Package: bkbase (via r-universe)

October 21, 2026

Title Package Utility Functions

Version 0.0.1

Description A collection of package utility functions written in 'base' R.

URL https://brettklamer.com/work/bkbase, https://bitbucket.org/bklamer/bkbase

License MIT + file LICENSE

LazyData TRUE

Depends R (>= 4.5.0)

Suggests tinytest

Roxygen list(markdown = TRUE)

Encoding UTF-8

Config/roxygen2/version 8.1.1

Repository https://bklamer.r-universe.dev

Date/Publication 2026-10-20 23:01:58 UTC

RemoteUrl https://bitbucket.org/bklamer/bkbase

RemoteRef HEAD

RemoteSha bf25ef8ca696998697f883e9be436c01c7b62934

Contents

%==% . . . 3
%in2% . . . 4
all2 . . . 5
and . . . 7
any2 . . . 8
as_na . . . 10
csw . . . 10
dots . . . 12
fdeparse . . . 13

fformula . . . 14
fmean . . . 15
fmedian . . . 16
fncol . . . 17
fnrow . . . 17
format_value . . . 18
fsd . . . 19
fstr_wrap . . . 20
ftabulate . . . 22
fvar . . . 23
glue2 . . . 24
is_atomic . . . 26
is_const . . . 27
is_date . . . 28
is_date_or_datetime . . . 29
is_datetime . . . 29
is_factor . . . 30
is_formula . . . 31
is_monotonic_decreasing . . . 31
is_monotonic_increasing . . . 32
is_probability . . . 33
is_scalar_character . . . 34
is_scalar_date . . . 35
is_scalar_date_or_datetime . . . 36
is_scalar_datetime . . . 36
is_scalar_double . . . 37
is_scalar_factor . . . 38
is_scalar_integer . . . 39
is_scalar_logical . . . 40
is_scalar_numeric . . . 40
is_scalar_probability . . . 41
is_scalar_whole_number . . . 42
is_scalar_whole_numeric . . . 43
is_strict_decreasing . . . 45
is_strict_increasing . . . 46
is_string . . . 47
is_whole_number . . . 47
is_whole_numeric . . . 49
list_clean . . . 50
list_flatten2 . . . 51
logical_to_integer . . . 52
match.arg2 . . . 53
match.call2 . . . 54
middle . . . 56
or . . . 57
quiet . . . 59
resample . . . 60
reset_rownames . . . 61

rle2 62
rm_na 64
round2 65
rowCumsums 66
rowMaxs 67
rowMaxs2 69
rowMeans2 70
rowSums2 72
seq2 74
str_trim2 75

Index **76**

`%==%` *Equality operator that is NA-aware*

Description

Infix equality operator that resolves missing values instead of propagating them.

Usage

```
x %==% y
```

Arguments

x	(atomic vector) The left-hand side values to compare.
y	(atomic vector) The right-hand side values to compare.

Details

`base::==` returns `NA` whenever either operand is `NA`. `%==%` resolves those positions and never returns `NA`.

For each pair of compared elements the result is

- `TRUE` when both `x` and `y` are `NA`.
- `FALSE` when exactly one of `x` and `y` is `NA`.
- The value of `x == y` when neither is `NA`.

`NaN` is treated the same as `NA`, because `is.na(NaN)` is `TRUE`. `NaN %==% NaN` is `TRUE` and `NaN %==% NA` is `TRUE`.

`x` and `y` are recycled to a common length following standard R rules.

Value

A logical vector of length `max(length(x), length(y))`. The result never contains `NA`.

See Also

base::Comparison, %in2%

Examples

```
#--------------------------------------------------------------------------
# %==% examples
#--------------------------------------------------------------------------
library(bkbase)

# both NA gives TRUE, exactly one NA gives FALSE
1:3 %==% c(1:2, NA)   # TRUE TRUE FALSE
NA %==% NA            # TRUE

# NaN is treated the same as NA
1:3 %==% c(1:2, NaN)  # TRUE TRUE FALSE
NaN %==% NaN          # TRUE
NaN %==% NA           # TRUE

# base `==` propagates NA instead
1:3 == c(1:2, NA)     # TRUE TRUE NA
1:3 == c(1:2, NaN)    # TRUE TRUE NA
```

%in2% *Value matching that propagates NA*

Description

Infix operator for matching that forces NA in x to propagate in results.

Usage

```
x %in2% table
```

Arguments

x	(atomic vector) Values to match in table.
table	(atomic vector) The reference set against which membership is evaluated.

Details

base::%in% treats NA in x as an ordinary value. NA %in% c(1, 2, 3) is FALSE and NA %in% c(NA) is TRUE. You may not want this behavior because membership of a missing value is itself unknown. %in2% instead returns NA for every position where x is NA, regardless of whether table contains NA. For all non-missing elements of x, the result equals base::%in%.

NaN is treated the same as NA and also propagates to NA.

Value

A logical vector with length `length(x)`.

See Also

base::%in%, `base::match()`, %==%

Examples

```
#------------------------------------------------------------------------
# %in2% examples
#------------------------------------------------------------------------
library(bkbase)

# base::`%in%` behavior
NA %in% c(1, 2, 3)            # FALSE
NA %in% c(NA)                 # TRUE

# bkbase::`%in2%` (always propagate NA from x)
NA %in2% c(1, 2, 3)           # NA
NA %in2% c(NA)                # NA

# Mixed vectors
c(NA, 1) %in%  c(1, 2, 3)     # FALSE TRUE
c(NA, 1) %in2% c(1, 2, 3)     # NA    TRUE

c(NA, 1) %in%  c(NA, 2, 3)    # TRUE  FALSE
c(NA, 1) %in2% c(NA, 2, 3)    # NA    FALSE
```

---

all2 *Are all values TRUE*

Description

Returns `TRUE` if all values in a logical vector are `TRUE`. A stricter, surprise-free version of `base::all()`.

Usage

```
all2(x, na.rm = FALSE)
```

Arguments

`x`	(`logical()`) The logical vector to be checked if all elements are `TRUE`. Any non-logical input returns `FALSE`.
`na.rm`	(`logical(1)`; default `FALSE`) Whether or not to remove `NA`s from `x` before evaluating.

Details

Suppose you have a numeric vector `y` and want to validate all values are greater than zero using `base::all(y > 0)`. If `y` contains `NA`s and all nonmissing elements are greater than `0`, the result will be `NA`. You then might use `base::all(y > 0, na.rm = TRUE)`, however, this assumes `NA`s are ok to exclude. In this scenario, `all2(y > 0)` would return `FALSE` instead of `NA`.

`all(NULL)` and `all(logical(0))` both return `TRUE`. `all2()` returns `FALSE` for both cases.

`all(NA, na.rm = TRUE)` returns `TRUE`. `all2(NA, na.rm = TRUE)` returns `FALSE`.

Value

`logical(1)`.

See Also

`base::all()`, `any2()`

Examples

```
#------------------------------------------------------------------------------
# all2() examples
#------------------------------------------------------------------------------
library(bkbase)

x <- c(1, 2, 3, NA)

all2(x > 0)
all(x > 0)

all2(x > 0, na.rm = TRUE)
all(x > 0, na.rm = TRUE)

all2(NULL)
all(NULL)

all2(logical(0))
all(logical(0))

all2(numeric(0))
all(numeric(0))

all2(NA, na.rm = TRUE)
all(NA, na.rm = TRUE)

all2(NA, na.rm = FALSE)
all(NA, na.rm = FALSE)
```

and *Logical AND with NA tolerance*

Description

Compute a position-wise logical AND across multiple logical inputs, allowing up to a specified fraction of missing values before returning NA.

Usage

```
and(..., .na_allowed = 0.2)
```

Arguments

`...`	(`logical()`, logical matrix, or data.frame) Logical vectors of equal length, or a single logical matrix/data.frame. Matrix/data.frame inputs must have at least one column. Inputs are coerced to logical via `as.logical`, preserving NA.
`.na_allowed`	(`numeric(1)`; default `0.2`; range `[0, 1]`) Maximum allowed fraction of NA per position before the result is set to NA when no input is FALSE.

Details

This is a tolerance-aware variant of logical conjunction over multiple inputs (e.g., `x & y & z`). Each output position corresponds to one vector index, or to one row when a matrix or data.frame is supplied. The result is FALSE when any input at that position is FALSE. Otherwise, the result is NA when the fraction of missing values exceeds `.na_allowed`. If the missing fraction is within tolerance, missing values are treated as TRUE and the result is TRUE.

For each position (row) i across p inputs, let m_i be the number of NAs and f_i be the number of FALSEs. Define the missing fraction $r_i = m_i/p$. Then:

$$\text{and}_i = \begin{cases} \text{NA}, & \text{if } r_i > \text{.na_allowed and } f_i = 0, \\ \text{TRUE}, & \text{if } r_i \leq \text{.na_allowed and } f_i = 0, \\ \text{FALSE}, & \text{if } f_i \geq 1. \end{cases}$$

Notes:

1. When `.na_allowed = 0`, behavior matches strict AND: NAs propagate only when no input is FALSE.
2. When .na_allowed $= 1$, missingness never forces NA. NAs are treated as TRUE, so the result is FALSE iff any FALSE is present, otherwise TRUE.

Value

A logical vector with values TRUE, FALSE, or NA per the rule in Details. The length equals the length of each input vector, or the number of rows of a matrix or data.frame.

See Also

base::Logic

Examples

```
#--------------------------------------------------------------------------
# and() examples
#--------------------------------------------------------------------------
library(bkbase)

x <- c(TRUE, TRUE, TRUE, FALSE, TRUE)
y <- c(TRUE, TRUE, NA, TRUE, NA)
z <- c(TRUE, NA, TRUE, TRUE, NA)

# Base R strict AND
x & y & z

# NA tolerance at 20%
and(x, y, z, .na_allowed = 0.2)

# NA tolerance at 40%
and(x, y, z, .na_allowed = 0.4)

# Single data.frame input
df <- data.frame(x = x, y = y, z = z)
and(df, .na_allowed = 0.4)

# FALSE dominates even when NA is present
and(FALSE, NA, .na_allowed = 0)
```

---

any2 *Are any nonmissing values TRUE*

Description

Returns TRUE if any nonmissing values in a logical vector are TRUE.

Usage

```
any2(x)
```

Arguments

x	(logical()) The logical vector to be checked if any element is TRUE.

Details

Missing values are excluded before evaluation. Compared with `base::any()`, `any2()` has stricter behavior:

- `any2()` never returns `NA`.
- `any2()` returns `FALSE` for non-logical input.
- `any2()` returns `FALSE` for `logical(0)` and `NULL`.

Value

`logical(1)`.

See Also

`base::any()`, `all2()`

Examples

```
#------------------------------------------------------------------------------
# any2() examples
#------------------------------------------------------------------------------
library(bkbase)

x <- c(1, 2, 3, NA)
any2(x < 0)
any(x < 0, na.rm = TRUE)
any(x < 0, na.rm = FALSE)

any2(NA)
any(NA)

any2(1L)
any(1L)

any2(c(TRUE, NA))
any2(c(FALSE, NA))
any2(c(NA, NA))

any2(logical(0))
any(logical(0))

any2(NULL)
any(NULL)
```

as_na *Coerce to NA*

Description

Coerces input to missing values (NA) of the same type.

Usage

```
as_na(x)
```

Arguments

x	(object) The object to coerce to NAs. Raw vectors, functions, and environments give an error.

Value

Object of same class as x. Each column of a data.frame keeps its type. NULL returns NULL.

