Package ‘common’

February 23, 2026

Type Package

Title Solutions for Common Problems in Base R

Version 1.1.5

Maintainer David Bosak <dbosak01@gmail.com>

Description Contains functions for solving commonly encountered problems while programming in R. This package is intended to provide a lightweight supplement to Base R, and will be useful for almost any R user.

License CC0

Encoding UTF-8

URL https://common.r-sassy.org, https://github.com/dbosak01/common

BugReports https://github.com/dbosak01/common/issues

Depends R (>= 3.6.0)

Suggests knitr, rmarkdown, testthat (>= 3.0.0), glue, box, rstudioapi

Enhances base

Imports utils

Config/testthat/edition 3

RoxygenNote 7.3.3

VignetteBuilder knitr

NeedsCompilation no

Author David Bosak [aut, cre],
Duong Tran [ctb]

Repository CRAN

Date/Publication 2026-02-23 08:30:09 UTC

Contents

`changed` *Identify changed values*

Description

The `changed` function identifies changes in a vector or data frame. The function is used to locate grouping boundaries. It will return a TRUE each time the current value is different from the previous value. The `changed` function is similar to the Base R `duplicated` function, except the `changed` function will return TRUE even if the changed value is not unique.

Usage

```
changed(x, reverse = FALSE, simplify = FALSE)
```

Arguments

`x`	A vector of values in which to identify changed values. Also accepts a data frame. In the case of a data frame, the function will use all columns. Input data can be any data type.
`reverse`	Reverse the direction of the scan to identify the last value in a group instead of the first.
`simplify`	If the input data to the function is a data frame, the simplify option will return a single vector of indicator values instead of a data frame of indicator values.

Details

For a data frame, by default, the function will return another data frame with an equal number of change indicator columns. The column names will be the original column names, with a ".changed" suffix.

To collapse the multiple change indicators into one vector, use the "simplify" option. In this case, the returned vector will essentially be an "or" operation across all columns.

Value

A vector of TRUE or FALSE values indicating the grouping boundaries of the vector or data frame. If the input data is a data frame and the "simplify" parameter is FALSE, the return value will be a data frame of logical vectors describing changed values for each column.

Examples

```
# Create sample vector
v1 <- c(1, 1, 1, 2, 2, 3, 3, 3, 1, 1)

# Identify changed values
res1 <- changed(v1)

# View results
res1
# [1] TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE

# Create sample data frame
v2 <- c("A", "A", "A", "A", "A", "A", "B", "B", "B", "B")
dat <- data.frame(v1, v2)

# View original data frame
dat
#    v1 v2
# 1   1  A
# 2   1  A
# 3   1  A
# 4   2  A
# 5   2  A
# 6   3  A
# 7   3  B
# 8   3  B
# 9   1  B
# 10  1  B

# Get changed values for each column
res2 <- changed(dat)

# View results
res2
#    v1.changed v2.changed
# 1        TRUE       TRUE
# 2       FALSE      FALSE
# 3       FALSE      FALSE
# 4        TRUE      FALSE
# 5       FALSE      FALSE
# 6        TRUE      FALSE
# 7       FALSE       TRUE
# 8       FALSE      FALSE
# 9        TRUE      FALSE
# 10      FALSE      FALSE

# Get changed values for all columns
res3 <- changed(dat, simplify = TRUE)

# View results
res3
# [1] TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE

# Get last items in each group instead of first
res4 <- changed(dat, reverse = TRUE)

# View results
res4
#    v1.changed v2.changed
# 1       FALSE      FALSE
# 2       FALSE      FALSE
# 3        TRUE      FALSE
# 4       FALSE      FALSE
# 5        TRUE      FALSE
# 6       FALSE       TRUE
# 7       FALSE      FALSE
# 8        TRUE      FALSE
# 9       FALSE      FALSE
# 10       TRUE       TRUE
```

copy.attributes *Copy attributes between two data frames*

Description

A function to copy column attributes from one data frame to another. The function will copy all attributes attached to each column. The column order does not matter, and the data frames do not need identical structures. The matching occurs by column name, not position. Any existing attributes on the target data frame that do not match the source data frame will be retained unaltered.

Usage

```
copy.attributes(source, target)
```

Arguments

`source`	A data frame to copy attributes from.
`target`	A data frame to copy attributes to.

Value

The data frame in the `target` parameter, with updated attributes from `source`.

See Also

Other overrides: `labels.data.frame()`, `sort.data.frame()`

Examples

```
# Prepare data
dat1 <- mtcars
dat2 <- mtcars

# Set labels for dat1
labels(dat1) <- list(mpg = "Miles Per Gallon",
                     cyl = "Cylinders",
                     disp = "Displacement")

# Copy labels from dat1 to dat2
dat2 <- copy.attributes(dat1, dat2)

# View results
labels(dat2)
# $mpg
# [1] "Miles Per Gallon"
#
# $cyl
# [1] "Cylinders"
#
# $disp
# [1] "Displacement"
```

`dir.find` *Search for directories*

Description

A function to find directories on the file system. The function starts from the directory specified in the `path` parameter, and searches outward in a radiating pattern for the ending directory name in the `pattern` parameter. The up and down parameters define the scope of the search. Results are returned as a vector of full paths in the order encountered. This function has an advantage over `list.dirs` in that it can search both up and down the file system, and limit the scope of the search.

