Package 'suppr'

September 7, 2026

Title Supplementary Idiomatic Utilities and Extensions

Version 1.0.1

Description Miscellaneous supplementary functions designed to follow idiomatic 'R' conventions. Some functions are simple wrappers that reduce repetitive code, while others address common tasks or extend existing 'R' functions.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 8.0.0

Suggests rlang, testthat (>= 3.0.0)

Config/testthat/edition 3

Imports methods, stats

Depends R (>= 4.0.0)

URL https://lj-jenkins.github.io/suppr/, https://github.com/LJ-Jenkins/suppr

BugReports https://github.com/LJ-Jenkins/suppr/issues

NeedsCompilation yes

Author Luke Jenkins [aut, cre, cph] (ORCID: <https://orcid.org/0000-0002-7206-7242>)

Maintainer Luke Jenkins <luke-jenkins-dev@outlook.com>

Repository CRAN

Date/Publication 2026-09-07 08:10:02 UTC

Contents

`addClass` *Add a Class to an Object*

Description

Add a class to an object, either appending to, or replacing, the existing class vector.

Usage

```
addClass(x, class, prepend = TRUE)
```

Arguments

`x`	an **R** object.
`class`	character vector, or object coercible to character, of one or more class names to add.
`prepend`	logical. If `TRUE` the new class(es) are prepended to the existing class vector. If `FALSE`, the class vector is replaced entirely.

Details

With prepend = TRUE, addClass() is equivalent to:

```
class(x) <- c(class, class(x))
x
```

With prepend = FALSE:

```
class(x) <- class
x
```

Value

The input object x with its class attribute updated.

See Also

class, inherits, is.

Examples

```
x <- structure(1, class = "base_class")
y <- addClass(x, c("new_class", "another_class"))
class(y)

y <- addClass(x, c("new_class", "another_class"), prepend = FALSE)
class(y)
```

---

anyZchar *Are any zero char or all whitespace elements present?*

Description

Tests if a character vector contains any zero character or all whitespace elements.

Usage

```
anyZchar(x)

anyWS(x, zchar = FALSE)
```

Arguments

x	a character vector.
zchar	if TRUE then zero character elements will be treated as all whitespace.

Details

anyZchar() and anyWS() return an index immediately when encountering a zero character or a whitespace element, respectively.

Whitespace elements are defined as any character that is a space, horizontal tab, carriage return or newline, aka "[\t\r\n]". See trimws for more details and how to instead match all unicode whitespace.

anyZchar() implements the same definition of a zero char as nzchar.

NA_character_ is not considered zero char or all whitespace.

Value

an integer or real vector of length one with value the 1-based index of the first zero char/all whitespace value if any, otherwise 0.

Note

Unlike the similar base functions, these do not coerce the input to character.

See Also

nzchar, trimws, anyNF

Examples

```
anyZchar(c("hi", "bye", " ", ""))
anyWS(c("hi", "bye", " ", ""))
anyWS(c("hi", "bye", "who", ""), zchar = TRUE)
```

---

bckQuote *Backquote Text*

Description

Backquote text by combining with backticks.

Usage

```
bckQuote(x)
```

Arguments

x	an **R** object, to be coerced to a character vector.

Value

A character vector of the same length as x (after any coercion).

See Also

Quotes, dQuote and sQuote.

Examples

```
bckQuote("example")
bckQuote(c("one", "two", "three"))
bckQuote(123)
```

`cat0` *Concatenate and Print with No Separator*

Description

Outputs the objects, concatenating the representations with no separator.

Usage

```
cat0(..., file = "", fill = FALSE, labels = NULL, append = FALSE)
```

Arguments

`...`	**R** objects (see cat for the types of objects allowed).
`file`	a connection, or a character string naming the file to print to. If "" (the default), `cat0` prints to the standard output connection, the console unless redirected by sink.
`fill`	a logical or (positive) numeric controlling how the output is broken into successive lines. If `FALSE` (default), only newlines created explicitly by `"\n"` are printed. Otherwise, the output is broken into lines with print width equal to the option width if fill is `TRUE`, or the value of fill if this is numeric. Linefeeds are only inserted between elements, strings wider than fill are not wrapped. Non-positive fill values are ignored, with a warning.
`labels`	character vector of labels for the lines printed. Ignored if fill is `FALSE`.
`append`	logical. Only used if the argument file is the name of file (and not a connection or `"|cmd"`). If `TRUE` output will be appended to file; otherwise, it will overwrite the contents of file.

Details

All arguments are passed to a call to cat with `sep` set to `""`.

Value

None (invisible `NULL`).

See Also

cat, print, format and paste which concatenates into a string.

Examples

```
cat(letters[1:3])
cat0(letters[1:3])

cat(
  paste(letters, 100 * 1:26),
  fill = TRUE, labels = paste0("{", 1:10, "}:")
)
cat0(
  paste(letters, 100 * 1:26),
  fill = TRUE, labels = paste0("{", 1:10, "}:")
)
```

---

collapse *Collapse a Vector of Strings into a Single String*

Description

Collapse a character vector into a single string, with an optional separator, and for `listing()` an optional conjunction and period, with options to quote the terms.

Usage

```
collapse(..., sep = "", recurse = FALSE)

collapse0(..., sep = "", recurse = FALSE)

listing(x, sep = ", ", conjunction = "and", period = TRUE, quote = NULL)
```

Arguments

`...`	one or more **R** objects, to be converted to character vectors.
`sep`	character string to collapse the terms, not `NA_character_`.
`recurse`	`logical`. If `TRUE`, collapse each argument separately first, before collapsing the results into a single string.
`x`	a character vector from which to create the human readable list.
`conjunction`	character string to use as a conjunction for the last two terms.
`period`	`logical`. If `TRUE`, append a period to the end of the string.
`quote`	`NULL` or a character string indicating the type of quotes to use for quoting the terms. If `NULL`, no quoting is done. If `"single"`, single quotes are used, if `"double"`, double quotes are used, and if `"back"`, backquotes are used.

Details

collapse and collapse0 are simple wrappers for paste0(..., collapse = sep) and paste(..., collapse = sep), with an option to collapse each argument separately first, before collapsing the results.

Value

a single character string.

See Also

paste0, sQuote, dQuote, bckQuote.

Examples

```
collapse(c("a", "b", "c"), "d")
collapse0(c("a", "b", "c"), "d")

collapse(c("a", "b", "c"), "d", sep = ", ")
collapse0(c("a", "b", "c"), "d", sep = ", ")

collapse(c("a", "b"), c("c", "d"), sep = ", ")
collapse(c("a", "b"), c("c", "d"), sep = ", ", recurse = TRUE)

listing(c("a", "b", "c"), sep = ", ", conjunction = "and")
listing(c("a", "b"), conjunction = "or", period = FALSE)

listing(c("a", "b", "c"), quote = "single")
listing(c("a", "b", "c"), quote = "double")
listing(c("a", "b", "c"), quote = "back")
```

---

empty.list *Create an Empty list of Given Length*

Description

For a given length, create an empty list of that length.

