Package 'R.AlphA.Home'

May 7, 2026

Type Package

Title Feel at Home using R, Thanks to Shortcuts Functions Making it Simple

Version 2.0.2

Description A collection of personal functions designed to simplify and streamline common R programming tasks. This package provides reusable tools and shortcuts for frequently used calculations and workflows.

License GPL-3

URL https://github.com/R-alpha-act/R.AlphA.Home

BugReports https://github.com/R-alpha-act/R.AlphA.Home/issues

Depends R (>= 4.0.0)

Imports data.table, diffobj, dplyr, ggplot2, lubridate, magrittr, readxl, writexl, R.utils, R6, rstudioapi, tibble

Suggests shiny, shinyWidgets, testthat (>= 3.0.0)

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation no

Author Raphaël Flambard [aut, cre], Adrien Cocuaud [ctb]

Maintainer Raphaël Flambard <raphael@ralpha.fr>

Repository CRAN

Date/Publication 2026-01-28 13:40:02 UTC

Contents

`cols_pad` *Add Variables to ease data usage in a Pivot Table*

Description

Adds dummy columns to reach the number specified by the user. this is mostly useful to ensure straightforward and easy data updating when using pivot tables in Excel. It allows replacement of the previous data sheet by the new one, without having to take care about the number of columns, which will always be the same.

Usage

```
cols_pad(data, nCols = 100, colPrefix = "x_")
```

Arguments

`data`	The data frame to which dummy columns will be added.
`nCols`	the total number of columns required : default is 100
`colPrefix`	A string used as the prefix for the names of dummy columns.

Value

A data frame with the specified total number of columns.

Examples

```
table <- data.frame(a = 1:5, b = letters[1:5])
extraTable <- cols_pad(table, nCols = 6, colPrefix = "extra_")
print(extraTable)
```

`compareVars` *Compare Table Variables*

Description

Compares column names in two tables based on a given pattern. Provides information about which columns are present in which tables.

Usage

```
compareVars(x, y, pattern = "")
```

Arguments

`x`	A data frame representing the first table.
`y`	A data frame representing the second table.
`pattern`	A string pattern used to filter and compare only a subset of variables (column names).

Value

A list containing:

- `all`: All column names from both tables.
- `common`: Column names found in both tables.
- `onlyX`: Column names found only in the first table (`x`).
- `onlyY`: Column names found only in the second table (`y`).
- `exclusive`: Column names found in only one of the two tables.

Examples

```
# Example tables
table1 <- data.frame(exclusive_1 = 1:5, common_1 = 6:10, common_2 = 11:15)
table2 <- data.frame(common_1 = 16:20, common_2 = 21:25, exclusive_2 = 26:30)

# Compare all columns (no pattern given)
compare_all <- compareVars(table1, table2)
compare_all$common
compare_all$exclusive
compare_all$onlyX
```

```
compare_all$onlyY

# compare only columns following a specific pattern
compare_wPattern <- compareVars(table1, table2, pattern = "1")
compare_wPattern$all
compare_wPattern$common
```

`countSwitches` *create an incremented Counter, based on Start/Stop Markers*

Description

This function aims at identifying sections and sub-sections numbers, based on markers of section starts and ends.

Given a data frame, and the name of a column giving the start/stop markers, it will add columns giving infos about the successive section levels

Usage

```
countSwitches(
  data,
  colNm,
  sttMark,
  endMark,
  includeStt = TRUE,
  includeEnd = TRUE
)
```

Arguments

`data`	A data frame containing the column to process.
`colNm`	A string specifying the column name in ‘data‘ to evaluate.
`sttMark`	A value indicating the start of a series.
`endMark`	A value indicating the end of a series.
`includeStt`	Logical. Should the start marker be included as part of the series? Default is ‘TRUE‘.
`includeEnd`	Logical. Should the end marker be included as part of the series? Default is ‘TRUE‘.