Examples

```
#--------------------------------------------------------------------------
# as_na() examples
#--------------------------------------------------------------------------
library(bkbase)

as_na(data.frame(a = 1:3))
as_na(matrix(1:3))
as_na(list(a = 1:3))
as_na(1:3)
as_na(letters[1:3])
as_na(factor(letters[1:3]))
as_na(as.Date("2020-01-01"))
```

csw *Comma-separated words*

Description

Join the elements of an atomic vector into one readable string, such as "'a', 'b', and 'c'". Use csw() to list values in messages, warnings, and errors.

Usage

```
csw(
  x,
  n = Inf,
  quote = !(is.numeric(x) || is.logical(x) || is.complex(x)),
  conjunction = "and",
  tail = "count",
  period = FALSE,
  width = Inf
)
```

Arguments

x	(atomic vector or NULL) Values to join. The function does not accept a matrix or an array.
n	(numeric(1); default Inf; range [1, Inf]) Maximum number of elements of x to show. Must be a whole number. Inf shows all elements.
quote	(logical(1); default !(is.numeric(x) || is.logical(x) || is.complex(x))) Whether to put quotes around each element that is not NA. By default, numeric, logical, and complex values get no quotes. Values of other types get quotes, for example character, factor, Date, and date-time values. An element that contains ' gets double quotes, and each " in that element gets a backslash before it. The remaining elements get single quotes.
conjunction	(character(1) or NULL; default "and") Word to put before the last element, such as "and", "or", or "and/or". If NULL, the function puts only commas between elements.
tail	(character(1); default "count") One of "ellipsis", "count", or "none". Text to add after the last element shown when x has more than n elements. "ellipsis" adds "...". "count" adds the number of elements not shown, as in "and 24 more". The "count" tail uses conjunction, or "and" if conjunction is NULL. "none" adds no text.
period	(logical(1); default FALSE) Whether to end the string with a period. Has no effect when the function adds the "ellipsis" tail.
width	(numeric(1); default Inf; range [1, Inf]) Maximum number of columns on each line. Must be a whole number. Inf turns off line wrapping.

Details

The function converts x to character with as.character(). Round or format numbers before you call csw() if you need a fixed number of digits. Numbers formatted as character, for example by format() or round2(), get quotes unless you set quote = FALSE.

When conjunction is a string, the function puts it before the last element. Two elements get no comma, as in "'a' and 'b'". Three or more elements get a comma before the conjunction, as in "'a', 'b', and 'c'".

When x has more than n elements, the function shows the first n elements and then adds the text that tail selects. Because the "count" tail is the last element of the list, it gets the conjunction and the period, as in "'a', 'b', and 24 more.". The "ellipsis" tail turns off conjunction and period.

By default, the function does not wrap lines. When the output is part of a longer message, wrap the full message instead, for example with fstr_wrap(). When width is finite, the function puts line breaks only between elements. It does not split an element or change the characters in it. An element that is wider than width gets a line of its own. The period counts toward width.

Because NA elements get no quotes, a missing value is different from the string "NA". The values NaN, Inf, and -Inf are not missing, and quote applies to them as to other values. NULL and zero-length x return character(0).

Value

character(1), or character(0) when x has length zero. Line breaks are "\n".

See Also

base::toString()

Examples

```
#--------------------------------------------------------------------------
# csw() example
#--------------------------------------------------------------------------
library(bkbase)

# Strings get quotes, and "and" goes before the last element.
csw(c("mpg", "cyl", "disp"))

# Numbers get no quotes. Show the first 3 values and count the remainder.
csw(c(2, 3, 5, 7, 11, 13), n = 3)

# Put the list in a message, then wrap the full message.
msg <- paste0(
  "Argument `x` must be one of ",
  csw(month.name, conjunction = "or"),
  "."
)
message(fstr_wrap(msg, width = 60))
```

---

dots *Capture unevaluated dots*

Description

Capture the expressions supplied in ... without evaluating them. Use dots() when the expressions should remain as language objects. Use dots_char() when the expressions should be returned as text.

Usage

```
dots(..., .names = TRUE, .duplicate_names = "warn")

dots_char(..., .names = TRUE, .duplicate_names = "warn")
```

Arguments

`...`	(expressions) Expressions to capture.
`.names`	(`logical(1)`; default `TRUE`) Whether to fill missing names with deparsed expressions. Explicit names are always preserved.
`.duplicate_names`	(`character(1)`; default `"warn"`) One of `"ignore"`, `"warn"`, or `"stop"`. Action to take when duplicate names are found. Values are matched exactly. Duplicate names are checked after missing names have been filled. Because `.names = TRUE` gives repeated unnamed expressions the same name, the duplicate-name policy applies to them. The duplicate-name policy is applied only if `.names = TRUE`.

Value

An object with one element for each expression in `...`. `dots()` returns a list of unevaluated expressions. `dots_char()` returns a character vector of deparsed expressions. If `.names = TRUE`, missing names are filled with the deparsed expression. If `.names = FALSE`, missing names are left missing. Empty `...` returns `list()` from `dots()` and `character(0)` from `dots_char()`.

Examples

```
#--------------------------------------------------------------------------
# dots() examples
#--------------------------------------------------------------------------
library(bkbase)

a <- 1:5

dots(a, 6:10, x = 3)
dots_char(a, 6:10, x = 3)
dots(a, 6:10, x = 3, .names = FALSE)
```

`fdeparse` *Fast deparse*

Description

A convenience function between `base::deparse()` and `base::deparse1()`.

Usage

```
fdeparse(expr, width.cutoff = 500L, ...)
```

Arguments

`expr`	(expression) Any R expression.
`width.cutoff`	(`integer(1)`; default `500L`; range `[20, 500]`) The cutoff at which line-breaking occurs. See `base::deparse1()` if you need more.
`...`	Further arguments passed to `base::deparse()`.

Value

`character()`.

Examples

```
#--------------------------------------------------------------------------
# fdeparse() example
#--------------------------------------------------------------------------
library(bkbase)

fdeparse(y ~ x)
```

`fformula` *Fast formula*

Description

Create a formula object from known-good string input. A fast, unsafe alternative to `stats::formula()`.

Usage

```
fformula(x, env = parent.frame())
```

Arguments

`x`	(`character(1)`) The string to be converted into a formula.
`env`	(environment; default `parent.frame()`) An environment to associate with the result.

Value

A formula object whose environment is set to `env`.

See Also

stats::formula()

Examples

```
#-------------------------------------------------------------------------
# fformula() example
#-------------------------------------------------------------------------
library(bkbase)

fformula("y ~ x")
```

---

fmean *Fast mean*

Description

Calculates the arithmetic mean after excluding missing values. A fast, unsafe alternative to base::mean(x, na.rm = TRUE) for numeric vectors.

Usage

```
fmean(x)
```

Arguments

x	(numeric()) The numeric vector used to calculate the mean.

Value

numeric(1).

See Also

base::mean()

Examples

```
#-------------------------------------------------------------------------
# fmean() example
#-------------------------------------------------------------------------
library(bkbase)

fmean(1:3)
fmean(c(1:3, NA))
fmean(NA)
fmean(numeric(0))
```

fmedian *Fast median*

Description

Calculates the median after excluding missing values. A fast, unsafe alternative to `stats::median(x, na.rm = TRUE)`.

Usage

```
fmedian(x, is_sorted = FALSE)
```

Arguments

`x`	(`numeric()`) The numeric vector used to calculate the median.
`is_sorted`	(`logical(1)`; default `FALSE`) Whether or not `x` is already sorted in ascending order. Only set `is_sorted = TRUE` when `x` is guaranteed to already be sorted.

Value

`numeric(1)`.

See Also

`stats::median()`

Examples

```
#--------------------------------------------------------------------------
# fmedian() example
#--------------------------------------------------------------------------
library(bkbase)

fmedian(1:3)
fmedian(c(1:3, NA))
fmedian(NA)
fmedian(numeric(0))
```

fncol *Fast number of columns in a data frame*

Description

Get the number of columns in a data frame. A fast, unsafe alternative to `base::ncol()`.

Usage

```
fncol(x)
```

Arguments

x	(data.frame) The data frame to get the number of columns.

Value

`integer(1)`.

See Also

`base::ncol()`

Examples

```
#--------------------------------------------------------------------------
# fncol() examples
#--------------------------------------------------------------------------
library(bkbase)

x <- data.frame(a = 1)
fncol(x)
```

fnrow *Fast number of rows in a data frame*

Description

Get the number of rows in a data frame. A fast, unsafe alternative to `base::nrow()`.

Usage

```
fnrow(x)
```

Arguments

x	(data.frame) The data frame to get the number of rows.

Value

`integer(1)`.

See Also

`base::nrow()`

Examples

```
#--------------------------------------------------------------------------
# fnrow() examples
#--------------------------------------------------------------------------
library(bkbase)

x <- data.frame(a = 1)
fnrow(x)
```

---

format_value *Argument value as a phrase for a message*

Description

Write an R object as a short phrase for an error, a warning, or a message. Use `format_value()` to state the value that an argument received, as in `"Argument `n` is 2.5, but must be a whole number."`.

Usage

```
format_value(x)
```

Arguments

x	(any R object) Value to write.

Details

The function writes a plain atomic vector of length 5 or less as R code, for example `3L`, `"bogus"`, `NA`, or `c(1, 5)`. A plain vector has no class and no dimensions. The function writes a longer plain vector by its length, as in `"a vector of length 10"`. It writes any other object by the first element of its class, as in `an object of class "factor"`. A matrix, a factor, a Date, a list, and a function are not plain vectors.

The R code keeps the type and the names of `x`. An integer gets the suffix `L`, and a typed missing value keeps its type, as in `NA_real_`. Other attributes appear in a call to `structure()`.

`deparse()` writes 15 significant digits by default. With 15 digits, `2 + 4e-16` gives `"2"`, and a message that rejects a value that is not whole then shows a whole number. When a double element of `x` needs more than 15 digits, the function writes all elements with 17 significant digits, as in `2.0000000000000004`. Other doubles keep the short form, such as `0.3`.

`NULL` gives `"NULL"`. Zero-length input gives R code such as `"character(0)"`. `NA`, `NaN`, `Inf`, and `-Inf` give the R code for the value.

Value

`character(1)`.

See Also

`match.arg2()`, `csw()`, `base::deparse1()`

Examples

```
#--------------------------------------------------------------------------
# format_value() example
#--------------------------------------------------------------------------
library(bkbase)

# State the received value in an error message.
n <- 2.5
msg <- paste0(
  "Argument `n` is ",
  format_value(n),
  ", but must be a whole number."
)
message(msg)

# Short vectors give R code. Long vectors and other objects give a phrase.
format_value(c(1, 5))
format_value(1:10)
format_value(factor("a"))

# A double that 15 digits do not give exactly gets 17 digits.
format_value(2 + 4e-16)
```

---

fsd *Fast standard deviation*

Description

Calculates the standard deviation after excluding missing values. A fast, unsafe alternative to `stats::sd(x, na.rm = TRUE)`.

Usage

```
fsd(x)
```

Arguments

x	(numeric()) The numeric vector to calculate the standard deviation.

Value

numeric(1).

See Also

stats::sd()

Examples

```
#--------------------------------------------------------------------------
# fsd() example
#--------------------------------------------------------------------------
library(bkbase)

fsd(1:3)
fsd(c(1:3, NA))
```

`fstr_wrap` *Fast string wrapping*

Description

Wrap each element of a character vector to a target width. CRLF/CR are normalized to LF when present and existing newlines may be preserved. Internal runs of whitespace are collapsed to single spaces between words.