Usage

```
dir.find(path = ".", pattern = NULL, up = 5, down = 2)
```

Arguments

`path`	The directory to start searching from. Default is the current working directory.
`pattern`	A full or partial name of the directory to find. If partial, use the question mark (?) or asterisk (*) characters to indicate the missing piece. Default is NULL, which will return all directories.
`up`	The number of levels above the base path to search. A value of zero (0) means that you do not want to search up. A value of -1 means you do not want to include the base directory in the search results. Default is 5 levels up.
`down`	The number of levels below the base path to search. A value of zero (0) means that you do not want to search down. A value of -1 means you do not want to include the base directory in the search results. Default is 2 levels down.

Details

The `dir.find` function attempts to find a directory based on a full or partial directory name. The directory name is passed on the `pattern` parameter. The `pattern` accepts both the single character question mark wild card (?), and the asterisk, multi-character wild card (*). Searches are case-insensitive.

Starting from the base path specified in the `path` parameter, the function searches both above and below the base path in an alternating pattern. The function will first search the base path, then up one level, then down one level, and so on. The boundaries of the search can be controlled by the `up` and `down` parameters.

You can control whether or not you want the base directory included in the results. To include this directory, ensure both `up` and `down` parameters are zero or greater. If either of these parameters is set to -1, the base path will be excluded. For example, `up = 3, down = 1` will search up three levels, and down one level. `up = 3, down = 0` will search up three levels and not search down, but will include the base directory. `up = 3, down = -1` will search up three levels, not search down, and not include the base directory in the results.

Value

A vector of one or more full directory paths that met the search criteria. The paths in the vector are returned in the order of matches, according to the search algorithm. That means the first directory found will be in position one, and the last directory found will be at the end of the vector. A NULL is returned if no directories met the search criteria.

See Also

Other fileops: `Sys.path()`, `file.find()`, `source.all()`

Examples

```
# Search for a directory named "prod"
file.find(pattern = "prod")

# Search for a directory that starts with "dat"
file.find(pattern = "dat*")
```

```
# Search for a directory up only
file.find(pattern = "dat*", up = 3, down = 0)

# Search for directories up only, skipping the current working directory
file.find(pattern = "dat*", up = 3, down = -1)
```

`file.find` *Search for files*

Description

A function to find files on the file system. The function starts from the directory specified in the `path` parameter, and searches outward in a radiating pattern for the file name in the `pattern` parameter. Results are returned as a vector of full paths in the order encountered. The up and down parameters define the scope of the search. This function has an advantage over `list.files` in that it can search both up and down the file system, and limit the scope of the search.

Usage

```
file.find(path = ".", pattern = NULL, up = 3, down = 1)
```

Arguments

`path`	The directory to start searching from. Default is the current working directory.
`pattern`	A full or partial name of the file to find. If partial, use the question mark (?) or asterisk (*) characters to indicate the missing piece. Default is NULL, which will return all files.
`up`	The number of levels above the base path to search. A value of zero (0) means that you do not want to search up. A value of -1 means you do not want to include the base directory in the search results. Default is 3 levels up.
`down`	The number of levels below the base path to search. A value of zero (0) means that you do not want to search down. A value of -1 means you do not want to include the base directory in the search results. Default is 1 level down.

Details

The `file.find` function attempts to find a file based on a full or partial file name. The file name is passed on the `pattern` parameter. The `pattern` accepts both the single character question mark wild card (?), and the asterisk multi-character wild card (*). Searches are case-insensitive.

Starting from the base path specified in the `path` parameter, the function searches both above and below the base path in an alternating pattern. The function will first search the base path, then up one level, then down one level, and so on. The boundaries of the search can be controlled by the up and down parameters.

You can control whether or not you want files from the base directory included in the results. To include these files, ensure both up and down parameters are zero or greater. If either of these parameters is set to -1, the base path will be excluded. For example, `up = 3, down = 1` will search up three levels, and down one level. `up = 3, down = 0` will search up three levels and not search down, but will include the base directory. `up = 3, down = -1` will search up three levels, not search down, and not include the base directory in the results.

Value

A vector of one or more full file paths that met the search criteria. The paths in the vector are returned in the order of matches, according to the search algorithm. That means the first file found will be in position one, and the last file found will be at the end of the vector. A NULL is returned if no files met the search criteria.

See Also

Other fileops: `Sys.path()`, `dir.find()`, `source.all()`

Examples

```
# Search for a file named "globals.R"
file.find(getwd(), "globals.R")

# Search for Rdata files
file.find(getwd(), "*.Rdata")

# Search for Rdata files up only
file.find(getwd(), "*.Rdata", up = 3, down = 0)

# Search for Rdata files up only, skipping the current working directory
file.find(getwd(), "*.Rdata", up = 3, down = -1)
```

`find.names` *Search for names*

Description

A function to search for variable names in a data.frame or tibble. The function features wild card pattern matching, start and end boundaries, and names to exclude.

Usage

```
find.names(
  x,
  pattern = NULL,
  exclude = NULL,
  start = NULL,
  end = NULL,
  ignore.case = TRUE
)
```

Arguments

`x`	A data frame or tibble whose names to search. Parameter also accepts a character vector of names.
`pattern`	A vector of patterns to search for. The asterisk (*) and question mark (?) characters may be used to indicate partial matches.
`exclude`	A vector of patterns to exclude from the search results. The asterisk (*) and question mark (?) characters may be used to indicate partial matches.
`start`	A variable name or position to start the search. Default is 1.
`end`	A variable name or position to end the search. Default is the length of the name vector.
`ignore.case`	Whether to perform a case sensitive or insensitive search. Valid values are TRUE and FALSE. Default is TRUE.

