Package 'bedrock'

October 16, 2026

Title Base Functions for the 'DescToolsX' Ecosystem

Version 0.1.20

Description Provides the low level utilities on which the 'DescToolsX' ecosystem is built. Covered are data manipulation and reshaping, predicates for data inspection and validation, vector and string operations, handling of labels and metadata, and routines from number theory and combinatorics. All functions share a common naming and argument scheme and are implemented as S3 generics wherever several input types are meaningful, with performance critical parts written in C++. The package is self contained and can be used on its own, independently of the higher level packages of the suite.

Depends R (>= 4.4.0)

License GPL (>= 2)

Encoding UTF-8

LinkingTo Rcpp

Imports Rcpp, abind, expm, tools, data.table, readxl, httr, cli

Suggests testthat (>= 3.0.0), haven, R.rsp, readr, tibble

Config/testthat/edition 3

VignetteBuilder R.rsp

LazyData true

URL https://andrisignorell.github.io/bedrock/, https://github.com/AndriSignorell/bedrock/

BugReports https://github.com/AndriSignorell/bedrock/issues

Config/roxygen2/version 8.1.0

NeedsCompilation yes

Author Andri Signorell [aut, cre] (ORCID: <https://orcid.org/0000-0003-4311-1969>),
R Core Team [ctb],
Hans W. Borchers [ctb],

Daniel Chessel [ctb],
Nicholas Cooper [ctb],
Stephane Dray [ctb],
Martin Elff [ctb],
Michael Friendly [ctb],
Friedrich Leisch [ctb],
Thomas Lumley [ctb],
Martin Maechler [ctb],
Nick Sabbe [ctb],
Karline Soetaert [ctb],
Terry Therneau [ctb],
Kevin Ushey [ctb],
Jeremy VanDerWal [ctb],
Bill Venables [ctb],
Gregory R. Warnes [ctb],
Daniel Wollschlaeger [ctb],
Thomas W. Yee [ctb]

Maintainer Andri Signorell <andri@signorell.net>

Repository CRAN

Date/Publication 2026-10-16 10:10:02 UTC

Contents

allDuplicated 5
allIdentical 6
appendEnum 7
appendRowNames 9
appendX 10
applySides 11
as.array.xtabs 12
asBinary 13
asCDateFmt 14
auditNames 16
between-operators 17
binaryTree 20
buildPath 21
callIf 22
Cards 23
char-ascii-conversion 24
checkConfLevel 26
checkCount 27
checkFlag 28
checkString 29
closest 30
coalesceX 32
collapseTable 33
columnWrap 35

combLevels 36
combN 37
combPairs 38
combSet 39
compareDataFrames 40
completeColumns 41
concepts 42
countCompCases 43
crossProd 44
crossProdN 45
dataDescription 46
digitSum 47
divisors 48
dotProd 49
dummy 50
extractArgs 52
factorize 53
fibonacci 54
findDownload 55
flags 56
funArgs 57
funCalls 58
funKeywords 59
funList 60
GCD-LCM 61
getDotsArg 62
intervals 63
isDichotomous 65
isEuclid 66
isFilePath 67
isLowCardinality 68
isNA 68
isNumeric 69
isOdd 71
isPrime 71
isUrl 73
isWholeLike 73
isZero 75
label 76
linScale 77
locf 78
logit 79
long-wide-reshape 81
mergeArgs 82
mGsub 83
midx 84
moveAvg 85
mReplace 87

multMerge . . . 88
naIf . . . 89
naReplace . . . 90
nf . . . 91
numeric-conversions . . . 92
nUnique . . . 94
nz . . . 95
openDataObject . . . 96
pairApply . . . 97
parseSasDatalines . . . 98
pdfManual . . . 99
peekFile . . . 100
percentRank . . . 102
permn . . . 103
Pizza . . . 104
precision . . . 106
primes . . . 108
printCharMatrix . . . 109
ptInPoly . . . 110
quot . . . 111
randGroupSplit . . . 113
range-operators . . . 114
rankX . . . 115
rBetaShape . . . 116
rdLabels . . . 118
rdTitle . . . 119
readCourseData . . . 120
readDownload . . . 121
recodeX . . . 122
recycle . . . 125
renameX . . . 126
resolveContingency . . . 128
resolveFormula . . . 130
resolveGroups . . . 136
revCode . . . 137
revX . . . 139
Roulette . . . 141
roundTo . . . 142
rSum21 . . . 144
sampleX . . . 145
setAttr-removeAttr-keepAttr . . . 146
setLength . . . 147
setNamesX . . . 148
sortX . . . 149
splitAt . . . 152
splitPath . . . 153
splitX . . . 154
stringsAsFactors . . . 156

strSplitToCol . . . 157
strSplitToDummy . . . 158
strX . . . 159
Tarot . . . 160
toBaseR . . . 161
trim . . . 163
type-aliases . . . 164
unirootAll . . . 165
untable . . . 167
unwhich . . . 169
urlExists . . . 170
vRot . . . 171
vShift . . . 172
winsorize . . . 173
withSeed . . . 174

Index **176**

`allDuplicated` *Logical Indicator for All Values Involved in Ties*

Description

The function `duplicated()` returns a logical vector indicating which elements of `x` are duplicates, but it does not flag the first occurrence of subsequently duplicated elements.

Usage

```
allDuplicated(x)
```

Arguments

`x`	a vector of any type.

Details

`allDuplicated` returns a logical vector indicating all elements of `x` that are involved in ties (i.e., have frequency > 1).

Note that `allDuplicated` flags all occurrences of tied values, not only the duplicates beyond the first occurrence.

Consequently, `!allDuplicated(x)` can be used to identify elements of `x` that appear exactly once.

