Package ‘maditr’
May 25, 2021

Type Package

Title Fast Data Aggregation, Modification, and Filtering with Pipes and ‘data.table’

Version 0.8.2

Maintainer Gregory Demin <gdemin@gmail.com>

Depends R (>= 3.3.0)

Imports data.table (>= 1.12.6), magrittr (>= 1.5)

Suggests knitr, tinytest, utils, rmarkdown, stats

Description Provides pipe-style interface for ‘data.table’. Package preserves all ‘data.table’ features without significant impact on performance. ‘let’ and ‘take’ functions are simplified interfaces for most common data manipulation tasks. For example, you can write ‘take(mtcars, mean(mpg), by = am)’ for aggregation or ‘let(mtcars, hp_wt = hp/wt, hp_wt_mpg = hp_wt/mpg)’ for modification. Use ‘take_if/let_if’ for conditional aggregation/modification. Additionally there are some conveniences such as automatic ‘data.frame’ conversion to ‘data.table’.

License GPL-2

URL https://github.com/gdemin/maditr

BugReports https://github.com/gdemin/maditr/issues

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.1.1

NeedsCompilation no

Author Gregory Demin [aut, cre]

Repository CRAN

Date/Publication 2021-05-24 22:00:02 UTC
**R topics documented:**

- coalesce .................................................. 2
- columns .................................................... 3
- copy ......................................................... 4
- dt_count .................................................... 5
- dt_left_join ................................................. 6
- dt_mutate ................................................... 7
- let_if ......................................................... 10
- maditr ....................................................... 17
- query_if ..................................................... 22
- text_expand ................................................ 27
- to_list ........................................................ 28
- to_long ....................................................... 31
- vlookup ...................................................... 35

### Description

It is an alias for data.table fcoalesce. For details see fcoalesce.

### Usage

```r
coalesce(...)```

### Arguments

```
... vectors```

### Value

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

### Examples

```r
# examples from dplyr
x = sample(c(1:5, NA, NA, NA))
coalesce(x, 0L)

y = c(1, 2, NA, NA, 5)
z = c(NA, NA, 3, 4, 5)
coalesce(y, z)```
columns

Selects columns or rows from the data set

Description

- **columns**: select columns from dataset. There are four ways of column selection:
  
  1. Simply by column names
  2. By variable ranges, e.g. `vs:carb`. Alternatively, you can use `from` instead of colon: `vs `to` carb`.
  3. With regular expressions. Characters which start with `^` or end with `$` considered as Perl-style regular expression patterns. For example, `^Petal` returns all variables started with `Petal`. `Width$` returns all variables which end with `Width`. Pattern `^:` matches all variables and pattern `^.*my_str` is equivalent to contains `"my_str"`.
  4. By character variables with interpolated parts. Expression in the curly brackets inside characters will be evaluated in the parent frame with `text_expand`. For example, `{a[1:3]` will be transformed to the names `'a1', 'a2', 'a3'. `cols` is just a shortcut for `columns`. See examples.

- **rows**: select rows from dataset by logical conditions.

Usage

```
columns(data, ...)

cols(data, ...)

rows(data, ...)
```

Arguments

- `data`: data.table/data.frame
- `...`: unquoted or quoted column names, regex selectors or variable ranges for `columns` and logical conditions for `rows`.

Value

- `data.frame/data.table`
Examples

```r
## columns
mtcars %>%
columns(vs:carb, cyl)
mtcars %>%
columns(-am, -cyl)

# regular expression pattern
columns(iris, "^Petal") %>% head()  # variables which start from 'Petal'
columns(iris, "Width\$") %>% head()  # variables which end with 'Width'
# move Species variable to the front.  
# pattern "\." matches all variables
columns(iris, Species, "^\.") %>% head()
# pattern "\.*i" means "contains 'i'"
columns(iris, "\.*i") %>% head()
# numeric indexing - all variables except Species
columns(iris, 1:4) %>% head()

# variable expansion
dims = c("Width", "Length")
columns(iris, "Petal.(dims)") %>% head()

# rows
mtcars %>%
rows(am==0) %>%
head()

# select rows with compound condition
mtcars %>%
rows(am==0 & mpg>mean(mpg))
```

---

**copy**

*Copy an entire object*

**Description**

Mainly intended to copy data.table objects because by default they are modified by reference. See example.

**Usage**

`copy(x)`

**Arguments**

- `x` object
**dt_count**

**Value**

copy of the object 'x'

**Examples**

```r
data(mtcars)
dt_mtcars = as.data.table(mtcars)
dt_mtcars2 = dt_mtcars
dt_mtcars3 = copy(dt_mtcars)
let(dt_mtcars, new = 1)

head(dt_mtcars2) # we see 'new' column
head(dt_mtcars3) # no 'new' column
```

---

**dt_count**  
**Additional useful functions**

**Description**

- `dt_count` calculates number of cases by groups, possibly weighted. `dt_add_count` adds number of cases to existing dataset.
- `dt_top_n` returns top n rows from each group.

**Usage**

```r
dt_count(data, ..., weight = NULL, sort = FALSE, name = "n")
dt_add_count(data, ..., weight = NULL, sort = FALSE, name = "n")
dt_top_n(data, n, by, order_by = NULL)
```

**Arguments**

- `data` : data.table/data.frame data.frame will be automatically converted to data.table.
- `...` : variables to group by.
- `weight` : optional. Unquoted variable name. If provided result will be the sum of this variable by groups.
- `sort` : logical. If TRUE result will be sorted in desending order by resulting variable.
- `name` : character. Name of resulting variable.
- `n` : numeric. number of top cases. If n is negative then bottom values will be returned.
- `by` : list or vector of grouping variables
- `order_by` : unquoted variable name by which result will be sorted. If not specified, defaults to the last variable in the dataset.
dt_left_join

Join two dataframes by common columns.

Description

Do different versions of SQL join operations. See examples.

Usage

dt_left_join(x, y, by = NULL, suffix = c(".x", ".y"))
dt_right_join(x, y, by = NULL, suffix = c(".x", ".y"))
dt_inner_join(x, y, by = NULL, suffix = c(".x", ".y"))
dt_full_join(x, y, by = NULL, suffix = c(".x", ".y"))
dt_semi_join(x, y, by = NULL)
dt_anti_join(x, y, by = NULL)

Arguments

x data.frame or data.table
y data.frame or data.table
by a character vector of variables to join by. If NULL, the default, *_join() will do a natural join, using all variables with common names across the two tables. A message lists the variables so that you can check they’re right (to suppress the message, simply explicitly list the variables that you want to join). To join by different variables on x and y use a named vector. For example, by = c("a" = "b") will match x.a to y.b.
suffix If there are non-joined duplicate variables in x and y, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.
Value

data.table

Examples

```
workers = fread("
    name company
    Nick Acme
    John Ajax
    Daniela Ajax
")

positions = fread("
    name position
    John designer
    Daniela engineer
    Cathie manager
")

workers %>% dt_inner_join(positions)
workers %>% dt_left_join(positions)
workers %>% dt_right_join(positions)
workers %>% dt_full_join(positions)