Value

A vector of variable names that met the search criteria.

Examples

```
# Show all names for reference
names(mtcars)
#  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear" "carb"

# Names that start with "c"
find.names(mtcars, "c*")
# [1] "cyl"  "carb"

# Names that start with "c" or "d"
find.names(mtcars, c("c*", "d*"))
# [1] "cyl"  "carb" "disp" "drat"

# Names between "disp" and "qsec"
find.names(mtcars, start = "disp", end = "qsec")
# [1] "disp" "hp"   "drat" "wt"   "qsec"

# Names that start with "c" or "d" after position 5
find.names(mtcars, c("c*", "d*"), start = 5)
# [1] "carb" "drat"

# Names between "disp" and "qsec" excluding "wt"
find.names(mtcars, start = "disp", end = "qsec", exclude = "wt")
# [1] "disp" "hp"   "drat" "qsec"
```

`labels.data.frame` *Get or set labels for a data frame*

Description

The `labels` function extracts all assigned labels from a data frame, and returns them in a named list. The function also assigns labels from a named list. This function is a data frame-specific implementation of the Base R `labels` function.

Usage

```
## S3 method for class 'data.frame'
labels(object, ...)

labels(x) <- value
```

Arguments

`object`	A data frame or tibble.
`...`	Follow-on parameters. Required for generic function.
`x`	A data frame or tibble
`value`	A named list of labels. The labels must be quoted strings.

Details

If labels are assigned to the "label" attributes of the data frame columns, the `labels` function will extract those labels. The function will return the labels in a named list, where the names correspond to the name of the column that the label was assigned to. If a column does not have a label attribute assigned, that column will not be included in the list.

When used on the receiving side of an assignment, the function will assign labels to a data frame. The labels should be in a named list, where each name corresponds to the data frame column to assign the label to.

Finally, if you wish to clear out the label attributes, assign a NULL value to the `labels` function.

Value

A named list of labels. The labels must be quoted strings.

See Also

Other overrides: `copy.attributes()`, `sort.data.frame()`

Examples

```
# Take subset of data
df1 <- mtcars[1:10, c("mpg", "cyl")]

# Assign labels
labels(df1) <- list(mpg = "Mile Per Gallon", cyl = "Cylinders")

# Examine attributes
str(df1)
# 'data.frame': 10 obs. of  2 variables:
# $ mpg: num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2
# ..- attr(*, "label")= chr "Mile Per Gallon"
# $ cyl: num  6 6 4 6 8 6 8 4 4 6
# ..- attr(*, "label")= chr "Cylinders"

# View assigned labels
labels(df1)
# $mpg
# [1] "Mile Per Gallon"
#
# $cyl
# [1] "Cylinders"

# Clear labels
labels(df1) <- NULL

# Display Cleared Labels
labels(df1)
# list()
```

`roundup` *Rounds numbers up*

Description

A function that rounds positive numbers up when the last digit is a 5. For negative numbers ending in 5, the function actually rounds down. "Round away from zero" is the most accurate description of this function.

Usage

```
roundup(x, digits = 0)
```

Arguments

`x`	A vector of values to round. Also accepts a data frame. In the case of a data frame, the function will round all numeric columns.
`digits`	A number of decimal places to round to. Default is zero.

Value

The rounded data vector.

Examples

```
# Round to even
round(2.4)   # 2
round(2.5)   # 2
round(-2.5)  # -2
round(2.6)   # 3

# Round up
roundup(2.4)  # 2
roundup(2.5)  # 3
roundup(-2.5) # -3
roundup(2.6)  # 3
```

`sort.data.frame` *Sorts a data frame*

Description

An overload to the Base R `sort` function for data frames. Allows multiple columns to be sorted easily. Also allows you to control the sort direction for each column independently.

Usage

```
## S3 method for class 'data.frame'
sort(
  x,
  decreasing = FALSE,
  ...,
  by = NULL,
  ascending = TRUE,
  na.last = TRUE,
  index.return = FALSE
)
```

Arguments

`x`	The input data frame to sort.
`decreasing`	This parameter was added to conform to the S3 generic method signature of the `sort` function, and will be ignored here. Please use the `ascending` parameter.
`...`	This parameter is required for the generic method signature. Anything passed on it will be ignored.

`by`	A vector of column names to sort by. If this parameter is not supplied, the function will sort by all columns in order from left to right.
`ascending`	A vector of TRUE or FALSE values corresponding to the variables on the by parameter. These values will determine the direction to sort each column. Ascending is TRUE, and descending is FALSE. The vector will be recycled if it is short, and truncated if it is long. By default, all variables will be sorted ascending.
`na.last`	Whether to put NA values first or last in the sort. If TRUE, NA values will sort to the bottom. If FALSE, NA values will sort to the top. The default is TRUE. Parameter also accepts a vector of TRUE/FALSE values, which correspond one to one with the number of by variables.
`index.return`	Whether to return the sorted data frame or a vector of sorted index values. If this parameter is TRUE, the function will return sorted index values. By default, the parameter is FALSE, and will return the sorted data frame.

Value

The function returns either a sorted data frame or a sorted vector of row index values, depending on the value of the `index.return` parameter. If `index.return` is FALSE, the function will return the sorted data frame. If the `index.return` parameter is TRUE, it will return a vector of row indices.

