Package 'm61r'

January 13, 2026

Type Package

Version 0.1.0

Title Package About Data Manipulation in Pure Base R

Description A lightweight, dependency-free data engine for R that provides a grammar for tabular and time-series manipulation. Built entirely on Base R, 'm61r' offers a fluent, chainable API inspired by modern data tools while prioritizing memory efficiency and speed. It includes optimized versions of common data verbs such as filtering, mutation, grouped aggregation, and approximate temporal joins, making it an ideal choice for environments where external dependencies are restricted or where performance in pure R is required.

Depends R (>= 4.2.0)

License MIT + file LICENSE

URL https://github.com/pv71u98h1/m61r/

BugReports https://github.com/pv71u98h1/m61r/issues/

Encoding UTF-8

Language en-GB

RoxygenNote 7.3.3

VignetteBuilder knitr

Suggests knitr, rmarkdown

NeedsCompilation no

Author Jean-Marie Lepioufle [aut, cre]

Maintainer Jean-Marie Lepioufle <pv71u98h1@gmail.com>

Repository CRAN

Date/Publication 2026-01-13 00:50:28 UTC

Contents

`m61r-package` *High-Performance Data Manipulation with Pure Base R*

Description

The `m61r` package provides a suite of optimized functions for tabular data manipulation, The design prioritizes computational speed and a clean, readable data-processing grammar. The package got highly inspired by modern data manipulation packages, while exclusively built upon the Base R environment.

Details

The core of `m61r` is the Base R Data Manipulation Grammar, implemented through two layers:

- **Primitives (e.g., `filter_`, `arrange_`):** These are the raw, optimized functions operating directly on Base R `data.frame` objects. They are designed for maximum computational efficiency.
- `m61r` **Object (`m61r()` constructor):** This S3 class provides a pipeline interface, allowing a sequence of operations (e.g., `$filter()`, `$mutate()`) to be chained cleanly.

The Base R Formula Domain-Specific Language: All manipulation functions (like `filter`, `mutate`, `group_by`, `select`, `summarise`) utilise a formula syntax (`~<expression>`). This ensures Non-Standard Evaluation can access column names directly within the context of the data frame.

Date and Time Handling: For complex grouping or filtering on date-time columns (`Date` or `POSIXct`), users must employ standard Base R functions within the formula expression to extract

components. For instance, to group by the year of a column named DateColumn, one must use Base R format function: ~format(DateColumn, "%Y")

For detailed documentation, see the individual function reference pages.

---

across *Apply a function across multiple columns*

Description

The across function allows you to apply the same transformation or aggregation to multiple columns simultaneously. It is designed to be used within summarise_, mutate_, or transmutate_ methods.

Usage

```
# across(cols, FUN, ...)
```

Arguments

cols	A character vector of column names, a numeric vector of column indices, or a predicate function (e.g., is.numeric) to select columns.
FUN	A function to be applied to each of the selected columns.
...	Additional arguments passed to the function FUN.

Details

This function provides a concise way to perform operations on multiple columns at once. It internally accesses the data subset (.SD) of the current group or data frame. If cols is a function, it acts as a filter to select all columns for which the function returns TRUE.

Value

A list where each element represents the result of FUN applied to a selected column. When used within summarise_, this list is automatically flattened into separate columns.

Examples

```
# Usage within an m61r pipeline for aggregation
p <- m61r(mtcars)
p$summarise(
  avg = ~across(c("mpg", "disp", "hp"), mean)
)
p[]

# Usage with a predicate function to select numeric columns
p <- m61r(iris)
p$summarise(
  stats = ~across(is.numeric, sd, na.rm = TRUE)
)
p[]
```

arrange *Arrange your data.frames*

Description

Re-arrange your data.frame in ascending or descending order given one or several columns.

Usage

```
# arrange_(df, ...)
# desange_(df, ...)
```

Arguments

df	A data.frame.
...	A formula used for arranging the data.frame (e.g., ~c(col1, col2)).