Missing values are considered equal to each other, so multiple `NA`s are flagged as ties. As the function builds on `duplicated()`, it also works for data frames (row-wise) and matrices.

Value

a logical vector of the same length as `x`.

See Also

duplicated() for identifying duplicate elements (excluding first occurrences).
unique() for extracting unique values.
split() for grouping tied values.
table() for counting frequencies.

Other data.equal: allIdentical(), compareDataFrames()

Examples

```
x <- c(1:10, 4:6)

allDuplicated(x)

# Compare with duplicated():
duplicated(x)

# Elements appearing exactly once
x[!allDuplicated(x)]

# Identify and analyse ties
x <- sample(letters[1:10], 20, replace = TRUE)
ties <- split(x, x)

# Number of tied groups
sum(sapply(ties, length) > 1)

# Sizes of tied groups
sizes <- sapply(ties, length)
sizes[sizes > 1]

# Same via table()
tab <- table(x)
tab[tab > 1]
```

---

allIdentical *Test Whether Multiple Objects Are Identical*

Description

Extends identical() to more than two objects. Returns TRUE if all supplied objects are exactly identical, and FALSE otherwise.

Usage

```
allIdentical(...)
```

Arguments

...	objects to compare.

Details

If zero or one object is supplied, the function returns `TRUE`.

Note that the objects themselves are compared, not their elements. So `allIdentical(list(A, B, C))` is `TRUE`, as a single object is trivially identical to itself. Use `do.call(allIdentical, myList)` to compare the elements of a list.

Value

logical scalar.

See Also

`identical()`

Other data.equal: `allDuplicated()`, `compareDataFrames()`

Examples

```
A <- LETTERS[1:5]
B <- LETTERS[1:5]
C <- LETTERS[1:5]
E <- factor(LETTERS[1:5])

allIdentical(A, B, C)        # TRUE
allIdentical(A, B, C, E)     # FALSE

allIdentical(1, 1L)          # FALSE (type matters)
```

---

appendEnum *Add an Enumeration Column*

Description

Prepends (or inserts) a column of enumeration labels – lowercase or uppercase Roman numerals, or Arabic numbers.

Usage

```
appendEnum(
  x,
  type = c("roman-lcase", "roman-ucase", "arabic"),
  suffix = ". ",
  startWith = 1L,
  after = 0L,
  colName = NULL
)
```

Arguments

x	a data.frame or matrix (vectors are coerced via `matrix()`).
type	enumeration style: `"roman-lcase"`, `"roman-ucase"` or `"arabic"`.
suffix	text appended to each enumeration label.
startWith	first enumeration index.
after	position after which the column is inserted (see `appendX()`); default `0L` prepends it.
colName	optional name for the new column; `NULL` (default) leaves it unnamed for matrices, while for data frames a default name (`"V1"`) is used.

Value

x with an additional enumeration column.

See Also

`append()`

Other data.append: `appendRowNames()`, `appendX()`, `multMerge()`

Examples

```
d <- data.frame(x = 1:3, y = c("a", "b", "c"))
appendEnum(d)

appendEnum(d, type = "arabic", suffix = ") ")

# insert after the first column instead of prepending
appendEnum(d, after = 1L, colName = "no")
```

appendRowNames *Append Rownames to a Data Frame or Matrix*

Description

Adds the row names of a data.frame or matrix as a column.

Usage

```
appendRowNames(x, colName = "rowname", after = 0L, removeRowNames = TRUE)
```

Arguments

x	a data.frame or matrix.
colName	name of the new column containing the row names.
after	position after which the column is inserted. Default is 0 (first column).
removeRowNames	logical; if TRUE, existing row names are removed.

Value

an object of the same class as x with the row names added as a column. Note that for matrices the result is coerced to the common mode, so appending (character) row names to a numeric matrix yields a character matrix.

See Also

append()

Other data.append: appendEnum(), appendX(), multMerge()

Examples

```
dd <- data.frame(x = 1:5, y = 6:10, z = LETTERS[1:5],
                 row.names = letters[1:5])
appendRowNames(dd)
```

appendX *Append Elements to Objects*

Description

Generic function to append or insert elements to vectors, matrices, and data frames.

Usage

```
appendX(x, values, after = NULL, ...)

## Default S3 method:
appendX(x, values, after = NULL, ...)

## S3 method for class 'matrix'
appendX(x, values, after = NULL, rows = FALSE, newNames = NULL, ...)

## S3 method for class 'data.frame'
appendX(x, values, after = NULL, rows = FALSE, newNames = NULL, ...)

## S3 method for class 'TOne'
appendX(x, values, after = NULL, rows = TRUE, newNames = NULL, ...)
```

Arguments

x	object to which values are appended.
values	values to insert into x. For matrices with rows = TRUE, values are read row by row.
after	position after which to insert. If NULL, values are appended at the end. Use 0 to prepend.
...	additional arguments.
rows	logical; if TRUE, insert rows instead of columns. Ignored for vectors. Note that the method for TOne objects defaults to rows = TRUE, as appending rows is the typical use case there.
newNames	optional names for the inserted elements: column names when inserting columns, row names when inserting rows. When inserting a column into a data.frame without giving newNames, default names ("V1", "V2", ...) are used.

Value

object of the same class as x.