See Also

Other overrides: `copy.attributes()`, `labels.data.frame()`

Examples

```
# Prepare unsorted sample data
dt <- mtcars[1:10, 1:3]
dt
#                    mpg cyl  disp
# Mazda RX4         21.0   6 160.0
# Mazda RX4 Wag     21.0   6 160.0
# Datsun 710        22.8   4 108.0
# Hornet 4 Drive    21.4   6 258.0
# Hornet Sportabout 18.7   8 360.0
# Valiant           18.1   6 225.0
# Duster 360        14.3   8 360.0
# Merc 240D         24.4   4 146.7
# Merc 230          22.8   4 140.8
# Merc 280          19.2   6 167.6

# Sort by mpg ascending
dt1 <- sort(dt, by = "mpg")
dt1
#                    mpg cyl  disp
# Duster 360        14.3   8 360.0
# Valiant           18.1   6 225.0
# Hornet Sportabout 18.7   8 360.0
# Merc 280          19.2   6 167.6
# Mazda RX4         21.0   6 160.0
# Mazda RX4 Wag     21.0   6 160.0
# Hornet 4 Drive    21.4   6 258.0
# Datsun 710        22.8   4 108.0
# Merc 230          22.8   4 140.8
# Merc 240D         24.4   4 146.7

# Sort by mpg descending
dt1 <- sort(dt, by = "mpg", ascending = FALSE)
dt1
#                    mpg cyl  disp
# Merc 240D         24.4   4 146.7
# Datsun 710        22.8   4 108.0
# Merc 230          22.8   4 140.8
# Hornet 4 Drive    21.4   6 258.0
# Mazda RX4         21.0   6 160.0
# Mazda RX4 Wag     21.0   6 160.0
# Merc 280          19.2   6 167.6
# Hornet Sportabout 18.7   8 360.0
# Valiant           18.1   6 225.0
# Duster 360        14.3   8 360.0

# Sort by cyl then mpg
dt1 <- sort(dt, by = c("cyl", "mpg"))
dt1
#                    mpg cyl  disp
# Datsun 710        22.8   4 108.0
# Merc 230          22.8   4 140.8
# Merc 240D         24.4   4 146.7
# Valiant           18.1   6 225.0
# Merc 280          19.2   6 167.6
# Mazda RX4         21.0   6 160.0
# Mazda RX4 Wag     21.0   6 160.0
# Hornet 4 Drive    21.4   6 258.0
# Duster 360        14.3   8 360.0
# Hornet Sportabout 18.7   8 360.0

# Sort by cyl descending then mpg ascending
dt1 <- sort(dt, by = c("cyl", "mpg"),
            ascending = c(FALSE, TRUE))
dt1
#                    mpg cyl  disp
# Duster 360        14.3   8 360.0
# Hornet Sportabout 18.7   8 360.0
# Valiant           18.1   6 225.0
# Merc 280          19.2   6 167.6
# Mazda RX4         21.0   6 160.0
# Mazda RX4 Wag     21.0   6 160.0
# Hornet 4 Drive    21.4   6 258.0
# Datsun 710        22.8   4 108.0
# Merc 230          22.8   4 140.8
# Merc 240D         24.4   4 146.7
```

source.all — *Source all programs in a directory*

Description

A function to source all programs in a specified directory. The function will run each R program file in the directory, and then return a data frame of results of the run.

Usage

```
source.all(path = ".", pattern = NULL, exclude = NULL, isolate = TRUE)
```

Arguments

`path`	The directory to source programs from. Default is the current working directory.
`pattern`	A full or partial name of the programs to source. If partial, use the question mark (?) or asterisk (*) characters to indicate the missing piece(s). Default is NULL, which will return all programs. You may pass multiple patterns as a vector. In that case, the function will perform an "or" operation on each pattern. Note that it is not necessary to include the ".R" file extension in your patterns. It is assumed that all source files have a ".R" extension.
`exclude`	A vector of patterns to exclude from the included programs. The exclusion patterns can be the names of specific programs or a wild card exclusion. The asterisk (*) and question mark (?) characters may be used to indicate partial matches. Similar to the "pattern" parameter, the ".R" file extension can be ignored.
`isolate`	Whether to isolate each source call to its own environment. Valid values are TRUE, FALSE, or an environment to run in. If the isolate parameter is FALSE, the programs will run in the global environment. Default is TRUE.

Details

The `source.all` function attempts to run all programs in a directory. This function is useful for batch runs. It has parameters to control which programs are run or not run. By default, the function will run all programs in the working directory. You can use the "pattern" and "exclude" parameters to specify individual program names, or wild card matches. Inclusion and exclusion patterns are case-insensitive.

Note that the function will run all programs, regardless of any errors. Errors will be indicated in the "Status" and "Message" columns of the result dataset.

Value

A data frame of the results of the source operation. The data frame will show each file sourced, the time started, the time ended, the status, and any error messages. The status value is either 0 (no errors) or 1 (errors).

Result Dataset

The `source.all` function returns a dataset showing the results of the source operation. There will be one row for each program executed. The return dataset has the following columns:

- **Filename**: The name of the program.
- **StartTime**: The date and time execution started.
- **EndTime**: The date and time execution ended.
- **Status**: A numeric value indicating whether or not the program returned errors or warnings. A zero (0) value indicates that no errors occurred. A one (1) value indicates that an error occurred. Warnings can also be generated along with an error, but the status will still be one (1). A two (2) value indicates that warnings occurred but no errors. Note that capture of warnings is less reliable than the capture of errors. It is possible that a program may generate a warning and still return a zero (0) status. If you want to ensure that warnings are detected, convert them to errors with `options(warn = 2)`.
- **Message**: If errors or warnings are returned from the program, they will be shown in this column. Multiple messages will be separated with a semi-colon (;) and a carriage return.