Value

The functions return an object of the same type as df. Properties:

- Columns are not modified.
- Output rows are in the order specified by the formula.
- Data frame attributes are preserved.

Examples

```
tmp <- arrange_(CO2, ~c(conc))
head(tmp)

co2 <- m61r(df = CO2)
co2$arrange(~c(conc))
co2$head()
```

case_when *Logic within a dataframe*

Description

The case_when function provides a vectorised approach to multiple if_else conditions in a readable and efficient way. It evaluates conditions sequentially and assigns values as soon as a condition is satisfied.

Usage

```
# case_when(...)
```

Arguments

`...`	A sequence of condition/value pairs, ending with a default value (`default`). Format: `condition1`, `value1`, `condition2`, `value2`, `...`, `default`.

Details

This function is optimized for use inside `mutate_` or `transmutate_` methods. The final argument acts as the fallback value (the "otherwise" branch) if all preceding conditions evaluate to `FALSE`.

Value

An atomic vector of the same length as the input conditions. The output type (e.g., character, numeric) is determined by the types of the values provided.

Examples

```
# Independent usage
x <- 1:10
res <- case_when(
  x <= 3, "Small",
  x <= 7, "Medium",
  "Large" # Default/Otherwise value
)

# Usage with an m61r pipeline
tmp <- m61r(mtcars)
tmp$mutate(
  efficiency_cat = ~case_when(
    mpg > 25, "Economical",
    mpg > 15, "Standard",
    "High Consumption"
  )
)
tmp
```

`cut_time` *Binning Date and Time Columns*

Description

`cut_time` is a helper function designed to generate expressions for binning `POSIXct` or `Date` columns into specific time intervals. It is primarily intended for use within `mutate` calls to create grouping variables for time-series analysis.

Usage

```
# cut_time(var, breaks_str)
```

Arguments

var	The symbol of the date or time column (e.g., timestamp).
breaks_str	A character string specifying the time interval (e.g., "hour", "day", "week", "month"). This is passed directly to the breaks argument of Base R cut function.

Details

This function uses substitute to create a symbolic call to base::cut. When used inside an m61r pipeline, it allows for high-performance temporal bucketing.

Value

A language object (call) representing the binning operation, which is evaluated within the context of the data frame.

Examples

```
df_time <- data.frame(
  timestamp = seq(as.POSIXct("2025-01-01"), by = "15 mins", length.out = 100),
  value = rnorm(100)
)

tmp <- m61r(df_time)


tmp$mutate(day_bin = ~eval(cut_time(timestamp, "day")))

tmp$group_by(~day_bin)
tmp$summarise(daily_avg = ~mean(value))

tmp$head()
```

---

explode *Flattening a List-column*

Description

The explode method flattens a list-column, creating a new row for every element in the list while duplicating the values of all other columns.

Usage

```
# Within an m61r object
# tmp$explode(column)
```

Arguments

column	A character string specifying the name of the list-column to be flattened.

Details

This operation is particularly useful after creating temporal sequences or ranges using `Map()` or `seq()`. It transforms "nested" data into a "long" format suitable for standard aggregations.

Technically, it uses `rep()` to replicate row indices based on the length of each list element, ensuring maximum performance for large data frames.

Value

The function updates the internal data frame of the `m61r` object invisibly.

Examples

```
df <- data.frame(
  id = 1:2,
  tags = I(list(c("A", "B"), c("C", "D", "E")))
)

tmp <- m61r(df)

# This will result in 2 rows for id 1 and 3 rows for id 2
tmp$explode("tags")

tmp

# Time-Series Example
df_time <- data.frame(
  id = 1,
  start = as.POSIXct("2025-01-01 08:00"),
  end   = as.POSIXct("2025-01-01 13:00")
)

tmp <- m61r(df_time)
# Create a sequence of hours
tmp$mutate(hour_slot = ~Map(function(s, e) seq(s, e, by = "hour"), start, end))
# Explode to get one row per hour
tmp$explode("hour_slot")
tmp
```

`expression` *Evaluate Formula Expressions on Data Subsets*

Description

The core engine for Non-Standard Evaluation within `m61r`. `expression_` evaluates a user-provided formula within the context of a data frame, optionally for calculated groups. The result relies on the Base R functions `with` and `eval`.