See Also

append()

Other data.append: appendEnum(), appendRowNames(), multMerge()

Examples

```
# vectors
appendX(1:5, 99, after = 2)

# matrices: insert a column / a row
m <- matrix(1:6, nrow = 2,
            dimnames = list(c("r1", "r2"), c("a", "b", "c")))
appendX(m, c(9, 9), after = 1, newNames = "z")
appendX(m, 7:9, after = 1, rows = TRUE, newNames = "r1b")

# data frames: insert a column / a row
d <- data.frame(a = 1:3, b = 4:6)
appendX(d, 7:9, after = 1, newNames = "z")
appendX(d, list(a = 99, b = 88), after = 0, rows = TRUE)
```

---

applySides *Open One Side of a Confidence Interval*

Description

Clamps a confidence interval to the range of the parameter and opens the side that a one-sided interval leaves free. One implementation for the whole suite, so that every function reports a one-sided bound the same way.

Usage

```
applySides(ci, sides = "two.sided", lo = -Inf, hi = Inf)
```

Arguments

`ci`	numeric vector of length two, the lower and upper bound in that order. `NA` bounds are passed through, and `c(NA, NA)` is accepted although it is logical rather than numeric - that is how an interval which could not be computed is usually written.
`sides`	character string, one of `"two.sided"` (default), `"left"` or `"right"`. It names the side carrying the *finite* bound, so `"left"` corresponds to `alternative = "greater"` in a test. Callers are expected to have resolved the value with `match.arg()` already; an unmatched value is an error rather than a partial match.
`lo`, `hi`	the range of the parameter, not infinities by default in spirit but in signature. See Details.

Details

`sides` names the side carrying the finite bound:

`"left"` the informative bound is the lower one; the upper one is opened to `hi`.

`"right"` the informative bound is the upper one; the lower one is opened to `lo`.

`lo` and `hi` are the parameter's range, not infinities. Most statistics are bounded, so reporting the open side at the boundary is the ordinary case rather than an exception: a correlation opens to ± 1, an association measure in $[0, 1]$ to 0 or 1, Pearson's C to $\sqrt{(m-1)/m}$. Where the parameter really is unbounded, $\pm$`Inf` is passed and the usual half-line comes back. Some statistics need one of each: Cronbach's alpha takes `lo = -Inf` and `hi = 1`, a relative risk `lo = 0` and `hi = Inf`.

The two-sided interval is clamped to $[lo, hi]$ as well, so an interval can never claim a value the statistic cannot take.

Value

a named numeric vector with the elements `lci` and `uci`.

See Also

`checkConfLevel()`, `checkFlag()`

Examples

```
ci <- c(0.12, 0.58)

applySides(ci, "two.sided", lo = 0, hi = 1)
applySides(ci, "left",      lo = 0, hi = 1)   # uci opens to 1
applySides(ci, "right",     lo = 0, hi = 1)   # lci opens to 0

# an unbounded parameter opens to infinity
applySides(c(-1.4, 2.6), "left", lo = -Inf, hi = Inf)

# and one of each: Cronbach's alpha is bounded above only
applySides(c(0.61, 0.94), "right", lo = -Inf, hi = 1)

# the two-sided interval is clamped too
applySides(c(-0.2, 1.3), "two.sided", lo = 0, hi = 1)

# NA bounds survive
applySides(c(NA, NA), "left", lo = -1, hi = 1)
```

---

as.array.xtabs *Coerce* xtabs *Object to Array or Matrix*

Description

Converts an object of class `"xtabs"` to a plain array or matrix by dropping all additional classes such as `"xtabs"` and `"table"`, along with the `call` attribute.

Usage

```
## S3 method for class 'xtabs'
as.array(x, ...)

## S3 method for class 'xtabs'
as.matrix(x, ...)
```

Arguments

x	an object of class "xtabs".
...	ignored.

Value

an array (or matrix in the two-dimensional case) with no additional classes.

See Also

Other data.coerce: toBaseR(), type-aliases

Examples

```
xt <- xtabs(~ cyl + gear, data = mtcars)
class(as.matrix(xt))
# "matrix" "array"
```

---

asBinary *Coerce a Vector to Binary (0/1)*

Description

A unified conversion utility for binary variables. Converts a logical, numeric, factor, or character vector to a binary numeric vector coded as 0 and 1.

Usage

```
asBinary(x, pos = NULL, warn = TRUE)
```

Arguments

x	a logical, numeric, integer, factor, or character vector.
pos	optional positive value. If supplied, observations equal to pos are coded as 1 and all others as 0. Must be one of the observed values or factor levels.
warn	logical. If TRUE (default), a warning is issued when a factor or character vector is coerced to binary without an explicit pos, indicating which value is coded as 1.

Details

For **logical** input, TRUE is mapped to 1 and FALSE to 0.

For **numeric** input, values must already be 0 or 1 (or NA); any other value raises an error.

For **factor** input, the vector must have exactly two levels. By default the second level is coded as 1. Use pos to specify which level should be coded as 1.

For **character** input, the vector must have exactly two distinct non-missing values. By default the alphabetically second value is coded as 1. The same pos logic applies.

Value

a numeric vector of 0s and 1s (and NAs where present in x). For factor and character input, the result carries a "coding" attribute, a named integer vector documenting which original value was mapped to 0 and which to 1.

See Also

Other data.recode: combLevels(), dummy(), mReplace(), nf(), recodeX(), revCode(), stringsAsFactors()

Examples

```
# logical
asBinary(c(TRUE, FALSE, TRUE))

# numeric (already binary)
asBinary(c(0, 1, 1, 0))

# factor: second level coded as 1 by default
asBinary(factor(c("control", "treatment", "control")))

# factor with explicit positive value
asBinary(factor(c("control", "treatment", "control")), pos = "treatment")

# character
asBinary(c("no", "yes", "no", "yes"))

# character with explicit positive value
asBinary(c("F", "U", "F", "U"), pos = "F")
```

---

asCDateFmt *Convert Custom Date Format to strftime Format*

Description

Translates a custom date format string using tokens like yyyy, mm, dd, mmm, etc. into a valid strftime-compatible format string (C-style).