Usage

```
empty.list(length = 0L)
```

Arguments

length	integer, specified length of the output list.

Details

This is a simple wrapper of vector("list", length) with a clearer naming convention.

Value

an empty list of given length.

See Also

vector, na.vector

Examples

```
empty.list()
empty.list(5L)
empty.list(length = 3L)
```

`grep-wrappers` *Pattern Matching*

Description

Strongly typed wrappers around `grep` that have the `fixed` and `ignore.case` arguments set internally.

Usage

```
grepf(pattern, x, value = FALSE, useBytes = FALSE, invert = FALSE)

grepi(
  pattern,
  x,
  perl = FALSE,
  value = FALSE,
  useBytes = FALSE,
  invert = FALSE
)
```

Arguments

`pattern`	character string containing a regular expression (or character string for `fixed = TRUE` to be matched in the given character vector. Coerced by as.character to a character string if possible. If a character vector of length 2 or more is supplied, the first element is used with a warning. Missing values are allowed.
`x`	a character vector where matches are sought, or an object which can be coerced by as.character to a character vector. Long vectors are supported.
`value`	logical. If `FALSE`, a vector containing the (integer) indices of the matches determined by grep is returned, and if `TRUE`, a vector containing the matching elements themselves is returned.
`useBytes`	logical. If `TRUE` the matching is done byte-by-byte rather than character-by-character.

`invert`	logical. If `TRUE` return indices or values for elements that do not match.
`perl`	logical. Should Perl-compatible regexps be used?

Details

`*f()` suffixed functions have `fixed = TRUE` and conflicting arguments (`perl` and `ignore.case`) set as `FALSE`.
`*i()` suffixed functions have `ignore.case = TRUE` and the conflicting `fixed` argument set as `FALSE`.

For full documentation of the wrapped functions, see the help pages for `grep`.

Value

with `value = FALSE`, return a vector of the indices of the elements of `x` that yielded a match (or not, for `invert = TRUE`). This will be an integer vector unless the input is a long vector, when it will be a double vector.

with `value = TRUE`, return a character vector containing the selected elements of `x` (after coercion, preserving names but no other attributes).

See Also

Other pattern-match-replacement-wrappers: `grepl-wrappers`, `grepv-wrappers`, `sub-wrappers`

Examples

```
grepf("foo", c("foo", "Foo", "bar"))
grepf("foo", c("foo", "Foo", "bar"), value = TRUE)
grepi("foo", c("foo", "Foo", "bar"))
grepi("foo", c("foo", "Foo", "bar"), value = TRUE)
```

---

grepl-wrappers *Pattern Matching*

Description

Strongly typed wrappers around `grepl` that have the `fixed` and `ignore.case` arguments set internally (as well as the conflicting arguments).

Usage

```
greplf(pattern, x, useBytes = FALSE)

grepli(pattern, x, perl = FALSE, useBytes = FALSE)
```

Arguments

pattern	character string containing a regular expression (or character string for fixed = TRUE to be matched in the given character vector. Coerced by as.character to a character string if possible. If a character vector of length 2 or more is supplied, the first element is used with a warning. Missing values are allowed.
x	a character vector where matches are sought, or an object which can be coerced by as.character to a character vector. Long vectors are supported.
useBytes	logical. If TRUE the matching is done byte-by-byte rather than character-by-character.
perl	logical. Should Perl-compatible regexps be used?

Details

*f() suffixed functions have fixed = TRUE and conflicting arguments (perl and ignore.case) set as FALSE.
*i() suffixed functions have ignore.case = TRUE and the conflicting fixed argument set as FALSE.

For full documentation of the wrapped functions, see the help pages for grep.

Value

logical vector (match or not for each element of x).

See Also

Other pattern-match-replacement-wrappers: grep-wrappers, grepv-wrappers, sub-wrappers

Examples

```
greplf("foo", c("foo", "Foo", "bar"))
grepli("foo", c("foo", "Foo", "bar"))
```

---

grepv-wrappers *Pattern Matching*

Description

Strongly typed wrappers around grep that have the value (see note), ignore.case and fixed arguments set internally (as well as the conflicting arguments).

If grepv is found in the current **R** version, it is reexported from base.

Usage

```
grepv(
  pattern,
  x,
  ignore.case = FALSE,
  perl = FALSE,
  value = TRUE,
  fixed = FALSE,
  useBytes = FALSE,
  invert = FALSE
)

grepvf(pattern, x, useBytes = FALSE, invert = FALSE)

grepvi(pattern, x, perl = FALSE, useBytes = FALSE, invert = FALSE)
```

Arguments

pattern	character string containing a regular expression (or character string for fixed = TRUE to be matched in the given character vector. Coerced by as.character to a character string if possible. If a character vector of length 2 or more is supplied, the first element is used with a warning. Missing values are allowed.
x	a character vector where matches are sought, or an object which can be coerced by as.character to a character vector. Long vectors are supported.
ignore.case	logical. if FALSE, the pattern matching is case sensitive and if TRUE, case is ignored during matching.
perl	logical. Should Perl-compatible regexps be used?
value	logical. If FALSE, a vector containing the (integer) indices of the matches determined by grep is returned, and if TRUE, a vector containing the matching elements themselves is returned.
fixed	logical. If TRUE, pattern is a string to be matched as is. Overrides all conflicting arguments.
useBytes	logical. If TRUE the matching is done byte-by-byte rather than character-by-character.
invert	logical. If TRUE return indices or values for elements that do not match.

Details

*f() suffixed functions have fixed = TRUE and conflicting arguments (perl and ignore.case) set as FALSE.
*i() suffixed functions have ignore.case = TRUE and the conflicting fixed argument set as FALSE.

For full documentation of the wrapped functions, see the help pages for grep.

Value

character vector containing the selected elements of x (after coercion, preserving names but no other attributes).

grepv() can also return a vector of the indices of the elements of x that yielded a match (or not, for invert = TRUE) if value = FALSE (see note). This will be an integer vector unless the input is a long vector, when it will be a double vector.

Note

suppr wrappers of the base grep/sub functions remove the arguments that relate to the strong typing, just like grepl does by not having a value argument. However, the base grepv implementation has kept the value argument, so the version here keeps the value argument for compatibility.

See Also

Other pattern-match-replacement-wrappers: grep-wrappers, grepl-wrappers, sub-wrappers

Examples

```
grepv("foo", c("foo", "Foo", "bar"))
grepvf("foo", c("foo", "Foo", "bar"))
grepvi("foo", c("foo", "Foo", "bar"))
```

---

is.datetype *Is object of a Date type?*

Description

Tests if an object inherits from some/all of **R**'s date types: Date, POSIXt, POSIXct and POSIXlt.