Usage

```
# expression_(df, group_info = NULL, fun_expr)
```

Arguments

df	data.frame
group_info	An optional list of grouping indices and keys, typically generated by get_group_indices_(). If NULL, the expression runs over the entire df.
fun_expr	A formula (~<expression>) that describes the R code to be executed.

Value

The function returns a list.

- If group_info is NULL, the list contains the result of fun_expr executed on the entire df.
- If group_info is provided, the list contains the results of fun_expr executed on each group's subset of the df.

Examples

```
# Non-Grouped Evaluation (for mutate)
expression_(CO2,fun_expr=~conc/uptake)

# Grouped Evaluation (for summarise)
group_info <- get_group_indices_(CO2, ~Type)
expression_(CO2, group_info = group_info, fun_expr=~mean(uptake))

# Complex Grouped Evaluation (results in a list per group)
expression_(CO2, group_info = group_info, fun_expr=~lm(uptake~conc))
```

---

filter *filter a data.frame*

Description

Filter rows of a data.frame with conditions.

Usage

```
# filter_(df, subset = NULL)
```

Arguments

df	data.frame
subset	formula that describes the conditions

Value

The function returns an object of the same type as `df`. Properties:

- Columns are not modified.
- Only rows following the condition determined by
- subset appear.
- Data frame attributes are preserved.

Examples

```
tmp <- filter_(CO2,~Plant=="Qn1")
head(tmp)

tmp <- filter_(CO2,~Type=="Quebec")
head(tmp)

# with m61r class
co2 <- m61r(df=CO2)

co2$filter(~Plant=="Qn1")
co2

co2$filter(~Type=="Quebec")
co2
```

`get_group_indices_` *Determine Grouping Structure for a data.frame*

Description

`get_group_indices_` calculates the necessary indices and keys for efficient grouped operations (like `summarise_`). This mechanism uses Base R `interaction` for group factor calculation.

Usage

```
# get_group_indices_(df, group = NULL)
```

Arguments

`df`	data.frame
`group`	A formula (`~<expression>`) that describes the grouping columns. Column names can be listed in a vector (e.g., `~c(colA, colB)`). Base R functions may be nested to process columns (e.g., for date-time components).

Value

get_group_indices_ returns a list containing: group_cols (names), indices (a list of row indices per group, for fast subsetting), and keys (a data frame of unique group combinations).

Examples

```
g_info <- get_group_indices_(CO2, ~c(Type, Treatment))
summarise_(CO2, group_info = g_info, mean = ~mean(uptake))


# Grouping with a Base R function: Group by the 'year' of a column 'Date'
df_date <- data.frame(
  Date = seq(as.Date("2020-01-01"), by = "month", length.out = 12),
  Value = 1:12
)

# Usage within the m61r pipeline:
df_date_m61r <- m61r(df_date)
df_date_m61r$group_by(~format(Date, "%Y"))
df_date_m61r$summarise(mean_val = ~mean(Value))
df_date_m61r
```

---

io_csv *CSV Input and Output Utilities*

Description

High-performance wrappers for reading and writing CSV files. These functions utilize Base R read.table and write.table engines while ensuring the resulting data frames are optimized for m61r pipelines.

Usage

```
# read_csv(file, header = TRUE, sep = ",", stringsAsFactors = FALSE, ...)

# Within an m61r pipeline
# p$write_csv(file, sep = ",", row.names = FALSE, quote = FALSE, ...)
```

Arguments

file	A character string specifying the file path.
header	Logical; does the file contain a header row?
sep	The field separator character.
stringsAsFactors	Logical; should character vectors be converted to factors?
row.names	Logical; should row names be written to the file?
quote	Logical; should character strings be quoted?
...	Additional arguments passed to the underlying read.table or write.table functions.