Usage

```
asCDateFmt(fmt)
```

Arguments

fmt	character string. Custom date format.

Details

The function parses the input string sequentially and replaces recognized tokens while leaving all other characters unchanged. This makes it robust to compact formats (e.g. yyyymmdd) and mixed text.

Supported tokens:

- d, dd, ddd, dddd
- m, mm, mmm, mmmm
- y, yy, yyyy

Mapping:

- yyyy -> %Y
- yy, y -> %y
- mm, m -> %m
- mmm -> %b
- mmmm -> %B
- dd -> %d
- d -> %e
- ddd -> %a
- dddd -> %A

Value

character string. A valid strftime format.

Examples

```
asCDateFmt("yyyy-mm-dd")
asCDateFmt("dd.mm.yy")
asCDateFmt("yyyymmdd")
asCDateFmt("mmm d, yyyy")
```

auditNames *Audit the Names of a Package Against the Design Rules*

Description

Checks the exported functions of an installed package, their arguments and the values of their enumerated arguments against the naming rules of the suite. Meant to run as a test in every package, so that a name which breaks a rule is found when it is introduced and not in a review.

Usage

```
auditNames(package, exceptions = NULL)
```

Arguments

package	character string, the name of an installed package.
exceptions	character vector of findings to accept, each written as the key of the result: "fun" for a function name, "fun(arg)" for an argument and "fun(arg = \"value\")" for an enumerated value. An exception that matches nothing is reported in the attribute "unused", so that the list cannot outlive its reasons.

Details

The rules checked, with the label used in the column rule:

camelCase exported functions and arguments are written in lowerCamelCase. A dot is kept in S3 methods, in coercion generics as.<Class> that really dispatch, and in argument names that are part of the shared vocabulary (na.rm, conf.level, sig.level, ...).

acronym only CI, QQ, XY, ECDF, SE and AD keep their capitals inside a name, GCD and LCM as whole names. Every other acronym is written like a word: Rgb, Url, Html. As arguments, FUN, R, N, S and X are capitals by convention.

collision a function whose name is taken by base, stats, graphics, utils or dplyr carries the suffix X. A name counts as taken if it exists there, or if it differs from one only in the case of the whole word (iqr against IQR()). An internal capital makes a name of its own: combN() is not combn().

suffixX and no function carries it without such a collision.

dotExport no function is exported with a leading dot, the graphics helpers of the plotting framework excepted.

forwarded an argument name taken over from another package (ignore.case, useNA, all.inside, ...) is kept only where the value is passed on to another function.

replaced names the rules have replaced: alpha (sig.level), seed (the caller uses set.seed()), pkg (package), fun in an exported function (FUN), dat (data), grp (groups), w (weights), level, color and cols (col), obs and resp (ref), the prefix num for a count (n), the suffix Args for a list of arguments, and g and horizontal in plot functions (groups, horiz).

`enumValue` the values of an enumerated argument are lower case, words joined by a hyphen (`"wald-cc"`). Values that belong to the function they are passed on to, such as `"two.sided"` or the names in p.adjust.methods, are left alone. A value that names an element of the result (`which = "tauB"` returning `$tauB`) follows the rule for results instead; the audit cannot see that and reports it, so such values are listed as exceptions.

Arguments of an S3 method that its generic defines are not checked: `print.foo(x, ...)` and `predict.foo(object, newdata, ...)` are given by the generic. Re-exported functions are skipped altogether.

What the audit cannot see is a word boundary that was never written: `maxlen` and `nlow` are single lower case words to it. Such names are found by reading, not by this function.

Value

a data frame with one row per finding and the columns `package`, `fun`, `arg` (`NA` for a finding on the function name), `rule`, `detail` and `key`, the form in which the finding is named in `exceptions`. It has no rows if the package complies. The attribute `"unused"` holds the exceptions that matched no finding.

See Also

`funArgs()`, `funList()`

Other pkg.funinfo: `funArgs()`, `funCalls()`, `funKeywords()`, `funList()`, `rdLabels()`, `rdTitle()`

Examples

```
res <- auditNames("bedrock")
res[, c("fun", "arg", "rule", "detail")]

# as a test: no findings beyond the documented exceptions, and no
# exception without a finding
accepted <- c("isNA" = "NA is R's own constant, not an acronym")
res <- auditNames("bedrock", exceptions = names(accepted))
attr(res, "unused")
```

`between-operators` *Operators To Check, If a Value Lies Within Or Outside a Given Range*

Description

The between and outside operators are used to check, whether a vector of given values x lie within a defined range (or outside respectively). The values can be numbers, text or dates. Ordered factors are supported.

Usage

```
x %[]% rng

x %[)% rng

x %(]% rng

x %()% rng

x %][% rng

x %](% rng

x %)[% rng

x %)(% rng
```

Arguments

x	a variable with at least ordinal scale, usually a numeric value, but can be an ordered factor or a text as well. Texts would be treated alphabetically.
rng	a vector of two values or a matrix with 2 columns, defining the minimum and maximum of the range for x. If rng is a matrix, x or rng will be recycled. Matrix ranges are supported for numeric (and date) x only.

Details

The "BETWEEN" operators basically combine two conditional statements into one and simplify the query process.
They are merely a wrapper for: `x >= rng[1] & x <= rng[2]`, where the round bracket `(` means *strictly greater (>)* and the square bracket `[` means *greater or equal (>=)*. Numerical values of x will be handled by C-code, which is significantly faster than two comparisons in R (especially when x is huge).

For the matching outside-operator, boundary elements of the corresponding between-range return `FALSE`; that is, they are not considered outside whenever the negated between-operator includes that boundary.

