Package 'tidytable'

April 16, 2021

Title Tidy Interface to 'data.table'

Version 0.6.1

Description A tidy interface to 'data.table' that is 'rlang' compatible,
giving users the speed of 'data.table' with the clean syntax of the tidyverse.

License MIT + file LICENSE

Encoding UTF-8

Imports data.table (>= 1.12.6), glue (>= 1.4.0), lifecycle (>= 0.2.0),
magrittr (>= 1.5), pillar (>= 1.5.0), rlang (>= 0.4.7),
tidyselect (>= 1.1.0), vctrs (>= 0.3.5)

RoxygenNote 7.1.1

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

NeedsCompilation no

Author Mark Fairbanks [aut, cre],
Abdessabour Moutik [ctb],
Matt Carlson [ctb],
Ivan Leung [ctb],
Ross Kennedy [ctb]

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

Repository CRAN

Date/Publication 2021-04-16 14:40:02 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`	Functions to pass. Can pass a 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

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

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

test_df %>%
  summarize.(across.(c(x, y), ~ mean(.x, na.rm = TRUE)), .by = z)

test_df %>%
  arrange.(across.(c(y, z)))
```

arrange. *Arrange/reorder rows*

Description

Order rows in ascending or descending order.

Note: `data.table` orders character columns slightly differently than `dplyr::arrange()` by ordering in the "C-locale". See `?data.table::setorder` for more details.

Usage

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

Arguments

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

Examples

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

test_df %>%
  arrange.(c, -a)

test_df %>%
  arrange.(c, desc(a))
```

arrange_across. *Arrange by a selection of variables*

Description

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

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

Examples

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

test_df %>%
  arrange_across.()

test_df %>%
  arrange_across.(a, desc.)
```

---

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 = c("check_unique", "unique", "universal", "minimal"),
  .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

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

test_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 of values
left, right	Boundary values

Examples

```
between.(1:10, 5, 7)

test_df <- data.table(
  x = sample(1:5, 10, replace = TRUE),
  y = sample(1:5, 10, replace = TRUE)
)

# Typically used in a filter.()
test_df %>%
  filter.(between.(x, 1,3))

# Can also use the %between% operator
test_df %>%
  filter.(x %between% c(1, 3))
```

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)
```

Arguments

...	Sequence of condition/value designations
default	Default value. Set to NA by default.

Examples

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

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

`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.(...)
```

Arguments

`...`	A sequence of two-sided formulas. The left hand side gives the conditions, the right hand side gives the values.

Examples

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

test_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.(...)
```

Arguments

...	Input vectors. Supports dynamic dots.

Examples

```
# Use a single value to replace all missing values
x <- sample(c(1:5, NA, 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())
```

Arguments

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

Examples

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

test_df %>%
  complete.(x, y)

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

count. *Count observations by group*

Description

Returns row counts of the dataset. If bare column names are provided, `count.()` returns counts by group.

Usage

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

Arguments

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

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

test_df %>%
  count.()

test_df %>%
  count.(z)

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

test_df %>%
  count.(z, wt = y, name = "y_sum")

test_df %>%
  count.(z, sort = TRUE)
```

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)
```

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

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

test_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

```
test_df <- data.table(
  a = c(1,2,3),
  b = c(4,5,6),
  c = c("a","a","b")
)

test_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

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

test_df %>%
  distinct.()

test_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

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

```
test_df <- data.table(
  x = 1:3,
  y = 4:5,
  z = c("a", "a", "b")
)

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

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

Usage

```
expand.(.df, ..., .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

Examples

```
test_df <- tidytable(x = 1:2, y = 1:2)

test_df %>%
  expand.(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

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

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

test_df %>%
  fill.(a, b)

test_df %>%
  fill.(a, b, .direction = "downup")

test_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

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

test_df %>%
  filter.(a >= 2, b >= 4)

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

---

get_dummies. *Convert character and factor columns to dummy variables*

Description

Convert character and factor columns to dummy variables

Usage

```
get_dummies.(
  .df,
  cols = c(where(is.character), where(is.factor)),
  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

```
test_df <- tidytable(
  col1 = c("a", "b", "c", NA),
  col2 = as.factor(c("a", "b", NA, "d")),
  var1 = rnorm(4,0,1))

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

# Can select one column
test_df %>%
  get_dummies.(col1)

# Can select one or multiple columns in a vector of unquoted column names
test_df %>%
  get_dummies.(c(col1, col2))

# Can drop certain columns using
test_df %>%
  get_dummies.(c(where(is.character), -col2))

test_df %>%
  get_dummies.(prefix_sep = ".", drop_first = TRUE)

test_df %>%
  get_dummies.(c(col1, col2), dummify_na = FALSE)
```

---

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	Should the list be named with labels that identify the group

Examples

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

test_df %>%
  group_split.(c, d)

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

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

---

ifelse. *Fast ifelse*

Description

ifelse.() utilizes data.table::fifelse() in the background, but automatically converts NAs to their proper type.

Usage

```
ifelse.(conditions, true, false, na = NA)
```

Arguments

conditions	Conditions to test on
true	Values to return if conditions evaluate to TRUE
false	Values to return if conditions evaluate to FALSE
na	Value to return if an element of test is NA.

Examples

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

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

test_df %>%
  mutate.(new_col = ifelse.(x < 3, 1, 0))
```

`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))
```

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_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)
```

`lags.` *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

```
lags.(x, n = 1L, default = NA)

leads.(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

leads.(x, 1)
lags.(x, 1)

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

test_df %>%
  mutate.(lag_x = lags.(x))
```

`left_join.` *Join two data.tables together*

Description

Join two data.tables together

Usage

```
left_join.(x, y, by = NULL)

inner_join.(x, y, by = NULL)

right_join.(x, y, by = NULL)

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

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.()`

Value

A data.table

Examples

```
df1 <- data.table(x = c("a","a","a","b","b"), y = 1:5)
df2 <- data.table(x = c("a","b"), z = 1:2)

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, ...)