In addition to the information shown above, the results dataset will have attributes assigned with the parameter values passed to the function. Those attributes can be observed with the Base R `attributes()` function.

Source Isolation

Multiple programs running in the same environment have a risk of conflicting variables or data. Variables created by the first program can possibly interfere with the running of the next program. Or they could conflict with variables in the global environment. To avoid such conflicts, each program is run in its own environment by default. `isolate = TRUE` starts each program with a clean workspace, and is the best choice for running programs in batch.

There may be situations, however, where you do not want to isolate the source calls. For example, if you are loading functions from a utility library, you may actually wanted them loaded into the global environment so they can by accessed by you or your programs. In this case, set the "isolate" parameter to FALSE.

Lastly, there may be situations where you want to intentionally share an environment, or extract values create by the running programs. In this case, you can instantiate a new environment yourself, and pass that to the "isolate" parameter instead of TRUE or FALSE. Note that this environment will be shared by all programs, but will not have access to the global environment.

See Also

Other fileops: `Sys.path()`, `dir.find()`, `file.find()`

Examples

```
# Create temp directory
tmp <- tempdir()

# Write program 1
p1 <- file(file.path(tmp, "prog1.R"))
```

```
writeLines("print('Hello from program 1')", p1)
close(p1)

# Write program 2
p2 <- file(file.path(tmp, "prog2.R"))
writeLines("stop('Error from program 2')", p2)
close(p2)

# Write program 3
p3 <- file(file.path(tmp, "prog3.R"))
writeLines("print('Hello from program 3')", p3)
close(p3)

# Example #1: Run all programs
res1 <- source.all(tmp)
# [1] "Hello from program 1"
# [1] "Hello from program 3"

# View results
res1
#   Filename           StartTime             EndTime Status              Message
# 1  prog1.R 2024-03-05 10:12:04 2024-03-05 10:12:04      0              Success
# 2  prog2.R 2024-03-05 10:12:04 2024-03-05 10:12:04      1 Error from program 2
# 3  prog3.R 2024-03-05 10:12:04 2024-03-05 10:12:04      0              Success

#' # Example #2: Exclusion criteria
res2 <- source.all(tmp, exclude = "prog2")
# [1] "Hello from program 1"
# [1] "Hello from program 3"

# View results
res2
# Filename           StartTime             EndTime Status Message
# 1  prog1.R 2024-03-05 10:13:24 2024-03-05 10:13:24      0 Success
# 2  prog3.R 2024-03-05 10:13:24 2024-03-05 10:13:24      0 Success

# Example #3: Inclusion criteria
res3 <- source.all(tmp, pattern = "*2")

# View results
res3
#   Filename           StartTime             EndTime Status              Message
# 1  prog2.R 2024-03-05 10:16:41 2024-03-05 10:16:41      1 Error from program 2

# View attributes
attributes(res3)
# $names
# [1] "Filename"  "StartTime" "EndTime"   "Status"    "Message"
#
# $class
# [1] "data.frame"
#
# $row.names
# [1] 1
#
# $path
# [1] "C:\Users\dbosa\AppData\Local\Temp\RtmpGAXYJl"
#
# $pattern
# [1] "*2.R"
#
# $errors
# [1] 1
```

---

spaces *Creates a string of blank spaces*

Description

A function to create a string of some number of blank spaces. This function is useful when trying to align things.

Usage

```
spaces(num)
```

Arguments

num	The desired number of spaces.

Value

A single character vector of blank spaces.

Examples

```
# Create spaces
spaces(10)
# [1] "          "

# Use spaces to separate something
str <- "Left" %p% spaces(40) %p% "Right"
str
# [1] "Left                                        Right"
```

`subsc` *Converts a string to UTF-8 subscript*

Description

The `subsc` function translates a normal character to a UTF-8 subscript character. The function can be used to generate subscripts for many common characters. All numeric characters and some lower case letters have UTF-8 subscripts. There are no upper case subscript letters. This function is useful because it saves you from having to look up the subscript character code, or copy and paste from the internet. If a corresponding subscript character code does not exist, a question mark will be returned for that character.

Usage

```
subsc(x)
```

Arguments

`x`	A string to be converted to subscript.

Value

The subscript version of the string passed to the function, if one exists. Otherwise, a question mark will be returned.

See Also

Other utf8: `supsc()`, `symbol()`

Examples

```
# Date string
paste0("December 5", subsc("th"))

# Chemistry
paste0("H", subsc("2"), "SO", subsc("4"))
```

`supsc` *Converts a string to UTF-8 superscript*

Description

The `supsc` function translates a normal character to a UTF-8 superscript character. The function can be used to generate superscripts for many common characters. Most alphabetic and numeric characters have UTF-8 superscripts. This function is useful because it saves you from having to look up the superscript character code, or copy and pasting from the internet. If a corresponding superscript character code does not exist, a question mark will be returned for that character.

Usage

```
supsc(x)
```

Arguments

`x`	A string to be converted to superscript.

Value

The superscript version of the string passed to the function, if one exists. Otherwise, a question mark will be returned.

