Package ‘maditr’

December 3, 2019

Type Package

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

Version 0.6.3

Maintainer Gregory Demin <gdemin@gmail.com>

Depends R (>= 3.3.0)

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

Suggests knitr, tinytest

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. 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. 'query_if' function translates its arguments one-to-one to '[.data.table' method.
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

LazyData true

RoxygenNote 7.0.1

NeedsCompilation no

Author Gregory Demin [aut, cre]

Repository CRAN

Date/Publication 2019-12-03 06:00:02 UTC

R topics documented:

`coalesce` *Return first non-missing element*

Description

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

Usage

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

Arguments

`...`	vectors

Value

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

Examples

```
# 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)
```

dcast *Convert data between wide and long forms.*

Description

The ‘dcast‘ formula takes the form ‘LHS ~ RHS‘, ex: ‘var1 + var2 ~ var3‘. The order of entries in the formula is essential. There are two special variables: ‘.‘ and ‘...‘. ‘.‘ represents no variable; ‘...‘ represents all variables not otherwise mentioned in formula. LHS variable values will be in rows. RHS variables values will become column names. ‘fun.aggregate(value.var)‘ will be cell values. For details see dcast and melt.

Usage

```
dcast(
  data,
  formula,
  fun.aggregate = NULL,
  sep = "_",
  ...,
  margins = NULL,
  subset = NULL,
  fill = NULL,
  drop = TRUE,
  value.var = guess(data),
  verbose = getOption("datatable.verbose")
)

melt(
  data,
  id.vars,
  measure.vars,
  variable.name = "variable",
  value.name = "value",
  ...,
  na.rm = FALSE,
  variable.factor = TRUE,
  value.factor = FALSE,
  verbose = getOption("datatable.verbose")
)

guess(data)
```

Arguments

`data`	A data.table/data.frame. `data.frame` will be automatically converted to data.table.
`formula`	A formula of the form LHS ~ RHS to cast. For details see dcast.

`fun.aggregate`	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.
`sep`	Character vector of length 1, indicating the separating character in variable names generated during casting. Default is _ for backwards compatibility.
`...`	Any other arguments that may be passed to the aggregating function.
`margins`	For details see dcast.
`subset`	Specified if casting should be done on a subset of the data.
`fill`	Value with which to fill missing cells. If fun.aggregate is present, takes the value by applying the function on a 0-length vector.
`drop`	FALSE will cast by including all missing combinations. c(FALSE, TRUE) will only include all missing combinations of formula LHS. And c(TRUE, FALSE) will only include all missing combinations of formula RHS.
`value.var`	Name of the column whose values will be filled to cast. Function 'guess()' tries to, well, guess this column automatically, if none is provided. It is possible to cast multiple 'value.var" columns simultaneously. For details see dcast.
`verbose`	For details see dcast.
`id.vars`	vector of id variables. Can be integer (corresponding id column numbers) or character (id column names) vector. If missing, all non-measure columns will be assigned to it. If integer, must be positive; see Details.
`measure.vars`	Measure variables for melting. Can be missing, vector, list, or pattern-based. For details see dcast.
`variable.name`	name for the measured variable names column. The default name is 'variable'.
`value.name`	name for the molten data values column(s). The default name is 'value'. Multiple names can be provided here for the case when measure.vars is a list, though note well that the names provided in measure.vars take precedence.
`na.rm`	If TRUE, NA values will be removed from the molten data.
`variable.factor`	If TRUE, the variable column will be converted to factor, else it will be a character column.
`value.factor`	If TRUE, the value column will be converted to factor, else the molten value type is left unchanged.

Value

data.table

Author(s)

Matt Dowle <mattjdowle@gmail.com>

Examples

```
# examples from 'tidyr' package
stocks = data.frame(
    time = as.Date('2009-01-01') + 0:9,
    X = rnorm(10, 0, 1),
    Y = rnorm(10, 0, 2),
    Z = rnorm(10, 0, 4)
)
stocksm = stocks %>%
    melt(id.vars = "time", variable.name = "stock", value.name = "price")
stocksm %>% dcast(time ~ stock)
stocksm %>% dcast(stock ~ time)

# dcast and melt are complements
df = data.frame(x = c("a", "b"), y = c(3, 4), z = c(5, 6))
df %>%
    dcast(z ~ x, value.var = "y") %>%
    melt(id.vars = "z", variable.name = "x", value.name = "y", na.rm = TRUE)
```

`dt_count` *Additional useful functions*

Description

- dt_count calculate 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

```
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.

Value

data.table

Examples

```
data(mtcars)

# dt_count
dt_count(mtcars, am, vs)
dt_add_count(mtcars, am, vs, name = "am_vs")[] # [] for autoprinting

# dt_top_n
dt_top_n(mtcars, 2, by  = list(am, vs))
dt_top_n(mtcars, 2, order_by = mpg, by  = list(am, vs))
```

`dt_left_join` *Join two data.frames 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` the same as `dt_summarize` but work over all non-grouping variables.
- `dt_filter` 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.
- `dt_arrange` sorts dataset by variable(-s). Use '-' to sort in descending order. If `data` is data.table then it modifies in-place.

Usage

```
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`, `dt_mutate_if`, `dt_mutate_if` 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.