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)
```

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
.y	A list or vector

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)
```

Arguments

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

Examples

```
test_df <- data.table(
  a = c(1,2,3),
  b = c(4,5,6),
  c = c("a","a","b")
)

test_df %>%
  mutate.(double_a = a * 2,
```

```
          a_plus_b = a + b)

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

`mutate_across.` *Mutate multiple columns simultaneously*

Description

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

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

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

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

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

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

`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, ...)
```

Arguments

`.df`	A data.table or data.frame
`...`	Columns to add/modify

Examples

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

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

# Use c_across.() to more easily select many variables
test_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

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

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

# The dplyr version `n()` also works
test_df %>%
  summarize.(count = n(), .by = z)
```

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

```
test_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))
)

test_df %>%
  nest.(data = c(a, b))

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

`nest_by.` *Nest data.tables*

Description

Nest data.tables by group

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

```
test_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))
)

test_df %>%
  nest_by.()

test_df %>%
  nest_by.(c, d)
```

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

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

`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 = list(),
  names_transform = list(),
  names_repair = "check_unique",
  values_drop_na = FALSE,
  values_ptypes = list(),
  values_transform = list(),
  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

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

test_df %>%
  pivot_longer.(cols = c(x, y))

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

---

pivot_wider. *Pivot data from long to wide*

Description

pivot_wider.() "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 = "check_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

```
test_df <- data.table(
  a = rep(c("a", "b", "c"), 2),
  b = c(rep("x", 3), rep("y", 3)),
  vals = 1:6
)

test_df %>%
  pivot_wider.(names_from = b, values_from = vals)

test_df %>%
  pivot_wider.(
    names_from = b, 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)
```

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

Examples

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

# Grab column by name
test_df %>%
  pull.(y)

# Grab column by position
test_df %>%
  pull.(1)

# Defaults to last column
test_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

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

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

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

test_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
...	Rename expression like dplyr::rename()

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, .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

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

test_df %>%
  rename_with.(toupper)

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

test_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 = NA)
```

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

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

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

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

row_number. *Return row number*

Description

Returns row number. This function is designed to work inside of mutate.()

Usage

```
row_number.()
```

Examples

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

test_df %>%
  mutate.(row = row_number.())

# The dplyr version `row_number()` also works
test_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

```
test_df <- data.table(
  x = c(1,1,1),
  y = c(4,5,6),
  double_x = c(2,2,2),
  z = c("a","a","b")
)

test_df %>%
  select.(x, y)

test_df %>%
  select.(x:z)

test_df %>%
  select.(-y, -z)

test_df %>%
  select.(starts_with("x"), z)

test_df %>%
```

```
  select.(where(is.character), x)

test_df %>%
  select.(stuff = x, y)
```

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

Description

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 or data.table
`col`	The column to split into multiple columns
`into`	New column names to split into. A character vector.
`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

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

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

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

separate_rows. *Separate a collapsed column into multiple rows*

Description

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

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

separate_rows.(test_df, y, z)

separate_rows.(test_df, y, z, convert = TRUE)
```

slice. *Choose rows in a data.table*

Description

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

Usage

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

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

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

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

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

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

Arguments

.df	A data.frame or data.table
...	Integer row values
.by	Columns to group by
n	Number of rows to grab
order_by	Variable to arrange by
prop	The proportion of rows to select
weight_by	Sampling weights
replace	Should sampling be performed with (TRUE) or without (FALSE, default) replacement

Examples

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

test_df %>%
  slice.(1:3)

test_df %>%
  slice.(1, 3)

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

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

test_df %>%
  slice_tail.(1, .by = z)
```

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

test_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 = FALSE)

summarise.(.df, ..., .by = NULL, .sort = 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*: Should the resulting data.table be sorted by the grouping columns?

Examples

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

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

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

summarize_across. *Summarize multiple columns*

Description

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

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

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

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

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

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

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

`tidytable` *Build a data.table/tidytable*

Description

`tidytable()` constructs a data.table, but one with nice printing features.

Usage

```
tidytable(...)
```

Arguments

`...`	Arguments passed to `data.table()`

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

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

test_df %>%
  top_n.(2, wt = y)

test_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

```
mtcars %>%
  transmute.(displ_l = disp / 61.0237)
```

---

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 = "new_col", ..., 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

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

test_df %>%
  unite.("new_col", b, c)

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

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

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

test_df %>%
  unite.()
```

unnest. *Unnest a nested data.table*

Description

Unnest a nested data.table.

Usage

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

Arguments

.df	A nested data.table
...	Columns to unnest. If empty, unnests all list columns. tidyselect compatible.
.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

```
nested_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))
  ) %>%
  nest_by.(c, d) %>%
  mutate.(pulled_vec = map.(data, ~ pull.(.x, a)))

nested_df %>%
  unnest.(data)

nested_df %>%
  unnest.(data, names_sep = "_")

nested_df %>%
  unnest.(data, pulled_vec)
```

`%notin%` *notin operator*

Description

"not in" operator - works best when used inside `filter.()`

Usage

```
x %notin% y
```

Arguments

x	vector or NULL
y	vector or NULL

Examples

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

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

Index