Usage

```
fstr_wrap(x, width = 72L, indent = 0L, exdent = 0L, keep_newlines = FALSE)

fstrwrap(x, width = 72L, indent = 0L, exdent = 0L, keep_newlines = FALSE)
```

Arguments

x	(character()) The text to wrap.
width	(numeric(1); default 72L; range [1, .Machine$integer.max]) Target column width in characters for wrapping, including the cost of any indent or exdent.
indent	(numeric(1); default 0L; range [0, .Machine$integer.max]) Number of leading spaces to prepend to the first line of each wrapped element. If the element does not wrap, the indent is still applied.
exdent	(numeric(1); default 0L; range [0, .Machine$integer.max]) Number of leading spaces to prepend to each line after the first. Has no effect if the element does not wrap.
keep_newlines	(logical(1); default FALSE) FALSE (default) collapses all whitespace including newlines into spaces before wrapping. TRUE splits the input at "\n" and wraps each line independently.

Details

Words are defined by splitting on the regular expression \\s+, and empty tokens are dropped. NA elements are returned as NA_character_. When keep_newlines = TRUE, input is split into lines at "\n", each line is word-wrapped independently, and empty lines are preserved as empty strings.

A greedy algorithm buffers contiguous words and flushes them to the output when adding the next word would exceed the target width. If the entire line fits within the width when accounting for indent, the line is returned unwrapped. Long words are not broken; they are placed on a line even if they exceed the width. Line length is computed as the sum of the prefix and buffered words plus inter-word spaces.

Names on the input vector are preserved on the output. The result for each element is a single string with any inserted wraps represented by "\n".

Value

Character vector of the same length as x.

See Also

base::strwrap()

Examples

```
#-------------------------------------------------------------------------
# fstr_wrap()
#-------------------------------------------------------------------------
library(bkbase)

# Basic wrapping
txt <- "This is a simple example sentence that will be wrapped."
cat(fstr_wrap(txt, width = 20))
```

```
# Indent for the first line and exdent for continuation lines
txt <- "Indented wrapping across multiple lines of text."
cat(fstr_wrap(txt, width = 20, indent = 2, exdent = 4))

# Existing newlines are preserved and each line is wrapped independently
txt <- "First line that is somewhat long.\nSecond line that is also long."
cat(fstr_wrap(txt, width = 25, exdent = 2))

txt <- "This is a sentence.
Next is the second sentence.
Finally, the third sentence."
cat(fstr_wrap(txt, keep_newlines = FALSE))
cat(fstr_wrap(txt, keep_newlines = TRUE))

# Names preserved and NA propagated
txt <- c(a = "short line", b = NA_character_)
fstr_wrap(txt, width = 25)
```

`ftabulate` *Fast tabulate*

Description

Calculates the number of times each unique element occurs after excluding missing values. A fast, generalized alternative to `base::tabulate()`.

Usage

```
ftabulate(x, is_sorted = FALSE)
```

Arguments

`x`	(atomic vector) The atomic vector for tabulation.
`is_sorted`	(`logical(1)`; default `FALSE`) Whether or not `x` is already sorted. Only set `is_sorted = TRUE` when `x` is guaranteed to already be sorted.

Details

`base::tabulate()` only counts positive integers. `ftabulate()` will count any nonmissing element type.

Value

`integer()`.

See Also

base::tabulate(), base::table()

Examples

```
#-------------------------------------------------------------------------
# ftabulate() example
#-------------------------------------------------------------------------
library(bkbase)

x <- sample(1:10, size = 100, replace = TRUE)
ftabulate(x)
tabulate(x)
table(x)
```

---

fvar *Fast variance*

Description

Calculates the variance after excluding missing values. A fast, unsafe alternative to `stats::var(x, na.rm = TRUE)`.

Usage

```
fvar(x)
```

Arguments

x	(numeric()) The numeric vector to calculate the variance.

Value

numeric(1).

See Also

stats::var()

Examples

```
#-------------------------------------------------------------------------
# fvar() example
#-------------------------------------------------------------------------
library(bkbase)

fvar(1:3)
```

```
fvar(c(1:3, NA))
```

`glue2` *String interpolation*

Description

Expressions enclosed by delimiters will be evaluated or extracted and inserted into the string.

Usage

```
glue2(
  ...,
  .env = parent.frame(),
  .open = "{",
  .close = "}",
  .sep = " ",
  .eval = TRUE
)
```

Arguments

`...`	(`character()`) One or more strings or character vectors to format. All inputs are coerced with `as.character()` and joined with `.sep` into a single template string.
`.env`	(environment, list, or data.frame; default `parent.frame()`) The object used to evaluate expressions or get named values.
`.open, .close`	(`character(1)`; default `"{"` and `"}"`) The opening and closing delimiters of the expressions. Each delimiter must be one character.
`.sep`	(`character(1)`; default `" "`) The separator used for concatenating strings in `...`
`.eval`	(`logical(1)`; default `TRUE`) If `TRUE` (default), the expressions in `...` are treated as R code and evaluated. If `FALSE`, expressions in `...` will be treated as variable names in `.env` (this is faster than evaluation).

Details

An expression can span more than one line. The function converts each value with `base::as.character()`. If a value has length zero, the result is `character(0)`, as in `glue::glue()`. Otherwise, the function recycles the values to the length of the longest value. The function stops with an error when the length of a value does not divide the length of the longest value.

Value

`character()`. Its length is the length of the longest value, or 1 if the template has no expressions. It is `character(0)` if a value has length zero.

Author(s)

This function was inspired by code by Posit (MIT license) and Mike Cheng (MIT license)

References

https://github.com/tidyverse/glue/blob/main/R/glue.R

https://github.com/coolbutuseless/gluestick/blob/main/R/gluestick.R

See Also

`glue::glue()`

Examples

```
#--------------------------------------------------------------------------
# glue2() examples
#--------------------------------------------------------------------------
library(bkbase)

# Extract value from data object
data <- list(name = "Fred")
glue2("Hello {name}", .env = data)

# Extract value from parent.frame()
name <- "Fred"
glue2("Hello {name}")

# A vector is returned if expressions contain vectors
glue2("Number {1:2}")

# Evaluate expression
year_current <- as.numeric(format(Sys.Date(),'%Y'))
year_past <- 1970
glue2(
  "Today is {Sys.Date()}",
  "and it has been {year_current - year_past} years since 1970.",
  .eval = TRUE
)
```

`is_atomic` *Is object an atomic vector*

Description

Returns `TRUE` when `x` is an atomic vector of type integer, double, character, logical, complex, or raw and has no `dim` attribute.

Usage

```
is_atomic(x)
```

Arguments

`x`	(any object) The object to be tested.

Details

This is a stricter alternative to `base::is.atomic()`.

Only integer, double, character, logical, complex, and raw vectors return `TRUE`.

A `dim` attribute makes the result `FALSE`. Matrices and arrays therefore return `FALSE` even when their underlying type is allowed.

The test uses `base::typeof()`, so classed objects built on an allowed type return `TRUE`. This includes `Date`, `POSIXct`, and `factor` objects. Objects stored as lists return `FALSE`, including `data.frame` and `POSIXlt`.

`NULL` returns `FALSE`. Zero-length vectors of an allowed type return `TRUE`.

Value

`logical(1)`.

See Also

`base::is.atomic()`

Examples

```
#--------------------------------------------------------------------------
# is_atomic() examples
#--------------------------------------------------------------------------
library(bkbase)

is_atomic(1)
is_atomic(FALSE)
is_atomic(c("a", "b"))
is_atomic(integer(0))
```

```
is_atomic(raw(1))

# FALSE because of a dim attribute
is_atomic(matrix(1))

# FALSE because the object is stored as a list
is_atomic(data.frame(1))
```

`is_const` *Is vector constant*

Description

Returns `TRUE` if `x` is a one-dimensional atomic vector where every element is equal to the first element. Zero-length and length-one vectors are constant.

Usage

```
is_const(x)
```

Arguments

`x`	(atomic vector) The vector to be tested.

Details

`NA` and `NaN` are treated as values rather than unknown comparison results. This means `c(NA, NA)` and `c(NaN, NaN)` are constant, while `c(NA, NaN)` and `c(1, NA)` are not constant.

Non-atomic objects and objects with dimensions return `FALSE`.

Value

`logical(1)`.

See Also

`base::match()`, `base::anyNA()`

Examples

```
#--------------------------------------------------------------------------
# is_const() examples
#--------------------------------------------------------------------------
library(bkbase)

is_const(c(1, 1, 1))
is_const(c(1, 2, 1))
```

```
is_const(c(NA, NA))
is_const(c(NaN, NaN))
is_const(c(NA, NaN))

is_const(integer(0))
is_const(NULL)

is_const(matrix(1, nrow = 1))
is_const(list(1, 1))
```

`is_date` *Is object class date*

Description

Returns `TRUE` if `x` is a date (class `Date`).

Usage

```
is_date(x)
```

Arguments

`x`	(any object) The object to be tested.

Value

`logical(1)`.

Examples

```
#----------------------------------------------------------------------------
# is_date() examples
#----------------------------------------------------------------------------
library(bkbase)

d <- as.Date("2019-01-01")
dt <- as.POSIXct("2019-01-01")
is_date(d)
is_date(dt)
```

is_date_or_datetime *Is object class date or datetime*

Description

Returns TRUE if x is a date (class Date) or datetime (class POSIXct).

Usage

```
is_date_or_datetime(x)
```

Arguments

x	(any object) The object to be tested.

Value

logical(1).

Examples

```
#---------------------------------------------------------------------------
# is_date_or_datetime() examples
#---------------------------------------------------------------------------
library(bkbase)

d <- as.Date("2019-01-01")
dt <- as.POSIXct("2019-01-01")
is_date_or_datetime(d)
is_date_or_datetime(dt)
```

is_datetime *Is object class datetime*

Description

Returns TRUE if x is a datetime (class POSIXct).

Usage

```
is_datetime(x)
```

Arguments

x	(any object) The object to be tested.

Value

`logical(1)`.

Examples

```
#------------------------------------------------------------------------
# is_datetime() examples
#------------------------------------------------------------------------
library(bkbase)

dt <- as.POSIXct("2019-01-01")
d <- as.Date("2019-01-01")
is_datetime(dt)
is_datetime(d)
```

---

is_factor *Is object a factor vector*

Description

Returns `TRUE` if `x` inherits from class `"factor"`.

Usage

```
is_factor(x)
```

Arguments

`x`	(any object) The object to be tested.

Value

`logical(1)`.

Examples

```
#------------------------------------------------------------------------
# is_factor() examples
#------------------------------------------------------------------------
library(bkbase)

is_factor(factor(c("a", "b")))
is_factor(factor("a", ordered = TRUE))
is_factor(factor())
is_factor("a")
is_factor(NULL)
```

is_formula *Is object a formula*

Description

Returns TRUE if x inherits from class "formula".

Usage

```
is_formula(x)
```

Arguments

x	(any object) The object to be tested.

Value

logical(1).

Examples

```
#----------------------------------------------------------------------------
# is_formula() examples
#----------------------------------------------------------------------------
library(bkbase)

is_formula(1:4)
is_formula("y ~ x")
is_formula(y ~ x)
```

is_monotonic_decreasing *Is vector monotonically decreasing*

Description

Returns TRUE if all values in x are monotonically decreasing. Returns FALSE for non-numeric input, NULL, and zero-length vectors.

Usage

```
is_monotonic_decreasing(x, na.rm = FALSE)
```

Arguments

x	(numeric()) The vector to be tested.
na.rm	(logical(1); default FALSE) Whether or not to remove/ignore missing values from x.

Value

logical(1).

Examples

```
#--------------------------------------------------------------------------
# is_monotonic_decreasing() examples
#--------------------------------------------------------------------------
library(bkbase)

is_monotonic_decreasing(10:-10)
is_monotonic_decreasing(c(3.5, 2, 2, 1, 1, 0))
is_monotonic_decreasing(1)

is_monotonic_decreasing(1:3)

is_monotonic_decreasing(c(3:1, NA))
is_monotonic_decreasing(c(3:1, NA), na.rm = TRUE)

is_monotonic_decreasing(NA_real_)
is_monotonic_decreasing(NA_real_, na.rm = TRUE)

is_monotonic_decreasing(numeric(0L))
is_monotonic_decreasing(NULL)
```

---

is_monotonic_increasing

Is vector monotonically increasing

Description

Returns TRUE if all values in x are monotonically increasing. Returns FALSE for non-numeric input, NULL, and zero-length vectors.