Usage

```
is.datetype(x)

is.Date(x)

is.POSIXt(x)

is.POSIXct(x)

is.POSIXlt(x)
```

Arguments

x	an **R** object.

Details

`is.datetype` checks if the object inherits from either `Date` or `POSIXt`.

`is.Date` checks if the object inherits from `Date`.

`is.POSIXt` checks if the object inherits from `POSIXt`.

`is.POSIXct` checks if the object inherits from `POSIXct`.

`is.POSIXlt` checks if the object inherits from `POSIXlt`.

Value

`TRUE` or `FALSE`.

See Also

as.Date, as.POSIXct, as.POSIXlt.

Examples

```
a <- "2020-01-01"
b <- as.Date("2020-01-01")
c <- 123

is.Date(a)
is.Date(b)
is.Date(c)

is.Date(as.POSIXct(a))
is.POSIXt(as.POSIXct(a))
is.POSIXct(as.POSIXct(a))
is.POSIXlt(as.POSIXct(a))
is.datetype(as.POSIXct(a))
```

`is.even` *Is a Number Even or Odd?*

Description

Show where a numeric input is even or odd.

Usage

```
is.even(x, noparity.na = FALSE)

is.odd(x, noparity.na = FALSE)
```

Arguments

x	numeric (logical, integer or double) vector.
noparity.na	logical, whether values without parity (e.g., NA, NaN, Inf, -Inf, and decimal numbers) should return NA (when noparity.na = TRUE) or FALSE (when noparity.na = FALSE).

Value

logical vector the length of x.

Examples

```
is.even(2)
is.odd(1)

is.even(-5:5)
is.odd(-5:5)

m <- matrix(1:4, nrow = 2, ncol = 2)
is.even(m)
is.odd(m)

# inputs without parity can be handled as NA or FALSE:
x <- c(2.0, 3.0, 2.2, 3.1, NA, Inf, -Inf, NaN)
is.even(x)
is.odd(x)
is.even(x, noparity.na = TRUE)
is.odd(x, noparity.na = TRUE)
```

---

is.nonfinite *Are non-finite values present?*

Description

Tests if a vector contains non-finite values (Inf, -Inf, NaN, or NA).

Usage

```
is.nonfinite(x)

is.nf(x)

anyNF(x)
```

Arguments

x	**R** object to be tested: the default methods handle atomic vectors.

Details

`is.nonfinite()` (and alias `is.nf()`) check for non-finite values, returning a logical vector of the same length as `x`, whereas `anyNF()` returns an index immediately when encountering a non-finite value.

`is.nonfinite()` and `anyNF()` are S3 generics, so custom methods can be defined for different object types.

Similar to `is.finite()` semantics, `is.nonfinite()` returns all `TRUE` for character and raw vectors, and `anyNF()` returns `1L`.

Value

For `is.nonfinite()` and `is.nf()`, a logical vector of the same length as `x`.

For `anyNF()`, an integer or real vector of length one with value the 1-based index of the first non-finite value if any, otherwise `0`.

Note

For character vectors use is.na and anyNA.

See Also

is.finite, is.whole, anyZchar, anyWS

Examples

```
is.nonfinite(1:10)
anyNF(1:10)

is.nonfinite(c(1, 2, NA, 4))
anyNF(c(1, 2, NA, 4))

is.nonfinite(c(1, 2, NaN, 4))
anyNF(c(1, 2, NaN, 4))

is.nonfinite(c(1, 2, Inf, 4))
anyNF(c(1, 2, Inf, 4))

is.nf(c(1, 2, -Inf, 4))
anyNF(c(1, 2, -Inf, 4))
```

---

is.whole *Are vectors whole or integer-like?*

Description

Tests if numeric vectors are integerish, or whole according to a given tolerance.

`is.whole()` and `is.integerish()` check if the entire vector is whole or integerish, whilst `is.wholenumber()` checks element-wise for wholeness.

Usage

```
is.whole(x, tol = .Machine$double.eps^0.5)

is.wholenumber(x, tol = .Machine$double.eps^0.5)

is.integerish(x)
```

Arguments

x	a logical, numeric, or complex vector.
tol	numeric tolerance for wholeness.

Details

`is.integerish()` tests if a vector is integerish by evaluating if the remainder of the absolute value of `x` divided by `1` is 'equal' to `0.0` in the `C` code.

`is.whole()` and `is.wholenumber()` test for wholeness by evaluating if the absolute value of `x` minus its rounded value is less than the given tolerance.

Both `is.integerish()` and `is.whole()` ignore non-finite values (i.e., treat them as integerish/whole) and only test finite values for integerishness or wholeness. If you do not want this behavior, use anyNF, e.g., `!anyNF(x) && is.whole(x)`.

`is.wholenumber()` returns `NA` for non-finite elements.

Value

For `is.whole()` and `is.integerish()` a single `TRUE` or `FALSE`.

For `is.wholenumber()`, a logical vector of the same length as `x`.

See Also

is.integerish, anyNF

Examples

```
is.integerish(1)
is.integerish(1.0)
is.integerish(1.0000000001)
is.integerish(1.0000000000000001)

is.wholenumber(1)
x <- c(1.0, 1.0000001, 1.0000000001)
is.wholenumber(x)
is.whole(x)


# ignores non-finite values:
is.integerish(c(Inf, -Inf, NaN, NA))
is.whole(c(Inf, -Inf, NaN, NA))
```

```
# non-finite values flag as NA:
is.wholenumber(c(Inf, -Inf, NaN, NA))

# all error on non-numeric vector inputs:
try(is.integerish("1"))
try(is.whole("1"))
try(is.wholenumber("1"))
```

---

isVector *Check if Any of Given Vector Types*

Description

Wrapper around is.vector that allows the mode argument to accept a character vector of multiple specific types. TRUE is returned if the given object is **any** of the given types.

Usage

```
isVector(x, mode = "any")
```

Arguments

x	an **R** object.
mode	character string (or chr vector) naming an atomic mode or "list" or "expression" or "any". When using "any", it must be given on its own.

Details

See is.vector for full details and for the types that can be checked with mode.

Value

TRUE or FALSE.

See Also

is.vector, typeof.

Examples

```
x <- c(a = 1, b = 2)
isVector(x) # default `mode` is "any"

# "any" can't be given with other types.
try(isVector(x, mode = c("numeric", "any")))

# `TRUE` is returned if *any* of the types are matched.
isVector(x, mode = c("character", "list", "logical", "numeric"))
isVector(x, mode = c("character", "list", "logical"))
```

`libraries` *Loading/Attaching Multiple Packages*

Description

Wraps `library()` and `require()` to load multiple packages.

