all *Create conditions on a selection of columns*

Description

Helpers to apply a filter across a selection of columns.

Usage

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

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

Arguments

.cols	Selection of columns
.fns	Function to create filter conditions
...	Other arguments passed to the function

Examples

```
iris %>%
  filter(if_any(ends_with("Width"), ~ .x > 4))

iris %>%
  filter(if_all(ends_with("Width"), ~ .x > 2))
```

if_else *Fast if_else*

Description

Fast version of base::ifelse().

Usage

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

Arguments

condition	Conditions to test on
true	Values to return if conditions evaluate to TRUE
false	Values to return if conditions evaluate to FALSE
missing	Value to return if an element of test is NA
...	These dots are for future extensions and must be empty.
ptype	Optional ptype to override output type
size	Optional size to override output size

Examples

```
x <- 1:5
if_else(x < 3, 1, 0)

# Can also be used inside of mutate()
df <- data.table(x = x)

df %>%
  mutate(new_col = if_else(x < 3, 1, 0))
```

---

inv_gc *Run invisible garbage collection*

Description

Run garbage collection without the gc() output. Can also be run in the middle of a long pipe chain. Useful for large datasets or when using parallel processing.

Usage

```
inv_gc(x)
```

Arguments

x	Optional. If missing runs gc() silently. Else returns the same object unaltered.

Examples

```
# Can be run with no input
inv_gc()

df <- tidytable(col1 = 1, col2 = 2)

# Or can be used in the middle of a pipe chain (object is unaltered)
df %>%
  filter(col1 < 2, col2 < 4) %>%
  inv_gc() %>%
  select(col1)
```

is_grouped_df *Check if the tidytable is grouped*

Description

Check if the tidytable is grouped

Usage

```
is_grouped_df(x)
```

Arguments

x	An object

Examples

```
df <- data.table(
  a = 1:3,
  b = c("a", "a", "b")
)

df %>%
  group_by(b) %>%
  is_grouped_df()
```

is_tidytable *Test if the object is a tidytable*

Description

This function returns TRUE for tidytables or subclasses of tidytables, and FALSE for all other objects.

Usage

```
is_tidytable(x)
```

Arguments

x	An object

Examples

```
df <- data.frame(x = 1:3, y = 1:3)

is_tidytable(df)

df <- tidytable(x = 1:3, y = 1:3)

is_tidytable(df)
```

`lag` *Get lagging or leading values*

Description

Find the "previous" or "next" values in a vector. Useful for comparing values behind or ahead of the current values.

Usage

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

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

Arguments

`x`	a vector of values
`n`	a positive integer of length 1, giving the number of positions to lead or lag by
`default`	value used for non-existent rows. Defaults to NA.

Examples

```
x <- 1:5

lag(x, 1)
lead(x, 1)

# Also works inside of `mutate()`
df <- tidytable(x = 1:5)

df %>%
  mutate(lag_x = lag(x))
```

left_join *Join two data.tables together*

Description

Join two data.tables together

Usage

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

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

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

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

anti_join(x, y, by = NULL)

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

Arguments

x	A data.frame or data.table
y	A data.frame or data.table
by	A character vector of variables to join by. If NULL, the default, the join will do a natural join, using all variables with common names across the two tables.
suffix	Append created for duplicated column names when using `full_join()`
...	Other parameters passed on to methods
keep	Should the join keys from both x and y be preserved in the output?

Examples

```
df1 <- data.table(x = c("a", "a", "b", "c"), y = 1:4)
df2 <- data.table(x = c("a", "b"), z = 5:6)

df1 %>% left_join(df2)
df1 %>% inner_join(df2)
df1 %>% right_join(df2)
df1 %>% full_join(df2)
df1 %>% anti_join(df2)
```

map *Apply a function to each element of a vector or list*

Description

The map functions transform their input by applying a function to each element and returning a list/vector/data.table.

- `map()` returns a list
- `_lgl()`, `_int`, `_dbl`,`_chr`, `_df` variants return their specified type
- `_dfr` & `_dfc` Return all data frame results combined utilizing row or column binding

Usage

```
map(.x, .f, ...)

map_lgl(.x, .f, ...)

map_int(.x, .f, ...)

map_dbl(.x, .f, ...)

map_chr(.x, .f, ...)

map_dfc(.x, .f, ...)

map_dfr(.x, .f, ..., .id = NULL)

map_df(.x, .f, ..., .id = NULL)

walk(.x, .f, ...)

map_vec(.x, .f, ..., .ptype = NULL)

map2(.x, .y, .f, ...)

map2_lgl(.x, .y, .f, ...)

map2_int(.x, .y, .f, ...)

map2_dbl(.x, .y, .f, ...)

map2_chr(.x, .y, .f, ...)

map2_dfc(.x, .y, .f, ...)

map2_dfr(.x, .y, .f, ..., .id = NULL)
```

```
map2_df(.x, .y, .f, ..., .id = NULL)

map2_vec(.x, .y, .f, ..., .ptype = NULL)

pmap(.l, .f, ...)

pmap_lgl(.l, .f, ...)

pmap_int(.l, .f, ...)

pmap_dbl(.l, .f, ...)

pmap_chr(.l, .f, ...)

pmap_dfc(.l, .f, ...)

pmap_dfr(.l, .f, ..., .id = NULL)

pmap_df(.l, .f, ..., .id = NULL)

pmap_vec(.l, .f, ..., .ptype = NULL)
```