See Also

Other utf8: `subsc()`, `symbol()`

Examples

```
# Single letter
paste0(supsc("a"), "Footnote")

# Single number
paste0(supsc("1"), "Footnote")

# Character string
paste0("December 5", supsc("th"))

# Formula
paste0("x", supsc("(a+1)"))
```

`symbol` *Gets UTF-8 symbol characters*

Description

The `symbol` function gets UTF-8 symbol characters. You can call this function to look up trademarks, Greek letters, and mathematical operators. The function uses HTML entity keywords to indicate which symbol to return. You may pass more than one keyword in a single call to get a combined result. Any characters not recognized as a keyword will be left alone. Characters surrounded by square brackets ([]) will be subscripted. Characters surrounded by square brackets and prefixed with an up arrow (^[]) will be superscripted.

Usage

```
symbol(keyword)
```

Arguments

`keyword`	A symbol keyword. This keyword follows HTML conventions. See the **Keywords** section for a complete list of all supported keywords.

Value

The requested UTF-8 symbol character or characters.

Keywords

The following symbol keywords are available:

- **Trademark and Copyright**: copy, reg, trade
- **Financial**: cent, euro, pound, rupee, ruble, yen, yuan
- **Mathmatical**: asymp, bcong, cong, coprod, empty, fnof, ge, int, Int, infin, le, ncong, ne, not, part, plusmn, prod, radic, sime, sum
- **Logical**: and, cap, cup, comp, cuvee, cuwed, exist, forall, fork, isin, nexist, ni, notin, notni, nsub, nsup, or, sub, sup, xcap, xcup, xvee, xwedge
- **Greek uppercase letters**: Alpha, Beta, Gamma, Delta, Epsilon, Zeta, Eta, Theta, Iota, Kappa, Lambda, Mu, Nu, Xi, Omicron, Pi, Rho, Sigma, Tau, Upsilon, Phi, Chi, Psi, Omega
- **Greek lowercase letters**: alpha, beta, gamma, delta, epsilon, zeta, eta, theta, iota, kappa, lambda, mu, nu, xi, omicron, pi, rho, sigma, tau, upsilon, phi, chi, psi, omega
- **Arrows**: rarr, larr, barr, uarr, darr, harr, rArr, lArr, uArr, dArr, hArr
- **Other Symbols**: dagger, ddagger, deg, permil, pertenk, sect

See Also

Other utf8: `subsc()`, `supsc()`

Examples

```
# Trademark symbol
symbol("My Companytrade")

# Registered Trademark symbol
symbol("My Companyreg")

# Dagger symbol concatenated
paste0(symbol("dagger"), "My footnotes")

# Alpha squared
symbol("alpha^[2]")

# Greek Symbols
symbol("SigmaPsiZeta")

# Useful Math Symbols
symbol("asymp ge le ne plusmn empty fnof radic sum")
```

```
# Useful Logical Symbols
symbol("forall isin notin cup cap and or")

# Chemistry
symbol("2H[2] + O[2] barr 2H[2]O")
```

Sys.path *Returns the path of the current program*

Description

A function that gets the full path of the currently running program. If the function fails to retrieve the path for some reason, it will return a NULL. The function takes no parameters.

Usage

```
Sys.path()
```

Value

The full path of the currently running program, or a NULL.

See Also

Other fileops: dir.find(), file.find(), source.all()

Examples

```
# Get current path
pth <- Sys.path()
pth
# [1] "C:/programs/myprogram.R"
```

v *Combine unquoted values*

Description

A function to quote and combine unquoted values. The function will return a vector of quoted values. This function allows you to use non-standard evaluation for any parameter that accepts a string or vector of strings.

Usage

```
v(...)
```

Arguments

...	One or more unquoted values.

Value

A vector of quoted values.

Examples

```
# Combine unquoted values
v(var1, var2, var3)
# [1] "var1" "var2" "var3"

# Data frame subset
dat <- mtcars[1:5, v(mpg, cyl, disp)]
dat
#                    mpg cyl disp
# Mazda RX4         21.0   6  160
# Mazda RX4 Wag     21.0   6  160
# Datsun 710        22.8   4  108
# Hornet 4 Drive    21.4   6  258
# Hornet Sportabout 18.7   8  360

# Data frame sort
dat2 <- sort(dat, by = v(cyl, mpg))
dat2
#                    mpg cyl disp
# Datsun 710        22.8   4  108
# Mazda RX4         21.0   6  160
# Mazda RX4 Wag     21.0   6  160
# Hornet 4 Drive    21.4   6  258
# Hornet Sportabout 18.7   8  360
```

`%eq%` *Check equality of two objects*

Description

The goal of the `%eq%` operator is to return a TRUE or FALSE value when any two objects are compared. The function provides a simple, reliable equality check that allows comparing of NULLs, NA values, and atomic data types without error.

The function also allows comparing of data frames. It will return TRUE if all values in the data frames are equal, and ignores differences in attributes.

Usage

```
x1 %eq% x2
```

Arguments

x1	The first object to compare
x2	The second object to compare

Value

A single TRUE or FALSE value depending on whether the objects are equal.

See Also

Other operators: %ge%(), %gt%(), %le%(), %lt%(), %ne%(), %p%()

Examples

```
# Comparing of NULLs and NA
NULL %eq% NULL        # TRUE
NULL %eq% NA          # FALSE
NA %eq% NA            # TRUE
1 %eq% NULL           # FALSE
1 %eq% NA             # FALSE

# Comparing of atomic values
1 %eq% 1              # TRUE
"one" %eq% "one"      # TRUE
1 %eq% "one"          # FALSE
1 %eq% Sys.Date()     # FALSE

# Comparing of vectors
v1 <- c("A", "B", "C")
v2 <- c("A", "B", "D")
v1 %eq% v1            # TRUE
v1 %eq% v2            # FALSE

# Comparing of data frames
mtcars %eq% mtcars    # TRUE
mtcars %eq% iris      # FALSE
iris %eq% iris[1:50,] # FALSE

# Mixing it up
mtcars %eq% NULL      # FALSE
v1 %eq% NA            # FALSE
1 %eq% v1             # FALSE
```

%ge% *Perform greater than or equal comparison between two objects*

Description

The goal of the comparison operators is to return a TRUE or FALSE value when any two objects are compared. The operators provides a simple, reliable equality check that allows comparing of NULLs, NA values, and atomic data types without error. This operator performs a greater than or equal comparison.