# filtering joins
workers %>% dt_anti_join(positions)
workers %>% dt_semi_join(positions)

# To suppress the message, supply 'by' argument
workers %>% dt_left_join(positions, by = "name")

# Use a named 'by' if the join variables have different names
positions2 = setNames(positions, c("worker", "position")) # rename first column in 'positions'
workers %>% dt_inner_join(positions2, by = c("name" = "worker"))
```

---

**dt_mutate**

'dplyr'-like interface for **data.table**.

**Description**

Subset of 'dplyr' verbs to work with **data.table**. Note that there is no `group_by` verb - use by or `keyby` argument when needed.

- `dt_mutate` adds new variables or modify existing variables. If data is **data.table** then it modifies in-place.
- `dt_summarize` computes summary statistics. Splits the data into subsets, computes summary statistics for each, and returns the result in the "**data.table" form.
- `dt_summarize_all` is the same as `dt_summarize` but work over all non-grouping variables.
dt_mutate

- **dt_mutate** selects rows/cases where conditions are true. Rows where the condition evaluates to NA are dropped.
- **dt_select** selects column/variables from the data set. Range of variables are supported, e.g. vs:carb. Characters which start with '^' or end with '$' considered as Perl-style regular expression patterns. For example, '^Petal' returns all variables started with 'Petal'. 'Width$' returns all variables which end with 'Width'. Pattern '^.' matches all variables and pattern '^.my_str' is equivalent to contains "my_str". See examples.
- **dt_arrange** sorts dataset by variable(-s). Use '-' to sort in descending order. If data is data.table then it modifies in-place.

**Usage**

```r
dt_mutate(data, ..., by)
dt_summarize(data, ..., by, keyby, fun = NULL)
dt_summarize_all(data, fun, by, keyby)
dt_summarise(data, ..., by, keyby, fun = NULL)
dt_summarise_all(data, fun, by, keyby)
dt_select(data, ...)
dt_filter(data, ...)
dt_arrange(data, ..., na.last = FALSE)
```

**Arguments**

- `data` data.table/data.frame data.frame will be automatically converted to data.table. dt_mutate modify data.table object in-place.
- `...` List of variables or name-value pairs of summary/modifications functions. The name will be the name of the variable in the result. In the mutate function we can use `a = b` or `a := b` notation. Advantages of `:=` are multiassignment (c("a","b") := list(1,2)) and parametric assignment ((a) := 2).
- `by` unquoted name of grouping variable of list of unquoted names of grouping variables. For details see data.table.
- `keyby` Same as by, but with an additional `setkey()` run on the by columns of the result, for convenience. It is common practice to use `keyby=` routinely when you wish the result to be sorted. For details see data.table.
- `fun` function which will be applied to all variables in dt_summarize and dt_summarize_all.
- `na.last` logical. FALSE by default. If TRUE, missing values in the data are put last; if FALSE, they are put first.

**Value**

data.table
Examples

# examples from 'dplyr'
# newly created variables are available immediately
mtcars %>%
  dt_mutate(
    cyl2 = cyl * 2,
    cyl4 = cyl2 * 2
  ) %>%
  head()

# you can also use dt_mutate() to remove variables and
# modify existing variables
mtcars %>%
  dt_mutate(
    mpg = NULL,
    disp = disp * 0.0163871 # convert to litres
  ) %>%
  head()

# window functions are useful for grouped mutates
mtcars %>%
  dt_mutate(
    rank = rank(-mpg, ties.method = "min"),
    keyby = cyl)

# You can drop variables by setting them to NULL
mtcars %>% dt_mutate(cyl = NULL) %>% head()

# A summary applied without by returns a single row
mtcars %>%
  dt_summarise(mean = mean(disp), n = .N)

# Usually, you'll want to group first
mtcars %>%
  dt_summarise(mean = mean(disp), n = .N, by = cyl)

# Multiple 'by' - variables
mtcars %>%
  dt_summarise(cyl_n = .N, by = list(cyl, vs))

# Newly created summaries immediately
# doesn't overwrite existing variables
mtcars %>%
  dt_summarise(disp = mean(disp),
               sd = sd(disp),
               by = cyl)
# You can group by expressions:
```r
mtcars %>%
  dt_summarise_all(mean, by = list(vsam = vs + am))
```

# filter by condition
```r
mtcars %>%
  dt_filter(am == 0)
```

# filter by compound condition
```r
mtcars %>%
  dt_filter(am == 0, mpg > mean(mpg))
```

# select
```r
mtcars %>% dt_select(vs:carb, cyl)
mtcars %>% dt_select(-am, -cyl)
```

# regular expression pattern
```r
dt_select(iris, '^Petal') # variables which start from 'Petal'
dt_select(iris, 'Width$') # variables which end with 'Width'
# move Species variable to the front.
# pattern "^." matches all variables
dt_select(iris, Species, "^.")
# pattern "^.*i" means "contains 'i'"
dt_select(iris, "^.*i")
dt_select(iris, 1:4) # numeric indexing - all variables except Species
```

# sorting
```r
dt_arrange(mtcars, cyl, disp)
dt_arrange(mtcars, -disp)
```

---

**let_if**

Modify, aggregate, select or filter data.frame/data.table

**Description**

- `let` adds new variables or modify existing variables. `let_if` make the same thing on the subset of rows.
- `take/take_if` aggregate data or aggregate subset of the data.
- `let_all` applies expressions to all variables in the dataset. It is also possible to modify the subset of the variables.
- `take_all` aggregates all variables in the dataset. It is also possible to aggregate the subset of the variables.

All functions return `data.table`. Expression in the `take_all` and `let_all` can use predefined variables: `:.x` is a value of current variable , `.name` is a name of the variable and `.index` is sequential number of the variable. `.value` is an alias to `.x`. 
- Add new variables: `let(mtcars, new_var = 42, new_var2 = new_var*hp)`
- Select variables: `take(mtcars, am, vs, mpg)`
- Aggregate data: `take(mtcars, mean_mpg = mean(mpg), by = am)`
- Aggregate all non-grouping columns: `take_all(mtcars, mean = mean(.x), sd = sd(.x), n = .N, by = am)`
- Aggregate all numeric columns: `take_all(iris, if(is.numeric(.x)) mean(.x))`
- To modify all non-grouping variables:

```r
iris %>%
  let_all(
    scaled = (.x - mean(.x))/sd(.x),
    by = Species)
```

- Aggregate specific columns: `take_all(iris, if(startsWith(.name, "Sepal")) mean(.x))`

You can use 'columns' inside expression in the 'take'/let'. 'columns' will be replaced with data.table with selected columns. In 'let' in the expressions with ':=', 'cols' or '%to%' can be placed in the left part of the expression. It is useful for multiple assignment. There are four ways of column selection:

1. Simply by column names
2. By variable ranges, e.g. `vs:carb`. Alternatively, you can use '%to%' instead of colon: `vs %to% carb`.
3. With regular expressions. Characters which start with '^' or end with '$' considered as Perl-style regular expression patterns. For example, '^Petal' returns all variables started with 'Petal'. 'Width$' returns all variables which end with 'Width'. Pattern '^.' matches all variables and pattern '^.*my_str' is equivalent to contains "my_str".
4. By character variables with interpolated parts. Expression in the curly brackets inside characters will be evaluated in the parent frame with `text_expand`. For example, `a{1:3}` will be transformed to the names 'a1', 'a2', 'a3'. 'cols' is just a shortcut for 'columns'. See examples.

**Usage**