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) %>%
    print()


# 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:
mtcars %>%
    dt_summarise_all(mean, by = list(vsam = vs + am))

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

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


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

# sorting
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 conditionally. `take` aggregates data or select subset of the data by rows or columns. Both functions return `data.table`.

- Add new variables: `let(mtcars,new_var = 42,new_var2 = new_var*hp)`
- Filter data: `take_if(mtcars,am==0)`
- Select variables: `take(mtcars,am,vs,mpg)`
- Aggregate data: `take(mtcars,mean_mpg = mean(mpg),by = am)`
- Aggregate all non-grouping columns: `take(mtcars,fun = mean,by = am)`

Usage

```
let_if(
  data,
  i,
  ...,
  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
)

take_if(
  data,
  i,
  ...,
  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,
  autoname = TRUE,
  fun = NULL
)

take(
  data,
  ...,
  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,
  autoname = TRUE,
  fun = NULL
)

let(
  data,
  ...,
  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
)

## Default S3 method:
let(
  data,
  ...,
  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
)
```

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

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))`
`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.
`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.
`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.

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()

# filter by condition
mtcars %>%
    take_if(am==0)

# filter by compound condition
mtcars %>%
    take_if(am==0 & mpg>mean(mpg))
```

```
# 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(fun = mean, by = list(vsam = vs + am))


# 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))

########################################

# 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
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, !2:4)                      # all rows other than 2:4
take_if(dat, -(2:4))                    # same

# select|compute columns
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
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
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
take_if(dat, x!="a", sum(v), by=x)                   # get sum(v) by "x" for each x != "a"
take_if(dat, c("b", "c"), sum(v), by = .EACHI, on="x")   # same

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

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


# add/update/delete by reference
# [] 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)

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

# 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)

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(dat, fun = 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) %>%                 # ordering results
    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.

- To select rows from data: `take_if(mtcars,am==0)`
- To select columns from data: `take(mtcars,am,vs,mpg)`
- To aggregate data: `take(mtcars,mean_mpg = mean(mpg),by = am)`
- To aggregate all non-grouping columns: `take(mtcars,fun = 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(mtcars,fun = mean)`
- Use magrittr pipe '%>%' to chain several operations:

  ```
  mtcars %>%
     let(mpg_hp = mpg/hp) %>%
     take(mean(mpg_hp), by = am)
  ```

- To modify variables or add new variables:

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

- To drop variable assign NULL: `let(mtcars,am = NULL) %>% head()`
- For parametric assignment use ':=':

  ```
  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

```
# 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()

# filter by condition
mtcars %>%
    take_if(am==0)

# filter by compound condition
```

```
mtcars %>%
    take_if(am==0 & mpg>mean(mpg))


# 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 = am)

# grouping by multiple variables
mtcars %>%
    take(mean = mean(disp), n = .N, by = list(am, vs))

# parametric evaluation:
var = quote(mean(cyl))
take(mtcars, eval(var))


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

---

query_if *One-to-one interface for data.table '[' method*

Description

Quote from data.table:

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

or,

```
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

```
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"),
  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 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
`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(dat, sum(v), keyby=x)          # same, but order the result on by cols
query(dat, sum(v), by=x) %>%
    query_if(order(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, !"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"))
query_if(dat,X, on="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
query_if(dat, X, sum(v)*foo, by=.EACHI, on="x")  # join inherited scope
query_if(dat, X, sum(v)*i.v, 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+)


# more on special symbols, see also ?"special-symbols"
query_if(dat, .N)                            # last row
query(dat, .N)                               # total number of rows in DT
query(dat, .N, by=x)                         # number of rows in each group
query(dat, .SD, .SDcols=x:y)                 # select columns 'x' and 'y'
query(dat, .SD[1])                           # first row of all columns
query(dat, .SD[1], by=x)                  # first row of 'y' and 'v' for each group in 'x'
query(dat, c(.N, lapply(.SD, sum)), by=x)  # get rows *and* sum columns 'v' and 'y' by group
query(dat, .I[1], by=x)                    # row number in DT corresponding to each group
query(dat, grp := .GRP, by=x) %>% head()     # add a group counter column
query(X, query_if(dat, .BY, y, on="x"), by=x)                # join within each group

# add/update/delete by reference (see ?assign)
query(dat, z:=42L) %>% head()         # add new column by reference
query(dat, z:=NULL) %>% head()        # remove column by reference
query_if(dat, "a", v:=42L, on="x") %>% head()  # subassign to existing v column by reference
query_if(dat, "b", v2:=84L, on="x") %>% head() # subassign to new column by reference (NA padded)

# NB: postfix [] is shortcut to print()
query(dat, m:=mean(v), by=x)[]               # add new column by reference by group

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

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

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

Index