Arguments

.x	A list or vector
.f	A function
...	Other arguments to pass to a function
.id	Whether map_dfr() should add an id column to the finished dataset
.ptype	ptype for resulting vector in map_vec()
.y	A list or vector
.l	A list to use in pmap

Examples

```
map(c(1,2,3), ~ .x + 1)

map_dbl(c(1,2,3), ~ .x + 1)

map_chr(c(1,2,3), as.character)
```

`mutate` *Add/modify/delete columns*

Description

With `mutate()` you can do 3 things:

- Add new columns
- Modify existing columns
- Delete columns

Usage

```
mutate(
  .df,
  ...,
  .by = NULL,
  .keep = c("all", "used", "unused", "none"),
  .before = NULL,
  .after = NULL
)
```

Arguments

`.df`	A data.frame or data.table
`...`	Columns to add/modify
`.by`	Columns to group by
`.keep`	*experimental*: This is an experimental argument that allows you to control which columns from `.df` are retained in the output: • `"all"`, the default, retains all variables. • `"used"` keeps any variables used to make new variables; it's useful for checking your work as it displays inputs and outputs side-by-side. • `"unused"` keeps only existing variables **not** used to make new variables. • `"none"`, only keeps grouping keys (like `transmute()`).
`.before, .after`	Optionally indicate where new columns should be placed. Defaults to the right side of the data frame.

Examples

```
df <- data.table(
  a = 1:3,
  b = 4:6,
  c = c("a", "a", "b")
)
```

```
df %>%
  mutate(double_a = a * 2,
         a_plus_b = a + b)

df %>%
  mutate(double_a = a * 2,
         avg_a = mean(a),
         .by = c)

df %>%
  mutate(double_a = a * 2, .keep = "used")

df %>%
  mutate(double_a = a * 2, .after = a)
```

`mutate_across.` *Mutate multiple columns simultaneously*

Description

Deprecated

Mutate multiple columns simultaneously.

Usage

```
mutate_across.(
  .df,
  .cols = everything(),
  .fns = NULL,
  ...,
  .by = NULL,
  .names = NULL
)
```

Arguments

`.df`	A data.frame or data.table
`.cols`	vector `c()` of unquoted column names. `tidyselect` compatible.
`.fns`	Functions to pass. Can pass a list of functions.
`...`	Other arguments for the passed function
`.by`	Columns to group by
`.names`	A glue specification that helps with renaming output columns. `{.col}` stands for the selected column, and `{.fn}` stands for the name of the function being applied. The default (`NULL`) is equivalent to `"{.col}"` for a single function case and `"{.col}_{.fn}"` when a list is used for `.fns`.

Examples

```
## Not run:
df <- data.table(
  x = rep(1, 3),
  y = rep(2, 3),
  z = c("a", "a", "b")
)

df %>%
  mutate_across.(where(is.numeric), as.character)

df %>%
  mutate_across.(c(x, y), ~ .x * 2)

df %>%
  mutate_across.(everything(), as.character)

df %>%
  mutate_across.(c(x, y), list(new = ~ .x * 2,
                               another = ~ .x + 7))
df %>%
  mutate_across.(
    .cols = c(x, y),
    .fns = list(new = ~ .x * 2, another = ~ .x + 7),
    .names = "{.col}_test_{.fn}"
  )

## End(Not run)
```

---

mutate_rowwise *Add/modify columns by row*

Description

Allows you to mutate "by row". this is most useful when a vectorized function doesn't exist.

Usage

```
mutate_rowwise(
  .df,
  ...,
  .keep = c("all", "used", "unused", "none"),
  .before = NULL,
  .after = NULL
)
```

Arguments

.df	A data.table or data.frame
...	Columns to add/modify
.keep	*experimental*: This is an experimental argument that allows you to control which columns from .df are retained in the output: • "all", the default, retains all variables. • "used" keeps any variables used to make new variables; it's useful for checking your work as it displays inputs and outputs side-by-side. • "unused" keeps only existing variables **not** used to make new variables. • "none", only keeps grouping keys (like transmute()).
.before, .after	Optionally indicate where new columns should be placed. Defaults to the right side of the data frame.