Usage

```
is_monotonic_increasing(x, na.rm = FALSE)
```

Arguments

x	(numeric()) The vector to be tested.
na.rm	(logical(1); default FALSE) Whether or not to remove/ignore missing values from x.

Value

logical(1).

Examples

```
#--------------------------------------------------------------------------
# is_monotonic_increasing() examples
#--------------------------------------------------------------------------
library(bkbase)

is_monotonic_increasing(-10:10)
is_monotonic_increasing(c(0, 1, 1, 2, 2, 3.5))
is_monotonic_increasing(1)

is_monotonic_increasing(3:1)

is_monotonic_increasing(c(1:3, NA))
is_monotonic_increasing(c(1:3, NA), na.rm = TRUE)

is_monotonic_increasing(NA_real_)
is_monotonic_increasing(NA_real_, na.rm = TRUE)

is_monotonic_increasing(numeric(0L))
is_monotonic_increasing(NULL)
```

---

is_probability *Is object a vector of probabilities*

Description

Returns TRUE if all values in x are probabilities. A probability is a nonmissing numeric value bounded by [0, 1].

Usage

```
is_probability(x)
```

Arguments

x	(any object) The object to be tested.

Value

logical(1).

Examples

```
#--------------------------------------------------------------------------
# is_probability() examples
#--------------------------------------------------------------------------
library(bkbase)

# TRUE
is_probability(c(0, 0.5, 1))
is_probability(1L)

# FALSE
is_probability(1.1)
is_probability(NA_real_)
is_probability(numeric(0L))
is_probability(NULL)
```

---

is_scalar_character *Is object a scalar character vector*

Description

Returns TRUE if x is a scalar character (character vector of length 1).

Usage

```
is_scalar_character(x)
```

Arguments

x	(any object) The object to be tested.

Value

logical(1).

Examples

```
#--------------------------------------------------------------------------
# is_scalar_character() examples
#--------------------------------------------------------------------------
library(bkbase)

# TRUE
```

```
is_scalar_character("a")
is_scalar_character(NA_character_)

# FALSE
is_scalar_character(c("a", "b"))
is_scalar_character(character(0L))
is_scalar_character(1L)
is_scalar_character(NULL)
```

`is_scalar_date` *Is object a scalar date*

Description

Returns `TRUE` if `x` is a scalar date (class `Date` of length 1).

Usage

```
is_scalar_date(x)
```

Arguments

`x`	(any object) The object to be tested.

Value

`logical(1)`.

Examples

```
#--------------------------------------------------------------------------
# is_scalar_date() examples
#--------------------------------------------------------------------------
library(bkbase)

is_scalar_date(as.Date("2024-01-01"))
is_scalar_date(as.POSIXct("2024-01-01"))

is_scalar_date(as.Date(c("2024-01-01", "2024-01-02")))
is_scalar_date(as.POSIXct(c("2024-01-01", "2024-01-02")))
```

is_scalar_date_or_datetime *Is object a scalar date or datetime*

Description

Returns `TRUE` if `x` is a scalar date or datetime (class `Date` or `POSIXct` of length 1).

Usage

```
is_scalar_date_or_datetime(x)
```

Arguments

`x`	(any object) The object to be tested.

Value

`logical(1)`.

Examples

```
#------------------------------------------------------------------------
# is_scalar_date_or_datetime() examples
#------------------------------------------------------------------------
library(bkbase)

is_scalar_date_or_datetime(as.Date("2024-01-01"))
is_scalar_date_or_datetime(as.POSIXct("2024-01-01"))

is_scalar_date_or_datetime(as.Date(c("2024-01-01", "2024-01-02")))
is_scalar_date_or_datetime(as.POSIXct(c("2024-01-01", "2024-01-02")))
```

is_scalar_datetime *Is object a scalar datetime*

Description

Returns `TRUE` if `x` is a scalar datetime (class `POSIXct` of length 1).

Usage

```
is_scalar_datetime(x)
```

Arguments

x	(any object) The object to be tested.

Value

`logical(1)`.

Examples

```
#-------------------------------------------------------------------------
# is_scalar_datetime() examples
#-------------------------------------------------------------------------
library(bkbase)

is_scalar_datetime(as.Date("2024-01-01"))
is_scalar_datetime(as.POSIXct("2024-01-01"))

is_scalar_datetime(as.Date(c("2024-01-01", "2024-01-02")))
is_scalar_datetime(as.POSIXct(c("2024-01-01", "2024-01-02")))
```

---

is_scalar_double *Is object a scalar double-precision vector*

Description

Returns `TRUE` if `x` is a scalar double (double-precision vector of length 1).

Usage

```
is_scalar_double(x)
```

Arguments

x	(any object) The object to be tested.

Value

`logical(1)`.

Examples

```
#--------------------------------------------------------------------------
# is_scalar_double() examples
#--------------------------------------------------------------------------
library(bkbase)

# TRUE
is_scalar_double(0)
is_scalar_double(1.5)
is_scalar_double(NA_real_)

# FALSE
is_scalar_double(1L)
is_scalar_double(c(1, 2))
is_scalar_double(numeric(0L))
is_scalar_double(NULL)
```

---

is_scalar_factor *Is object a scalar factor vector*

Description

Returns `TRUE` if `x` is a scalar factor (factor vector of length 1).

Usage

```
is_scalar_factor(x)
```

Arguments

`x`	(any object) The object to be tested.

Value

`logical(1)`.

Examples

```
#--------------------------------------------------------------------------
# is_scalar_factor() examples
#--------------------------------------------------------------------------
library(bkbase)

# TRUE
is_scalar_factor(factor("a"))
is_scalar_factor(factor("a", ordered = TRUE))
is_scalar_factor(factor(NA))
```

```
# FALSE
is_scalar_factor(factor(c("a", "b")))
is_scalar_factor(factor())
is_scalar_factor("a")
is_scalar_factor(NULL)
```

`is_scalar_integer` *Is object a scalar integer vector*

Description

Returns `TRUE` if `x` is a scalar integer (integer vector of length 1).

Usage

```
is_scalar_integer(x)
```

Arguments

`x`	(any object) The object to be tested.

Value

`logical(1)`.

Examples

```
#--------------------------------------------------------------------------
# is_scalar_integer() examples
#--------------------------------------------------------------------------
library(bkbase)

# TRUE
is_scalar_integer(1L)
is_scalar_integer(NA_integer_)

# FALSE
is_scalar_integer(1)
is_scalar_integer(1:2)
is_scalar_integer(integer(0L))
is_scalar_integer(NULL)
```

is_scalar_logical *Is object a scalar logical vector*

Description

Returns TRUE if x is a scalar logical (logical vector of length 1).

Usage

```
is_scalar_logical(x)
```

Arguments

x	(any object) The object to be tested.

Value

logical(1).

Examples

```
#------------------------------------------------------------------------
# is_scalar_logical() examples
#------------------------------------------------------------------------
library(bkbase)

# TRUE
is_scalar_logical(TRUE)
is_scalar_logical(FALSE)
is_scalar_logical(NA)

# FALSE
is_scalar_logical(c(TRUE, FALSE))
is_scalar_logical(logical(0L))
is_scalar_logical(1L)
is_scalar_logical(NULL)
```

is_scalar_numeric *Is object a scalar numeric vector*

Description

Returns TRUE if x is a scalar numeric (numeric vector of length 1).

Usage

```
is_scalar_numeric(x)
```

Arguments

x	(any object) The object to be tested.

Value

`logical(1)`.

Examples

```
#--------------------------------------------------------------------------
# is_scalar_numeric() examples
#--------------------------------------------------------------------------
library(bkbase)

# TRUE
is_scalar_numeric(1)
is_scalar_numeric(1L)
is_scalar_numeric(1.5)
is_scalar_numeric(NA_real_)
is_scalar_numeric(NA_integer_)

# FALSE
is_scalar_numeric(1:2)
is_scalar_numeric(numeric(0L))
is_scalar_numeric(NA)
is_scalar_numeric(as.Date("2024-10-01"))
is_scalar_numeric(factor("a"))
is_scalar_numeric(NULL)
```

---

is_scalar_probability *Is object a scalar probability*

Description

Returns `TRUE` if `x` is a scalar probability. A scalar probability is a length 1 nonmissing numeric value bounded by [0, 1].

Usage

```
is_scalar_probability(x)
```

Arguments

x	(any object) The object to be tested.

Value

`logical(1)`.

Examples

```
#--------------------------------------------------------------------------
# is_scalar_probability() examples
#--------------------------------------------------------------------------
library(bkbase)

# TRUE
is_scalar_probability(0)
is_scalar_probability(0.5)
is_scalar_probability(1)

# FALSE
is_scalar_probability(1.1)
is_scalar_probability(NA_real_)
is_scalar_probability(numeric(0L))
is_scalar_probability(NULL)
```

---

is_scalar_whole_number

Is object a scalar whole number

Description

Returns `TRUE` if `x` is a single whole number to some tolerance. See `is_scalar_whole_numeric()` to test for a scalar whole number without a tolerance. See `base::is.integer()` to test for integer storage type.

Usage

```
is_scalar_whole_number(x, tol = 1e-08)
```

Arguments

x	(any object) The object to be tested.
tol	(`numeric(1)`; default `1e-08`; range `[0, Inf)`) Absolute differences smaller than or equal to `tol` are ignored, thus assumed to be a whole number. `tol = 0` tests for an exact whole number.

Details

- NA, NaN, Inf, and -Inf return FALSE.
- Zero-length input and NULL return FALSE.
- Inputs with length other than 1 return FALSE.

Value

logical(1).

See Also

is_whole_number(), is_scalar_whole_numeric(), base::is.integer()

Examples

```
#------------------------------------------------------------------------
# is_scalar_whole_number() examples
#------------------------------------------------------------------------
library(bkbase)

# TRUE
is_scalar_whole_number(1L)
is_scalar_whole_number(1)
is_scalar_whole_number(9.0)
is_scalar_whole_number(sqrt(2)^2)
is_scalar_whole_number(1 + 2^-30)
is_scalar_whole_number(1e100)
is_scalar_whole_number(1.00000001)

# FALSE
is_scalar_whole_number(1.0000001)
is_scalar_whole_number(Inf)
is_scalar_whole_number(c(1, 2, 3))

# Missing values and zero-length
is_scalar_whole_number(NA_integer_) # FALSE
is_scalar_whole_number(NA_real_)    # FALSE
is_scalar_whole_number(NaN)         # FALSE
is_scalar_whole_number(numeric(0))  # FALSE
is_scalar_whole_number(NULL)        # FALSE
```

---

is_scalar_whole_numeric

Is object a scalar whole numeric

Description

Returns TRUE if x is a single integer, or a single double whose value is exactly equal to its truncated form. See is_scalar_whole_number() to test for a scalar whole number within a chosen tolerance. See base::is.integer() to test for integer storage type.

Usage

```
is_scalar_whole_numeric(x)
```

Arguments

x	(any object) The object to be tested.

Details

- Inf and -Inf in double storage return TRUE because Inf == trunc(Inf).
- NA and NaN return FALSE.
- Zero-length input and NULL return FALSE.
- Inputs with length other than 1 return FALSE.

Value

logical(1).