Usage

```
libraries(
  ...,
  pos = 2,
  lib.loc = NULL,
  character.only = FALSE,
  logical.return = FALSE,
  warn.conflicts,
  quietly = FALSE,
  verbose = getOption("verbose"),
  mask.ok,
  exclude,
  include.only,
  attach.required = missing(include.only)
)

requires(
  ...,
  lib.loc = NULL,
  quietly = FALSE,
  warn.conflicts,
  character.only = FALSE,
  mask.ok,
  exclude,
  include.only,
  attach.required = missing(include.only)
)
```

Arguments

`...`	the names of the packages, given as names, character strings, a combination of both, or character vectors (see `character.only` below).
`pos`	the position on the search list at which to attach the loaded namespace. Can also be the name of a position on the current search list as given by `search()`.
`lib.loc`	a character vector describing the location of **R** library trees to search through, or `NULL`. The default value of `NULL` corresponds to all libraries currently known to `.libPaths()`. Non-existent library trees are silently ignored.

`character.only` a logical indicating whether package or help can be assumed to be character strings. If `TRUE`, dot arguments are coerced to be character before being unlisted into a single character vector.

`logical.return` logical. If it is `TRUE`, `FALSE` or `TRUE` is returned to indicate success.

`warn.conflicts` logical. If `TRUE`, warnings are printed about conflicts from attaching the new package. A conflict is a function masking a function, or a non-function masking a non-function. The default is `TRUE` unless specified as `FALSE` in the `conflicts.policy` option.

`quietly` a logical. If `TRUE`, no message confirming package attaching is printed, and most often, no errors/warnings are printed if package attaching fails.

`verbose` a logical. If `TRUE`, additional diagnostics are printed.

`mask.ok` character vector of names of objects that can mask objects on the search path without signaling an error when strict conflict checking is enabled.

`exclude, include.only`
character vector of names of objects to exclude or include in the attached frame. Only one of these arguments may be used in a call to `libraries` or `requires`.

`attach.required`
logical specifying whether required packages listed in the `Depends` clause of the `DESCRIPTION` file should be attached automatically.

Details

All non-dot arguments are passed in their entirety to each `library` or `require` call, so the same arguments are used for each package.

For the `help` of a package, call `library(help = <package>)` directly.

For full details see library.

Value

`libraries()` and `requires()` return the value of the last dot argument evaluated, invisibly:

Normally `library` returns (invisibly) the list of attached packages, but `TRUE` or `FALSE` if `logical.return` is `TRUE`. When called as `library()` it returns an object of class `"libraryIQR"`, and for `library(help=)`, one of class `"packageInfo"`.

`require` returns (invisibly) a logical indicating whether the required package is available.

See Also

attach, detach, install.packages

Examples

```
libraries("stats", "graphics", methods)
requires("stats", graphics, "methods")

x <- c("stats", "graphics")
libraries(x, "methods", character.only = TRUE)
try(requires(methods, x, character.only = TRUE))
```

match.argv *Argument Verification*

Description

match.argv() matches a given argument against a list of candidate values as specified by choices.

Usage

```
match.argv(arg, choices, match.fn = NULL)
```

Arguments

arg	an **R** object.
choices	a list of candidate values to match against that will be extracted with [[.
match.fn	a function to use for matching the argument against the choices. This function must take two arguments: the first is the argument to be matched, and the second is a candidate value extracted from choices with [[. It must return a single TRUE or FALSE value indicating whether the argument matches the candidate value. If NULL, the default matching function is used, which checks for equality using identical() with ignore.environment = TRUE.

Details

If arg is the same as choices, then the first element of arg is returned. This check is done using identical() with ignore.environment = TRUE, regardless of the match.fn argument.

In the one-argument form match.argv(arg), the choices are obtained from a default setting for the formal argument arg of the function from which match.argv was called.

Value

If arg is identical to choices, then the first element of arg is returned. If arg is matched against a candidate value in choices, then arg is returned. If no match is found, an error is thrown.

See Also

match.arg, match.call, match.fun.

Examples

```
# default matching function is strict.
match.argv(1:10, list(c("a", "b"), list(1, 2), 1:10))
try(match.argv(1, list(c("a", "b"), list(1, 2), 1:10)))

# NULL can be given as a candidate.
match.argv(NULL, list(NULL, 1:10))
```

```
# default matching function is strict.
try(match.argv(NA, list(NA_real_, NA_integer_, NA_character_)))

# a custom matching function can be used.
match.argv("A", list("a", "b", "c"), match.fn = function(x, y) {
  is.character(x) && length(x) == 1L && tolower(x) == tolower(y)
})
```

`na.refill` *Refill the NAs for a 'na.action' Object*

Description

For a na.action object, use the indices of where NAs were removed to refill the object back to its original size with NAs in the appropriate positions.

Usage

```
na.refill(object, ...)
```

Arguments

`object`	a na.action object (atomic vector, matrix or data.frame).
`...`	further arguments special methods could require.

Details

For objects where na.omit removes whole rows (e.g., matrices, data.frames), the information about that row is lost, so `na.refill` will refill those **entire rows** with NAs.

Value

vector, matrix or data.frame with the indices from `na.action` refilled with NAs. rownames and colnames are preserved.

If the object given is not of those types, or not a `na.action` object, it is returned unchanged.

See Also

na.action, na.omit

Examples

```
x <- c(1, 2, NA, 4, NA, 6)
na_x <- na.omit(x)
na_x
na.refill(na_x)

m <- matrix(1:9, 3, 3)
```

```
m[c(1, 3), 1:2] <- NA
dimnames(m) <- list(c("r1", "r2", "r3"), c("a", "b", "c"))
na_m <- na.omit(m)
na_m
na.refill(na_m) # previous data in NA row is lost

df <- data.frame(x = 1:5, y = c("a", NA, "c", "d", NA))
na_df <- na.omit(df)
na_df
na.refill(na_df)
```

`na.vector` *Create a vector of NA's*

Description

For a given type and length, create a vector of NA's.

Usage

```
na.vector(
  length = 1L,
 type = c("logical", "integer", "double", "character", "complex", "numeric", "list")
)
```

Arguments

length	integer, length of the output vector.
type	character string naming an atomic type that has an equivalent NA value (i.e., not raw), or "list".

Details

This function also offers a "list" type, which gives a list of single (logical) NA values. To initialize an empty list (of NULL's) of a given length, use empty.list instead.

Value

vector of given mode and length filled with NA values.

See Also

vector, whichNA, setNA

Examples

```
na.vector(5L)
x <- na.vector(3L, "character")
class(x)

x <- complex(1:5, 6:10)
y <- na.vector(length(x), typeof(x))
class(y)
length(y)

na.vector(2L, "list")
```

---

predapply *Apply a predicate function over a list or vector*

Description

Returns a logical of the same length as X, each element of which is the result of applying predicate FUN to the corresponding element of X.