Examples

```
df <- data.table(x = 1:3, y = 1:3 * 2, z = 1:3 * 3)

# Compute the mean of x, y, z in each row
df %>%
  mutate_rowwise(row_mean = mean(c(x, y, z)))

# Use c_across() to more easily select many variables
df %>%
  mutate_rowwise(row_mean = mean(c_across(x:z)))
```

---

n *Number of observations in each group*

Description

Helper function that can be used to find counts by group.

Can be used inside summarize(), mutate(), & filter()

Usage

```
n()
```

Examples

```
df <- data.table(
  x = 1:3,
  y = 4:6,
  z = c("a","a","b")
 )

df %>%
  summarize(count = n(), .by = z)
```

na_if *Convert values to* NA

Description

Convert values to NA.

Usage

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

Arguments

x	A vector
y	Value to replace with NA

Examples

```
vec <- 1:3
na_if(vec, 3)
```

nest *Nest data.tables*

Description

Nest data.tables

Usage

```
nest(.df, ..., .names_sep = NULL)
```

Arguments

.df	A data.table or data.frame
...	Columns to be nested.
.names_sep	If NULL, the names will be left alone. If a string, the names of the columns will be created by pasting together the inner column names and the outer column names.

Examples

```
df <- data.table(
  a = 1:10,
  b = 11:20,
  c = c(rep("a", 6), rep("b", 4)),
  d = c(rep("a", 4), rep("b", 6))
)

df %>%
  nest(data = c(a, b))

df %>%
  nest(data = where(is.numeric))
```

`nest_by` *Nest data.tables*

Description

Nest data.tables by group.

Note: `nest_by()` *does not* return a rowwise tidytable.

Usage

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

Arguments

`.df`	A data.frame or data.table
`...`	Columns to group by. If empty nests the entire data.table. `tidyselect` compatible.
`.key`	Name of the new column created by nesting.
`.keep`	Should the grouping columns be kept in the list column.

Examples

```
df <- data.table(
  a = 1:5,
  b = 6:10,
  c = c(rep("a", 3), rep("b", 2)),
  d = c(rep("a", 3), rep("b", 2))
)

df %>%
  nest_by()

df %>%
  nest_by(c, d)
```

```
df %>%
  nest_by(where(is.character))

df %>%
  nest_by(c, d, .keep = TRUE)
```

`nest_join` *Nest join*

Description

Join the data from y as a list column onto x.

Usage

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

Arguments

`x`	A data.frame or data.table
`y`	A data.frame or data.table
`by`	A character vector of variables to join by. If NULL, the default, the join will do a natural join, using all variables with common names across the two tables.
`keep`	Should the join keys from both `x` and `y` be preserved in the output?
`name`	The name of the list-column created by the join. If `NULL` the name of `y` is used.
`...`	Other parameters passed on to methods

Examples

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

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

new_tidytable *Create a tidytable from a list*

Description

Create a tidytable from a list

Usage

```
new_tidytable(x = list())
```

Arguments

x	A named list of equal-length vectors. The lengths are not checked; it is the responsibility of the caller to make sure they are equal.

Examples

```
l <- list(x = 1:3, y = c("a", "a", "b"))

new_tidytable(l)
```

n_distinct *Count the number of unique values in a vector*

Description

This is a faster version of `length(unique(x))` that calls `data.table::uniqueN()`.

Usage

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

Arguments

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

Examples

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

pick *Selection version of* `across()`

Description

Select a subset of columns from within functions like `mutate()`, `summarize()`, or `filter()`.

Usage

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

Arguments

`...`	Columns to select. Tidyselect compatible.

Examples

```
df <- tidytable(
  x = 1:3,
  y = 4:6,
  z = c("a", "a", "b")
)

df %>%
  mutate(row_sum = rowSums(pick(x, y)))
```

pivot_longer *Pivot data from wide to long*

Description

`pivot_longer()` "lengthens" the data, increasing the number of rows and decreasing the number of columns.

Usage

```
pivot_longer(
  .df,
  cols = everything(),
  names_to = "name",
  values_to = "value",
  names_prefix = NULL,
  names_sep = NULL,
  names_pattern = NULL,
  names_ptypes = NULL,
  names_transform = NULL,
```

```
  names_repair = "check_unique",
  values_drop_na = FALSE,
  values_ptypes = NULL,
  values_transform = NULL,
  fast_pivot = FALSE,
  ...
)
```