```r
let_if(data, i, ..., by, keyby)
take_if(data, i, ..., by, keyby, .SDcols, autoname = TRUE, fun = NULL)
take(data, ..., by, keyby, .SDcols, autoname = TRUE, fun = NULL)
let(data, ..., by, keyby)
```

## S3 method for class 'data.frame'
let(data, ..., by, keyby, i)

## S3 method for class 'etable'
let(data, ..., by, keyby, i)

sort_by(data, ..., na.last = FALSE)

let_all(data, ..., by, keyby, .SDcols, suffix = TRUE, sep = "_", i)

take_all(data, ..., by, keyby, .SDcols, suffix = TRUE, sep = "_", i)

Arguments

data  data.table/data.frame data.frame will be automatically converted to data.table. `let` modify data.table object in-place.

i  integer/logical vector. Supposed to use to subset/conditional modifications of `data`. For details see `data.table`

...  List of variables or name-value pairs of summary/modifications functions. The name will be the name of the variable in the result. In the `let` and `take` functions we can use `a = b` or `a := b` notation. Advantages of `:=` is parametric assignment, e.g. `(a) := 2` create variable with name which are stored in `a`. In `let :=` can be used for multiassignment (`c("a","b") := list(1,2)`). Expression in the `take_all` and `let_all` can use predefined variables: `.x` is a value of current variable, `.name` is a name of the variable and `.index` is sequential number of the variable. `.value` is is an alias to `.x`.

by  unquoted name of grouping variable of list of unquoted names of grouping variables. For details see `data.table`

keyby  Same as `by`, but with an additional `setkey()` run on the by columns of the result, for convenience. It is common practice to use `keyby=` routinely when you wish the result to be sorted. For details see `data.table`.

.SDcols  Specifies the columns of `x` to be included in the special symbol `.SD` which stands for Subset of `data.table`. May be character column names or numeric positions. For details see `data.table`.

autoname  logical. TRUE by default. Should we create names for unnamed expressions in `take`?

fun  Function which will be applied to all variables in `take`. If there are no variables in `take` then it will be applied to all non-grouping variables in the `data`.

na.last  logical. FALSE by default. If TRUE, missing values in the data are put last; if FALSE, they are put first.

suffix  logical TRUE by default. For 'let_all'/`take_all`. If TRUE than we append summary name to the end of the variable name. If FALSE summary name will be added at the beginning of the variable name.

sep  character. "_" by default. Separator between the old variables name and prefix or suffix for 'let_all' and 'take_all'.
Value

data.table. let returns its result invisibly.

Examples

# examples form 'dplyr' package
data(mtcars)

# Newly created variables are available immediately
mtcars %>%
  let(
    cyl2 = cyl * 2,
    cyl4 = cyl2 * 2
  ) %>% head()

# You can also use let() to remove variables and
# modify existing variables
mtcars %>%
  let(
    mpg = NULL,
    disp = disp * 0.0163871 # convert to litres
  ) %>% head()

# window functions are useful for grouped computations
mtcars %>%
  let(rank = rank(-mpg, ties.method = "min"),
      by = cyl) %>%
  head()

# You can drop variables by setting them to NULL
mtcars %>% let(cyl = NULL) %>% head()

# keeps all existing variables
mtcars %>%
  let(displ_l = disp / 61.0237) %>%
  head()

# keeps only the variables you create
mtcars %>%
  take(displ_l = disp / 61.0237)

# can refer to both contextual variables and variable names:
var = 100
mtcars %>%
  let(cyl = cyl * var) %>%
  head()

# A 'take' with summary functions applied without 'by' argument returns an aggregated data
mtcars %>%
let_if

take(mean = mean(disp), n = .N)

# Usually, you’ll want to group first
mtcars %>%
  take(mean = mean(disp), n = .N, by = cyl)

# You can group by expressions:
mtcars %>%
  take_all(mean, by = list(vsam = vs + am))

# modify all non-grouping variables in-place
mtcars %>%
  let_all((.x - mean(.x))/sd(.x), by = am) %>%
  head()

# modify all non-grouping variables to new variables
mtcars %>%
  let_all(scaled = (.x - mean(.x))/sd(.x), by = am) %>%
  head()

# conditionally modify all variables
iris %>%
  let_all(mean = if(is.numeric(.x)) mean(.x)) %>%
  head()

# modify all variables conditionally on name
iris %>%
  let_all(
    mean = if(startsWith(.name, "Sepal")) mean(.x),
    median = if(startsWith(.name, "Petal")) median(.x),
    by = Species
  ) %>%
  head()

# aggregation with 'take_all'
mtcars %>%
  take_all(mean = mean(.x), sd = sd(.x), n = .N, by = am)

# conditionally aggregate all variables
iris %>%
  take_all(mean = if(is.numeric(.x)) mean(.x))

# aggregate all variables conditionally on name
iris %>%
  take_all(
    mean = if(startsWith(.name, "Sepal")) mean(.x),
    median = if(startsWith(.name, "Petal")) median(.x),
    by = Species
  )

# parametric evaluation:
var = quote(mean(cyl))
mtcars %>%
```r
let(mean_cyl = eval(var)) %>%
head()
take(mtcars, eval(var))

# all together
new_var = "mean_cyl"
mtcars %>%
let((new_var) := eval(var)) %>%
head()
take(mtcars, (new_var) := eval(var))

# variable selection

# range selection
iris %>%
let(
  avg = rowMeans(Sepal.Length %to% Petal.Width)
) %>%
head()

# multiassignment
iris %>%
let(
  # starts with Sepal or Petal
  multipled1 %to% multipled4 := cols("^(Sepal|Petal)\)*2
) %>%
head()

mtcars %>%
let(
  # text expansion
  cols("scaled_{names(mtcars)}") := lapply(cols("(names(mtcars))"), scale)
) %>%
head()

# range selection in 'by'
# range selection + additional column
mtcars %>
take(
  res = sum(cols(mpg, disp %to% drat)),
  by = vs %to% gear
)
```

# examples from data.table
dat = data.table(
  x=rep(c("b","a","c"), each=3),
  y=c(1,3,6),
  v=1:9
)

# basic row subset operations

```r
take_if(dat, 2)  # 2nd row
take_if(dat, 3:2)  # 3rd and 2nd row
take_if(dat, order(x))  # no need for order(dat$x)
take_if(dat, y>2)  # all rows where dat$y > 2
take_if(dat, y>2 & v>5)  # compound logical expressions
take_if(dat, 1:2:4)  # all rows other than 2:4
take_if(dat, -(2:4))  # same
```

# select|compute columns

```r
take(dat, v)  # v column (as data.table)
take(dat, sum(v))  # return data.table with sum of v (column autonamed 'sum(v)')
take(dat, sv = sum(v))  # same, but column named "sv"
take(dat, v, v*2)  # return two column data.table, v and v*2
```

# subset rows and select|compute

```r
take_if(dat, 2:3, sum(v))  # sum(v) over rows 2 and 3
take_if(dat, 2:3, sv = sum(v))  # same, but return data.table with column sv
```

# grouping operations

```r
take(dat, sum(v), by = x)  # ad hoc by, order of groups preserved in result
take(dat, sum(v), keyby = x)  # same, but order the result on by cols
```

# all together now

```r
take_if(dat, x!="a", sum(v), by=x)  # get sum(v) by "x" for each x != "a"
```

# more on special symbols, see also ?"data.table::special-symbols"

```r
take_if(dat, .N)  # last row
take(dat, .N)  # total number of rows in DT
take(dat, .N, by=x)  # number of rows in each group
```

```r
take(dat, .I[1], by=x)  # row number in DT corresponding to each group
```

# add/update/delete by reference