Both arguments, `x` and `rng`, will be recycled to the highest dimension, which is either the length of the vector (`x`) or the number of rows of the matrix (`rng`).
See also the routines used to check, whether two ranges overlap (`overlapSize()`, `distance()`).

The "OUTSIDE" operators are the negations of the corresponding "BETWEEN" operators, matched by *meaning* rather than by mirrored bracket symbols:

- `%][%` negates `%()%` (strictly outside both bounds)
- `%](%` negates `%(]%`
- `%)[%` negates `%[)%`
- `%)(%` negates `%[]%` (strictly outside, both bounds of the between-operator were closed)

Value

a logical vector of the same length as x.

See Also

if(), ifelse()

Other data.interval: intervals, range-operators

Examples

```
x <- 1:9
x %[]% c(3,5)

# outside
x <- 1:9
x %][% c(3,5)

c(x,NA) %[]% c(3,5)

x %(]% c(3,5)

# no result when from > to:
x %[]% c(5,3)
x %(]% c(5,5)

# no problem:
ordered(x) %[]% c(3,5)

# not meaningful:
factor(x) %[]% c(3,5)

# characters
letters[letters %(]% c("d","h")]

# select numbers between 0.4 and 0.5
x <- runif(20)
x %[]% c(0.4, 0.5)

# use it with an ordered factor
x <- ordered(sample(LETTERS, 20), levels = LETTERS)
x %[)% c("G","K")


# use multiple ranges
2 %[]% cbind(1:4,2:5)

# both arguments are recycled
c(2,3) %[]% cbind(1:4,2:5)
```

`binaryTree` *Binary Tree*

Description

Create a binary tree of a given number of nodes `n`. Can be used to organize a sorted numeric vector as a binary tree.

Usage

```
binaryTree(n)
```

Arguments

`n`	integer, size of the tree.

Details

If we index the nodes of the tree as 1 for the top, 2–3 for the next horizontal row, 4–7 for the next, ... then the parent-child traversal becomes particularly easy. The basic idea is that the rows of the tree start at indices 1, 2, 4, ...

`binaryTree(13)` yields the vector `c(8, 4, 9, 2, 10, 5, 11, 1, 12, 6, 13, 3, 7)` meaning that the smallest element will be in position 8 of the tree, the next smallest in position 4, etc.

Value

an integer vector of length n.

Note

Substantially based on code by Terry Therneau, with major extensions and improvements by the package author.

See Also

`pharos::plotBinaryTree`

Other data.order: `revX()`, `sortX()`

Examples

```
binaryTree(12)
```

buildPath *Construct a Normalized File Path*

Description

Safely constructs a file path from a directory and a filename, independent of whether the directory ends with a trailing slash. The resulting path uses forward slashes.

Usage

```
buildPath(dir, filename)
```

Arguments

dir	character string. Directory path.
filename	character string. File name to append to dir.

Details

Trailing slashes (or backslashes) in dir are removed before the components are joined, so buildPath("data", "file.csv") and buildPath("data/", "file.csv") yield the same result.

The path is then passed through normalizePath()with mustWork = FALSE, so paths that do not (yet) exist are allowed. Note that normalizePath resolves existing paths to absolute form, while non-existing paths are returned as constructed (i.e. possibly relative).

Both arguments are vectorized in the usual file.path() manner.

Value

a character string representing the file path.

Note

Converting between forward slashes and backslashes is a frequent necessity – and a hassle – especially in Windows. The cycleSlashes() function in the swissValet package is useful for this purpose.

See Also

Other file.path: findDownload(), isFilePath(), isUrl(), splitPath(), urlExists()

Examples

```
buildPath("data", "file.csv")
buildPath("data/", "file.csv")
```

`callIf` *Conditionally Call a Function*

Description

Conditionally evaluate a function depending on the value of an argument. This is a convenient helper for optional features such as plotting, logging, or callbacks, where the user can enable, disable, or parameterize a function call via a single argument.

Usage

```
callIf(FUN, arg, defaults = NULL, forbidden = NULL, warn = TRUE)
```

Arguments

`FUN`	a function to be called.
`arg`	controls whether and how `FUN` is called: • `FALSE`, `NULL`, or `NA`: `FUN` is not called and `NULL` is returned invisibly. • `TRUE`: `FUN` is called with `defaults` (if provided), or with no arguments. • a fully named list: `FUN` is called with the list elements as arguments. If `defaults` is provided, it is merged with `arg`, where elements of `arg` override those in `defaults`.
`defaults`	a named list of default arguments passed to `FUN` when `arg = TRUE`, or used as a base when `arg` is a list. Default is `NULL`.
`forbidden`	optional character vector of argument names that are not allowed. If any of these appear in `arg`, they are removed before calling `FUN`. A warning is issued unless `warn = FALSE`.
`warn`	logical. If `TRUE` (default), a warning is issued when forbidden arguments are removed.

Details

This function implements a flexible pattern for optional function calls:

- Enable/disable behavior with `TRUE`/`FALSE`
- Customize behavior with a list of arguments
- Provide safe defaults and restrict certain arguments

When merging `defaults` and `arg`, user-supplied arguments take precedence. Unlike `modifyList()`, elements with the value `NULL` are preserved and passed on to `FUN` (so that an explicit `NULL` can be used to reset an argument).

Value

returns the result of `FUN(...)` if called. If `arg` is `FALSE`, `NULL`, or `NA`, returns `NULL` invisibly.