Arguments

.df	A data.table or data.frame
cols	Columns to pivot. tidyselect compatible.
names_to	Name of the new "names" column. Must be a string.
values_to	Name of the new "values" column. Must be a string.
names_prefix	Remove matching text from the start of selected columns using regex.
names_sep	If names_to contains multiple values, names_sep takes the same specification as separate().
names_pattern	If names_to contains multiple values, names_pattern takes the same specification as extract(), a regular expression containing matching groups.
names_ptypes, values_ptypes	A list of column name-prototype pairs. See "?vctrs::'theory-faq-coercion"' for more info on vctrs coercion.
names_transform, values_transform	A list of column name-function pairs. Use these arguments if you need to change the types of specific columns.
names_repair	Treatment of duplicate names. See ?vctrs::vec_as_names for options/details.
values_drop_na	If TRUE, rows will be dropped that contain NAs.
fast_pivot	*experimental*: Fast pivoting. If TRUE, the names_to column will be returned as a factor, otherwise it will be a character column. Defaults to FALSE to match tidyverse semantics.
...	Additional arguments to passed on to methods.

Examples

```
df <- data.table(
  x = 1:3,
  y = 4:6,
  z = c("a", "b", "c")
)

df %>%
  pivot_longer(cols = c(x, y))

df %>%
  pivot_longer(cols = -z, names_to = "stuff", values_to = "things")
```

pivot_wider *Pivot data from long to wide*

Description

"Widens" data, increasing the number of columns and decreasing the number of rows.

Usage

```
pivot_wider(
  .df,
  names_from = name,
  values_from = value,
  id_cols = NULL,
  names_sep = "_",
  names_prefix = "",
  names_glue = NULL,
  names_sort = FALSE,
  names_repair = "unique",
  values_fill = NULL,
  values_fn = NULL
)
```

Arguments

.df	A data.frame or data.table
names_from	A pair of arguments describing which column (or columns) to get the name of the output column name_from, and which column (or columns) to get the cell values from values_from). tidyselect compatible.
values_from	A pair of arguments describing which column (or columns) to get the name of the output column name_from, and which column (or columns) to get the cell values from values_from. tidyselect compatible.
id_cols	A set of columns that uniquely identifies each observation. Defaults to all columns in the data table except for the columns specified in names_from and values_from. Typically used when you have additional variables that is directly related. tidyselect compatible.
names_sep	the separator between the names of the columns
names_prefix	prefix to add to the names of the new columns
names_glue	Instead of using names_sep and names_prefix, you can supply a glue specification that uses the names_from columns (and special .value) to create custom column names
names_sort	Should the resulting new columns be sorted.
names_repair	Treatment of duplicate names. See ?vctrs::vec_as_names for options/details.
values_fill	If values are missing, what value should be filled in

values_fn — Should the data be aggregated before casting? If the formula doesn't identify a single observation for each cell, then aggregation defaults to length with a message.

Examples

```
df <- tidytable(
  id = 1,
  names = c("a", "b", "c"),
  vals = 1:3
)

df %>%
  pivot_wider(names_from = names, values_from = vals)

df %>%
  pivot_wider(
    names_from = names, values_from = vals, names_prefix = "new_"
  )
```

---

pull *Pull out a single variable*

Description

Pull a single variable from a data.table as a vector.

Usage

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

Arguments

.df — A data.frame or data.table

var — The column to pull from the data.table as:

- a variable name
- a positive integer giving the column position
- a negative integer giving the column position counting from the right

name — Optional - specifies the column to be used as names for the vector.

Examples

```
df <- data.table(
  x = 1:3,
  y = 1:3
)

# Grab column by name
```

```
df %>%
  pull(y)

# Grab column by position
df %>%
  pull(1)

# Defaults to last column
df %>%
  pull()
```

`relocate` *Relocate a column to a new position*

Description

Move a column or columns to a new position

Usage

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

Arguments

`.df`	A data.frame or data.table
`...`	A selection of columns to move. `tidyselect` compatible.
`.before`	Column to move selection before
`.after`	Column to move selection after

Examples

```
df <- data.table(
  a = 1:3,
  b = 1:3,
  c = c("a", "a", "b"),
  d = c("a", "a", "b")
)

df %>%
  relocate(c, .before = b)

df %>%
  relocate(a, b, .after = c)

df %>%
  relocate(where(is.numeric), .after = c)
```

rename *Rename variables by name*

Description

Rename variables from a data.table.

Usage

```
rename(.df, ...)
```

Arguments

.df	A data.frame or data.table
...	new_name = old_name pairs to rename columns

Examples

```
df <- data.table(x = 1:3, y = 4:6)

df %>%
  rename(new_x = x,
         new_y = y)
```

rename_with *Rename multiple columns*

Description

Rename multiple columns with the same transformation

Usage

```
rename_with(.df, .fn = NULL, .cols = everything(), ...)
```

Arguments

.df	A data.table or data.frame
.fn	Function to transform the names with.
.cols	Columns to rename. Defaults to all columns. tidyselect compatible.
...	Other parameters to pass to the function

Examples

```
df <- data.table(
  x = 1,
  y = 2,
  double_x = 2,
  double_y = 4
)

df %>%
  rename_with(toupper)

df %>%
  rename_with(~ toupper(.x))

df %>%
  rename_with(~ toupper(.x), .cols = c(x, double_x))
```

---

replace_na *Replace missing values*

Description

Replace NAs with specified values

Usage

```
replace_na(.x, replace)
```

Arguments

.x	A data.frame/data.table or a vector
replace	If .x is a data frame, a list() of replacement values for specified columns. If .x is a vector, a single replacement value.