Value

A modified version of the input data frame with additional columns including:

- ‘catLvl‘: The current series level calculated as the difference between the cumulative counts of start and end markers.
- ‘lvl_1‘, ‘lvl_2‘, ‘lvl_3‘: Final series counts returned for each respective level.

Note

This function is currently mostly useful internally, to perform foldAllBr().

Examples

```
# example code
library(dplyr)
tribble(
~step
, "start"
, "content of section 1"
, "start"
, "subsection 1.1"
, "end"
, "end"
, "out of any section"
, "start"
, "section 2"
, "start"
, "subsection 2.1"
, "end"
, "start"
, "subsection 2.2"
, "end"
, "end"
) %>%
countSwitches(colNm = "step", "start", "end")
```

`foldAllBr` *Easily Fold Code Parts*

Description

This function works with code split into parts identified by brackets. The format is as follows:

```
{
  ...
  code from part 1
  ...
} # part 1
{
  ...
} # part 2
```

It automatically identifies parts to fold/unfold easily.

Shortcuts required: Suggestion is to have one shortcut for this function, foldAllBr, and another one for "expand fold" command. Here are the suggested shortcuts depending on Windows or Mac :

- on Mac : use ctrl+shift -> + up (fold) and down (expand)
- on Win : use shift+alt -> + S (fold) and D (expand)

Usage

```
foldAllBr(time = FALSE, debug_getTbl = FALSE)
```

Arguments

time	Logical. If 'TRUE', the function will return ggplot object visualizing execution times for each step.
debug_getTbl	Logical. If 'TRUE', returns the 'docContent' table with tags for debugging purposes.

Value

A list containing:

- debug_info: A data frame with debugging information if debug_getTbl = TRUE.
- timer_plot: A ggplot object visualizing execution times if time = TRUE.

If both parameters are FALSE, the function returns a list with NULL values.

---

importAll *Function to Import and Concatenate Multiple Data Files*

Description

Imports multiple files into a list, concatenates them into a single table, and adds an 'fName' variable. The function automatically handles type harmonization when different file types are mixed and supports various formats including CSV, Excel, and RDS files.

The files can be selected either by giving a file list (character vector), or by specifying a pattern. The function also supports column renaming and file exclusion patterns. When type conflicts are detected across files, the function automatically harmonizes column types using a priority system (character > numeric > Date > integer).

Usage

```
importAll(
  path = ".",
  pattern = "",
  ignore.case = FALSE,
  importFunction = NULL,
  fill = FALSE,
  fileList = NULL,
  renameTable = data.frame(oldName = character(), newName = character()),
  excludePattern = NULL
)
```

Arguments

path	Character. Path to the directory, passed to ‘list.files‘. Default is current directory (".”).
pattern	Character. Pattern to match file names, passed to ‘list.files‘. Default is empty string (all files).
ignore.case	Logical. If ‘TRUE‘, ignores case when matching file names. Passed to ‘list.files‘. Default behavior is case-sensitive (‘FALSE‘).
importFunction	Function. A custom function for importing files. If not set, the function selects an import method based on the file extension.
fill	Logical. Passed to ‘rbind‘ to allow filling missing columns with NA values. Default is ‘FALSE‘.
fileList	Character vector. A vector of file names to import (used instead of ‘pattern‘). Can contain absolute or relative paths.
renameTable	Data.frame. A data.frame with 2 columns (oldName/newName). importAll will rename the columns of each file following this table. Default is empty data.frame.
excludePattern	Character. Pattern to exclude files from import, applied after initial file selection. Default is ‘NULL‘ (no exclusion).

Value

A data.table containing the concatenated table with the fName column indicating the source file for each row. All imported data is converted to data.table format with automatic type harmonization when necessary.