```r
[] at the end of expression is for autoprinting

let(dat, grp = .GRP, by=x)[]  # add a group counter column
let(dat, z = 42L)[]  # add new column by reference
let(dat, z = NULL)[]  # remove column by reference
let_if(dat, x=="a", v = 42L)[]  # subassign to existing v column by reference
let_if(dat, x=="b", v2 = 84L)[]  # subassign to new column by reference (NA padded)
```

```r
let(dat, m = mean(v), by=x)[]  # add new column by reference by group
```

# advanced usage

```r
dat = data.table(x=rep(c("b","a","c"), each=3),
                 v=c(1,1,1,2,2,1,1,2,2),
                 y=c(1,3,6),
                 a=1:9,
                 b=9:1)
```
```r
take(dat, sum(v), by=list(y%%2))  # expressions in by

take(dat, sum(v), by=list(bool = y%%2))  # same, using a named list to change by column name

take_all(dat, sum, by=x)  # sum of all (other) columns for each group

take(dat,
    MySum=sum(v),
    MyMin=min(v),
    MyMax=max(v),
    by = list(x, y%%2)  # by 2 expressions
)

take(dat, seq = min(a):max(b), by=x)  # j is not limited to just aggregations

dat %>%
    take(V1 = sum(v), by=x) %>%
    take_if(V1<20)  # compound query

dat %>%
    take(V1 = sum(v), by=x) %>%
    sort_by(-V1) %>%
    head()
```

---

**maditr**

*maditr: Pipe-Style Interface for ‘data.table’*

**Description**

Package provides pipe-style interface for `data.table`. It preserves all `data.table` features without significant impact on performance. ’let’ and ’take’ functions are simplified interfaces for most common data manipulation tasks.

**Details**

- To select rows from data: `rows(mtcars,am==0)`
- To select columns from data: `columns(mtcars,mpg,vs:carb)`
- To aggregate data: `take(mtcars,mean_mpg = mean(mpg),by = am)`
- To aggregate all non-grouping columns: `take_all(mtcars,mean,by = am)`
- To aggregate several columns with one summary: `take(mtcars,mpg,hp,fun = mean,by = am)`
- To get total summary skip by argument: `take_all(mtcars,mean)`
- Use magrittr pipe ’%>%’ to chain several operations:

```r
mtcars %>%
    let(mpg_hp = mpg/hp) %>%
    take(mean(mpg_hp), by = am)
```
• To modify variables or add new variables:

```r
mtcars %>%
  let(new_var = 42,
      new_var2 = new_var*hp) %>%
  head()
```

• To modify all non-grouping variables:

```r
iris %>%
  let_all(
      scaled = (.x - mean(.x))/sd(.x),
      by = Species) %>%
  head()
```

• To drop variable assign NULL: `let(mtcars, am = NULL) %>% head()`

• To aggregate all variables conditionally on name:

```r
iris %>%
  take_all(
      mean = if(startsWith(.name, "Sepal")) mean(.x),
      median = if(startsWith(.name, "Petal")) median(.x),
      by = Species
  )
```

• For parametric assignment use `:=`:

```r
new_var = "my_var"
old_var = "mpg"
mtcars %>%
  let((new_var) := get(old_var)*2) %>%
  head()
```

• For more sophisticated operations see `query`/`query_if`: these functions translates its arguments one-to-one to `[.data.table` method. Additionally there are some conveniences such as automatic `data.frame` conversion to `data.table`.

**Examples**

```r
# examples form 'dplyr' package
data(mtcars)

# Newly created variables are available immediately
mtcars %>%
  let(
      cyl2 = cyl * 2,
      cyl4 = cyl2 * 2
  ) %>%
  head()
```
# You can also use `let()` to remove variables and modify existing variables
```r
mtcars %>%
  let(
    mpg = NULL,
    disp = disp * 0.0163871  # convert to litres
  ) %>%
  head()
```

# window functions are useful for grouped computations
```r
mtcars %>%
  let(rank = rank(-mpg, ties.method = "min"),
        by = cyl) %>%
  head()
```

# You can drop variables by setting them to NULL
```r
mtcars %>% let(cyl = NULL) %>% head()
```

# keeps all existing variables
```r
mtcars %>%
  let(displ_l = disp / 61.0237) %>%
  head()
```

# keeps only the variables you create
```r
mtcars %>%
  take(displ_l = disp / 61.0237)
```

# can refer to both contextual variables and variable names:
```r
var = 100
mtcars %>%
  let(cyl = cyl * var) %>%
  head()
```

# select rows
```r
mtcars %>%
  rows(am==0) %>%
  head()
```

# select rows with compound condition
```r
mtcars %>%
  rows(am==0 & mpg>mean(mpg))
```

# select columns
```r
mtcars %>%
  columns(vs:carb, cyl)
```

mtcars %>%
  columns(-am, -cyl)

# regular expression pattern
```r
columns(iris, "^Petal") # variables which start from 'Petal'
```
columns(iris, "Width$") # variables which end with 'Width'

# move Species variable to the front
# pattern "\^\." matches all variables
columns(iris, Species, "^\.")

# pattern "\^\.*al" means "contains 'al'"
columns(iris, "\^\.*al")

# numeric indexing - all variables except Species
columns(iris, 1:4)

# A 'take' with summary functions applied without 'by' argument returns an aggregated data
mtcars %>%
  take(mean = mean(disp), n = .N)

# Usually, you'll want to group first
mtcars %>%
  take(mean = mean(disp), n = .N, by = cyl)

# You can group by expressions:
mtcars %>%
  take_all(mean, by = list(vsam = vs + am))

# modify all non-grouping variables in-place
mtcars %>%
  let_all((.x - mean(.x))/sd(.x), by = am) %>%
  head()

# modify all non-grouping variables to new variables
mtcars %>%
  let_all(scaled = (.x - mean(.x))/sd(.x), by = am) %>%
  head()

# conditionally modify all variables
iris %>%
  let_all(mean = if(is.numeric(.x)) mean(.x)) %>%
  head()

# modify all variables conditionally on name
iris %>%
  let_all(
    mean = if(startsWith(.name, "Sepal")) mean(.x),
    median = if(startsWith(.name, "Petal")) median(.x),
    by = Species
  ) %>%
  head()

# aggregation with 'take_all'
mtcars %>%
  take_all(mean = mean(.x), sd = sd(.x), n = .N, by = am)

# conditionally aggregate all variables
iris %>%
take_all(mean = if(is.numeric(.x)) mean(.x))

# aggregate all variables conditionally on name
iris %>%
take_all(
  mean = if(startsWith(.name, "Sepal")) mean(.x),
  median = if(startsWith(.name, "Petal")) median(.x),
  by = Species
)

# parametric evaluation:
var = quote(mean(cyl))
mtcars %>%
  let(mean_cyl = eval(var)) %>%
  head()
  take(mtcars, eval(var))

# all together
new_var = "mean_cyl"
mtcars %>%
  let((new_var) := eval(var)) %>%
  head()
  take(mtcars, (new_var) := eval(var))

# variable selection
# range selection
iris %>%
  let(
    avg = rowMeans(Sepal.Length %to% Petal.Width)
  ) %>%
  head()

# multiassignment
iris %>%
  let(
    # starts with Sepal or Petal
    multiplied1 %to% multiplied4 := cols("^(Sepal|Petal)")*2
  ) %>%
  head()

mtcars %>%
  let(
    # text expansion
    cols("scaled_{names(mtcars)}") := lapply(cols("{names(mtcars)}"), scale)
  ) %>%
  head()

# range selection in 'by'
# range selection + additional column
mtcars %>%
  take(
    res = sum(cols(mpg, disp %to% drat)),
    by = vs %to% gear
  )

query_if

---

**query_if**

One-to-one interface for data.table `[` method

---

**Description**

Quote from data.table:

```r
query(data, j, by) # + extra arguments
  | | -------> grouped by what?
  | | what to do?
```

or,

```r
query_if(data, i, j, by) # + extra arguments
  | | | -------> grouped by what?
  | | what to do?
  | ---+ on which rows?
```

If you don’t need `'i'` argument, use `'query'`. In this case you can avoid printing leading comma inside brackets to denote empty `'i'`.

**Usage**

```r
query_if(
  data,
  i,
  j,
  by,
  keyby,
  with = TRUE,
  nomatch = getOption("datatable.nomatch"),
  mult = "all",
  roll = FALSE,
  rollends = if (roll == "nearest") c(TRUE, TRUE) else if (roll >= 0) c(FALSE, TRUE)
    else c(TRUE, FALSE),
  which = FALSE,
  .SDcols,
  verbose = getOption("datatable.verbose"),
  allow.cartesian = getOption("datatable.allow.cartesian"),
  ...)
```
```r
query_if

  drop = NULL,
  on = NULL

)

query(
  data,
  j,
  by,
  keyby,
  with = TRUE,
  nomatch = getOption("datatable.nomatch"),
  mult = "all",
  roll = FALSE,
  rollends = if (roll == "nearest") c(TRUE, TRUE) else if (roll >= 0) c(FALSE, TRUE)
     else c(TRUE, FALSE),
  which = FALSE,
  .SDcols,
  verbose = getOption("datatable.verbose"),
  allow.cartesian = getOption("datatable.allow.cartesian"),
  drop = NULL,
  on = NULL
)

Arguments

data          data.table/data.frame data.frame will be automatically converted to data.table.
i            Integer, logical or character vector, single column numeric matrix, expression
of column names, list, data.frame or data.table. integer and logical vectors work
the same way they do in [.data.frame except logical NAs are treated as FALSE.
expression is evaluated within the frame of the data.table (i.e. it sees column
names as if they are variables) and can evaluate to any of the other types. For
details see data.table
j          When with=TRUE (default), j is evaluated within the frame of the data.table;
i.e., it sees column names as if they are variables. This allows to not just select
columns in j, but also compute on them e.g., x[, a] and x[, sum(a)] returns x$a
and sum(x$a) as a vector respectively. x[, (a, b)] and x[, (sa=sum(a), sb=sum(b))]
returns a two column data.table each, the first simply selecting columns a, b and
the second computing their sums. For details see data.table.
by          unquoted name of grouping variable of list of unquoted names of grouping vari-
ables. For details see data.table
keyby        Same as by, but with an additional setkey() run on the by columns of the result,
for convenience. It is common practice to use 'keyby=\' routinely when you wish
the result to be sorted. For details see data.table
with       logical. For details see data.table.
nomatch     Same as nomatch in match. For details see data.table.
mult       For details see data.table.
```
roll
For details see data.table.
rollends
For details see data.table.
which
For details see data.table.
.SDcols
Specifies the columns of x to be included in the special symbol .SD which stands for Subset of data.table. May be character column names or numeric positions. For details see data.table.
verbose
logical. For details see data.table.
allow.cartesian
For details see data.table.
drop
For details see data.table.
on
For details see data.table.

Value
It depends. For details see data.table.

Examples

# examples from data.table
dat = data.table(x=rep(c("b","a","c"),each=3), y=c(1,3,6), v=1:9)
dat
# basic row subset operations
query_if(dat, 2)                       # 2nd row
query_if(dat, 3:2)                     # 3rd and 2nd row
query_if(dat, order(x))               # no need for order(dat$x)
query_if(dat, y>2)                     # all rows where dat$y > 2
query_if(dat, y>2 & v>5)               # compound logical expressions
query_if(dat, !2:4)                    # all rows other than 2:4
query_if(dat, -(2:4))                  # same

# select|compute columns data.table way
query(dat, v)                          # v column (as vector)
query(dat, list(v))                    # v column (as data.table)
query(dat, sum(v))                     # sum of column v, returned as vector
query(dat, list(sum(v)))              # same, but return data.table (column autonamed V1)
query(dat, list(v, v*2))               # return two column data.table, v and v*2

# subset rows and select|compute data.table way
query_if(dat, 2:3, sum(v))             # sum(v) over rows 2 and 3, return vector
query_if(dat, 2:3, list(sum(v)))      # same, but return data.table with column V1
query_if(dat, 2:3, list(sv=sum(v)))   # same, but return data.table with column sv
query_if(dat, 2:5, cat(v, "\n"))      # just for j's side effect

# select columns the data.frame way
query(dat, 2, with=FALSE)             # 2nd column, returns a data.table always
colNum = 2
query(dat, colNum, with=FALSE)       # same, equivalent to DT[, .SD, .SDcols=colNum]

# grouping operations - j and by
query(dat, sum(v), by=x)              # ad hoc by, order of groups preserved in result
query_if(dat, sum(v), keyby=x)  # same, but order the result on by cols
query_if(order(x), by=x)  # same but by chaining expressions together

# fast ad hoc row subsets (subsets as joins)
# same as x == "a" but uses binary search (fast)
query_if(dat, "a", on="x")
# same, for convenience, no need to quote every column
query_if(dat, "a", on=list(x))
query_if(dat, .("a"), on="x")  # same
# same, single "==" internally optimised to use binary search (fast)
query_if(dat, x=="a")
# not yet optimized, currently vector scan subset
query_if(dat, x!="b" | y!=3)
# join on columns x,y of 'dat'; uses binary search (fast)
query_if(dat, .("b", 3), on=c("x", "y"))
query_if(dat, .("b", 3), on=list(x, y))  # same, but using on=list()
query_if(dat, .("b", 1:2), on=c("x", "y"))  # no match returns NA
query_if(dat, .("b", 1:2), on.=.(x, y), nomatch=0)  # no match row is not returned
# locf, nomatch row gets rolled by previous row
query_if(dat, .("b", 1:2), on=c("x", "y"), roll=Inf)
query_if(dat, .("b", 1:2), on.=.(x, y), roll=-Inf)  # nocb, nomatch row gets rolled by next row
# on rows where dat$x == "b", calculate sum(v*y)
query_if(dat, "b", sum(v*y), on="x")

# all together now
query_if(dat, x!="a", sum(v), by=x)  # get sum(v) by "x" for each i != "a"
query_if(dat, x!="a", sum(v), by=.EACHI, on="x")  # same, but using subsets-as-joins
query_if(dat, c("b", "c"), sum(v), by=.EACHI, on="x")  # same
query_if(dat, c("b", "c"), sum(v), by=.EACHI, on.=.(x))  # same, using on.=()

# joins as subsets
X = data.table(x=c("c","b"), v=8:7, foo=c(4,2))
X

query_if(dat, X, on="x")  # right join
query_if(X, dat, on="x")  # left join
query_if(dat, X, on="x", nomatch=0)  # inner join
query_if(dat, !X, on="x")  # not join
# join using column "y" of 'dat' with column "v" of X
query_if(dat, X, on=c(y="v"))  # same as above (v1.9.8+)
query_if(dat, X, on = .(y<=foo))  # NEW non-equi join (v1.9.8+)
query_if(dat, X, on="y<=foo")  # same as above
query_if(dat, X, on=c("y<=foo"))  # same as above
query_if(dat, X, on.=.(y<=foo))  # NEW non-equi join (v1.9.8+)
query_if(dat, X, on.=.(x, y<=foo))  # NEW non-equi join (v1.9.8+)
query_if(dat, X, .(x,y,x,y,v), on.=.(x, y<=foo))  # Select x's join columns as well
query_if(dat, X, on="x", mult="first")  # first row of each group
query_if(dat, X, on="x", mult="last")  # last row of each group
query_if(dat, X, sum(v), by=.EACHI, on="x")  # join and eval j for each row in i
```r
# join inherited scope
query_if(dat, X, sum(v)*foo, by=.EACHI, on="x")