Details

read_csv is an optimized loader that automatically strips row names after reading, ensuring a clean index for subsequent m61r operations.

write_csv is designed to be used as a terminal step in an m61r pipeline. It accesses the internal result_ buffer of the object and exports it to the specified file path.

Value

read_csv returns a data.frame. write_csv returns invisible() and is used for its side effect of file creation.

Examples

```
# df <- read_csv("data.csv")
# p <- m61r(df)

p <- m61r(mtcars)
p$filter(~mpg > 20)
p$mutate(hp_per_cyl = ~hp / cyl)

# Export results
# p$write_csv("filtered_mtcars.csv")
```

---

join *Join two data.frames*

Description

Join two data.frames.

Usage

```
# left_join_(df, df2, by = NULL, by.x = NULL, by.y = NULL)
# anti_join_(df, df2, by = NULL, by.x = NULL, by.y = NULL)
# full_join_(df, df2, by = NULL, by.x = NULL, by.y = NULL)
# inner_join_(df, df2, by = NULL, by.x = NULL, by.y = NULL)
# right_join_(df, df2, by = NULL, by.x = NULL, by.y = NULL)
# semi_join_(df, df2, by = NULL, by.x = NULL, by.y = NULL)
```

Arguments

df	data.frame
df2	data.frame
by	column names of the pivot of both data.frame 1 and data.frame 2 if they are identical. Otherwise, better to use by.x and by.y
by.x	column names of the pivot of data.frame 1
by.y	column names of the pivot of data.frame 2

Value

The functions return a data frame. The output has the following properties:

- For functions `left_join()`, `inner_join()`, `full_join()`, and `right_join()`, output includes all `df1` columns and all `df2` columns. For columns with identical names in `df1` and `df2`, a suffix '.x' and '.y' is added. For `left_join()`, all `df1` rows with matching rows of `df2` For `inner_join()`, a subset of `df1` rows matching rows of `df2`. For `full_join()`, all `df1` rows, with all `df2` rows. For `right_join()`, all `df2` rows with matching rows of `df1`.
- For functions `semi_join()` and `anti_join()`, output include columns of `df1` only. For `semi_join()`, all `df1` rows with a match in `df2`. For `anti_join()`, a subset of `df1` rows not matching rows of `df2`.

Examples

```
books <- data.frame(
             name = I(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"),
              other.author = c(NA, "Ripley", NA, NA, NA, NA,"Venables & Smith"))

authors <- data.frame(
             surname = I(c("Tukey", "Venables", "Tierney", "Ripley", "McNeil","Asimov")),
               nationality = c("US", "Australia", "US", "UK", "Australia","US"),
               deceased = c("yes", rep("no", 4),"yes"))

tmp <- left_join_(books,authors, by.x = "name", by.y = "surname")
head(tmp)

tmp <- inner_join_(books,authors, by.x = "name", by.y = "surname")
head(tmp)

tmp <- full_join_(books,authors, by.x = "name", by.y = "surname")
head(tmp)

tmp <- right_join_(books,authors, by.x = "name", by.y = "surname")
head(tmp)

tmp <- semi_join_(books,authors, by.x = "name", by.y = "surname")
head(tmp)

tmp <- anti_join_(books,authors, by.x = "name", by.y = "surname")
head(tmp)


  # with m61r class
```

```
## inner join
tmp <- m61r(df=authors)

tmp$inner_join(books, by.x = "surname", by.y = "name")
tmp

## left join
tmp$left_join(books, by.x = "surname", by.y = "name")
tmp

## right join
tmp$right_join(books, by.x = "surname", by.y = "name")
tmp

## full join
tmp$full_join(books, by.x = "surname", by.y = "name")
tmp

## semi join
tmp$semi_join(books, by.x = "surname", by.y = "name")
tmp

## anti join #1
tmp$anti_join(books, by.x = "surname", by.y = "name")
tmp

## anti join #2
tmp2 <- m61r(df=books)
tmp2$anti_join(authors, by.x = "name", by.y = "surname")
tmp2

## with two m61r objects
tmp1 <- m61r(books)
tmp2 <- m61r(authors)
tmp3 <- anti_join(tmp1,tmp2, by.x = "name", by.y = "surname")
tmp3
```

`join_asof` *Join Two Data Frames Based on Nearest Key*

Description

Performs an "As-Of" join, matching rows from two data frames where the keys are close but not necessarily equal. This is the primary tool for time-series synchronization, mimicking 'Polars' `join_asof`.