Examples

```
# Directory containing test files
test_path <- tempdir()

# Create test files
write.csv(data.frame(a = 1:3, b = 4:6), file.path(test_path, "file1.csv"))
write.csv(data.frame(a = 7:9, b = 10:12), file.path(test_path, "file2.csv"))
write.csv(data.frame(a = 3:5, b = 8:10), file.path(test_path, "file3.csv"))
saveRDS(data.frame(a = 1:5, b = 6:10), file.path(test_path, "file1.rds"))
saveRDS(data.frame(a = 11:15, b = 16:20), file.path(test_path, "file2.rds"))

# Example 1: Import all csv files
result <- importAll(path = test_path, pattern = "\\.csv$")
print(result)

# Example 2: Import only selected files
file_list <- c("file1.csv", "file2.csv")
result <- importAll(path = test_path, fileList = file_list)
print(result)

# Example 3: Import all .rds files
result <- importAll(path = test_path, pattern = "\\.rds$")
```

```
print(result)

# Example 4: Use a custom import function
custom_import <- function(file) {
  data <- read.csv(file, stringsAsFactors = FALSE)
  return(data)
}
result <- importAll(path = test_path, pattern = "\\.csv$", importFunction = custom_import)
print(result)
```

`left_join_checks` *Left Join with Validation Checks*

Description

a custom usage of left_join, with more detailed checks. Performs a left join and verifies that no unexpected duplicates or mismatches occur. In cas of unexpected results, gives details about what caused the problem.

Usage

```
left_join_checks(
  x,
  y,
  ...,
  req_xAllMatch = 1,
  req_preserved_x = 1,
  behavior = "error",
  showNotFound = FALSE,
  showProblems = TRUE,
  time = FALSE
)
```

Arguments

`x`	A data.table representing the left table.
`y`	A data.table representing the right table.
`...`	Additional arguments passed to ‘dplyr::left_join‘.
`req_xAllMatch`	Logical. Ensure that all rows in ‘x‘ find a match in ‘y‘. Default: FALSE.
`req_preserved_x`	Logical. Ensure that the number of rows in ‘x‘ remains unchanged after the join. Default: TRUE.
`behavior`	Character. Specifies behavior if validation fails. Options: ‘"warning"‘ or ‘"error"‘. (default: ‘"warning"‘)
`showNotFound`	Logical. Show rows from ‘x‘ that did not match with ‘y‘. Default: FALSE.

showProblems	Logical. Display the problems encountered during the joining process, if any.
time	Logical. Internal argument used only for testing purposes, timing the function steps

Value

A data.table containing the joined table.

Examples

```
library(data.table)
library(dplyr)

# Example 1: Simple left join with all matches
table_left <- data.table(id = 1:3, value_left = c("A", "B", "C"))
table_right <- data.table(id = 1:3, value_right = c("X", "Y", "Z"))
result <- left_join_checks(table_left, table_right, by = "id", req_preserved_x = TRUE)
print(result) # Ensures all rows in table_left are preserved

# Example 2: Left join with missing matches
table_left <- data.table(id = 1:5, value_left = c("A", "B", "C", "D", "E"))
table_right <- data.table(id = c(1, 3, 5), value_right = c("X", "Y", "Z"))
result <- left_join_checks(
  table_left,
  table_right,
  by = "id",
  req_preserved_x = TRUE,
  showNotFound = TRUE,
  behavior = "warning"
)
print(result) # Rows from table_left with no matches in table_right are shown
```

---

loadCheck *Load and Install Package if Necessary*

Description

This function checks if a specified package is available in the current R environment. If the package is not installed, it automatically installs it with dependencies and then loads it. #' The function suppresses startup messages to provide a clean loading experience.

Usage

```
loadCheck(package_names)
```

Arguments

package_names	A character vector specifying the name(s) of the package(s) to install (if necessary), and load.

Value

No return value.

Examples

```
# Load a commonly used package
loadCheck("dplyr")

# Load a package that might not be installed
loadCheck("ggplot2")
```

`lum_0_100` *Adjust the Brightness of the Graphics Window for confortable viewing when using ggplot2*

Description

Modifies the brightness level of the active graphics window by adjusting its background color.

This is especially useful when using dark RStudio themes, where a 100 graphic window creates an unconfortable contrast.

Usage

```
lum_0_100(lum = NULL)
```

Arguments

`lum`	Numeric. Brightness level, ranging from 0 (completely dark) to 100 (maximum brightness).