See Also

is_whole_numeric(), is_scalar_whole_number(), base::is.integer()

Examples

```
#------------------------------------------------------------------------------
# is_scalar_whole_numeric() examples
#------------------------------------------------------------------------------
library(bkbase)

# TRUE
is_scalar_whole_numeric(1L)
is_scalar_whole_numeric(1)
is_scalar_whole_numeric(9.0)
is_scalar_whole_numeric(1e100)
is_scalar_whole_numeric(Inf)

# FALSE
is_scalar_whole_numeric(1.00000001)
is_scalar_whole_numeric(sqrt(2)^2)
is_scalar_whole_numeric(1 + 2^-30)
is_scalar_whole_numeric(c(1, 2, 3))

# Missing values and zero-length
```

```
is_scalar_whole_numeric(NA_integer_) # FALSE
is_scalar_whole_numeric(NA_real_)    # FALSE
is_scalar_whole_numeric(NaN)         # FALSE
is_scalar_whole_numeric(numeric(0))  # FALSE
is_scalar_whole_numeric(NULL)        # FALSE
```

`is_strict_decreasing` *Is vector strictly decreasing*

Description

Returns `TRUE` if all values in `x` are strictly decreasing. Returns `FALSE` for non-numeric input, `NULL`, and zero-length vectors.

Usage

```
is_strict_decreasing(x, na.rm = FALSE)
```

Arguments

`x`	(`numeric()`) The vector to be tested.
`na.rm`	(`logical(1)`; default `FALSE`) Whether or not to remove/ignore missing values from `x`.

Value

`logical(1)`.

Examples

```
#--------------------------------------------------------------------------
# is_strict_decreasing() examples
#--------------------------------------------------------------------------
library(bkbase)

is_strict_decreasing(10:-10)
is_strict_decreasing(c(3.5, 2, 2, 1, 1, 0))
is_strict_decreasing(1)

is_strict_decreasing(1:3)

is_strict_decreasing(c(3:1, NA))
is_strict_decreasing(c(3:1, NA), na.rm = TRUE)

is_strict_decreasing(NA_real_)
is_strict_decreasing(NA_real_, na.rm = TRUE)
```

```
is_strict_decreasing(numeric(0L))
is_strict_decreasing(NULL)
```

`is_strict_increasing` *Is vector strictly increasing*

Description

Returns `TRUE` if all values in `x` are strictly increasing. Returns `FALSE` for non-numeric input, `NULL`, and zero-length vectors.

Usage

```
is_strict_increasing(x, na.rm = FALSE)
```

Arguments

`x`	(`numeric()`) The vector to be tested.
`na.rm`	(`logical(1)`; default `FALSE`) Whether or not to remove/ignore missing values from `x`.

Value

`logical(1)`.

Examples

```
#--------------------------------------------------------------------------
# is_strict_increasing() examples
#--------------------------------------------------------------------------
library(bkbase)

is_strict_increasing(-10:10)
is_strict_increasing(c(0, 1, 1, 2, 2, 3.5))
is_strict_increasing(1)

is_strict_increasing(3:1)

is_strict_increasing(c(1:3, NA))
is_strict_increasing(c(1:3, NA), na.rm = TRUE)

is_strict_increasing(NA_real_)
is_strict_increasing(NA_real_, na.rm = TRUE)

is_strict_increasing(numeric(0L))
is_strict_increasing(NULL)
```

is_string *Is object a string*

Description

Returns TRUE if x is a string (character vector of length 1).

Usage

```
is_string(x)
```

Arguments

x	(any object) The object to be tested.

Value

logical(1).

Examples

```
#--------------------------------------------------------------------------
# is_string() examples
#--------------------------------------------------------------------------
library(bkbase)

# TRUE
is_string("a")
is_string(NA_character_)

# FALSE
is_string(c("a", "b"))
is_string(character(0L))
is_string(1L)
is_string(NULL)
```

is_whole_number *Is object a vector of whole numbers*

Description

Returns TRUE if x consists of whole numbers to some tolerance. See is_whole_numeric() to test for whole numbers without a tolerance. See base::is.integer() to test for integer storage type.

Usage

```
is_whole_number(x, tol = 1e-08)
```

Arguments

x	(any object) The object to be tested.
tol	(numeric(1); default 1e-08; range [0, Inf)) Absolute differences smaller than or equal to tol are ignored, thus assumed to be a whole number. tol = 0 tests for exact whole numbers.

Details

- NA, NaN, Inf, and -Inf return FALSE.
- Zero-length input and NULL return FALSE.

Value

logical(1).

See Also

is_whole_numeric(), is_scalar_whole_number(), base::is.integer()

Examples

```
#------------------------------------------------------------------------------
# is_whole_number() examples
#------------------------------------------------------------------------------
library(bkbase)

# TRUE
is_whole_number(1L)
is_whole_number(1)
is_whole_number(9.0)
is_whole_number(sqrt(2)^2)
is_whole_number(1 + 2^-30)
is_whole_number(1e100)
is_whole_number(1.00000001)

# FALSE
is_whole_number(1.0000001)
is_whole_number(Inf)

# Missing values and zero-length
is_whole_number(NA_integer_) # FALSE
is_whole_number(NA_real_)    # FALSE
is_whole_number(NaN)         # FALSE
is_whole_number(numeric(0))  # FALSE
is_whole_number(NULL)        # FALSE
```

`is_whole_numeric` *Is object a vector of whole numerics*

Description

Returns `TRUE` if `x` is an integer vector, or a double vector whose values are exactly equal to their truncated form. See `is_whole_number()` to test for whole numbers within a chosen tolerance. See `base::is.integer()` to test for integer storage type.

Usage

```
is_whole_numeric(x)
```

Arguments

`x`	(any object) The object to be tested.

Details

- `Inf` and `-Inf` in double storage return `TRUE` because `Inf == trunc(Inf)`.
- `NA` and `NaN` return `FALSE`.
- Zero-length input and `NULL` return `FALSE`.

Value

`logical(1)`.

See Also

`is_whole_number()`, `is_scalar_whole_numeric()`, `base::is.integer()`

Examples

```
#--------------------------------------------------------------------------
# is_whole_numeric() examples
#--------------------------------------------------------------------------
library(bkbase)

# TRUE
is_whole_numeric(1L)
is_whole_numeric(1)
is_whole_numeric(9.0)
is_whole_numeric(1e100)
is_whole_numeric(Inf)

# FALSE
is_whole_numeric(1.00000001)
```

```
is_whole_numeric(sqrt(2)^2)
is_whole_numeric(1 + 2^-30)

# Missing values and zero-length
is_whole_numeric(NA_integer_) # FALSE
is_whole_numeric(NA_real_)    # FALSE
is_whole_numeric(NaN)         # FALSE
is_whole_numeric(numeric(0))  # FALSE
is_whole_numeric(NULL)        # FALSE
```

`list_clean` *Clean lists*

Description

Recursively remove `NULL` and zero-length elements from a list.

Usage

```
list_clean(x)
```

Arguments

`x`	(list) A list to clean. If `x` is not a list, it is returned unchanged.

Value

If `inherits(x, "list")` is `FALSE` (for example, a `data.frame`, an atomic vector, `NULL`, or any S3 object whose class vector does not contain `"list"`), `x` is returned unchanged. Otherwise, a list is returned with `NULL`, zero-length, and recursively-empty elements removed at every level of nesting.

Examples

```
#-------------------------------------------------------------------------
# list_clean() examples
#-------------------------------------------------------------------------
library(bkbase)

# Removes NULL and zero-length elements, preserves names
x <- list(a = 1, b = NULL, c = logical(0L), d = 2)
list_clean(x)

# Recursive cleaning, including fully-empty sublists
y <- list(a = list(1, NULL), b = list(NULL, logical(0L)))
list_clean(y)

# Non-list inputs are returned unchanged
list_clean(data.frame(x = 1:3))
```

```
list_clean(1:5)
list_clean(NULL)
```

`list_flatten2` *Flatten a nested list to a single level*

Description

Recursively flattens any nested lists into a single-level list. Non-list objects are treated as leaves and kept as single elements. If `preserve_df = TRUE`, `data.frames` are treated as leaves and not exploded.

Usage

```
list_flatten2(x, preserve_df = TRUE, keep_names = TRUE)
```

Arguments

`x`	(list) A possibly nested list that should be flattened. If `x` is not a list, the result is `list(x)`. This also applies to `NULL`, and to a data.frame when `preserve_df = TRUE`.
`preserve_df`	(`logical(1)`; default `TRUE`) If `TRUE` (default), `data.frames` are treated as leaves and not flattened inside.
`keep_names`	(`logical(1)`; default `TRUE`) If `TRUE` (default), names of leaf elements are retained. The function keeps only the name of the leaf itself and drops the names of the enclosing lists. Because of this, the result can hold repeated names, as for `list(a = list(x = 1), b = list(x = 2))`.

Value

A single-level list containing all leaf elements from `x`, in left-to-right order.

Examples

```
#------------------------------------------------------------------------------
# list_flatten2() examples
#------------------------------------------------------------------------------
library(bkbase)

# Flatten
x <- list(1, list(2, list(3, 4)), 5)
list_flatten2(x)

# Preserves leaf names and data frames as leaves
y <- list(list(a = 1:2), data.frame(x = 1:2), list(b = "z"))
```

```
list_flatten2(y)

# Drop leaf names
z <- list(list(a = c("name1" = 1, "name2" = 2)), list(b = "z"))
list_flatten2(z, keep_names = FALSE)
```

`logical_to_integer` *Convert logical values to 0/1*

Description

Converts logical vectors and matrices to integer type. Values `FALSE`, `TRUE`, and `NA` are converted to `0`, `1`, or `NA_integer_`, respectively.

Usage

```
logical_to_integer(x)
```

Arguments

`x`	(logical vector, matrix, or array) The logical object to convert to 0/1 integer.

Details

`logical_to_integer()` will not drop dimensional attributes (as done by `as.integer()`).

Value

integer vector, matrix, or array with the same dimensions as `x`.

Examples

```
#--------------------------------------------------------------------------
# logical_to_integer() examples
#--------------------------------------------------------------------------
library(bkbase)

logical_to_integer(c(TRUE, FALSE, NA))
logical_to_integer(matrix(c(TRUE, FALSE, TRUE), nrow = 3))
```

match.arg2 *Exact match of a choice argument*

Description

Resolve an argument that takes one string from a fixed set, as base::match.arg() does. Use match.arg2() at the start of a function whose formal default lists the allowed values, as in link = c("logit", "probit").

Usage

```
match.arg2(arg, choices, name = NULL)
```

Arguments

arg	(character(1) or character()) Value of the argument to match. Must be one element of choices, or identical to choices.
choices	(character()) Allowed values, with length 1 or more and no NA. If missing, the function uses the default of the formal argument name of the calling function.
name	(character(1) or NULL; default NULL) Name of the argument. The function uses it in the error message and to find choices when choices is missing. If NULL, the function uses the expression passed as arg. A symbol such as link gives "link", and a call such as x$type gives "x$type".

Details

match.arg2() is different from base::match.arg() in these ways:

1. The function accepts only an exact match. The value "log" does not match "logit".
2. NULL gives an error. match.arg() returns the first choice for NULL.
3. The error message names the argument and states the value that it received, with format_value().
4. The function has no several.ok argument, and it always returns one choice.

When arg is identical to choices, the function returns the first choice. The caller gets this result when it does not supply the argument, because the value of arg is then the full default vector.

When choices is missing, the function finds the choices in the default of the formal argument name of the calling function. It evaluates that default in the frame of the caller. Because of this, a default can use a constant of the namespace of the caller. The lookup works only when the caller passes the argument under its formal name, as with match.arg(). For example, l <- link; match.arg2(l) finds no choices, because the caller has no argument l. In this case, pass choices. When the formal argument has no default, R stops with the error "argument is missing, with no default", as with match.arg().

The error message for a string that is not a choice states the string, `""` included. For `NA_character_`, the message states only the choices. The error has no call, because the call of `match.arg2()` does not help the user.