Usage

```
predapply(X, FUN, ..., reduce = NULL, na.as = NA)
```

Arguments

X	a vector (atomic or list) or an expression object. Other objects (including classed objects) will be coerced by as.list.
FUN	a predicate function to be applied to each element of X that returns a single logical value.
...	optional arguments to FUN.
reduce	NULL or one of "all", "any", or "none". When non-NULL, the output logical vector is reduced to a single boolean value using all, any or "none" (implemented as all(!logi)).
na.as	logical value to return for NA values in the output logical vector (NA, TRUE or FALSE).

Details

na.as is most meaningful when reduce is non-NULL, as it allows control flow calls (e.g., if (predapply(...))) to proceed without error. See examples.

Value

logical vector or boolean if reduce is non-NULL.

See Also

apply, lapply, mapply, all, any.

Examples

```
x <- list(a = 1, b = 2, c = NA)
predapply(x, is.numeric)
predapply(x, is.numeric, reduce = "any")

x <- list(a = 1, b = 2, c = 3)
predapply(x, is.numeric, reduce = "all")

x <- list(a = "1", b = "2", c = "3")
predapply(x, is.numeric, reduce = "none")

x <- c(1, 2, NA)
predapply(x, function(x) x > 0)
predapply(x, function(x) x > 0, reduce = "all")
predapply(x, function(x) x > 0, reduce = "all", na.as = TRUE)
```

---

repeated *Determine Repeated Elements*

Description

Return a logical vector, indices, or the values of repeated elements in a vector.

Usage

```
repeated(x, ...)

## S3 method for class 'array'
repeated(x, MARGIN = 1L, ...)

whichRepeated(x, ...)

## S3 method for class 'array'
whichRepeated(x, MARGIN = 1L, ...)

repeats(x, ...)

## S3 method for class 'array'
repeats(x, MARGIN = 1L, ...)

non.unique(x, ...)
```

Arguments

x	a vector, a data frame, an array, or NULL.
...	additional arguments passed to methods.
MARGIN	the array margin to be held fixed: see apply, and note that MARGIN = 0 may be useful.

Details

The repeated functions determine which elements of a vector or data frame are duplicates of elements with smaller subscripts. They are very similar in functionality to duplicated but instead mark **all** duplicates (not just those after the first/last occurrence), and do not have an incomparables argument.

These are generic functions with methods for vectors (including lists and expressions), data frames and arrays (including matrices).

The array method calculates for each element of the sub-array specified by MARGIN if the dimensions are identical to those for an earlier or later element (in row-major order). This would most commonly be used to find repeated rows (the default) or columns (with MARGIN = 2). Note that MARGIN = 0 returns an array of the same dimensionality attributes as x.

non.unique() is an alias for repeats().

Value

repeated(): For a vector input, a logical vector of the same length as x. For a data frame, a logical vector with one element for each row. For a matrix or array, and when MARGIN = 0, a logical array with the same dimensions and dimnames.

whichRepeated(): For a vector input, an integer vector giving the indices of the repeated values. For a data frame, an integer vector giving the indices of the repeated rows. For a matrix or array, an integer vector giving the indices of the repeated elements across the margin specified.

repeats() and its alias non.unique(): an object of the same type as x, containing the repeated values (vector input), rows (data frame input), or elements across the margin (matrix/array input).

See Also

duplicated and unique.

Examples

```
# Repeated values in a vector
x <- c(1, 2, 3, 2, 1)

repeated(x)
whichRepeated(x)
repeats(x)

# Repeated rows in a data frame
df <- data.frame(
  x = c(1, 2, 1),
  y = c("a", "b", "a")
```

```
)

repeated(df)
repeats(df)

# Repeated rows/columns in a matrix
m <- cbind(
  c(1, 2),
  c(3, 4),
  c(1, 2)
)

repeated(m)
repeated(m, MARGIN = 2)
repeats(m, MARGIN = 2)

# non.unique() is an alias for repeats()
identical(repeats(x), non.unique(x))
```

`rm.first` *Remove First or Last N Elements*

Description

Remove the first or last `n` elements of an **R** object.

Usage

```
rm.first(x, n = 1L, ...)

## Default S3 method:
rm.first(x, n = 1L, ...)

rm.first(x, ...) <- value

## Default S3 replacement method:
rm.first(x, ...) <- value

rm.last(x, n = 1L, ...)

## Default S3 method:
rm.last(x, n = 1L, ...)

rm.last(x, ...) <- value

## Default S3 replacement method:
rm.last(x, ...) <- value
```

Arguments

x	an **R** object with a [method, e.g., a vector, matrix, list.
n	integer, number of elements to remove from the beginning or end.
...	additional arguments passed to methods.
value	integer, same as n.

Details

The default methods operate on atomic vectors and lists, removing n elements from the beginning or end of the object.

Dimensional objects (matrix, data.frame, array, etc.) are handled by removing n entries along the first dimension ('row-wise' for matrices/data.frames and for each slice of higher-dimensional arrays). Remaining dimensions are preserved.

Value

The modified object with the first or last n elements removed.

Examples

```
x <- 1:10
rm.first(x, 3)
rm.last(x, 3)

x <- matrix(1:10, nrow = 5)
rm.first(x, 2)
rm.last(x, 2)

x <- list(a = 1, b = 2, c = 3, d = 4)
rm.first(x, 2)
rm.last(x, 2)

x <- 1:10
rm.first(x) <- 3
rm.last(x) <- 3
x
```

---

setNA *Set given indices as NA*

Description

For given indices, set those indices of an object as NA.

Usage

```
setNA(x, indices)

setNA(x) <- value
```

Arguments

x	an **R** object.
indices	integer vector of indices to set as NA.
value	integer vector of indices to set as NA for the replacement function.

Details

This function is a S3 generic. The default method sets indices to NA using [<-, passing the indices first to arrayInd if the input has a non-NULL dim attribute.

The setNA<- function is meant to be a direct replacement for is.na<- with (in my opinion) a clearer naming convention. The base methods are implemented verbatim (for factor and numeric_version objects), whereas the default method differs by using arrayInd (see above), whereas is.na<- is implemented just as x[value] <- NA.

Value

the modified object with given indices set to NA.

Note

Complex inputs will have indices set to NA_complex_, meaning that both the real and imaginary parts will be set to NA, not just the real part which can happen (depending on **R** version) with x_complex[indices] <- NA.

See Also

is.na<-, whichNA

Examples

```
setNA(1:5, c(1, 4))
setNA(c("hi", "hello", "bye", "goodbye"), c(1, 4))
setNA(matrix(1:4, 2, 2), c(1, 4))
setNA(list(1, 2, 3, list(1, 2)), c(1, 4))

x <- 1:10
setNA(x) <- c(1, 7, 9)
x
```

---

stop2 *Display Warnings and Errors with Call Information*

Description

Wrappers around stop and warning that enable the call. argument to derive a call from the stack.