Usage

```
# Primitive function
```

```
# join_asof_(x, y, by_x, by_y, direction = "backward")

# Within an m61r pipeline
# p$join_asof(y, by_x, by_y, direction = "backward")
```

Arguments

x, result_	The left data frame (primary timeline).
y	The right data frame (reference timeline). Must be sorted by the join key.
by_x	The column name in the left data frame used for joining.
by_y	The column name in the right data frame used for joining.
direction	Direction of the search: "backward" (default) finds the nearest value $\leq$ key; "forward" finds the nearest value $\geq$ key.

Details

The "As-Of" join is fundamentally different from a standard join. It does not look for exact matches but finds the closest record in a reference table.

- **Backward**: Matches the observation in y that is most recent relative to the time in x (where y_time <= x_time).
- **Forward**: Matches the next upcoming observation in y (where y_time >= x_time).

For maximum speed, m61r utilizes the findInterval function, which performs a binary search in C, ensuring that even with millions of rows, the join remains nearly instantaneous.

Value

A data frame (or updates the m61r object) containing all columns from x and the matched columns from y.

Examples

```
quotes <- data.frame(
  time = as.POSIXct("2025-01-01 10:00") + c(0, 10, 20),
  price = c(100.1, 100.5, 100.3)
)

trades <- data.frame(
  time = as.POSIXct("2025-01-01 10:00:05"),
  volume = 50
)

# This matches the trade at 10:00:05 with the price at 10:00:00 (100.1)
p <- m61r(trades)
p$join_asof(quotes, by_x = "time", by_y = "time", direction = "backward")

print(p)
```

`m61r` *Create m61r object*

Description

Create a m61r object that enables to run a sequence of operations on a data.frame.

Usage

```
# m61r(df = NULL)

## S3 method for class 'm61r'
x[i, j, ...]

## S3 replacement method for class 'm61r'
x[i, j] <- value

## S3 method for class 'm61r'
print(x, ...)

## S3 method for class 'm61r'
names(x, ...)

## S3 method for class 'm61r'
dim(x, ...)

## S3 method for class 'm61r'
as.data.frame(x, ...)

## S3 method for class 'm61r'
rbind(x, ...)

## S3 method for class 'm61r'
cbind(x, ...)
```

Arguments

`df`	data.frame
`x`	object of class `m61r`
`i`	row
`j`	column
`...`	further arguments passed to or from other methods
`value`	value to be assigned

Value

The function m61r returns an object of type m61r.

Argument df get stored internally to the object m61r. One manipulates the internal data.frame by using internal functions similar to the ones implemented in package m61r for data.frames as arrange, desange, filter, join and its relatives, mutate and transmutate, gather and spread, select, group_by, summarise, values and modify. The result of the last action is stored internally to the object m61r until the internal function values get called. It is thus possible to create a readable sequence of actions on a data.frame.

In addition,

- [.m61r returns a subset of the internal data.frame embedded to the object m61r.
- [<-.m61r assigns value to the internal data.frame embedded to the object m61r.
- print.m61r prints the internal data.frame embedded to the object m61r.
- names.m61r provides the names of the column of the internal data.frame embedded to the object m61r.
- dim.m61r provides the dimensions of the internal data.frame embedded to the object m61r.
- as.data.frame.m61r extracts the internal data.frame embedded to the object m61r.
- cbind.m61r combines by columns two objects m61r.
- rbind.m61r combines by rows two objects m61r.
- left_join, anti_join, full_join, inner_join, right_join, semi_join join two objects m61r.