Value

`character(1)`, the matched element of `choices` with no names. The function stops with an error when `arg` does not match, or when it finds no choices.

See Also

`base::match.arg()`

Examples

```
#-------------------------------------------------------------------------
# match.arg2() example
#-------------------------------------------------------------------------
library(bkbase)

# The formal default lists the choices. No value gives the first choice.
fit <- function(link = c("logit", "probit", "cloglog")) {
  link <- match.arg2(link)
  link
}
fit()
fit("probit")

# A prefix of a choice does not match, and NULL gives an error.
try(fit("log"))
try(fit(NULL))

# Pass `choices` for a value that is not a formal argument.
opts <- list(type = "b")
match.arg2(opts$type, c("a", "b"))
```

`match.call2` *Argument matching*

Description

Wraps `base::match.call()` with two additions.

1. Argument `n` selects a call further up the call stack rather than the immediate caller.
2. Argument `defaults = TRUE` inserts formal defaults for arguments the caller did not supply.

Usage

```
match.call2(
  n = 0L,
  definition = sys.function(sys.parent(n + 1L)),
  call = sys.call(sys.parent(n + 1L)),
  expand.dots = TRUE,
  envir = parent.frame(n + 2L),
  defaults = FALSE
)
```

Arguments

n	(integer(1); default 0L; range [0, Inf)) How far up the call stack to match a call. n = 0 is the caller of match.call2(), and each step up goes to the parent frame. n counts only parent frames (see base::sys.parent()). Other frames on the call stack, such as the frames of base::tryCatch(), do not count. Only the defaults of definition, call, and envir use n. When one of these defaults is used, the frame n steps up must exist, or the function gives an error. At n = 0, match.call2() also works at the top level when you give definition and call, the same as base::match.call().
definition	(function; default sys.function(sys.parent(n + 1L))) The function whose call is being matched. By default, this is the function of the frame that n selects.
call	(call; default sys.call(sys.parent(n + 1L))) An unevaluated call to the function specified by definition. By default, this is the call of the frame that n selects.
expand.dots	(logical(1); default TRUE) Whether arguments matching ... are expanded into the call as named arguments (TRUE), or kept grouped under a single ... slot (FALSE).
envir	(environment; default parent.frame(n + 2L)) The environment used to look up ... when call contains a literal ... symbol. This applies, for example, when a wrapper forwards ... to definition. By default, this is the environment in which definition was called.
defaults	(logical(1); default FALSE) Whether or not to include unspecified default arguments in the call. Formals without a default value are not inserted.

Value

An object of class call. When defaults = TRUE, formal arguments with defaults that the caller did not supply are inserted into the call as unevaluated expressions. Caller-supplied arguments appear in formal order, as with base::match.call(), and the inserted defaults are appended after them in formal order.

See Also

base::match.call()

Examples

```
#--------------------------------------------------------------------------
# match.call2() examples
#--------------------------------------------------------------------------
library(bkbase)

# Basic usage: capture the matched call from within a function.
fit <- function(x, y, method = "ols", tol = 1e-8) {
  match.call2()
}
fit(1:5, 6:10)

# defaults = TRUE inserts unevaluated formal defaults for arguments the
# caller did not supply, producing a self-describing call useful for
# logging or reproducing results.
fit2 <- function(x, y, method = "ols", tol = 1e-8) {
  match.call2(defaults = TRUE)
}
fit2(1:5, 6:10)
fit2(1:5, 6:10, method = "ridge")

# n > 0 reaches further up the call stack, so a helper can inspect how
# its caller was invoked.
log_caller <- function() match.call2(n = 1)
wrapper <- function(a, b = 2) log_caller()
wrapper(a = 10)

# expand.dots = FALSE keeps the ... arguments grouped under one slot.
inspect <- function(x, ...) match.call2(expand.dots = FALSE)
inspect(x = 1, extra1 = "a", extra2 = "b")
```

`middle` *Middle sorted element*

Description

Return the middle element of an atomic vector after sorting. Missing values are automatically removed.

Usage

```
middle(x)
```

Arguments

`x`	(atomic vector) An atomic vector.

Details

For odd-length input, returns the unique middle element of the sorted vector. For even-length input, returns the lower of the two middle elements (`sort(x)[ceiling(length(x) / 2)]`), not the average. This differs from `stats::median()` and allows `middle()` to work on any atomic type, including character and factor inputs.

Value

A scalar of the same type as `x`. Empty or all-`NA` input gives a zero-length vector of that type, and `NULL` gives `NULL`.

Examples

```
#-------------------------------------------------------------------------
# middle() example
#-------------------------------------------------------------------------
library(bkbase)

# Odd-length input returns the single middle element
middle(c(1, 2, 3, 4, 5))

# Even-length input returns the lower of the two middle elements
middle(c(1, 2, 3, 4))

# Works on any atomic type
middle(c("a", "b", "c", "d", "e"))

# Missing values are removed before selecting the middle element
middle(c(1, 2, 3, NA, NA))
```

`or` *Logical OR with NA tolerance*

Description

Compute a position-wise logical OR across multiple logical inputs. When no `TRUE` is present, it allows up to a specified fraction of missing values per position before returning `NA`.

Usage

```
or(..., .na_allowed = 0.2)
```

Arguments

`...`	(`logical()`, logical matrix, or data.frame) Logical vectors of equal length, or a single logical matrix/data.frame. Matrix/data.frame inputs must have at least one column. Inputs are coerced to logical via `as.logical`, preserving `NA`.

`.na_allowed` (`numeric(1)`; default `0.2`; range `[0, 1]`)
Maximum allowed fraction of `NA` per position before the result is set to `NA` when no `TRUE` is present.

Details

Standard logical disjunction over multiple vectors (e.g., `x | y | z`). Each output position corresponds to one vector index, or to one row when a matrix or data.frame is supplied. Returns `TRUE` for a position if at least one element is `TRUE`, regardless of missing values. Returns `NA` for a position if no `TRUE` is present and the fraction of `NA` exceeds `.na_allowed`. Otherwise, missing values are treated as `FALSE` when evaluating the OR.

For each position (row) i across p inputs, let m_i be the number of `NA`s and t_i be the number of `TRUE`s. Define the missing fraction $r_i = m_i/p$. Then:

$$\text{or}_i = \begin{cases} \text{TRUE}, & \text{if } t_i \geq 1, \\ \text{NA}, & \text{if } t_i = 0 \text{ and } r_i > \text{.na_allowed}, \\ \text{FALSE}, & \text{if } t_i = 0 \text{ and } r_i \leq \text{.na_allowed}. \end{cases}$$

Notes:

1. When `.na_allowed = 0`, behavior matches strict OR: `NA` propagates only when no `TRUE` is present.
2. When .na_allowed $= 1$, missingness never forces `NA`. `NA`s are treated as `FALSE`, so the result is `TRUE` iff any `TRUE` is present, otherwise `FALSE`.

Value

A logical vector with values `TRUE`, `FALSE`, or `NA` per the rule in Details. The length equals the length of each input vector, or the number of rows of a matrix or data.frame.

See Also

base::Logic

Examples

```
#--------------------------------------------------------------------------
# or() examples
#--------------------------------------------------------------------------
library(bkbase)

x <- c(FALSE, FALSE, FALSE, TRUE,  FALSE)
y <- c(FALSE, FALSE, NA,    FALSE, NA)
z <- c(FALSE, NA,    NA,    NA,    TRUE)

# Base R strict OR
x | y | z

# NA tolerance at 20%
or(x, y, z, .na_allowed = 0.2)
```

```
# NA tolerance at 40%
or(x, y, z, .na_allowed = 0.4)

# Single data.frame input
df <- data.frame(x = x, y = y, z = z)
or(df, .na_allowed = 0.4)

# TRUE dominates even when NA is present
or(TRUE, NA, .na_allowed = 0)
```

`quiet` *Suppress printed output*

Description

Output written to `stdout` during the evaluation of `x` is suppressed. Messages, warnings, and errors are not suppressed.

Usage

```
quiet(x)
```

Arguments

`x`	(expression) An expression.

Value

The value of `x`, returned invisibly.

See Also

`base::suppressMessages()`

Examples

```
#-------------------------------------------------------------------------
# quiet() examples
#-------------------------------------------------------------------------
library(bkbase)

bad_fun <- function(x) {
  cat("Unwanted output\n")
  x
}

# Direct call writes to stdout and returns the value:
```

```
bad_fun(2)

# Wrapped call suppresses stdout and returns invisibly:
quiet(bad_fun(2))

# The return value is preserved:
result <- quiet(bad_fun(2))
result
```

`resample` *Random sample*

Description

`base::sample()` treats a single numeric `x` with `x >= 1` as `1:x` and samples from that sequence. This surprises callers when `x` may have varying length, such as in `sample(x)`. `resample()` always samples from the elements of `x` itself. See `base::sample()` for more details.

Usage

```
resample(x, size, replace = FALSE, prob = NULL)
```

Arguments

`x`	(atomic vector) The atomic vector to sample from.
`size`	(`integer(1)`; range `[1, Inf)`) The number of elements to sample.
`replace`	(`logical(1)`; default `FALSE`) Whether or not sampling should be done with replacement.
`prob`	(`numeric(length(x))` or `NULL`; default `NULL`) A non-negative numeric vector of length `length(x)` whose elements sum to 1. If `NULL`, every element of `x` has the same probability. The function accepts a sum that differs from 1 by `1e-7` or less. Element `prob[i]` is the probability of sampling `x[i]`. `NA` values are not allowed.

Details

When `replace = FALSE`, `size` must not exceed `length(x)`.

Value

A vector of length `size` with elements sampled from `x`.

See Also

`base::sample()`

Examples

```
#--------------------------------------------------------------------------
# resample() examples
#--------------------------------------------------------------------------
library(bkbase)

set.seed(2)

# Sample without replacement
resample(x = 1:10, size = 5)

# Sample with replacement
resample(x = c("a", "b", "c"), size = 5, replace = TRUE)

# Weighted sampling
resample(
  x = c("heads", "tails"),
  size = 3, replace = TRUE,
  prob = c(0.7, 0.3)
)

# A length-1 vector is sampled as-is, unlike base::sample()
resample(x = 5L, size = 1L)
sample(x = 5L, size = 1L) # samples from 1:5 instead
```

---

reset_rownames *Reset data frame row names*

Description

Quickly reset data frame row names to integers using a pipe-compatible function.

Usage

```
reset_rownames(data)

remove_rownames(data)
```

Arguments

data	(data.frame) The data frame whose row names should be reset.

Value

A data frame identical to `data` except that the `row.names` attribute is reset to `1:nrow(data)`.

Examples

```
#------------------------------------------------------------------------------
# reset_rownames() examples
#------------------------------------------------------------------------------
library(bkbase)

# Subsetting a data frame preserves the original row names.
df <- data.frame(a = 1:5)
sub <- df[c(3, 5), , drop = FALSE]
sub

# reset_rownames() renumbers them as 1:nrow(sub).
reset_rownames(sub)

# remove_rownames() is an alias for reset_rownames().
remove_rownames(sub)

# Pipe-compatible.
df[c(3, 5), , drop = FALSE] |> reset_rownames()
```

`rle2` *Run length encoding*

Description

Returns the lengths and values of runs of consecutive equal elements in a vector.

Usage

```
rle2(x, order = FALSE, index = TRUE, na.last = TRUE)
```

Arguments

`x`	(atomic vector) The vector to be run length encoded.
`order`	(`logical(1)`; default `FALSE`) Whether or not to order the data frame based on the values in `x`.
`index`	(`logical(1)`; default `TRUE`) Whether or not to add the index position for each run/consecutive value in columns named `'start'` and `'stop'`.
`na.last`	(`logical(1)`; default `TRUE`) Controls placement of `NA` runs when `order = TRUE`. `TRUE` puts them last, `FALSE` puts them first, `NA` drops them. Ignored when `order = FALSE`.