See Also

Other pkg.args: `extractArgs()`, `getDotsArg()`, `mergeArgs()`, `recycle()`

Examples

```
# Simple usage: skip
callIf(message, FALSE)

# Call with defaults
callIf(message, TRUE, defaults = list("Hello world"))

# Call with explicit arguments
callIf(message, list(x = "Hello from callIf"))

# With defaults + override
callIf(plot, list(x = 1:5),
       defaults = list(y = 1:5, type = "l"))

# Forbid arguments
callIf(plot,
       list(x = 1:5, y = 1:5, col = "red"),
       forbidden = "col")

# Typical use case: optional plotting
x <- 1:10
y <- x^2
callIf(plot, TRUE, defaults = list(x, y))
```

Cards *Playing Cards dataset*

Description

A dataset representing a standard deck of playing Cards. Each row corresponds to a single card and includes information such as suit, rank and numerical value.

Usage

```
Cards
```

Format

A data frame with 52 observations and X variables:

card Name of the card.

rank Rank of the card (e.g. 2–10, Jack, Queen, King, Ace).

suit Suit of the card (e.g. hearts, diamonds, clubs, spades).

Details

The dataset can be used for simulations, probability calculations and teaching examples involving discrete outcomes and combinatorics.

The dataset represents a complete deck without jokers. It is suitable for probability experiments, simulations and demonstrations of categorical data.

Source

Simulated data.

See Also

Other datasets: `Pizza`, `Roulette`, `Tarot`, `readCourseData()`

Examples

```
head(Cards)
table(Cards$suit)
```

---

char-ascii-conversion *Character <-> ASCII Conversion*

Description

Convert characters to their numeric character codes and vice versa.

Usage

```
charToAscii(x, output = c("vector", "list"))

asciiToChar(x)
```

Arguments

`x`	for `charToAscii()` a character vector, for `asciiToChar()` an integer vector of character codes (1–255).
`output`	character string specifying the output representation. One of `"vector"` (simplify the result whenever possible, the default) or `"list"` (always return a list). See Details.

Details

`charToAscii()` converts each character in a string to its corresponding numeric code.

`asciiToChar()` converts numeric codes back to characters.

Only values in the range `1:127` belong to the ASCII standard and therefore have the same meaning across all systems. Values `128:255` depend on the current character encoding (for example ISO-8859-1 or Windows-1252) and may produce different characters on different platforms.

Note that `0` (NUL) cannot be represented in R character strings and is therefore not supported.

The `output` argument controls the representation returned by `charToAscii()`:

`"vector"` Simplifies the result whenever possible.

Returns an integer vector if:

- the input consists of a single string, or
- all input strings have length one.

Otherwise, a list of integer vectors is returned.

`"list"` Always returns a list of integer vectors.

Value

- `charToAscii()` returns either an integer vector or a list of integer vectors, depending on `output`.
- `asciiToChar()` returns a character vector.

See Also

`charToRaw()`, `rawToChar()`

Other string.transform: `mGsub()`, `strSplitToCol()`, `strSplitToDummy()`

Examples

```
# basic usage
x <- charToAscii("Silvia")
x

asciiToChar(x)

# multiple strings
charToAscii(c("A", "BC"), output = "list")

# split into individual characters
strsplit(asciiToChar(x), split = NULL)

# comparison with raw representation
charToRaw("Silvia")
```

checkConfLevel *Validate a Confidence Level*

Description

Checks that `conf.level` is a single number in $(0, 1)$, or `NA`. Intended for the confidence-interval functions across the suite, so that all of them accept the same values and refuse the rest with the same message.

Usage

```
checkConfLevel(conf.level, allowNA = TRUE)
```

Arguments

`conf.level`	the value to check.
`allowNA`	logical. `TRUE` (default) accepts `NA`, the suite's way of saying "no interval wanted". Functions whose result always carries an interval - the tests, mostly - pass `FALSE`.

Details

The order of the tests is the point of this function. `NA` is *logical*, so a check that leads with `!is.numeric()` rejects the very default most of these functions carry. And `is.na()` on a vector of length other than one turns the surrounding `if` into the error message, which then talks about the condition instead of the argument. Length first, then type, then range.

`NaN` is excluded explicitly: `is.na(NaN)` is `TRUE`, so without that test a `NaN` would be silently accepted as "no interval wanted".

Value

`conf.level`, invisibly,
so the check can be used in an assignment: `conf.level <- checkConfLevel(conf.level)`.

See Also

`checkFlag()`, `checkCount()`, `checkString()`

Examples

```
checkConfLevel(0.95)
checkConfLevel(NA)

# all of these are rejected:
try(checkConfLevel(c(0.9, 0.95)))   # length
try(checkConfLevel(NULL))           # length
try(checkConfLevel(NaN))            # not a level, and not NA either
try(checkConfLevel(0))              # range is open
```

```
# where the interval is part of the result, NA is refused as well
try(checkConfLevel(NA, allowNA = FALSE))
```

---

checkCount *Validate a Count*

Description

Checks that an argument is a single finite integer, not smaller than `min`. Meant for the many size arguments in the suite - `digits`, `sep`, `width`, `nPerm`, `R` and the like - which are conceptually counts rather than numbers and were previously spelled out by hand wherever they occur.

Usage

```
checkCount(x, min = 0L, name = deparse(substitute(x)))
```

Arguments

`x`	the value to check.
`min`	the smallest admissible value, `0` by default. Pass `1` for the arguments that must be positive, e.g. a width or a number of replicates.
`name`	the argument name to use in the message. Defaults to the expression that was passed.

Details

A whole number stored as a double is accepted, as that is what arithmetic on integers produces and what a user typing `2` supplies. `TRUE` is not, although it would survive `as.integer()`: a flag that reaches a count argument is a mistake, not a shorthand for one.

The order of the tests is the same as in `checkConfLevel()`, length first, then type, then value, so that the message names the argument rather than the condition that failed.