Examples

```
df <- data.table(
  x = c(1, 2, NA),
  y = c(NA, 1, 2)
)

# Using replace_na() inside mutate()
df %>%
  mutate(x = replace_na(x, 5))

# Using replace_na() on a data frame
df %>%
  replace_na(list(x = 5, y = 0))
```

rowwise *Convert to a rowwise tidytable*

Description

Convert to a rowwise tidytable.

Usage

```
rowwise(.df)
```

Arguments

.df	A data.frame or data.table

Examples

```
df <- tidytable(x = 1:3, y = 1:3 * 2, z = 1:3 * 3)

# Compute the mean of x, y, z in each row
df %>%
  rowwise() %>%
  mutate(row_mean = mean(c(x, y, z)))

# Use c_across() to more easily select many variables
df %>%
  rowwise() %>%
  mutate(row_mean = mean(c_across(x:z))) %>%
  ungroup()
```

row_number *Ranking functions*

Description

Ranking functions:

- row_number(): Gives other row number if empty. Equivalent to frank(ties.method = "first") if provided a vector.
- min_rank(): Equivalent to frank(ties.method = "min")
- dense_rank(): Equivalent to frank(ties.method = "dense")
- percent_rank(): Ranks by percentage from 0 to 1
- cume_dist(): Cumulative distribution

Usage

```
row_number(x)

min_rank(x)

dense_rank(x)

percent_rank(x)

cume_dist(x)
```

Arguments

x	A vector to rank

Examples

```
df <- data.table(x = rep(1, 3), y = c("a", "a", "b"))

df %>%
  mutate(row = row_number())
```

---

select *Select or drop columns*

Description

Select or drop columns from a data.table

Usage

```
select(.df, ...)
```

Arguments

.df	A data.frame or data.table
...	Columns to select or drop. Use named arguments, e.g. new_name = old_name, to rename selected variables. `tidyselect` compatible.

Examples

```
df <- data.table(
  x1 = 1:3,
  x2 = 1:3,
  y = c("a", "b", "c"),
  z = c("a", "b", "c")
)
```

```
df %>%
  select(x1, y)

df %>%
  select(x1:y)

df %>%
  select(-y, -z)

df %>%
  select(starts_with("x"), z)

df %>%
  select(where(is.character), x1)

df %>%
  select(new = x1, y)
```

`separate` *Separate a character column into multiple columns*

Description

Superseded

`separate()` has been superseded by `separate_wider_delim()`.

Separates a single column into multiple columns using a user supplied separator or regex.

If a separator is not supplied one will be automatically detected.

Note: Using automatic detection or regex will be slower than simple separators such as "," or ".".

Usage

```
separate(
  .df,
  col,
  into,
  sep = "[^[:alnum:]]+",
  remove = TRUE,
  convert = FALSE,
  ...
)
```

Arguments

`.df`	A data frame
`col`	The column to split into multiple columns
`into`	New column names to split into. A character vector. Use `NA` to omit the variable in the output.

sep	Separator to split on. Can be specified or detected automatically
remove	If TRUE, remove the input column from the output data.table
convert	TRUE calls type.convert() with as.is = TRUE on new columns
...	Arguments passed on to methods

Examples

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

# "sep" can be automatically detected (slower)
df %>%
  separate(x, into = c("c1", "c2"))

# Faster if "sep" is provided
df %>%
  separate(x, into = c("c1", "c2"), sep = ".")
```

---

separate_longer_delim *Split a string into rows*

Description

If a column contains observations with multiple delimited values, separate them each into their own row.

Usage

```
separate_longer_delim(.df, cols, delim, ...)
```

Arguments

.df	A data.frame or data.table
cols	Columns to separate
delim	Separator delimiting collapsed values
...	These dots are for future extensions and must be empty.

Examples

```
df <- data.table(
  x = 1:3,
  y = c("a", "d,e,f", "g,h"),
  z = c("1", "2,3,4", "5,6")
)

df %>%
  separate_longer_delim(c(y, z), ",")
```

separate_rows *Separate a collapsed column into multiple rows*

Description

Superseded

`separate_rows()` has been superseded by `separate_longer_delim()`.

If a column contains observations with multiple delimited values, separate them each into their own row.

Usage