# 'i,v' refers to X's v column
query_if(dat, X, on=(x, v>=v), sum(y)*foo, by=.EACHI)

# NEW non-equi join with by=.EACHI (v1.9.8+)
query_if(dat, X, on=(x, v>=v), sum(y)*foo, by=.EACHI)

# more on special symbols, see also ?"special-symbols"

# last row
query_if(dat, .N)

# total number of rows in DT
query_if(dat, .N, by=x)

# number of rows in each group
query_if(dat, .SD, .SDcols=x:y)

# select columns 'x' and 'y'
query_if(dat, .SD[1])

# first row of all columns
query_if(dat, .SD[1], by=x)

# first row of 'y' and 'v' for each group in 'x'
query_if(dat, c(.N, lapply(.SD, sum)), by=x)

# get rows *and* sum columns 'v' and 'y' by group
query_if(dat, .I[1], by=x)

# row number in DT corresponding to each group
query_if(dat, grp := .GRP, by=x)

query(X, query_if(dat, .BY, y, on="x"), by=x)

# add/update/delete by reference (see ?assign)
query_if(dat, z:=42L, by=x)

query_if(dat, z:=NULL, by=x)

query_if(dat, v:=42L, by=x)

query_if(dat, v2:=84L, by=x)

# NB: postfix [[] is shortcut to print()
query_if(dat, m:=mean(v), by=x)

# advanced usage

dat = data.table(x=rep(c("b","a","c"),each=3),
                   v=c(1,1,1,2,2,1,1,2,2),
                   y=c(1,3,6),
                   a=1:9,
                   b=9:1)

dat

query_if(dat, sum(v), by=(x, y%%2))(# expressions in by
query_if(dat, sum(v), by=(y = y%%2)) # same, using a named list to change by column name
query_if(dat, .SD[2], by=x) # get 2nd row of each group
query_if(dat, tail(.SD,2), by=x) # last 2 rows of each group
query_if(dat, lapply(.SD, sum), by=x) # sum of all (other) columns for each group
query_if(dat, .SD[which.min(v)], by=x) # nested query by group

query_if(dat, list(MySum=sum(v),
                   MyMin=min(v),
                   MyMax=max(v)),
                   by=(x, y%%2)) # by 2 expressions

query_if(dat, (a = (a, b = (b))), by=x) # list columns
query_if(dat, .(seq = min(a):max(b)), by=x) # j is not limited to just aggregations
query_if(dat, sum(v), by=x) # by
query_if(V1<20) # compound query
query_if(dat, sum(v), by=x) %>%
  setorder(-V1) %>%
  head() # ordering results
```
text_expand

query(dat, c(.N, lapply(.SD,sum)), by=x)  # get number of observations and sum per group

# anonymous lambda in 'j', j accepts any valid
# expression. TO REMEMBER: every element of
# the list becomes a column in result.
query(dat,
  {tmp = mean(y);
   .(a = a-tmp, b = b-tmp)
  },
  by=x)

## Not run:
pdf("new.pdf")
query(dat, plot(a,b), by=x)  # can also plot in 'j'
dev.off()

## End(Not run)
# using rleid, get max(y) and min of all cols in .SDcols for each consecutive run of 'v'
query(dat,
  c(.(y=max(y)), lapply(.SD, min)),
  by=rleid(v),
  .SDcols=v:b
)

text_expand

Evaluate expressions in curly brackets inside strings

Description

text_expand is simple string interpolation function. It searches in its arguments expressions in
curly brackets {expr}, evaluate them and substitute with the result of evaluation. See examples.

Usage

text_expand(..., delim = c("\"\{", "\"\")

Arguments

... character vectors
delim character vector of length 2 - pair of opening and closing delimiters for the
    templating tags. By default it is curly brackets. Note that delim will be used in
    the perl-style regular expression so you need to escape special characters, e. g.
    use "\\" instead of ".

Value

Vector of characters
Examples

```r
i = 1:5
text_expand("q(i)")
```

```r
i = 1:3
ej = 1:3
text_expand("q1_{i}_{j}")
```

data(iris)
text_expand("'iris' has \{nrow(iris)\} rows.")

---

**to_list**

*Apply an expression to each element of a list or vector*

**Description**

- `to_list` always returns a list, each element of which is the result of expression `expr` on the elements of `data`. By default, NULL's will be removed from the result. You can change this behavior with `skip_null` argument.
- `to_vec` is the same as `to_list` but tries to convert its result to vector via `unlist`.
- `to_df` and `to_dfr` try to combine its results to `data.table` by rows.
- `to_dfc` tries to combine its result to `data.table` by columns.

Expression can use predefined variables: `.x` is a value of current list element, `.name` is a name of the element and `.index` is sequential number of the element.

**Usage**

```r
to_list(
data,
expr = NULL,
..., 
skip_null = TRUE,
trace = FALSE,
trace_step = 1L
)
```