Value

`x`, invisibly.

See Also

`checkConfLevel()`, `checkFlag()`, `checkString()`

Examples

```
sep <- 2
checkCount(sep)

width <- 80
checkCount(width, min = 1)

# all of these are rejected:
try(checkCount(1.5))           # not a whole number
try(checkCount(-1))            # below the default minimum
try(checkCount(TRUE))          # a flag is not a count
```

---

checkFlag *Validate a Logical Flag*

Description

Checks that an argument is a single non-missing TRUE or FALSE. Meant for the many switches in the suite - correct, unbiased, scaled, paired and the like - which were previously either unchecked or checked in three different ways.

Usage

```
checkFlag(x, name = deparse(substitute(x)))
```

Arguments

x	the value to check.
name	the argument name to use in the message. Defaults to the expression that was passed, which is right in the ordinary case checkFlag(correct); supply it explicitly when the caller passes something else, e.g. checkFlag(args$correct, "correct").

Details

NA is rejected on purpose. It is a logical of length one and therefore passes is.logical(), but a flag that is neither on nor off has no meaning for a switch - and it propagates silently, because if (NA) is an error somewhere further down rather than here.

Value

x, invisibly.

See Also

checkConfLevel(), checkCount(), checkString()

Examples

```
correct <- TRUE
checkFlag(correct)

correct <- NA
try(checkFlag(correct))        # "'correct' must be a single ..."
```

---

checkString *Validate a Character String*

Description

Checks that an argument is a single non-missing character string. Meant for the labelling arguments across the suite - `dataName`, captions, axis titles - where a vector or an `NA` would otherwise travel unnoticed into printed output.

Usage

```
checkString(x, name = deparse(substitute(x)))
```

Arguments

`x`	the value to check.
`name`	the argument name to use in the message. Defaults to the expression that was passed.

Details

An optional argument that may also be `NULL` is guarded by the caller, `if (!is.null(dataName)) checkString(dataName)`, rather than by a further argument here: whether the absence of a label is admissible is a decision of the function, not of the check.

The empty string is accepted. It is a legitimate label, and a caller that needs a non-empty one says so itself.

Value

`x`, invisibly.

See Also

`checkConfLevel()`, `checkFlag()`, `checkCount()`

Examples

```
dataName <- "smoking by sex"
checkString(dataName)

# all of these are rejected:
try(checkString(NA_character_))  # a missing label is not a label
try(checkString(c("a", "b")))    # length
try(checkString(42))             # type
```

`closest` *Find the Closest Value*

Description

Find the value(s) in a vector closest to a reference value. Multiple values are returned if ties occur or if duplicate values share the same minimum distance.

Usage

```
closest(x, val, output = "value", na.rm = FALSE)
```

Arguments

`x`	a numeric vector to search in.
`val`	the reference value. May be a vector; see Details.
`output`	character string specifying the output representation. One of `"value"` (return the closest value(s), the default) or `"index"` (return the index position(s) in `x`). May be a vector; recycled to the length of `val`.
`na.rm`	logical. If `TRUE`, `NA` values in `x` are ignored before searching. Default is `FALSE`.

Details

Distance is computed as $|x_i - a|$. Ties are detected via `isZero()` rather than exact equality, which avoids spurious misses due to floating-point representation (e.g. `0.3 - 0.2 != 0.1`).

When `na.rm = TRUE`, `NA` elements are excluded from the search but the original index positions of the remaining elements are preserved, so `output = "index"` always refers to positions in the original `x`.

When `val` or `output` are vectors, each element is processed independently and a list is returned.

Recycling follows standard R rules.

Value

if `val` and `output` are scalar:

- numeric vector if `output = "value"`.
- integer vector if `output = "index"`.

If `val` or `output` are vectors: a list with one element per value of `val`.

Returns `NA` if `x` is empty or all-`NA` (with `na.rm = TRUE`).

See Also

`which()`

Other math.basic: `crossProd()`, `crossProdN()`, `dotProd()`, `roundTo()`, `unirootAll()`

Examples

```
# basic
set.seed(8)
x <- runif(10) * 10

closest(x, 3.1)

sort(x)

y <- sample(10, size = 10, replace = TRUE)

# multiple observations of the same closest value
closest(y, val = 6)

# get the relevant positions
closest(y, val = 6, output = "index")

# two different values having the same distance (tie)
closest(c(2, 3, 4, 5), val = 3.5)

# na.rm preserves original index positions
closest(
  c(NA, 5, 8),
  val = 6,
  output = "index",
  na.rm = TRUE
)  # 2, not 1

# vectorize "a"
closest(c(2, 3, 4, 5), val = c(3.1, 3.9))

# vectorize "output"
closest(
  c(2, 3, 4, 5),
  val = 3.1,
  output = c("value", "index")
```

```
)

closest(
  c(2, 3, 4, 5),
  val = c(3.1, 3.9),
  output = c("value", "index")
)
```

`coalesceX` *Return the First Element Not Being NA*

Description

If several vectors are supplied, the evaluation will be elementwise, resp. rowwise if x is a data.frame or a matrix. The first element of the result is the first non `NA` element of the first elements of all the arguments, the second element of the result is the one of the second elements of all the arguments and so on.
Shorter inputs (of non-zero length) are NOT recycled: if all inputs have length greater than 1, they must have the same length (the function will bark otherwise). If any input has length 1 or 0, all inputs are flattened into a single vector (dropping `NULL`s) and the first valid element is returned, in the manner of a scalar SQL `COALESCE`.
The idea is borrowed from SQL. Might sometimes be useful when preparing data in R instead of in SQL.