Value

no return value : only apply the theme_set() function

`printif` *Conditionally Print an Object*

Description

This function prints ‘x‘ if ‘show‘ is ‘TRUE‘; otherwise, it returns ‘x‘ unchanged. Its main usage is to "close" dplyr-like treatment chains (using This creates an extremely handy way to process accurate line-by-line debugging, printing results when necessary and removing the print option easily without having to rewrite everything or take care about the last element in the chain.

This saves much code writing and debugging time.

It is also useful to design functions so that users can easily stop elements from being printed

Given the purpose of this function, it is much more convenient to use 1 and 0 for the ’show’ argument than TRUE or FALSE, as this can be switched easily in the editor.

Usage

```
printif(x, show = FALSE, ...)
```

Arguments

x	Any object.
show	A logical value indicating whether to print ‘x‘ (default: ‘FALSE‘).
...	Additional arguments passed to ‘print()‘.

Value

The object ‘x‘, printed if ‘show‘ is ‘TRUE‘.

Examples

```
# Basic usage
printif(42, show = TRUE)
printif(42, show = FALSE)

# Using numeric shortcuts
printif("Hello", 1)
printif("Hello", 0)

# Most useful usage : in a pipeline (requires dplyr)
if (requireNamespace("dplyr", quietly = TRUE)) {
  library(dplyr)
  mtcars %>%
    filter(mpg > 20) %>%
    summarise(mean_hp = mean(hp)) %>%
    printif(1)
}
```

---

quickExport *Quick Export of Data to Excel with Column Padding*

Description

Exports a data frame to an Excel file with optional column padding to ensure a consistent number of columns. This function combines data export functionality with column padding, making it particularly useful for creating Excel files that maintain the same structure across different datasets, especially when used with pivot tables.

Usage

```
quickExport(
  data,
  sheetName = "data_",
  saveDir = root(),
  saveName = "tmp_export.xlsx",
  nCols = 100,
  colPrefix = "x_",
  overwrite = TRUE
)
```

Arguments

data	A data frame to be exported to Excel.
sheetName	A character string specifying the name of the Excel sheet. Default is "data_".
saveDir	A character string specifying the directory path where the file will be saved. Default uses root() function.
saveName	A character string specifying the filename for the Excel file. Default is "tmp_export.xlsx".
nCols	An integer specifying the total number of columns required after padding. Default is 100.
colPrefix	A character string used as the prefix for the names of dummy columns added during padding. Default is "x_".
overwrite	A logical value indicating whether to overwrite existing files. Default is TRUE.

Value

No explicit return value. The function writes an Excel file to the specified location and prints a message with the file path.

Examples

```
## Not run:
# Basic usage with default parameters
df <- data.frame(name = c("Alice", "Bob"), age = c(25, 30))
quickExport(df, sheetName = "employees", saveName = "employee_data.xlsx")

# Custom column padding and file location
sales_data <- data.frame(product = c("A", "B"), sales = c(100, 200))
quickExport(sales_data, nCols = 50, colPrefix = "col_",
           saveName = "sales_report.xlsx", overwrite = FALSE)

## End(Not run)
```

quickSave *Save File in a Directory storing saves, prefixing it with current date*

Description

Saves a file with current date in its name in a sub directory located in the same directory as the original file. Optionally, a note is added after the file name.

Usage

```
quickSave(
  saveDir,
  filePath = NULL,
  saveNote = NULL,
  overwrite = FALSE,
  verbose = FALSE
)
```

Arguments

saveDir	Choose the directory used to store saves. Suggested : 'old'
filePath	Optional, if you want to save another file than the current one : full path of the file you want to save.
saveNote	An optional custom note to append to the file name for the save, allowing to keep track of why this save has been done.
overwrite	Logical. Should an existing save with the same name be overwritten? Default is 'FALSE'.
verbose	logical. If turned to 'TRUE', the save path is displayed

Value

the output value of the function used to copy file

ralpha_fold *Fold Code Sections in RStudio*

Description

Automatically fold code sections in RStudio editor to improve code readability and navigation

Usage

```
ralpha_fold(get_time = getOption("fab_time", default = FALSE))
```

Arguments

get_time	Logical value indicating whether to track and display function execution time. Default is taken from option "fab_time" or FALSE if not set.