Usage

```
stop2(..., call. = TRUE, domain = NULL)

warning2(..., call. = TRUE, domain = NULL)
```

Arguments

...	zero or more objects which can be coerced to character (and which are pasted together with no separator).
call.	call, logical, integer, or environment. logical, indicating if the calling call should become part of the error message with same semantics as stop. integer, specifying how many calls to go 'up' the call stack to extract a call for the message. A value of `0` will give the call to `stop2()` itself, `1` will give the call of the caller, and so on. Numeric values are coerced to integer and absolute values are taken. Values outside either boundary of the call stack will be clamped to the nearest boundary. environment, which will be matched against the calling stack and the corresponding call will be shown.
domain	see gettext. If `NA`, messages will not be translated.

Details

These functions derive a call to be displayed and then construct their own 'simple' conditions using simpleError and simpleWarning.

If a condition object is given as the first argument, it will be treated in the same way as the base functions do, by warning that other arguments will be ignored and then signalling the condition.

See stop and warning for full details.

Value

Called for side effects only.

See Also

stopifnot2 and stopifnot.with for validations with call information.

Examples

```
f1 <- function(call.) stop2("error", call. = call.)
f2 <- function(call.) f1(call. = call.)
f <- function(call.) f2(call. = call.)

try(f(call. = FALSE))
try(f(call. = TRUE))

try(f(call. = 0))
try(f(call. = 1))
try(f(call. = 2))

f <- function() {
```

```
  e <- environment()
  f1(call. = e)
}

try(f())

f1 <- function(call.) warning2("warning", call. = call.)
try(f())
```

`stopifnot.with` *Ensure the Truth of R Expressions in a Data Environment*

Description

Wrapper around stopifnot2 that evaluates **R** expressions in an environment constructed from `data`.

Usage

```
stopifnot.with(data, ..., call. = TRUE)
```

Arguments

`data`	data to use for constructing an environment. This may be an `environment`, a `list`, a `data.frame`, or an `integer` as in `sys.call`.
`...`	any number of **R** expressions, which should each evaluate to (a logical vector of all) `TRUE`. If named, the names will be used in lieu of the default error message.
`call.`	call, logical, integer, or environment. logical, indicating if the calling call should become part of the error message with same semantics as stop. integer, specifying how many calls to go 'up' the call stack to extract a call for the message. A value of `0` will give the call to `stop2()` itself, `1` will give the call of the caller, and so on. Numeric values are coerced to integer and absolute values are taken. Values outside either boundary of the call stack will be clamped to the nearest boundary. environment, which will be matched against the calling stack and the corresponding call will be shown.

Details

If any of the expressions are not all `TRUE`, stop is called, producing an error message indicating the first expression which was not (all) true. See stopifnot and stopifnot2 for full details.

Special care must be taken for handlers on the call stack, as they may affect the call displayed in the error message. In such instances, passing an environment to `call.` may be helpful.

Value

If no errors are thrown, the function returns `data` invisibly.

See Also

stop2 and warning2 for errors and warnings with call information.

Examples

```
try(stopifnot.with(data.frame(x = 1, y = 2), x == y))
try(stopifnot.with(list(x = 1, y = 2), all.equal(x, y)))

f1 <- function(x, ..., call.) stopifnot.with(x, ..., call. = call.)
f2 <- function(x, ..., call.) f1(x, ..., call. = call.)
f <- function(x, ..., call.) f2(x, ..., call. = call.)

x <- list(a = 1, b = 2)
try(f(x, a == b, call. = FALSE))
try(f(x, a == b, call. = TRUE))

try(f(x, a == 1, b < 1, call. = 0))
try(f(x, b > 3, call. = 1))
try(f(x, a != 1, call. = 2))

f <- function(x, ...) {
  e <- environment()
  f1(x, ..., call. = e)
}

try(f(x, a == 1, b < 1))
```

`stopifnot2` *Ensure the Truth of R Expressions with Call Information*

Description

Wrapper around stopifnot that leaves only the `...` argument and adds a `call.` argument that shows a call in the message that is derived from the call stack. `warningifnot()` implements the same functionality but produces a warning instead of an error.

Usage

```
stopifnot2(..., call. = TRUE)

warningifnot(..., warn.all = FALSE, call. = TRUE)
```

Arguments

`...`	any number of **R** expressions, which should each evaluate to (a logical vector of all) `TRUE`. If named, the names will be used in lieu of the default error/warning message.

`call.` call, logical, integer, or environment. logical, indicating if the calling call should become part of the error message with same semantics as stop. integer, specifying how many calls to go 'up' the call stack to extract a call for the message. A value of `0` will give the call to `stop2()` itself, `1` will give the call of the caller, and so on. Numeric values are coerced to integer and absolute values are taken. Values outside either boundary of the call stack will be clamped to the nearest boundary. environment, which will be matched against the calling stack and the corresponding call will be shown.

`warn.all` logical, indicating if all failed expressions should produce warnings, or only the first failed expression. Default is `FALSE`, which means only the first failed expression will produce a warning.

Details

If any of the expressions are not all `TRUE`, stop or warning is called, producing an error/warning message indicating the first (or all, if `warn.all = TRUE` for `warningifnot()`) expression which was not (all) true. See stopifnot for full details.

Special care must be taken for handlers on the call stack, as they may affect the call displayed in the error or warning message. In such instances, passing an environment to `call.` may be helpful.

Value

Called for side effects only.

See Also

stopifnot.with for a data-masked version of this function.

Examples

```
f1 <- function(call.) stopifnot2(1 == 2, call. = call.)
f2 <- function(call.) f1(call. = call.)
f <- function(call.) f2(call. = call.)

try(f(call. = FALSE))
try(f(call. = TRUE))

try(f(call. = 0))
try(f(call. = 1))
try(f(call. = 2))

f <- function() {
  e <- environment()
  f1(call. = e)
}

try(f())

try(stopifnot2(all.equal(1, 2)))
```

```
warningifnot(1 == 2, 2 == 3)
warningifnot(1 == 2, 2 == 3, warn.all = TRUE)
```

---

sub-wrappers *Pattern Replacement Wrappers*

Description

Strongly typed wrappers around sub and gsub that have the fixed and ignore.case arguments set internally (as well as the conflicting arguments).