The m61r object maintains an internal state. For advanced transformations, users can use .SD() within formulas. .SD() (Subset of Data) refers to the current data.frame being processed. If a group_by operation is active, .SD() refers to the current group only.

Finally, it is possible to clone a m61r object into a new one by using the internal function clone.

Examples

```
# init
co2 <- m61r(df=CO2)

# filter
co2$filter(~Plant=="Qn1")
co2

co2$filter(~Type=="Quebec")
co2

# select
co2$select(~Type)
co2

co2$select(~c(Plant,Type))
co2

co2$select(~-Type)
```

```
co2

co2$select(variable=~-(Plant:Treatment))
co2

# mutate/transmutate
co2$mutate(z=~conc/uptake)
co2


#co2$mutate(mean=~mean(uptake))
#co2
#Warning message:
#In FUN(X[[i]], ...) : Expression mean has incompatible length.

co2$mutate(z1=~uptake/conc,y=~conc/100)
co2

co2$transmutate(z2=~uptake/conc,y2=~conc/100)
co2

# summarise
co2$summarise(mean=~mean(uptake),sd=~sd(uptake))
co2

co2 = m61r(CO2)
co2$group_by(~cbind(Type,Treatment))
co2$summarise(mean=~mean(uptake),sd=~sd(uptake))
co2

# arrange/dessange
co2$arrange(~c(conc))
co2

co2$arrange(~c(Treatment,conc,uptake))
co2

co2$desange(~c(Treatment,conc,uptake))
co2


# Reshape

## gather
df3 <- data.frame(id = 1:4,
                  age = c(40,50,60,50),
                  dose.a1 = c(1,2,1,2),
                  dose.a2 = c(2,1,2,1),
                  dose.a14 = c(3,3,3,3))

df4 <- m61r::m61r(df3)
df4$gather(pivot = c("id","age"))
df4
```

```
## spread
df3 <- data.frame(id = 1:4,
                  age = c(40,50,60,50),
                  dose.a1 = c(1,2,1,2),
                  dose.a2 = c(2,1,2,1),
                  dose.a14 = c(3,3,3,3))

df4 <- m61r::gather_(df3,pivot = c("id","age"))
df4 <- rbind(df4,
  data.frame(id=5, age=20,parameters="dose.a14",values=8),
  data.frame(id=6, age=10,parameters="dose.a1",values=5))

tmp <- m61r::m61r(df4)
tmp$spread(col_name="parameters",col_values="values",pivot=c("id","age"))
tmp


# equivalence
co2           # is not equivalent to co2[]
co2[]         # is equivalent to co2$values()
co2[1,]       # is equivalent to co2$values(1,)
co2[,2:3]     # is equivalent to co2$values(,2:3)
co2[1:10,1:3] # is equivalent to co2$values(1:10,2:3)
co2[1,"Plant"]# is equivalent to co2$values(1,"Plant")

# modification on m61r object only stay for one step
co2[1,"conc"] <- 100
co2[1,] # temporary result
co2[1,] # back to normal

# WARNING:Keep the brackets to manipulate the intern data.frame
# ... OR you will destroy co2, and only keep the data.frame
# co2 <- co2[-1,]
# class(co2) # data.frame

# descriptive manipulation
names(co2)
dim(co2)
str(co2)

## cloning
# The following will only create a second variable that point on
# the same object (!= cloning)
foo <- co2
str(co2)
str(foo)

# Instead, cloning into a new environemnt
foo <- co2$clone()
str(co2)
str(foo)
```

mutate *Transformative selections of a data.frame*

Description

Transformative selections of a data.frame.

Usage

```
# mutate_(df, ...)
# transmutate_(df, ...)
```

Arguments

df	data.frame
...	formula used for transformative selections the data.frame

Details

mutate_ and transmutate_ execute expressions non-grouped. If the m61r object is in a grouped state (via $group_by()), that grouping state is ignored by the primitive functions, ensuring Base R speed for vectorised operations.