Value

Invisible NULL. The function is called for its side effect of folding code sections in the RStudio editor.

Examples

```
## Not run:
# Fold code sections in the current RStudio editor
ralpha_fold()

# Fold code sections and display timing information
ralpha_fold(get_time = TRUE)

## End(Not run)
```

---

ralpha_unfold *Unfold Code Sections in RStudio*

Description

ralpha_fold() and ralpha_unfold() allow usage of the R.AlphA code format that keeps long scripts easily readable.

This format is based on identifying code parts with brackets, and an optional but recommended comment at the end :

```
{
  ...
  code from part 1
  ...
} # part 1
{
  ...
} # part 2
```

then appearing as

```
{...} # part 1
{...} # part 2
```

To stay easy to manipulate, this format requires shortcuts to easily open or close the different sections.

ralpha_fold() will fold the different code parts and go back to beginning of current part

ralpha_unfold() will unfold a code part and jump to the next braces when relevant.

both combined will provide a convenient way to manage what is displayed on screen, ensuring a constant global overview of the document.

Shortcuts required: Here are the suggested shortcuts, both for Mac and Windows :

- ralpha_fold : use ctrl+up
- ralpha_unfold : use ctrl+down

Usage

```
ralpha_unfold()
```

Value

NULL (invisibly). This function performs actions only (cursor movement and unfolding)

---

rdate *Generate Random Dates, with a similar usage as the r* functions*

Description

Generates a vector of random dates within a specified range. This function tries to replicate the usage of the r* functions from stats package, such as runif(), rpois(), ...

Usage

```
rdate(
  x,
  min = paste0(format(Sys.Date(), "%Y"), "-01-01"),
  max = paste0(format(Sys.Date(), "%Y"), "-12-31"),
  sort = FALSE,
  include_hours = FALSE
)
```

Arguments

x	Integer. Length of the output vector (number of random dates to generate).
min	Date. Optional. The minimum date for the range. Defaults to the 1st of January of the current year.
max	Date. Optional. The maximum date for the range. Defaults to the 31st of December of the current year.
sort	Logical. Should the dates be sorted in ascending order? Default is 'FALSE'.
include_hours	Logical. Should the generated dates include time? Default is 'FALSE' (dates only). this will slow down the function

Value

A vector of random dates of length ‘x‘.

Examples

```
# Generate 5 random dates between two specific dates, sorted
rdate(5, min = as.Date("2020-01-01"), max = as.Date("2020-12-31"), sort = TRUE)

# Generate 7 random datetime values (with hours)
rdate(7, include_hours = TRUE)
```

---

ret_lum *Adjust the Brightness of a Hex Color*

Description

Modifies the brightness of a color by multiplying its RGB components by a specified factor.

Mostly for internal usage inside lum_0_100 function.

Usage

```
ret_lum(hexCol, rgbFact)
```

Arguments

hexCol	Character. The color to adjust, specified in hexadecimal format (e.g., "#FF5733").
rgbFact	Numeric. The luminosity factor : - use a factor between 0 and 1 to decrease luminosity - use a factor >1 to increase it The final Brightness value will be maintained between 0 and 1.

Value

A modified hex color in hexadecimal format.

Examples

```
# Example 1: Lightening a color
ret_lum("#FF5733", 1.5)  # Returns a lighter version of the input color

# Example 2: Darkening a color
ret_lum("#FF5733", 0.7)  # Returns a darker version of the input color
```

root *Get Root Directory of Current Source File*

Description

Returns the directory path where the current source code file is located, optionally the full file path, or builds a path relative to it.