```r
to_vec(
data,
expr = NULL,
..., 
skip_null = TRUE,
trace = FALSE,
trace_step = 1L,
```
to_list

recursive = TRUE,
use.names = TRUE
)

to_df(
  data,
  expr = NULL,
  ...
  trace = FALSE,
  trace_step = 1L,
  idvalue = NULL,
  idname = "item_id"
)

to_dfr(
  data,
  expr = NULL,
  ...
  trace = FALSE,
  trace_step = 1L,
  idvalue = NULL,
  idname = "item_id"
)

to_dfc(data, expr = NULL, ..., trace = FALSE, trace_step = 1)

Arguments

data data.frame/list/vector
expr expression or function. Expression can use predefined variables: `.x` is a value
of current list element, `.name` is a name of the element and `.index` is sequential
number of the element.
...
  further arguments provided if `expr` is function.
skip_null logical Should we skip NULL’s from result? Default is TRUE
trace FALSE by default. Should we report progress during execution? Possible values
are TRUE, FALSE, "pb" (progress bar) or custom expression in `quote`, e. g.
`quote(print(.x))`. Expression can contain `.x`, `.name`, and `.index` variables.
trace_step integer. 1 by default. Step for reporting progress. Ignored if `trace` argument is
equal to FALSE.
recursive logical. Should unlisting be applied to list components of x? For details see
unlist.
use.names logical. TRUE by default. Should names of source list be preserved? Setting
it to FALSE in some cases can greatly increase performance. For details see
unlist.
idvalue expression for calculation id column. Usually it is just unquoted symbols: one
of the `.name`, `.index` or `.x`.
idname character, `item_id` by default. Name for the id column.
Value

'to_list' returns list, 'to_vec' tries to return vector and other functions return data.table

Examples

1:5 %%>
   to_list(rnorm(n = 3, .x))

# or in 'lapply' style
1:5 %%>
   to_list(rnorm, n = 3) %>%
   to_vec(mean)

# or use an anonymous function
1:5 %>
   to_list(function(x) rnorm(3, x))

# Use to_vec() to reduce output to a vector instead
# of a list:
# filtering - return only even numbers
   to_vec(1:10, if(.x %% 2 == 0) .x)

# filtering - calculate mean only on the numeric columns
   to_vec(iris, if(is.numeric(.x)) mean(.x))

# mean for numerics, number of distincts for others
   to_vec(iris, if(is.numeric(.x)) mean(.x) else uniqueN(.x))

# means for Sepal
   to_vec(iris, if(startsWith(.name, "Sepal")) mean(.x))

# A more realistic example: split a data frame into pieces, fit a
# model to each piece, summarise and extract R^2
   mtcars %>%
     split(.$.cyl) %>%
     to_list(summary(lm(mpg ~ wt, data = .x))) %>%
     to_vec(.x$r.squared)

# If each element of the output is a data frame, use
# to_df to row-bind them together:
   mtcars %>%
     split(.$.cyl) %>%
     to_list(lm(mpg ~ wt, data = .x)) %>%
     to_df(c(cyl = .name, coef(.x)))

## Not run:  
# read all csv files in "data" to data.frame
   all_files = dir("data", pattern = "csv"$, full.names = TRUE) %>%
     to_df(fread,
        idvalue = basename(.x),
        idname = "filename",
        trace = "pb"
to_long

## End(Not run)

---

**to_long**

Convert data to long or to wide form

**Description**

to_long increases number of rows in the dataset and reduce number of columns. to_wide makes invert transformation. You can use cols for selecting variables in the arguments. See examples.

**Usage**

to_long(
  data,
  columns = NULL,
  keep = NULL,
  names_in = "variable",
  values_in = "value",
  drop_na = FALSE,
  names_factor = TRUE,
  value_factor = FALSE,
  ...
)

to_wide(
  data,
  keep = NULL,
  names_in = variable,
  values_in = value,
  fun = identity,
  sep = ".",
  fill = NA,
  missing_comb = c("none", "rows", "columns", "all"),
  ...
)

**Arguments**

data A data.frame to convert
columns unquoted names of variables for stacking. When missing, we will stack all columns outside keep columns.
keep unquoted names of columns which will be kept as is, e. g. only recycled or deduplicated. If missing, it is all columns except stacked or unstacked. If FALSE then nothing will be kept.
names_in name of the stacked variable names column. The default name is 'variable'. It is quoted in the to_long and unquoted in to_wide. If FALSE in the to_wide than nothing will be widening.

values_in name(-s) of the stacked data values column(s). The default name is 'value'. Multiple names can be provided here for the case when columns is a list, though note well that the names provided in columns take precedence. It is quoted in the to_long and unquoted in to_wide.

drop_na If TRUE, NA values will be removed from the stacked data.

names_factor If TRUE, the column with names will be converted to factor, else it will be a character column. TRUE by default.

value_factor If TRUE, the value column will be converted to factor, else the stacked values type is left unchanged. FALSE by default.

... other arguments passed to data.table::melt/data.table::dcast

fun Should the data be aggregated before casting? By default, it is identity - no aggregation. To use multiple aggregation functions, pass a list; see Examples.

sep Character vector of length 1, indicating the separating character in variable names generated during casting. Default is "_".

fill Value with which to fill missing cells. NA by default. If fun is present, takes the value by applying the function on a 0-length vector.

missing_comb One of "none" (the default), "rows" - include missing combinations in rows, "columns" - include missing combinations in columns, and "all" include all missing combinations.

Value
data.table in the wide or long form.

Examples
data(iris)

# 'to_long'

long_iris = iris %>%
to_long(keep = Species)

long_iris

iris_with_stat = long_iris %>%
take(mean = mean(value),
   sd = sd(value),
   n = .N*1.0,
   by = .(Species, variable)
) %>%
to_long(columns = c(mean, sd, n), names_in = "stat")

# 'to_wide' - table with multiple stats
iris_with_stat %>%


to_long

```r
to_wide()

iris_with_stat %>%
  to_wide(names_in = c(variable, stat))

iris_with_stat %>%
  to_wide(names_in = c(variable, Species))

# 'to_wide' - aggregation function
long_iris %>%
  to_wide(fun = list(Mean = mean, SD = sd, N = length))

# multiple variables
iris %>%
  to_long(list(Sepal = cols("^Sepal"), Petal = cols("^Petal"))) %>%
  let(
    variable = factor(variable, levels = 1:2, labels = c("Length", "Width"))
  ) %>%
  to_wide(values_in = c(Sepal, Petal))

# '%to%' selector - example from tidyr::pivot_longer
data(anscombe)
anscombe %>%
  to_long(
    list(x = x1 %to% x4, y = y1 %to% y4), names_in = "set"
  )
```

### Examples from data.table melt/dcast

```r
set.seed(45)
DT = data.table(
  i_1 = c(1:5, NA)*1.0,
  i_2 = c(NA,6,7,8,9,10)*1.0,
  f_1 = factor(sample(c(letters[1:3], NA), 6, TRUE)),
  f_2 = factor(c("z", "a", "x", "c", "x", "x"), ordered=TRUE),
  c_1 = sample(c(letters[1:3], NA), 6, TRUE),
  d_1 = as.Date(c(1:3,NA,4:5), origin="2013-09-01"),
  d_2 = as.Date(6:1, origin="2012-01-01")
)