```
separate_rows(.df, ..., sep = "[^[:alnum:].]+", convert = FALSE)
```

Arguments

`.df`	A data.frame or data.table
`...`	Columns to separate across multiple rows. `tidyselect` compatible
`sep`	Separator delimiting collapsed values
`convert`	If TRUE, runs `type.convert()` on the resulting column. Useful if the resulting column should be type integer/double.

Examples

```
df <- data.table(
  x = 1:3,
  y = c("a", "d,e,f", "g,h"),
  z = c("1", "2,3,4", "5,6")
)

separate_rows(df, y, z)

separate_rows(df, y, z, convert = TRUE)
```

separate_wider_delim *Separate a character column into multiple columns*

Description

Separates a single column into multiple columns

Usage

```
separate_wider_delim(
  .df,
  cols,
  delim,
  ...,
  names = NULL,
  names_sep = NULL,
  names_repair = "check_unique",
  too_few = c("align_start", "error"),
  too_many = c("drop", "error"),
  cols_remove = TRUE
)
```

Arguments

.df	A data frame
cols	Columns to separate
delim	Delimiter to separate on
...	These dots are for future extensions and must be empty.
names	New column names to separate into
names_sep	Names separator
names_repair	Treatment of duplicate names. See ?vctrs::vec_as_names for options/details.
too_few	What to do when too few column names are supplied
too_many	What to do when too many column names are supplied
cols_remove	Should old columns be removed

Examples

```
df <- tidytable(x = c("a", "a_b", "a_b", NA))

df %>%
  separate_wider_delim(x, delim = "_", names = c("left", "right"))

df %>%
  separate_wider_delim(x, delim = "_", names_sep = "")
```

---

separate_wider_regex *Separate a character column into multiple columns using regex patterns*

Description

Separate a character column into multiple columns using regex patterns

Usage

```
separate_wider_regex(
  .df,
  cols,
  patterns,
  ...,
  names_sep = NULL,
  names_repair = "check_unique",
  too_few = "error",
  cols_remove = TRUE
)
```

Arguments

.df	A data frame
cols	Columns to separate
patterns	patterns
...	These dots are for future extensions and must be empty.
names_sep	Names separator
names_repair	Treatment of duplicate names. See ?vctrs::vec_as_names for options/details.
too_few	What to do when too few column names are supplied
cols_remove	Should old columns be removed

Examples

```
df <- tidytable(id = 1:3, x = c("m-123", "f-455", "f-123"))

df %>%
  separate_wider_regex(x, c(gender = ".", ".", unit = "\\d+"))
```

---

slice_head *Choose rows in a data.table*

Description

Choose rows in a data.table. Grouped data.tables grab rows within each group.

Usage

```
slice_head(.df, n = 5, .by = NULL)

slice_tail(.df, n = 5, .by = NULL)

slice_max(.df, order_by, n = 1, ..., with_ties = TRUE, .by = NULL)
```

```
slice_min(.df, order_by, n = 1, ..., with_ties = TRUE, .by = NULL)

slice(.df, ..., .by = NULL)

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

Arguments

.df	A data.frame or data.table
n	Number of rows to grab
.by	Columns to group by
order_by	Variable to arrange by
...	Integer row values
with_ties	Should ties be kept together. The default TRUE may return can return multiple rows if they are equal. Use FALSE to ignore ties.
prop	The proportion of rows to select
weight_by	Sampling weights
replace	Should sampling be performed with (TRUE) or without (FALSE, default) replacement

Examples

```
df <- data.table(
  x = 1:4,
  y = 5:8,
  z = c("a", "a", "a", "b")
)

df %>%
  slice(1:3)

df %>%
  slice(1, 3)

df %>%
  slice(1:2, .by = z)

df %>%
  slice_head(1, .by = z)

df %>%
  slice_tail(1, .by = z)

df %>%
  slice_max(order_by = x, .by = z)

df %>%
  slice_min(order_by = y, .by = z)
```

summarize *Aggregate data using summary statistics*

Description

Aggregate data using summary statistics such as mean or median. Can be calculated by group.

Usage

```
summarize(
  .df,
  ...,
  .by = NULL,
  .sort = TRUE,
  .groups = "drop_last",
  .unpack = FALSE
)

summarise(
  .df,
  ...,
  .by = NULL,
  .sort = TRUE,
  .groups = "drop_last",
  .unpack = FALSE
)
```

Arguments

.df	A data.frame or data.table
...	Aggregations to perform
.by	Columns to group by. • A single column can be passed with .by = d. • Multiple columns can be passed with .by = c(c, d) • tidyselect can be used: – Single predicate: .by = where(is.character) – Multiple predicates: .by = c(where(is.character), where(is.factor)) – A combination of predicates and column names: .by = c(where(is.character), b)
.sort	*experimental*: Default TRUE. If FALSE the original order of the grouping variables will be preserved.
.groups	Grouping structure of the result • "drop_last": Drop the last level of grouping • "drop": Drop all groups

- "keep": Keep all groups

.unpack *experimental*: Default FALSE. Should unnamed data frame inputs be unpacked. The user must opt in to this option as it can lead to a reduction in performance.