This function is especially useful when the same source code is used by multiple users, each using their own environment with different file paths. It helps avoid writing full paths in raw text inside source codes by dynamically retrieving the location of the currently active source file in RStudio.

Usage

```
root(..., includeFName = FALSE)
```

Arguments

`...`	Path components to append to the root directory. If empty, returns only the directory path. If provided, builds a path using `file.path()`.
`includeFName`	Logical. If `TRUE`, returns the full file path including the filename instead of just the directory. Ignored if `...` is provided.

Value

A character string representing either:

- The absolute path of the directory containing the current source file (default)
- The full absolute path including the filename (if `includeFName = TRUE`)
- A path built from the root directory and the provided components (if `...` given)

Note

This function requires RStudio and will only work within the RStudio IDE. It relies on `rstudioapi::getSourceEditorCont`
to retrieve the active source file location.

Examples

```
## Not run:
# Get only the directory path of the current source file
my_dir <- root()
print(my_dir)
# Example output: "/home/user/my_project/R"

# Get the full path including filename
my_file <- root(includeFName = TRUE)
print(my_file)
# Example output: "/home/user/my_project/R/my_script.R"
```

```
# Build a path relative to root
data_path <- root("data", "input.csv")
print(data_path)
# Example output: "/home/user/my_project/R/data/input.csv"

## End(Not run)
```

`Rtimer` *Timer Class for Performance Measurement*

Description

Timer Class for Performance Measurement

Timer Class for Performance Measurement

Details

An R6 class for measuring and tracking execution time of code segments. Provides functionality to add timing checkpoints, calculate time differences, and generate summary reports of performance metrics.

Public Methods

`new()` Initialize a new Timer instance.

`add(...)` Add a timing checkpoint with optional labels.

`get(fill = TRUE)` Generate timing results as data.table.

Methods

Public methods:

- `Rtimer$new()`
- `Rtimer$add()`
- `Rtimer$get()`

Method `new()`**:** Create a new Timer instance

Usage:

```
Rtimer$new()
```

Returns: A Rtimer object

Method `add()`**:** Add a timestamp

Usage:

```
Rtimer$add(...)
```

Arguments:

`...` Optional named labels attached to the timestamp.

Returns: The object itself (invisible) for chaining

Method `get()`**:** Return the collected timings as a `data.table`

Usage:

```
Rtimer$get(fill = TRUE)
```

Arguments:

`fill` Logical; if `TRUE`, fill missing columns when combining entries

Returns: A `data.table` containing timestamps and time differences

Examples

```
## Not run:
tmr <- Rtimer$new()
tmr$add("start")
# some code
tmr$add("end")
result <- tmr$get()
print(result)

## End(Not run)
```

`sepThsd` *Quick Number Formatting with Custom Defaults*

Description

A wrapper for the ‘format‘ function, designed to format numbers with custom defaults for thousands separator, number of significant digits, and scientific notation.

Usage

```
sepThsd(x, big.mark = " ", digits = 1, scientific = FALSE)
```

Arguments

`x`	Numeric. The input values to format.
`big.mark`	Character. The separator for thousands (e.g., ‘" "‘ for "1 000" or ‘","‘ for "1,000"). Default is ‘" "‘.
`digits`	Integer. The number of significant digits to display. Default is ‘1‘.
`scientific`	Logical. Should the numbers be displayed in scientific notation? Default is ‘FALSE‘.

Value

A character vector of formatted numbers.

Examples

```
# Format with a comma as a thousands separator and 3 significant digits
sepThsd(1234567.89, big.mark = ",", digits = 3)
# Use scientific notation
sepThsd(1234567.89, scientific = TRUE)
```

`setOption` *Set Global Option from Named List Element*

Description

This function takes an element from a named list as an argument, and sets a global option based on the list's name.

Where : optionName$element == "value", calling setOption(optionName$element) triggers options(optionName = "value")

Usage

```
setOption(listElement)
```

Arguments

`listElement`	An element from a named list, specified as 'myList$element'.

Details

The function automatically extracts the list name from the argument. The option is then dynamically set using 'options(list_name = element)'.