For data frames, the operator will compare all values in all columns, and return a single TRUE if all values in the second data frame are greater than or equal to the corresponding values in the first data frame.

Usage

```
x1 %ge% x2
```

Arguments

x1	The first object to compare
x2	The second object to compare

Details

Additional details...

Value

A single TRUE or FALSE value indicating the results of the comparison.

See Also

Other operators: `%eq%()`, `%gt%()`, `%le%()`, `%lt%()`, `%ne%()`, `%p%()`

Examples

```
# Comparing of NULLs and NA
NULL %ge% NULL       # TRUE
NULL %ge% NA         # FALSE
NA %ge% NA           # TRUE
1 %ge% NULL          # FALSE
1 %ge% NA            # FALSE

# Comparing of atomic values
1 %ge% 1             # TRUE
2 %ge% 1             # TRUE
1 %ge% 2             # FALSE
"one" %ge% "one"     # TRUE
1 %ge% "one"         # FALSE
1 %ge% Sys.Date()    # FALSE
Sys.Date() %ge% 1    # TRUE (Sys.Date() is a number)

# Comparing of vectors
v1 <- c(0, 1, 2)
```

```
v2 <- c(1, 2, 3)
v3 <- c(2, 3, 4)
v1 %ge% v1            # TRUE
v1 %ge% v2            # FALSE
v2 %ge% v1            # TRUE
v3 %ge% v1            # TRUE

# Comparing of data frames
d1 <- data.frame(A = v1, B = v2)
d2 <- data.frame(A = v2, B = v3)
d1 %ge% d1            # TRUE
d1 %ge% d2            # FALSE
d2 %ge% d1            # TRUE

# Mixing it up
d1 %ge% NULL          # FALSE
v1 %ge% d1            # FALSE
1 %ge% v1             # FALSE
```

%gt% *Perform greater than comparison between two objects*

Description

The goal of the comparison operators is to return a TRUE or FALSE value when any two objects are compared. The operators provides a simple, reliable equality check that allows comparing of NULLs, NA values, and atomic data types without error. This operator performs a greater than comparison.

For data frames, the operator will compare all values in all columns, and return a single TRUE if all values in the second data frame are greater than the corresponding values in the first data frame.

Usage

```
x1 %gt% x2
```

Arguments

x1	The first object to compare
x2	The second object to compare

Value

A single TRUE or FALSE value indicating the results of the comparison.

See Also

Other operators: `%eq%()`, `%ge%()`, `%le%()`, `%lt%()`, `%ne%()`, `%p%()`

Examples

```
# Comparing of NULLs and NA
NULL %gt% NULL        # FALSE
NULL %gt% NA          # FALSE
NA %gt% NA            # FALSE
1 %gt% NULL           # FALSE
1 %gt% NA             # FALSE

# Comparing of atomic values
1 %gt% 1              # FALSE
2 %gt% 1              # TRUE
1 %gt% 2              # FALSE
"one" %gt% "one"      # FALSE
1 %gt% "one"          # FALSE
1 %gt% Sys.Date()     # FALSE
Sys.Date() %gt% 1     # TRUE (Sys.Date() is a number)

# Comparing of vectors
v1 <- c(0, 1, 2)
v2 <- c(1, 2, 3)
v3 <- c(2, 3, 4)
v1 %gt% v1            # FALSE
v1 %gt% v2            # FALSE
v2 %gt% v1            # TRUE
v3 %gt% v1            # TRUE

# Comparing of data frames
d1 <- data.frame(A = v1, B = v2)
d2 <- data.frame(A = v2, B = v3)
d1 %gt% d1            # FALSE
d1 %gt% d2            # FALSE
d2 %gt% d1            # TRUE

# Mixing it up
d1 %gt% NULL          # FALSE
v1 %gt% d1            # FALSE
1 %gt% v1             # FALSE
```

`%le%` *Perform less than or equal comparison between two objects*

Description

The goal of the comparison operators is to return a TRUE or FALSE value when any two objects are compared. The operators provides a simple, reliable equality check that allows comparing of NULLs, NA values, and atomic data types without error. This operator performs a less than or equal to comparison.

For data frames, the operator will compare all values in all columns, and return a single TRUE if all values in the second data frame are less than or equal to the corresponding values in the first data frame.

Usage

```
x1 %le% x2
```

Arguments

x1	The first object to compare
x2	The second object to compare

Details

Additional details...

Value

A single TRUE or FALSE value indicating the results of the comparison.