Value

The functions return a data frame. The output has the following properties:

- For function mutate_(), output includes all df columns. In addition, new columns are created according to argument ... and placed after the others.
- For function transmutate_(), output includes only columns created according to argument ... and placed after the others.

Examples

```
tmp <- mutate_(CO2,z=~conc/uptake)
head(tmp)

# Return an warning: expression mean(uptake) get a result with 'nrow' different from 'df'
# tmp <- mutate_(CO2,mean=~mean(uptake))

tmp <- mutate_(CO2,z1=~uptake/conc,y=~conc/100)
head(tmp)

tmp <- transmutate_(CO2,z2=~uptake/conc,y2=~conc/100)
head(tmp)

# with m61r class
co2 <- m61r(df=CO2)
```

```
co2$mutate(z=~conc/uptake)
co2

# not allowed
#co2$mutate(mean=~mean(uptake))
#co2
#Warning message:
#In FUN(X[[i]], ...) : Expression mean has incompatible length.

co2$mutate(z1=~uptake/conc,y=~conc/100)
co2

co2$transmutate(z2=~uptake/conc,y2=~conc/100)
co2
```

---

reshape *Reshape a data.frame*

Description

Reshape a data.frame.

Usage

```
# gather_(df, new_col_name = "parameters", new_col_values = "values", pivot)
# spread_(df, col_name, col_values, pivot)
```

Arguments

df	data.frame
new_col_name	name of the new column 'parameters'
new_col_values	name of the new columns 'values'
col_name	name of the column 'parameters'
col_values	name of the new columns 'values'
pivot	name of the columns used as pivot

Details

A data frame is said 'wide' if several of its columns describe connected information of the same record. A data frame is said 'long' if two of its columns provide information about records, with one describing their name and the second their value. Functions `gather_()` and `spread_()` enable to reshape a data frames from a 'wide' format to a 'long' format, and vice-versa.

Value

The functions return a data frame.

- Output from function gather_() get 'pivot' columns determined by argument pivot, and 'long' columns named according to arguments new_col_name and new_col_values.
- Output from function spread_() get 'pivot' columns determined by argument pivot, and 'wide' columns named according to values in column determined by argument col_name. For 'wide' columns, each row corresponds to values present in column determined by argument col_values.

Examples

```
df3 <- data.frame(id = 1:4,
                    age = c(40,50,60,50),
                    dose.a1 = c(1,2,1,2),
                    dose.a2 = c(2,1,2,1),
                    dose.a14 = c(3,3,3,3))

gather_(df3,pivot = c("id","age"))

df4 <- gather_(df3,pivot = c("id","age"))
df5 <- rbind(df4,
    data.frame(id=5, age=20,parameters="dose.a14",values=8),
    data.frame(id=6, age=10,parameters="dose.a1",values=5))

spread_(df5,col_name="parameters",col_values="values",pivot=c("id","age"))

 # with m61r class
co2 <- m61r(df=CO2)

## gather
df3 <- data.frame(id = 1:4,
                age = c(40,50,60,50),
                dose.a1 = c(1,2,1,2),
                dose.a2 = c(2,1,2,1),
                dose.a14 = c(3,3,3,3))

df4 <- m61r(df3)
df4$gather(pivot = c("id","age"))
df4

## spread
df3 <- data.frame(id = 1:4,
                age = c(40,50,60,50),
                dose.a1 = c(1,2,1,2),
                dose.a2 = c(2,1,2,1),
                dose.a14 = c(3,3,3,3))

df4 <- gather_(df3,pivot = c("id","age"))
df4 <- rbind(df4,
data.frame(id=5, age=20,parameters="dose.a14",values=8),
data.frame(id=6, age=10,parameters="dose.a1",values=5))
```

```
tmp <- m61r(df4)
tmp$spread(col_name="parameters",col_values="values",pivot=c("id","age"))
tmp
```

`select` *select columns of a data.frame*

Description

Select columns of a data.frame.