Usage

```
subf(pattern, replacement, x, useBytes = FALSE)

subi(pattern, replacement, x, perl = FALSE, useBytes = FALSE)

gsubf(pattern, replacement, x, useBytes = FALSE)

gsubi(pattern, replacement, x, perl = FALSE, useBytes = FALSE)
```

Arguments

pattern	character string containing a regular expression (or character string for fixed = TRUE to be matched in the given character vector. Coerced by as.character to a character string if possible. If a character vector of length 2 or more is supplied, the first element is used with a warning. Missing values are allowed.
replacement	a replacement for the matched pattern in sub and gsub. Coerced to character if possible. For fixed = FALSE this can include backreferences "\1" to "\9" to parenthesized subexpressions of pattern. For perl = TRUE only, it can also contain "\U" or "\L" to convert the rest of the replacement to upper or lower case and "\E" to end case conversion. If a character vector of length 2 or more is supplied, the first element is used with a warning. If NA, all elements in the result corresponding to matches will be set to NA.
x	a character vector where matches are sought, or an object which can be coerced by as.character to a character vector. Long vectors are supported.
useBytes	logical. If TRUE the matching is done byte-by-byte rather than character-by-character.
perl	logical. Should Perl-compatible regexps be used?

Details

*f() suffixed functions have fixed = TRUE and conflicting arguments (perl and ignore.case) set as FALSE.
*i() suffixed functions have ignore.case = TRUE and the conflicting fixed argument set as FALSE.

For full documentation of the wrapped functions, see the help pages for grep.

Value

a character vector of the same length and with the same attributes as `x` (after possible coercion to character). Elements of character vectors `x` which are not substituted will be returned unchanged (including any declared encoding if `useBytes = FALSE`). If `useBytes = FALSE` a non-ASCII substituted result will often be in UTF-8 with a marked encoding (e.g., if there is a UTF-8 input, and in a multibyte locale unless `fixed = TRUE`). Such strings can be re-encoded by `enc2native`. If any of the inputs is marked as "bytes", elements of character vectors `x` which are substituted will be returned marked as "bytes", but the encoding flag on elements not substituted is unspecified (it may be the original or "bytes"). If none of the inputs is marked as "bytes", but `useBytes = TRUE` is given explicitly, the encoding flag is unspecified even on the substituted elements (it may be "bytes" or "unknown", possibly invalid in the current encoding). Mixed use of "bytes" and other marked encodings is discouraged, but if still desired one may use `iconv` to re-encode the result e.g. to UTF-8 with suitably substituted invalid bytes.

See Also

Other pattern-match-replacement-wrappers: `grep-wrappers`, `grepl-wrappers`, `grepv-wrappers`

Examples

```
subf("foo", "X", c("foo", "Foo", "bar"))
subi("foo", "X", c("foo", "Foo", "bar"))

gsubf("foo", "X", c("foo foo", "Foo", "bar"))
gsubi("foo", "X", c("foo foo", "Foo", "bar"))
```

---

suppr-dots *Process the ... arguments of a function.*

Description

`dotsNames` returns the names of the `...` arguments, or a character vector of empty strings if they are unnamed (without evaluating `...`).

`subDots` returns the `...` arguments as a list of expressions.

`dp1Dots` returns the `...` arguments as a character vector of deparsed expressions.

`checkDots` errors or warns about extraneous arguments in the `...` of its caller.

Usage

```
dotsNames(...)

subDots(...)

dp1Dots(..., collapse = " ", width.cutoff = 500L)

checkDots(..., error = TRUE, which.call = -1, allowed = character(0))
```

Arguments

`...`	"the dots", as passed from the caller.
`collapse`	a string, passed to paste.
`width.cutoff`	integer in `[20, 500]` determining the cutoff (in bytes) at which line-breaking is tried.
`error`	a logical value indicating whether to throw an error (`TRUE`) or a warning (`FALSE`) for extraneous arguments.
`which.call`	passed to sys.call. A caller may use `-2` if the message should mention its caller.
`allowed`	character vector of named elements in `...` which are "allowed" and hence do not cause an error or warning.

Details

`dotsNames` is an implementation of allNames for `...` arguments.

`subDots` is a simple wrapper for `as.list(substitute(...()))`.

`dp1Dots` applies deparse1 to each element of `subDots(...)`.

`checkDots` is a variation of chkDots for use in functions that want to error in case of extraneous arguments, not just warn. `checkDots` also shows whether extraneous arguments are named or unnamed, and if unnamed, the message will show the deparsed expressions of the unnamed arguments. See examples.

Value

For `dotsNames`, a character vector of the names of the `...` arguments.

For `subDots`, a list of expressions.

For `dp1Dots`, a character vector of deparsed expressions.

For `checkDots`, `NULL` (invisibly), called for its side effects.

See Also

..., chkDots, stop, warning, substitute, deparse1.

Examples

```
f <- function(...) dotsNames(...)
f(1, 2, mean(1:10))
f(a = 1, b = 2, mean(1:10))

f <- function(...) subDots(...)
f(a = 1, b = 2, mean(1:10))

f <- function(...) dp1Dots(...)
f(a = 1, b = 2, mean(1:10))

f <- function(x, ...) checkDots(..., allowed = "b")
f(1, b = 1)
try(f(1, a = 1, b = 2, mean(1:10)))
```

suppr-helpers *Miscellaneous Helper Functions*

Description

`path()` builds a file path using file.path, before normalizing the path with normalizePath.

`dims()` returns the dimensions of an object (from dim), but for objects without dimensions the length of the object is returned along with `0L` (e.g., `c(length(x), 0L)`).

`enumerate` maps a list to each element of a vector, containing the index, value, and name of each element.

Usage

```
path(..., sharedDrive = FALSE, mustWork = NA)

dims(x)

enumerate(x)
```

Arguments

`...`	character vectors. Long vectors are not supported.
`sharedDrive`	logical, whether the path is on a shared drive. If `TRUE`, then `.Platform$file.sep` is given as the first element of the path, e.g., `"mysd"`, `"mydir"` becomes `"\\\\mysd\\mydir"` (on windows).
`mustWork`	logical: if `TRUE` then an error is given if the result cannot be determined; if `NA` then a warning; if `FALSE` then no error or warning is given.
`x`	For `dims()`, an **R** object. For `enumerate()`, a vector or list to be enumerated.

Details

`path()` will expand paths, see example.

Unnamed elements given to `enumerate()` will have an empty string (`""`) as their name.

Value

For `path()`, a character string of the expanded, normalized path.

For `dims()`, an integer of length 2 or greater that specifies the dimensions of an object.

For `enumerate()`, a list of lists, where each inner list has three elements: `idx`, `val`, and `name`, which are the index, value, and name of the corresponding element.