Details

`base::rle()` returns a list with a custom print method. `rle2()` returns a `data.frame`, optionally annotated with the start and stop indices of each run, and optionally sorted by value.

Unlike `base::rle()`, consecutive `NA`s collapse into a single run. A transition between `NA` and a non-`NA` value also counts as a run boundary.

A zero-length `x` returns a zero-row `data.frame` with the documented columns.

The returned `data.frame` is not compatible with `base::inverse.rle()`.

Value

`data.frame` with structure:

Column	Name	Class	Note
1	`value`	`class(x)`	
2	`length`	`integer`	
3	`start`	`integer`	Optional
4	`stop`	`integer`	Optional

See Also

`base::rle()`

Examples

```
#--------------------------------------------------------------------------
# rle2() examples
#--------------------------------------------------------------------------
library(bkbase)

x <- c(NA, NA, rev(rep(6:10, 1:5)), NA)

# base::rle() splits NAs into single-element runs and returns a list.
rle(x)

# rle2() collapses consecutive NAs and returns a data frame.
rle2(x)

# Drop the start and stop columns.
rle2(x, index = FALSE)

# Sort by value. NA runs are placed last by default.
rle2(x, order = TRUE)

# Sort by value and drop NA runs entirely.
rle2(x, order = TRUE, na.last = NA)

# Works on any atomic vector, including character.
rle2(c("a", "a", "b", "a", "a", "a"))
```

`rm_na` *Remove missing values*

Description

Remove missing values from an atomic vector using a pipe-compatible function.

Usage

```
rm_na(x)

rmna(x)
```

Arguments

`x`	(vector) The vector to remove missing values from. The function also accepts a list, and removes each top-level element that is a single `NA`.

Details

- `NaN` values are also removed because `is.na(NaN)` returns `TRUE`.
- `NULL` input returns `NULL` because `anyNA(NULL)` is `FALSE`.
- Zero-length input is returned unchanged.
- Attributes such as names, factor levels, and Date/POSIXct class are preserved by the `[` subset.
- A matrix, an array with two or more dimensions, or a data.frame gives an error. Use `stats::na.omit()` to remove rows that hold missing values.

Value

A vector of the same type and class as `x` with `NA` values removed.

See Also

`stats::na.omit()`, `as_na()`

Examples

```
#--------------------------------------------------------------------------
# rm_na() examples
#--------------------------------------------------------------------------
library(bkbase)

# No NAs: input is returned unchanged
rm_na(1:3)

# NAs are removed
```

```
rm_na(c(1, NA, 2, NA, 3))

# NaN is also removed
rm_na(c(1, NaN, 2))

# Names and other attributes are preserved
rm_na(c(a = 1, b = NA, c = 3))
rm_na(factor(c("a", NA, "b")))

# rmna() is an alias for rm_na()
c(1, NA, 2) |> rmna()
```

---

round2 *Round mixed-type numbers*

Description

Round the numeric elements of an atomic vector and return a character vector. Elements that do not represent a single number are left unchanged.

Usage

```
round2(x, digits = 3L)
```

Arguments

x	(atomic vector) The vector whose numeric elements should be rounded.
digits	(integer(1); default 3L) The number of decimal places for rounded numbers. Must be finite.

Details

Each element is coerced to character, then parsed with base::as.numeric(). Elements that parse to a finite number or to Inf/-Inf are rounded to digits decimal places. All other elements pass through unchanged, including text such as "$99.99", "NaN", and NA.

A nonzero value that rounds to zero loses all magnitude information. Such values are shown in compact scientific notation (for example "1.0e-04") instead of "0". Exact zero is returned as "0".

Other results with an absolute value below 1e15 use fixed notation, as in "100000". The fixed string has the same significant digits as the result of base::as.character(), so both strings give the same number. Results with an absolute value of 1e15 or more keep the format of base::as.character(), as in "1e+15".

Logical input is rendered as "TRUE"/"FALSE" rather than 0/1. Names on x are preserved.

Value

A character vector the same length as x.

See Also

base::round(), base::sprintf()

Examples

```
#------------------------------------------------------------------------
# round2() example
#------------------------------------------------------------------------
library(bkbase)

round2(c("hello", "0.0001", "1.23456", "hello 1.23456"))
```

`rowCumsums` *Row-wise cumulative sums*

Description

Compute the cumulative sums along each row of a matrix. Column j of the result holds the sum of columns 1 to j of `x`. For example, `rowCumsums()` changes a matrix of category probabilities into cumulative probabilities.

Usage

```
rowCumsums(x)
```

Arguments

`x`	(numeric or logical matrix) Matrix to sum along each row. The function does not accept a data frame. Use `as.matrix()` to convert one first.

Details

The result in row i and column j is:

$$S_{ij} = \sum_{k=1}^{j} x_{ik}$$

The function adds each column to the column before it, from left to right.

An `NA` or `NaN` in column j gives `NA` or `NaN` in column j and in each column to its right. The columns to its left keep their values. R does not specify which of `NA` and `NaN` a sum of the two gives (see base::NaN). `Inf` and `-Inf` are values, and their sum is `NaN`.

A double matrix gives a double matrix, with sums in double precision. `base::cumsum()` keeps its running sum in long double when the platform supports it. Because of this, the results of `rowCumsums(x)` and `t(apply(x, 1, cumsum))` can differ in the last bits.

An integer or a logical matrix gives an integer matrix. The function counts FALSE as 0 and TRUE as 1. A sum larger than .Machine$integer.max gives NA with a warning, as integer addition does. To prevent this, convert x to double first.

A numeric matrix with zero or one column returns unchanged. A matrix with zero rows returns with its dimensions.

Value

A matrix with the dimensions and the dimnames of x. The type is integer when x is integer or logical, and double otherwise.

See Also

base::cumsum()

Other rowwise: rowMaxs(), rowMaxs2(), rowMeans2(), rowSums2()

Examples

```
#------------------------------------------------------------------------
# rowCumsums() example
#------------------------------------------------------------------------
library(bkbase)

# Change category probabilities into cumulative probabilities.
p <- matrix(
  c(0.2, 0.5, 0.3,
    0.1, 0.1, 0.8),
  nrow = 2,
  byrow = TRUE,
  dimnames = list(NULL, c("low", "mid", "high"))
)
rowCumsums(p)

# An NA changes its column and the columns to its right.
rowCumsums(matrix(c(1, NA, 3, 4), nrow = 1))
```

---

rowMaxs *Row-wise maxima*

Description

Compute row-wise maxima from a logical/numeric matrix or data.frame. Optionally ignore missing values. Row names are preserved on ordinary non-empty results.

Usage

```
rowMaxs(x, na.rm = FALSE)
```

Arguments

x	(logical/numeric matrix or data.frame) A rectangular object whose rows will be aggregated via maxima. Each column must be logical or numeric.
na.rm	(logical(1); default FALSE) If TRUE, ignore NA values when computing row-wise maxima, returning NA only when all values in a row are missing. If FALSE, the presence of any NA in a row yields an NA result for that row.

Details

Let p be the number of columns in x. For row i, define the row-wise maximum M_i as follows.

If na.rm = TRUE, the maximum is computed over the observed values only.

$$M_i = \max\{x_{ij} : x_{ij} \text{ not NA}\}.$$

NaN is treated as missing because missingness is determined by base::is.na(). If all entries in row i are missing, the returned value for that row is NA.

If na.rm = FALSE, the maximum includes missing values.

$$M_i = \max\{x_{ij}\}.$$

In this case, any NA present in row i yields an NA result for that row.

Inf and -Inf are treated as observed values and can be returned as row maxima. Logical values are evaluated as FALSE = 0 and TRUE = 1. If x has row names, those names are assigned to the resulting vector.

Value

A vector of length nrow(x) containing the row-wise maxima. The storage type is integer when x is integer or logical, and double otherwise. If x has zero rows, a zero-length vector of the appropriate storage type is returned. If x has zero columns, an unnamed length-nrow(x) vector of NA of the appropriate storage type is returned. Rows return NA according to the rules described by na.rm. For inputs with at least one row and one column, row names are preserved when present.

See Also

Other rowwise: rowCumsums(), rowMaxs2(), rowMeans2(), rowSums2()

Examples

```
#------------------------------------------------------------------------
# rowMaxs() examples
#------------------------------------------------------------------------
library(bkbase)

# Example matrix
m <- matrix(
  c(1, NA, 3, 4,
```

```
    2,  5, NA, 6),
  nrow = 2,
  byrow = TRUE
)
m

# Row-wise maxima without removing NAs (rows containing NA yield NA)
rowMaxs(m, na.rm = FALSE)

# Row-wise maxima removing NAs
rowMaxs(m, na.rm = TRUE)

# Row names are preserved on ordinary non-empty results
rownames(m) <- c("row_a", "row_b")
rowMaxs(m, na.rm = TRUE)
```

`rowMaxs2` *Row-wise maxima with NA allowance*

Description

Compute row-wise maxima from a logical/numeric matrix or data.frame. Rows exceeding a specified missing-value proportion are set to `NA`. Row names are preserved on ordinary non-empty results.

Usage

```
rowMaxs2(x, na_allowed = 0.2)
```

Arguments

`x`	(logical/numeric matrix or data.frame) A rectangular object whose rows will be aggregated via maxima. Each column must be logical or numeric.
`na_allowed`	(`numeric(1)`; default `0.2`; range `[0, 1]`) Maximum allowed proportion of missing values per row (π_i) before the result is set to `NA`. Rows with $\pi_i >$ na_allowed return `NA`. Rows with $\pi_i \leq$ na_allowed are computed using observed values only.

Details

For row i, let π_i be the proportion of missing values in that row.

If $\pi_i > \tau$ where τ is `na_allowed`, the returned maximum for row i is `NA`. Otherwise, the maximum is computed over the observed values only:

$$M_i = \max\{x_{ij} : x_{ij} \text{ not NA}\}.$$

If all entries in a row are `NA`, the result for that row is `NA` regardless of `na_allowed`. `NaN` is treated as missing because missingness is determined by `base::is.na()`. It counts toward π_i and is excluded

from the maximum. Inf and -Inf are treated as observed values and can be returned as row maxima. Logical values are evaluated as FALSE = 0 and TRUE = 1. If x has row names, those names are assigned to the resulting vector.

Value

A vector of length nrow(x) containing the row-wise maxima, with NA where the missing proportion exceeds na_allowed. The storage type is integer when x is integer or logical, and double otherwise. If x has zero rows, a zero-length vector of the appropriate storage type is returned. If x has zero columns, an unnamed length-nrow(x) vector of NA of the appropriate storage type is returned. For inputs with at least one row and one column, row names are preserved when present.

See Also

Other rowwise: rowCumsums(), rowMaxs(), rowMeans2(), rowSums2()

Examples

```
#------------------------------------------------------------------------
# rowMaxs2() examples
#------------------------------------------------------------------------
library(bkbase)

# Example matrix
m <- matrix(
  c(1, NA, 3, 4,
    2,  5, NA, 6),
  nrow = 2,
  byrow = TRUE
)
m

# Allow up to 25% missing per row: both rows are computed
rowMaxs2(m, na_allowed = 0.25)

# Tighten allowance to 0%: any row with an NA becomes NA
rowMaxs2(m, na_allowed = 0)

# Row names are preserved on ordinary non-empty results
rownames(m) <- c("row_a", "row_b")
rowMaxs2(m, na_allowed = 0.5)
```

---

rowMeans2 *Row-wise means with NA allowance*

Description

Compute row-wise means from a logical/numeric matrix or data.frame. Rows exceeding a specified missing-value proportion are set to NA.