Usage

```
# select_(df, variable = NULL)
```

Arguments

`df`	data.frame
`variable`	formula that describes the selection

Value

`select_()` returns a data frame. Properties:

- Only columns following the condition determined by
- variable appear.
- Rows are not modified.

Examples

```
tmp <- select_(CO2,~Type)
head(tmp)

tmp <- select_(CO2,~c(Plant,Type))
head(tmp)

tmp <- select_(CO2,~-Type)
head(tmp)

tmp <- select_(CO2,variable=~-(Plant:Treatment))
head(tmp)

# with m611r class
co2 <- m61r(df=CO2)
```

```
co2$select(~Type)
co2

co2$select(~c(Plant,Type))
co2

co2$select(~-Type)
co2

co2$select(variable=~-(Plant:Treatment))
co2
```

---

summarise *Summarise Formula on Groups*

Description

Summarise of formulas on a data.frame.

Usage

```
# summarise_(df, group_info = NULL, ...)
```

Arguments

df	data.frame
group_info	formula that describes the group
...	formulas to be generated

Details

summarise_ is the aggregation function. It expects the grouping information from get_group_indices_. When a formula expression (e.g., ~mean(uptake)) is run, it is executed for each group subset, relying on Base R lapply over the pre-calculated group indices for performance. **All expressions within summarise_ must return an atomic vector of length 1 for each group.**

Value

summarise_() returns a data frame. If argument group_info is not NULL, output get its first columns called according to the names present in argument group_info. The following columns are called according to the name of each argument present in .... Each row corresponds to processed expressions determined in ... for each group determined in group_info, or over the whole data frame if group_info is NULL.

Examples

```
summarise_(CO2,a=~mean(uptake),b=~sd(uptake))

g_info <- get_group_indices_(CO2, ~c(Type, Treatment))
tmp <- summarise_(CO2, group_info=g_info,mean=~mean(uptake),sd=~sd(uptake))
tmp

# with m61r class
co2 <- m61r(df=CO2)

# summarise
co2$summarise(mean=~mean(uptake),sd=~sd(uptake))
co2

co2 = m61r(CO2)
co2$group_by(~cbind(Type,Treatment))
co2$summarise(mean=~mean(uptake),sd=~sd(uptake))
co2
```

`value` *get or assign a value to a data.frame*

Description

Get or assign a value to a data.frame

Usage

```
# value_(df, i, j)
# 'modify_<-'(df,i,j,value)
```

Arguments

`df`	data.frame
`i`	row
`j`	column
`value`	value to be assigned

Value

The functions `value_` and `'modify_<-'` return a data frame. Properties:

- Only rows determined by
- i appear. If
- i is missing, no row is filtered.
- Only columns determined by

- j appear. If
- j is missing, no column is filtered.

Besides,

- For function `value_`: If argument `i` is non-missing and argument `j` is missing, the function returns an object of the same type as `df`. If both arguments `i` and `j` are missing, the function returns an object of the same type as `df`.
- For function `'modify_<-'`: The function returns an object of the same type as `df`.

Examples

```
tmp <- value_(CO2,1,2)
attributes(tmp) # data frame

tmp <- value_(CO2,1:2,2)
attributes(tmp) # data frame

tmp <- value_(CO2,1:2,2:4)
attributes(tmp) # data frame

tmp <- value_(CO2,,2)
attributes(tmp) # data frame

tmp <- value_(CO2,2)
attributes(tmp) # same as CO2

tmp <- value_(CO2)
attributes(tmp) # same as CO2

df3 <- data.frame(id = 1:4,
                      age = c(40,50,60,50),
                      dose.a1 = c(1,2,1,2),
                      dose.a2 = c(2,1,2,1),
                      dose.a14 = c(3,3,3,3))


'modify_<-'(df3,1,2,6)

'modify_<-'(df3,1:3,2:4,data.frame(c(20,10,90),c(9,3,4),c(0,0,0)))
```

Index