Examples

```
path("path", "expansion", "occurs", mustWork = FALSE)
path("mysd", "mydir", sharedDrive = TRUE, mustWork = FALSE)

# objects with dimensions give same output as dim():
dims(matrix(1:6, nrow = 2))

# those without give length and 0:
dims(1:5)
dims(list(a = 1, b = 2, c = 3))

for (x in enumerate(c(a = 1, b = 2, 3))) print(x)
```

`suppr-infix` *Infix Operator Helpers*

Description

Infix operators for common tasks.
`%''%` and `%""%` return the right-hand side if the left-hand side is an empty string (`""`).
`%!||%` returns the right-hand side if the left-hand side is not `NULL`.
`%0%` returns the right-hand side if the left-hand side has length 0.
`%allin%` returns `TRUE` if all elements of `x` are in `table`.
`%anyin%` returns `TRUE` if any elements of `x` are in `table`.
`%nonein%` returns `TRUE` if none of the elements of `x` are in `table`.
`%onein%` returns `TRUE` if exactly one element of `x` is in `table`.
`%notin%` returns `TRUE` for elements of `x` that are not in `table`. This is implemented in the same way as base **R** and will be replaced by the base version in the **R** versions that have it.

Usage

```
lhs %''% rhs

lhs %""% rhs

lhs %!||% rhs

lhs %0% rhs

x %allin% table

x %anyin% table

x %nonein% table

x %onein% table

x %notin% table
```

Arguments

lhs	left-hand side object.
rhs	right-hand side object.
x	vector or NULL: the values to be matched. Long vectors are supported.
table	vector or NULL: the values to be matched against. Long vectors are not supported.

Details

The %*in% operators follow the semantics of %in% for NULL values:

- Singular NULL's on the lhs always returns a length 0 logical vector.
- NULL elements are not considered equal to singular NULL's so will give a TRUE for %nonein% and %notin% and FALSE for %allin%, %anyin% and %onein%.

See %in% semantics:

```
NULL %in% NULL
#> logical(0)
NULL %in% list(1, NULL)
#> logical(0)
list(1, NULL) %in% NULL
#> [1] FALSE FALSE
list(1, NULL) %in% list(1, NULL)
#> [1] TRUE TRUE
```

Value

For the non-%*in% operators, either the left-hand side or right-hand side, depending on the result of the operator.

For %allin%, %anyin%, %nonein%, and %onein% a single logical value (or empty logical if x has length 0) is returned. For %notin%, a logical vector of the same length as x is returned.

Examples

```
"" %''% "default" # if lhs is "", return rhs

NULL %!||% "default" # if lhs is NULL, return *lhs* (NULL)
# useful for when using NULL as a default value
# or when using elements of a list that may be NULL
# e.g.,
lst <- list(a = 1, b = 2)
x <- lst$nope %!||% mean(lst$nope) # returns NULL

integer(0) %0% 5L # if lhs is length 0, return rhs

c(1, 2, 3) %allin% c(1, 2, 3, 4) # TRUE

c(1, 2, 3) %anyin% c(1, 2, 3, 4) # TRUE
```

```
c(1, 2, 3) %nonein% c(4, 5, 6) # TRUE

c(1, 2, 3) %onein% c(1, 4, 5) # TRUE

c(1, 2, 3) %notin% c(4, 5, 6) # TRUE
```

---

suppr-predicates *Value Predicates for Common Object Types*

Description

is.boolean checks if an object is a single (non-NA) logical value (TRUE or FALSE).

is.string checks if an object is a single (non-NA) character string.

nzstring checks if an object is a single (non-NA) non-empty character string.

Usage

```
is.boolean(x)

is.string(x)

nzstring(x)
```

Arguments

x	an object to be tested.

Details

The string helpers differ slightly from what you *may* expect from base **R** as NA_character_ is not considered a string. For example:

```
nzchar(NA_character_)
#> [1] TRUE
is.string(NA_character_)
#> [1] FALSE
nzstring(NA_character_)
#> [1] FALSE
```

Value

TRUE or FALSE.

Examples

```
is.boolean(TRUE)
is.boolean(NA)

is.string("hello")
is.string(NA_character_)
is.string("")

nzstring("")
```

`whichMin` *Where is the min or max?*

Description

Determines the location (i.e., index) of the (first, last, or all) minima or maxima of a numeric (or logical) vector.

Usage

```
whichMin(x, loc = c("first", "last", "all"))

whichMax(x, loc = c("first", "last", "all"))

whichMin(x, loc = c("first", "last", "all")) <- value

whichMax(x, loc = c("first", "last", "all")) <- value
```

Arguments

`x`	numeric (logical, integer or double) vector or an **R** object for which the internal coercion to double works whose min or max is searched for.
`loc`	`'first'`, `'last'` or `'all'` to specify which index/indices to return.
`value`	value/s to replace the min or max values with when using the assignment functions.

Value

integer of indices. For the assignment functions, the modified object with min or max values replaced by `value`.

Note

For `logical` vectors, `which(x)` is faster than `whichMax(x, loc = "all")`, but `whichMin(x, loc = "all")` can be faster than `which(!x)` for medium to large vectors due to not having the performance cost of negating the vector.

See Also

which, which.max and which.min

Examples

```
x <- c(1:4, 0:5, 11, 1:4, 0:5, 11)
whichMin(x)
whichMin(x, loc = "last")
whichMax(x)
whichMax(x, loc = "all")

# it *does* work with NA's present, by discarding them:
presidents[1:30]
whichMin(presidents) # 28
whichMax(presidents) #  2

# Find the first occurrence, i.e. the first TRUE, if there is at least one:
x <- rpois(10000, lambda = 10)
x[sample.int(50, 20)] <- NA
# where is the first value >= 20 ?
whichMax(x >= 20)
whichMax(x >= 20, loc = "last")
whichMax(x >= 20, loc = "all")

# objects are coerced to numeric vectors if possible:
whichMin(list(A = 7, pi = pi)) ##  ->  c(pi = 2L)

x <- 1:4
whichMin(x) <- 999
whichMax(x) <- -999
x
```

`whichNA` *Which indices are NA?*

Description

Give the indices of `NA` values, allowing for array indices. Use the assignment function `whichNA<-` to replace `NA` values with a given value/s.

Usage

```
whichNA(x)

whichNA(x) <- value
```

Arguments

`x`	numeric **R** object. Does not accept `complex` or `raw`.
`value`	value/s to replace `NA` values with when using the assignment function.

Details

whichNA<- follows **R**'s usual recycling rules when replacing NA values with value. If there are no NA values in x, no replacement is made.

Value

If using whichNA, integer vector of indices of NA values in x, or matrix of array indices if arr.ind is TRUE. If using whichNA<-, the modified object with NA values replaced by value.

See Also

is.na, setNA, whichMin, whichMax

Examples

```
x <- c(1, NA, 3, NA, 5)
whichNA(x)
whichNA(x) <- 0
x

m <- matrix(c(1, NA, 3, NA, 5, NA), nrow = 2)
whichNA(m)

y <- c("a" = 1, "b" = NA, "c" = 2, "d" = NA, "e" = NA, "f" = NA)
whichNA(y)
whichNA(y) <- c(91, 92) # value recycled to number of NA's
y
```

Index