# id, values as character/integer/numeric vectors
to_long(DT, f_1, keep = 1:2)
to_long(DT, f_1, keep = c(1_1, i_2))
to_long(DT, f_1, keep = i_1 %to% i_2)
to_long(DT, f_1, keep = cols(i_1:i_2), names_factor = FALSE)
to_long(DT, f_1, keep = cols("i_(1:2)"))
to_long(DT, f_1, keep = cols("*i_"))
```
to_long(DT, f_1, keep = cols("^i_"), names_in = "var", values_in = "val")

col_var = "^i_"
to_long(DT, 3, keep = cols(col_var))

to_long(DT, cols("^f_"), keep = cols("^i_"), value_factor = TRUE)

to_long(mtcars)
to_long(mtcars, keep = am)
to_long(mtcars, columns = c(am, vs, mpg))
to_long(mtcars, columns = c(am, vs, mpg), keep = FALSE)
to_long(DT, keep = f_1, columns = c(i_1, i_2), drop_na = TRUE)
to_long(DT, keep=1:2, columns = list(cols("^f_"), cols("^d_")), value_factor=TRUE)

data("ChickWeight")
names(ChickWeight) = tolower(names(ChickWeight))
DT = to_long(ChickWeight, keep=2:4)

to_wide(DT, keep = time, fun = mean)
to_wide(DT, keep = FALSE, fun = mean)
to_wide(DT, keep = diet, fun = mean)
to_wide(DT, keep = c(diet, chick), names_in = time, missing_comb = "all")
to_wide(DT, keep = c(diet, chick), names_in = time, missing_comb = "all", fill = 0)
to_wide(DT, chick, time, fun = mean)

# using FALSE
DT = data.table(v1 = rep(1:2, each = 6),
                 v2 = rep(rep(1:3, 2), each = 2),
                 v3 = rep(1:2, 6),
                 v4 = rnorm(6))

## for each combination of (v1, v2), add up all values of v4
to_wide(DT,
        cols("^v(1|2)"),
        names_in = FALSE,
        values_in = v4,
        fun = sum)

# multiple values_in and multiple fun
DT = data.table(x=sample(5,20,TRUE),
                 y=sample(2,20,TRUE),
                 z=sample(letters[1:2], 20,TRUE),
                 d1 = runif(20),
                 d2=1L)

# multiple values_in
to_wide(DT,
        keep = c(x, y),
        names_in = z,
        values_in = c(d1, d2),
vlookup

Look up values in dictionary.

Description

vlookup function is inspired by VLOOKUP spreadsheet function. It looks for a lookup_value in the lookup_column of the dict, and then returns values in the same rows from result_column. xlookup is simplified version of vlookup. It searches for a lookup_value in the lookup_vector and return values in the same position from the result_vector.

Usage

vlookup(
  lookup_value,
  dict,
  result_column = 2,
  lookup_column = 1,
  no_match = NA
)

xlookup(lookup_value, lookup_vector, result_vector, no_match = NA)
Arguments

- **lookup_value**: Vector of looked up values
- **dict**: data.frame. Dictionary.
- **result_column**: numeric or character. Resulting columns in the dict. Default value for `result_column` is 2 - for frequent case of dictionary with keys in the first column and results in the second column.
- **lookup_column**: Column of dict in which lookup value will be searched. By default, it is the first column of the dict.
- **no_match**: vector of length one. NA by default. Where a valid match is not found, return the 'no_match' value you supply.
- **lookup_vector**: vector in which 'lookup_value' will be searched during 'xlookup'.
- **result_vector**: vector with resulting values for 'xlookup'.

Value

xlookup always return vector, vlookup returns vector if the result_column is single value. In the opposite case data.frame will be returned.

Examples

```r
# with data.frame
dict = data.frame(num=1:26, small=letters, cap=LETTERS)
vlookup(1:3, dict)
vlookup(c(45,1:3,58), dict, result_column='cap')
vlookup(c(45,1:3,58), dict, result_column='cap', no_match = "Not found")

# the same with xlookup
xlookup(1:3, dict$num, dict$small)
xlookup(c(45,1:3,58), dict$num, dict$cap)
xlookup(c(45,1:3,58), dict$num, dict$cap, no_match = "Not found")

# example from base 'merge'
authors = data.table(
  surname = c("Tukey", "Venables", "Tierney", "Ripley", "McNeil"),
  nationality = c("US", "Australia", "US", "UK", "Australia"),
  deceased = c("yes", rep("no", 4))
)

books = data.table(
  surname = c("Tukey", "Venables", "Tierney",
    "Ripley", "Ripley", "McNeil", "R Core"),
  title = c("Exploratory Data Analysis",
    "Modern Applied Statistics ...",
    "LISP-STAT",
    "Spatial Statistics", "Stochastic Simulation",
    "Interactive Data Analysis",
    "An Introduction to R")
)
let(books,
    c(“author_nationality”, “author_deceased”) := vlookup(surname,
        dict = authors,
        result_column = 2:3
    )
)

# Just for fun. Examples borrowed from Microsoft Excel.
# It is not the R way of doing things.

# Example 2

ex2 = fread("Item_ID Item Cost Markup
ST-340 Stroller 145.67 0.30
BI-567 Bib 3.56 0.40
DI-328 Diapers 21.45 0.35
WI-989 Wipes 5.12 0.40
AS-469 Aspirator 2.56 0.45")

# Calculates the retail price of diapers by adding the markup percentage to the cost.
vlookup("DI-328", ex2, 3) * (1 + vlookup("DI-328", ex2, 4)) # 28.9575

# Calculates the sale price of wipes by subtracting a specified discount from
# the retail price.
(vlookup("WI-989", ex2, "Cost") * (1 + vlookup("WI-989", ex2, "Markup"))) * (1 - 0.2) # 5.7344

A2 = ex2[["Item_ID"]][1]
A3 = ex2[["Item_ID"]][2]

# If the cost of an item is greater than or equal to $20.00, displays the string
# "Markup is nn%"; otherwise, displays the string "Cost is under $20.00".
ifelse(vlookup(A2, ex2, "Cost") >= 20,
    paste0("Markup is " , 100 * vlookup(A2, ex2, "Markup"), "%"),
    "Cost is under $20.00") # Markup is 30%

# If the cost of an item is greater than or equal to $20.00, displays the string
# "Markup is nn%"; otherwise, displays the string "Cost is $n.nn".
ifelse(vlookup(A3, ex2, "Cost") >= 20,
    paste0("Markup is: " , 100 * vlookup(A3, ex2, "Markup"), "%"),
    paste0("Cost is $", vlookup(A3, ex2, "Cost"))) #Cost is $3.56

# Example 3

ex3 = fread('> quotesingle.Var
ID Last_name First_name Title Birth_date
1 Davis Sara "Sales Rep." 12/8/1968
2 Fontana Olivier "V.P. of Sales" 2/19/1952
3 Leal Karina "Sales Rep." 8/30/1963

# VLOOKUP
4 Patten Michael "Sales Rep." 9/19/1958
5 Burke Brian "Sales Mgr." 3/4/1955
6 Sousa Luis "Sales Rep." 7/2/1963

')

# If there is an employee with an ID of 5, displays the employee's last name;
# otherwise, displays the message "Employee not found".
vlookup(5, ex3, "Last_name", no_match = "Employee not found") # Burke

# Many employees
vlookup(1:10, ex3, "Last_name", no_match = "Employee not found")

# For the employee with an ID of 4, concatenates the values of three cells into
# a complete sentence.
paste0(vlookup(4, ex3, "First_name"), " ",
      vlookup(4, ex3, "Last_name"), " is a ",
      vlookup(4, ex3, "Title")) # Michael Patten is a Sales Rep.
Index

coaalesce, 2
cols, 31
cols (columns), 3
columns, 3
copy, 4
data.table, 8, 12, 22-24
dt_add_count (dt_count), 5
dt_anti_join (dt_left_join), 6
dt_arrange (dt_mutate), 7
dt_count, 5
dt_filter (dt_mutate), 7
dt_full_join (dt_left_join), 6
dt_inner_join (dt_left_join), 6
dt_left_join, 6
dt_mutate, 7
dt_right_join (dt_left_join), 6
dt_select (dt_mutate), 7
dt_semi_join (dt_left_join), 6
dt_summarise (dt_mutate), 7
dt_summarise_all (dt_mutate), 7
dt_summarize (dt_mutate), 7
dt_summarize_all (dt_mutate), 7
dt_top_n (dt_count), 5

take_all (let_if), 10
take_if (let_if), 10
text_expand, 3, 11, 27
to_df (to_list), 28
to_dfc (to_list), 28
to_dfr (to_list), 28
to_list, 28
to_long, 31
to_vec (to_list), 28
to_wide (to_long), 31
unlist, 28, 29
vlookup, 35
xlookup (vlookup), 35

fcoalesce, 2
let (let_if), 10
let_all (let_if), 10
let_if, 10

maditr, 17

query (query_if), 22
query_if, 22

rows (columns), 3
sort_by (let_if), 10

take (let_if), 10