Value

The function does not return anything but sets an option that can be retrieved using 'getOption(list_name)'.

Examples

```
# Create a temporary list for demonstration
modelOption <- list(model1 = "model_1", model2 = "model_2", model3 = "model_3")

# Set the option
setOption(modelOption$model1)

# Retrieve the option
getOption("modelOption")  # Returns "model_1"

# Clean up
options(modelOption = NULL)
```

shiny_lum_0_100 *Set Shiny Background and Sidebar Colors to a Chosen Shade of Grey*

Description

Adjust the background color of a Shiny app's main body and sidebar based on a specified luminosity level.

The purpose is the same as lum_0_100() function, avoiding problems with high contrast between with graphic windows and dark themes.

Usage

```
shiny_lum_0_100(lum)
```

Arguments

lum	Numeric. Luminosity level, ranging from 0 (black) to 100 (white).

Value

The HTML tags for setting the background and sidebar colors.

show_diff *Compare two texts or files with diffobj*

Description

This function compares two inputs (files or text strings) and displays the differences using the diffobj package with syntax highlighting.

Usage

```
show_diff(input1, input2)
```

Arguments

input1	A character string. Either a file path or text content to compare.
input2	A character string. Either a file path or text content to compare.

Value

A diffobj object containing the visual comparison of the two inputs.

Examples

```
# Compare two text strings
show_diff("Hello\nWorld", "Hello\nR World")

# Compare two files
## Not run:
show_diff("file1.txt", "file2.txt")

## End(Not run)

# Mix file and text
## Not run:
show_diff("file.txt", "New content\nWith changes")

## End(Not run)
```

`timer` *allow organized tracking of R code execution time*

Description

The ‘timer‘ function allows you to append timeStamps to a data.table, and include additional metadata provided as arguments. The last call calculates time differences between timeStamps.

Usage

```
timer(timer_table = data.table(), end = FALSE, ...)
```

Arguments

`timer_table`	A data.table containing the timer log to continue from. Defaults to an empty ‘data.table().
`end`	A logical, inidicating the end of the timer, defaulted to FALSE. ’timer()’ calls must be placed at the beginning of each part : therefore, this ’closing’ step is necessary to compute time for the last part. Time differences between timeStamps are calculated only when closing the timer.
`...`	Additional specifications. Use named arguments to provide documentation on the code parts you are timing : naming the current step, the version of the code you are trying, or any other useful specification

Value

A ‘data.table‘ containing the original data, plus one new timeStamp, and optionally computed time differences :

- ‘timeStamp‘: The current timeStamp (‘POSIXct‘).
- ‘timeStamp_num‘: timeStamp converted to numeric, useful for intermediary calculations.

- ‘dt_num‘: The time difference in seconds between consecutive rows as a numeric value.
- ‘dt_text‘: The formatted time difference in seconds with milliseconds as a character string.
- Additional columns for any information provided by the user via ‘...‘. It allows documentation about the current step running, substeps, which version is being tested, ...

Examples

```
# compare code speed between using a loop, or the mean() function
library(data.table)
library(dplyr)
tmr <- data.table()  # Initialize timer
vec <- rnorm(1e6)    # Example vector

tmr <- timer(tmr, method = "loop")   # timeStamp : 1st step =================
total <- 0
for (i in seq_along(vec)) total <- total + vec[i]
mean_loop <- total / length(vec)

tmr <- timer(tmr, method = "mean()") # timeStamp : 1st step =================
mean_func <- mean(vec)

tmr <- timer(tmr, end = TRUE)        # timeStamp : close timer ==============

t_step1 <- tmr[method == "loop"]$dt_num
t_step2 <- tmr[method == "mean()"]$dt_num
diff_pc <- (t_step2/t_step1 - 1) * 100
diff_txt <- format(diff_pc, nsmall = 0, digits = 1)

# view speed difference
print(tmr %>% select(-matches("_num$")))
paste0("speed difference : ", diff_txt, "%")
```

Index