Usage

```
rowMeans2(x, na_allowed = 0.2, round_digits = NULL)
```

Arguments

x	(logical/numeric matrix or data.frame) A rectangular object whose rows will be aggregated via means. Each column must be logical or numeric.
na_allowed	(numeric(1); default 0.2; range [0, 1]) Maximum allowed proportion of missing values per row (π_i) before the result is set to NA. Rows with $\pi_i >$ na_allowed return NA; rows with $\pi_i \leq$ na_allowed are computed.
round_digits	(integer(1) or NULL; default NULL; range [0, Inf)) If not NULL, round the resulting means to the specified number of digits.

Details

Let p be the number of columns in x. For row i, let π_i be the proportion of missing values in that row.

If $\pi_i > \tau$ where τ is na_allowed, the returned mean for row i is NA. Otherwise, when at least one observed value exists, the mean is computed over the observed values only:

$$M_i = \frac{1}{|\{j : x_{ij} \text{ not missing}\}|} \sum_{j : x_{ij} \text{ not missing}} x_{ij}.$$

If all entries in a row are NA or NaN, the result for that row is NA regardless of na_allowed. NaN is treated as missing because missingness is determined by base::is.na(). Inf and -Inf are treated as observed values and are passed to base::rowMeans() arithmetic. A row with both observed Inf and -Inf can therefore return NaN. Logical values are averaged as FALSE = 0 and TRUE = 1.

Rounding is disabled by default. Set round_digits to round the resulting means.

Value

A double vector of length nrow(x) containing the row-wise means, with NA where the missing proportion exceeds na_allowed. For inputs with at least one row and one column, row names are preserved. If x has zero rows, a zero-length double vector is returned. If x has zero columns, an unnamed length-nrow(x) double vector of NA is returned.

See Also

Other rowwise: rowCumsums(), rowMaxs(), rowMaxs2(), rowSums2()

Examples

```
#------------------------------------------------------------------------
# rowMeans2() examples
#------------------------------------------------------------------------
library(bkbase)
```

```
m <- matrix(
  c(1, NA, 3, 4,
    2,  5, NA, 6),
  nrow = 2,
  byrow = TRUE
)
m

# Allow up to 25% missing per row
rowMeans2(m, na_allowed = 0.25)

# Enable rounding to 2 digits
rowMeans2(m, na_allowed = 0.5, round_digits = 2)
```

`rowSums2` *Row-wise sums with NA allowance*

Description

Compute row-wise sums from a logical/numeric matrix or data.frame. Rows exceeding a specified missing-value proportion are set to `NA`. Optionally impute missing values using the row mean when the proportion of missing values does not exceed the specified threshold.

Usage

```
rowSums2(x, na_allowed = 0.2, mean_impute = FALSE, round_digits = NULL)
```

Arguments

`x`	(logical/numeric matrix or data.frame) A rectangular object whose rows will be aggregated via sums. Each column must be logical or numeric.
`na_allowed`	(`numeric(1)`; default `0.2`; range `[0, 1]`) Maximum allowed proportion of missing values per row (π_i) before the result is set to `NA`. Rows with $\pi_i >$ na_allowed return `NA`; rows with $\pi_i \leq$ na_allowed are computed.
`mean_impute`	(`logical(1)`; default `FALSE`) Let π_i be the proportion of missing values in rows. If `TRUE`, rows with at least one missing value and $\pi_i \leq$ na_allowed are imputed using the row mean scaled by the number of columns. If `FALSE`, sums are computed over observed values only.
`round_digits`	(`integer(1)` or `NULL`; default `NULL`; range `[0, Inf)`) If not `NULL`, round the resulting sums to the specified number of digits.

Details

Let p be the number of columns in `x`. For row i, let π_i be the proportion of missing values in that row.

If $\pi_i > \tau$ where τ is `na_allowed`, the returned sum for row i is `NA`. Otherwise:

$$S_i = \sum_{j:x_{ij} \text{ not NA}} x_{ij}$$

when `mean_impute = FALSE`.

$$S_i = p \cdot \bar{x}_{i,\text{obs}}$$

when `mean_impute = TRUE`, where $\bar{x}_{i,\text{obs}}$ is the mean of the non-missing values in row i.

If all entries in a row are `NA` or `NaN`, the row has no observed mean. With `mean_impute = TRUE`, such rows return `NA` even when `na_allowed = 1`. With `mean_impute = FALSE`, such rows return `0` when allowed by `na_allowed`, matching `base::rowSums()` with `na.rm = TRUE`. `NaN` is treated as missing because missingness is determined by `base::is.na()`. `Inf` and `-Inf` are treated as observed values and are passed to `base::rowSums()` arithmetic. A row with both observed `Inf` and `-Inf` can therefore return `NaN`. Logical values are summed as `FALSE = 0` and `TRUE = 1`.

Rounding is disabled by default. Set `round_digits` to round the resulting sums.

Value

A double vector of length `nrow(x)` containing the row-wise sums, with `NA` where the missing proportion exceeds `na_allowed`. For inputs with at least one row and one column, row names are preserved. If `x` has zero rows, a zero-length double vector is returned. If `x` has zero columns, an unnamed length-`nrow(x)` double vector of `NA` is returned.

See Also

Other rowwise: `rowCumsums()`, `rowMaxs()`, `rowMaxs2()`, `rowMeans2()`

Examples

```
#------------------------------------------------------------------------
# rowSums2() examples
#------------------------------------------------------------------------
library(bkbase)

m <- matrix(
  c(1, NA, 3, 4,
    2,  5, NA, 6),
  nrow = 2,
  byrow = TRUE
)
m

# No imputation, allow up to 25% missing per row
rowSums2(m, na_allowed = 0.25, mean_impute = FALSE)

# Mean imputation for rows with some missing but within threshold
```

```
rowSums2(m, na_allowed = 0.5, mean_impute = TRUE)

# Enable rounding to 0 digits
rowSums2(m, na_allowed = 0.5, mean_impute = TRUE, round_digits = 0)
```

---

seq2 *Nondecreasing integer sequence*

Description

Similar to `seq.int`, but returns an empty vector (`integer(0)`) if the starting point is larger than the end point.

Usage

```
seq2(from, to)
```

Arguments

`from`	(`integer(1)`) The beginning value of the integer sequence.
`to`	(`integer(1)`) The ending value of the integer sequence.

Details

Safer than using something like `from:to` or `seq.int` in a programming scenario.

Value

`integer()`.

See Also

`base::seq.int()`, `base::seq()`, `base::seq_len()`, `base::seq_along()`

Examples

```
#----------------------------------------------------------------------
# seq2() examples
#----------------------------------------------------------------------
library(bkbase)

seq2(from = 1, to = 5)
seq2(from = 1, to = 1)
seq2(from = 1, to = 0)
```

`str_trim2` *Remove whitespace*

Description

Removes whitespace from the left and right sides of a string.

Usage

```
str_trim2(x, whitespace = "[ \t\r\n]")
```

Arguments

`x`	(`character()`) The character vector to remove left and right side whitespace.
`whitespace`	(`character(1)`; default `"[ \t\r\n]"`) The regular expression used to match whitespace. The function treats the whole expression as one unit and removes one or more matches of it at each end. For example, `whitespace = "a|b"` removes `"aab"` from the start of `"aabx"`.

Details

`NULL` is returned as `character(0L)`.

Value

`character()`.

See Also

`base::trimws()`

Examples

```
#--------------------------------------------------------------------------
# str_trim2() examples
#--------------------------------------------------------------------------
library(bkbase)

x <- " example "
str_trim2(x)

str_trim2(c(" a ", "\tb\n", NA))
str_trim2("--x--", whitespace = "-")
```

Index

∗ **rowwise**
    rowCumsums, 66
    rowMaxs, 67
    rowMaxs2, 69
    rowMeans2, 70
    rowSums2, 72
%==%, 3, *5*
%in2%, 4, *4*

all2, 5
all2(), *9*
and, 7
any2, 8
any2(), *6*
as.character(), *11*
as.matrix(), *66*
as_na, 10
as_na(), *64*

base::%in%, *5*
base::all(), *5, 6*
base::any(), *9*
base::anyNA(), *27*
base::as.character(), *24*, *65*
base::as.numeric(), *65*
base::Comparison, *4*
base::cumsum(), *66, 67*
base::deparse(), *13, 14*
base::deparse1(), *13, 14*, *19*
base::inverse.rle(), *63*
base::is.atomic(), *26*
base::is.integer(), *42–44*, *47–49*
base::is.na(), *68, 69*, *71*, *73*
base::Logic, *8*, *58*
base::match(), *5*, *27*
base::match.arg(), *53, 54*
base::match.call(), *54, 55*
base::mean(), *15*
base::NaN, *66*
base::ncol(), *17*
base::nrow(), *17, 18*
base::rle(), *63*
base::round(), *66*
base::rowMeans(), *71*
base::rowSums(), *73*
base::sample(), *60*
base::seq(), *74*
base::seq.int(), *74*
base::seq_along(), *74*
base::seq_len(), *74*
base::sprintf(), *66*
base::strwrap(), *21*
base::suppressMessages(), *59*
base::sys.parent(), *55*
base::table(), *23*
base::tabulate(), *22, 23*
base::toString(), *12*
base::trimws(), *75*
base::tryCatch(), *55*
base::typeof(), *26*

csw, 10
csw(), *19*

deparse(), *19*
dots, 12
dots_char *(*dots*)*, 12

fdeparse, 13
fformula, 14
fmean, 15
fmedian, 16
fncol, 17
fnrow, 17
format(), *11*
format_value, 18
format_value(), *53*
fsd, 19
fstr_wrap, 20
fstr_wrap(), *12*

fstrwrap *(fstr_wrap)*, 20
ftabulate, 22
fvar, 23

glue2, 24
glue::glue(), *24, 25*

is_atomic, 26
is_const, 27
is_date, 28
is_date_or_datetime, 29
is_datetime, 29
is_factor, 30
is_formula, 31
is_monotonic_decreasing, 31
is_monotonic_increasing, 32
is_probability, 33
is_scalar_character, 34
is_scalar_date, 35
is_scalar_date_or_datetime, 36
is_scalar_datetime, 36
is_scalar_double, 37
is_scalar_factor, 38
is_scalar_integer, 39
is_scalar_logical, 40
is_scalar_numeric, 40
is_scalar_probability, 41
is_scalar_whole_number, 42
is_scalar_whole_number(), *44*, *48*
is_scalar_whole_numeric, 43
is_scalar_whole_numeric(), *42, 43*, *49*
is_strict_decreasing, 45
is_strict_increasing, 46
is_string, 47
is_whole_number, 47
is_whole_number(), *43*, *49*
is_whole_numeric, 49
is_whole_numeric(), *44*, *47, 48*

list_clean, 50
list_flatten2, 51
logical_to_integer, 52

match.arg2, 53
match.arg2(), *19*
match.call2, 54
middle, 56

or, 57

quiet, 59

remove_rownames *(reset_rownames)*, 61
resample, 60
reset_rownames, 61
rle2, 62
rm_na, 64
rmna *(rm_na)*, 64
round2, 65
round2(), *11*
rowCumsums, 66
rowCumsums(), *68*, *70, 71*, *73*
rowMaxs, 67
rowMaxs(), *67*, *70, 71*, *73*
rowMaxs2, 69
rowMaxs2(), *67, 68*, *71*, *73*
rowMeans2, 70
rowMeans2(), *67, 68*, *70*, *73*
rowSums2, 72
rowSums2(), *67, 68*, *70, 71*

seq2, 74
stats::formula(), *14, 15*
stats::median(), *16*, *57*
stats::na.omit(), *64*
stats::sd(), *20*
stats::var(), *23*
str_trim2, 75
structure(), *19*