Examples

```
df <- data.table(
  a = 1:3,
  b = 4:6,
  c = c("a", "a", "b"),
  d = c("a", "a", "b")
)

df %>%
  summarize(avg_a = mean(a),
            max_b = max(b),
            .by = c)

df %>%
  summarize(avg_a = mean(a),
            .by = c(c, d))
```

---

summarize_across. *Summarize multiple columns*

Description

Deprecated

Summarize multiple columns simultaneously

Usage

```
summarize_across.(
  .df,
  .cols = everything(),
  .fns = NULL,
  ...,
  .by = NULL,
  .names = NULL
)

summarise_across.(
  .df,
  .cols = everything(),
  .fns = NULL,
  ...,
  .by = NULL,
  .names = NULL
)
```

Arguments

.df	A data.frame or data.table
.cols	vector c() of unquoted column names. tidyselect compatible.
.fns	Functions to pass. Can pass a list of functions.
...	Other arguments for the passed function
.by	Columns to group by
.names	A glue specification that helps with renaming output columns. {.col} stands for the selected column, and {.fn} stands for the name of the function being applied. The default (NULL) is equivalent to "{.col}" for a single function case and "{.col}_{.fn}" when a list is used for .fns.

Examples

```
## Not run:
df <- data.table(
  a = 1:3,
  b = 4:6,
  z = c("a", "a", "b")
)

# Pass a single function
df %>%
  summarize_across.(c(a, b), mean, na.rm = TRUE)

# Single function using purrr style interface
df %>%
  summarize_across.(c(a, b), ~ mean(.x, na.rm = TRUE))

# Passing a list of functions (with .by)
df %>%
  summarize_across.(c(a, b), list(mean, max), .by = z)

# Passing a named list of functions (with .by)
df %>%
  summarize_across.(c(a, b),
                    list(avg = mean,
                         max = ~ max(.x)),
                    .by = z)

# Use the `.names` argument for more naming control
df %>%
  summarize_across.(c(a, b),
                    list(avg = mean,
                         max = ~ max(.x)),
                    .by = z,
                    .names = "{.col}_test_{.fn}")

## End(Not run)
```

tidytable *Build a data.table/tidytable*

Description

Constructs a data.table, but one with nice printing features.

Usage

```
tidytable(..., .name_repair = "unique")
```

Arguments

...	A set of name-value pairs
.name_repair	Treatment of duplicate names. See ?vctrs::vec_as_names for options/details.

Examples

```
tidytable(x = 1:3, y = c("a", "a", "b"))
```

top_n *Select top (or bottom) n rows (by value)*

Description

Select the top or bottom entries in each group, ordered by wt.

Usage

```
top_n(.df, n = 5, wt = NULL, .by = NULL)
```

Arguments

.df	A data.frame or data.table
n	Number of rows to return
wt	Optional. The variable to use for ordering. If NULL uses the last column in the data.table.
.by	Columns to group by

Examples

```
df <- data.table(
  x = 1:5,
  y = 6:10,
  z = c(rep("a", 3), rep("b", 2))
)

df %>%
  top_n(2, wt = y)

df %>%
  top_n(2, wt = y, .by = z)
```

`transmute` *Add new variables and drop all others*

Description

Unlike `mutate()`, `transmute()` keeps only the variables that you create

Usage

```
transmute(.df, ..., .by = NULL)
```

Arguments

`.df`	A data.frame or data.table
`...`	Columns to create/modify
`.by`	Columns to group by

Examples

```
df <- data.table(
  a = 1:3,
  b = 4:6,
  c = c("a", "a", "b")
)

df %>%
  transmute(double_a = a * 2)
```

uncount *Uncount a data.table*

Description

Uncount a data.table

Usage

```
uncount(.df, weights, .remove = TRUE, .id = NULL)
```

Arguments

.df	A data.frame or data.table
weights	A column containing the weights to uncount by
.remove	If TRUE removes the selected weights column
.id	A string name for a new column containing a unique identifier for the newly uncounted rows.

Examples

```
df <- data.table(x = c("a", "b"), n = c(1, 2))

uncount(df, n)

uncount(df, n, .id = "id")
```

unite *Unite multiple columns by pasting strings together*

Description

Convenience function to paste together multiple columns into one.