See Also

Other operators: `%eq%()`, `%ge%()`, `%gt%()`, `%lt%()`, `%ne%()`, `%p%()`

Examples

```
# Comparing of NULLs and NA
NULL %le% NULL        # TRUE
NULL %le% NA          # FALSE
NA %le% NA            # TRUE
1 %le% NULL           # FALSE
1 %le% NA             # FALSE

# Comparing of atomic values
1 %le% 1              # TRUE
2 %le% 1              # FALSE
1 %le% 2              # TRUE
"one" %le% "one"      # TRUE
1 %le% "one"          # FALSE
1 %le% Sys.Date()     # TRUE  (Sys.Date() is a number)

# Comparing of vectors
v1 <- c(0, 1, 2)
v2 <- c(1, 2, 3)
v3 <- c(2, 3, 4)
v1 %le% v1            # TRUE
v1 %le% v2            # TRUE
v2 %le% v1            # FALSE
v2 %le% v3            # TRUE

# Comparing of data frames
d1 <- data.frame(A = v1, B = v2)
d2 <- data.frame(A = v2, B = v3)
d1 %le% d1            # TRUE
d1 %le% d2            # TRUE
```

```
d2 %le% d1           # FALSE

# Mixing it up
d1 %le% NULL         # FALSE
v1 %le% d1           # FALSE
1 %le% v1            # FALSE
```

`%lt%` *Perform less than comparison between two objects*

Description

The goal of the comparison operators is to return a TRUE or FALSE value when any two objects are compared. The operators provides a simple, reliable equality check that allows comparing of NULLs, NA values, and atomic data types without error. This operator performs a less than comparison.

For data frames, the operator will compare all values in all columns, and return a single TRUE if all values in the second data frame are less than the corresponding values in the first data frame.

Usage

```
x1 %lt% x2
```

Arguments

x1	The first object to compare
x2	The second object to compare

Value

A single TRUE or FALSE value indicating the results of the comparison.

See Also

Other operators: `%eq%`(), `%ge%`(), `%gt%`(), `%le%`(), `%ne%`(), `%p%`()

Examples

```
# Comparing of NULLs and NA
NULL %lt% NULL       # FALSE
NULL %lt% NA         # FALSE
NA %lt% NA           # FALSE
1 %lt% NULL          # FALSE
1 %lt% NA            # FALSE

# Comparing of atomic values
1 %lt% 1             # FALSE
2 %lt% 1             # FALSE
```

```
1 %lt% 2              # TRUE
"one" %lt% "one"      # FALSE
1 %lt% "one"          # FALSE
1 %lt% Sys.Date()     # TRUE (Sys.Date() is a number)

# Comparing of vectors
v1 <- c(0, 1, 2)
v2 <- c(1, 2, 3)
v3 <- c(2, 3, 4)
v1 %lt% v1            # FALSE
v1 %lt% v2            # TRUE
v2 %lt% v1            # FALSE
v2 %lt% v3            # TRUE

# Comparing of data frames
d1 <- data.frame(A = v1, B = v2)
d2 <- data.frame(A = v2, B = v3)
d1 %lt% d1            # FALSE
d1 %lt% d2            # TRUE
d2 %lt% d1            # FALSE

# Mixing it up
d1 %lt% NULL          # FALSE
v1 %lt% d1            # FALSE
1 %lt% v1             # FALSE
```

`%ne%` *Check for inequality of two objects*

Description

The goal of the `%ne%` operator is to return a TRUE or FALSE value when any two objects are compared. The function is the opposite of the equality operator. It returns a TRUE when the objects are not equal.

This operator also allows comparing of data frames. It will return TRUE if any values in the data frames are not equal, and ignores differences in attributes.

Usage

```
x1 %ne% x2
```

Arguments

`x1`	The first object to compare
`x2`	The second object to compare

Value

A single TRUE or FALSE value depending on whether the objects are not equal.

See Also

Other operators: %eq%(), %ge%(), %gt%(), %le%(), %lt%(), %p%()

Examples

```
# Comparing of NULLs and NA
NULL %ne% NULL        # FALSE
NULL %ne% NA          # TRUE
NA %ne% NA            # FALSE
1 %ne% NULL           # TRUE
1 %ne% NA             # TRUE

# Comparing of atomic values
1 %ne% 1              # FALSE
"one" %ne% "one"      # FALSE
1 %ne% "one"          # TRUE
1 %ne% Sys.Date()     # TRUE

# Comparing of vectors
v1 <- c("A", "B", "C")
v2 <- c("A", "B", "D")
v1 %ne% v1            # FALSE
v1 %ne% v2            # TRUE

# Comparing of data frames
mtcars %ne% mtcars    # FALSE
mtcars %ne% iris      # TRUE
iris %ne% iris[1:50,] # TRUE

# Mixing it up
mtcars %ne% NULL      # TRUE
v1 %ne% NA            # TRUE
1 %ne% v1             # TRUE
```

%p% *An infix operator for* `paste0()`

Description

This function provides an infix operator for the `paste0` function to concatenate strings. The operator will concatenate a vector of one or more values. The functionality is identical to `paste0()`, but more convenient to use in some situations.

Usage

```
x %p% y
```

Arguments

x	A value for the left side of the paste infix operator.
y	A value for the right side of the paste infix operator.

Value

The concatenated or pasted value. No spaces will be inserted in between the values to paste. If a vector of values is supplied, a vector of pasted values will be returned.

See Also

Other operators: `%eq%()`, `%ge%()`, `%gt%()`, `%le%()`, `%lt%()`, `%ne%()`

Examples

```
# Paste together two strings
str <- "Hello" %p% "World"
str
# [1] "HelloWorld"

# Paste together number and string
str <- 100 %p% " Kittens"
str
# [1] "100 Kittens"

# Paste together two vectors
v1 <- c("A", "B", "C")
v2 <- c(1, 2, 3)
str <- v1 %p% v2
str
# [1] "A1" "B2" "C3"
```

Index