Usage

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

Arguments

.df	A data.frame or data.table
col	Name of the new column, as a string.
...	Selection of columns. If empty all variables are selected. tidyselect compatible.
sep	Separator to use between values
remove	If TRUE, removes input columns from the data.table.
na.rm	If TRUE, NA values will be not be part of the concatenation

Examples

```
df <- tidytable(
    a = c("a", "a", "a"),
    b = c("b", "b", "b"),
    c = c("c", "c", NA)
)

df %>%
  unite("new_col", b, c)

df %>%
  unite("new_col", where(is.character))

df %>%
  unite("new_col", b, c, remove = FALSE)

df %>%
  unite("new_col", b, c, na.rm = TRUE)

df %>%
  unite()
```

---

unnest *Unnest list-columns*

Description

Unnest list-columns.

Usage

```
unnest(
  .df,
  ...,
  keep_empty = FALSE,
  .drop = TRUE,
  names_sep = NULL,
  names_repair = "unique"
)
```

Arguments

.df	A data.table
...	Columns to unnest If empty, unnests all list columns. `tidyselect` compatible.
keep_empty	Return `NA` for any `NULL` elements of the list column
.drop	Should list columns that were not unnested be dropped

names_sep	If NULL, the default, the inner column names will become the new outer column names. If a string, the name of the outer column will be appended to the beginning of the inner column names, with names_sep used as a separator.
names_repair	Treatment of duplicate names. See ?vctrs::vec_as_names for options/details.

Examples

```
df1 <- tidytable(x = 1:3, y = 1:3)
df2 <- tidytable(x = 1:2, y = 1:2)
nested_df <-
  data.table(
    a = c("a", "b"),
    frame_list = list(df1, df2),
    vec_list = list(4:6, 7:8)
  )

nested_df %>%
  unnest(frame_list)

nested_df %>%
  unnest(frame_list, names_sep = "_")

nested_df %>%
  unnest(frame_list, vec_list)
```

---

unnest_longer *Unnest a list-column of vectors into regular columns*

Description

Turns each element of a list-column into a row.

Usage

```
unnest_longer(
  .df,
  col,
  values_to = NULL,
  indices_to = NULL,
  indices_include = NULL,
  keep_empty = FALSE,
  names_repair = "check_unique",
  simplify = NULL,
  ptype = NULL,
  transform = NULL
)
```

Arguments

.df	A data.table or data.frame
col	Column to unnest
values_to	Name of column to store values
indices_to	Name of column to store indices
indices_include	Should an index column be included? Defaults to TRUE when col has inner names.
keep_empty	Return NA for any NULL elements of the list column
names_repair	Treatment of duplicate names. See ?vctrs::vec_as_names for options/details.
simplify	Currently not supported. Errors if not NULL.
ptype	Optionally a named list of ptypes declaring the desired output type of each component.
transform	Optionally a named list of transformation functions applied to each component.

Examples

```
df <- tidytable(
  x = 1:3,
  y = list(0, 1:3, 4:5)
)

df %>% unnest_longer(y)
```

---

unnest_wider *Unnest a list-column of vectors into a wide data frame*

Description

Unnest a list-column of vectors into a wide data frame

Usage

```
unnest_wider(
  .df,
  col,
  names_sep = NULL,
  simplify = NULL,
  names_repair = "check_unique",
  ptype = NULL,
  transform = NULL
)
```

Arguments

.df	A data.table or data.frame
col	Column to unnest
names_sep	If NULL, the default, the names will be left as they are. If a string, the inner and outer names will be pasted together with names_sep as the separator.
simplify	Currently not supported. Errors if not NULL.
names_repair	Treatment of duplicate names. See ?vctrs::vec_as_names for options/details.
ptype	Optionally a named list of ptypes declaring the desired output type of each component.
transform	Optionally a named list of transformation functions applied to each component.

Examples

```
df <- tidytable(
  x = 1:3,
  y = list(0, 1:3, 4:5)
)

# Automatically creates names
df %>% unnest_wider(y)

# But you can provide names_sep for increased naming control
df %>% unnest_wider(y, names_sep = "_")
```

%in% *Fast* %in% *and* %notin% *operators*

Description

Check whether values in a vector are in or not in another vector.

Built using data.table::'%chin%' and vctrs::vec_in() for performance.

Usage

```
x %in% y

x %notin% y
```

Arguments

x	A vector of values to check if they exist in y
y	A vector of values to check if x values exist in

Details

Falls back to `base::'%in%'` when x and y don't share a common type. This means that the behaviour of `base::'%in%'` is preserved (e.g. `"1" %in% c(1, 2)` is `TRUE`) but loses the speedup provided by `vctrs::vec_in()`.

Examples

```
df <- tidytable(x = 1:4, y = 1:4)

df %>%
  filter(x %in% c(2, 4))

df %>%
  filter(x %notin% c(2, 4))
```

Index