Usage

```
coalesceX(..., method = c("is.na", "is.null", "is.finite"), flatten = TRUE)
```

Arguments

`...`	the elements to be evaluated. This can either be a single vector, several vectors of same length, a matrix, a data.frame or a list of vectors (of same length). See examples.
`method`	one out of `"is.na"` (default), `"is.null"` or `"is.finite"`. With `"is.na"`, `Inf` values are treated as valid; with `"is.finite"` they are skipped. `"is.null"` operates on the arguments themselves and returns the first one that is not `NULL`.
`flatten`	logical, defines whether lists are going to be flattened (default `TRUE`).

Value

return a single vector of the first non `NA` element(s) of the given data structure.

See Also

`is.na()`, `is.finite()`

Other vector.na: `isNA()`, `locf()`, `naIf()`, `naReplace()`

Examples

```
coalesceX(c(NA, NA, NA, 5, 3))
coalesceX(c(NA, NULL, "a"))
coalesceX(NULL, 5, 3)

d.frm <- data.frame(matrix(c(
  1, 2, NA, 4,
  NA, NA, 3, 1,
  NaN, 2, 3, 1,
  NA, Inf, 1, 1), nrow=4, byrow=TRUE)
)

coalesceX(d.frm)
coalesceX(as.matrix(d.frm))
coalesceX(d.frm$X1, d.frm$X2, d.frm$X3, d.frm$X4)
coalesceX(d.frm$X1, d.frm$X2, d.frm$X3, d.frm$X4, method="is.finite")
coalesceX(list(d.frm[,1], d.frm[,2]))

# returns the first finite element (skips NA, Inf, NaN)
coalesceX(d.frm, method="is.finite")

# returns the first argument that is not NULL
coalesceX(NULL, NULL, 7, method = "is.null")

# with characters (take care, factors won't work!)
# is.finite does not make sense here...
d.frm <- data.frame(matrix(c(
  "a", "b", NA, "4",
  NA, NA, "g", "m",
  NA_character_,"hfdg", "rr", "m",
  NA, Inf, 1, 1), nrow=4, byrow=TRUE)
, stringsAsFactors = FALSE)

coalesceX(d.frm$X1, d.frm$X2, d.frm$X3, d.frm$X4)
coalesceX(d.frm)
coalesceX(as.list(d.frm))
```

---

collapseTable *Collapse Table Dimensions by Remapping Factor Levels*

Description

Aggregates a `table` or `ftable` object by reassigning the levels of one or more dimensions according to user-supplied mappings, and summing the frequencies within each resulting level combination.

Usage

```
collapseTable(x, ..., strict = TRUE)
```

Arguments

x	a table or ftable object with named dimensions.
...	named or unnamed mapping vectors specifying how levels of each dimension should be collapsed. Each mapping vector must have length equal to the number of levels in the corresponding dimension.
strict	logical (default TRUE). Controls error handling.

Details

Mapping vectors define how factor levels are reassigned. Element i specifies the new label for the i-th original level. Repeated values in a mapping vector cause the corresponding levels to be merged.

Argument matching

- Named arguments are matched to dimensions by name (e.g., age = c("young", "adult")).
- Unnamed arguments are assigned to dimensions in order.
- Mixed usage assigns named arguments first, then remaining unnamed arguments in order.

Constraints

- Each dimension may be specified at most once.
- Mapping vectors must have the same length as the number of levels of the corresponding dimension.
- NA values in mapping vectors are not allowed.

Level ordering The order of the resulting levels follows the first occurrence of each value in the mapping vector.

Error handling

- If strict = TRUE, unknown dimension names result in an error, and positional assignment of unnamed arguments produces a warning.
- If strict = FALSE, unknown dimensions produce a warning (and are ignored), and positional assignment is silent.

Value

a collapsed table object with updated dimensions and aggregated frequencies.

See Also

Other data.reshape: long-wide-reshape, splitAt(), splitX(), untable()

Examples

```
tab <- xtabs(~ gear + cyl, data = mtcars)
tab

# merge the 4- and 6-cylinder categories
collapseTable(tab, cyl = c("le6", "le6", "8"))

# positional assignment (first dimension), silent with strict = FALSE
collapseTable(tab, c("3-4", "3-4", "5"), strict = FALSE)
```

`columnWrap` *Column Wrap*

Description

Wraps text in a character matrix so that it's displayed over more than one line.

Usage

```
columnWrap(x, width = NULL)
```

Arguments

`x`	a character vector, typically one row of a matrix (e.g. via `apply(m, 1, columnWrap)`).
`width`	integer, the width of the columns in characters, recycled to the length of `x`. Defaults to an equal share of `getOption("width")` per column.

Details

A data.frame containing character columns with long texts is often wrapped by columns. This can lead to a loss of overview. `columnWrap()` wraps the lines within the columns.

Value

a character matrix with one column per element of `x` and one row per wrapped line.

See Also

`strwrap()`

Other data.print: `printCharMatrix()`

Examples

```
print(columnWrap("This is a very long text for a table", 12))
```

`combLevels` *Combine Levels from Multiple Inputs*

Description

Extracts and combines the levels from one or more vectors or factors. Non-factor inputs are coerced to factors before extracting levels.

Usage

```
combLevels(..., sorted = FALSE, na = FALSE)
```

Arguments

`...`	one or more vectors or factors.
`sorted`	logical; if `TRUE`, the resulting levels are sorted.
`na`	logical; if `TRUE`, `NA` is treated as a valid level (i.e., included in the result).

Details

Each input is coerced to a factor (if not already one), and its levels are extracted. The union of all levels is returned. Unused levels of factor inputs are preserved.

By default, missing values (`NA`) are not included as a level. Set `na = TRUE` to include them; `NA` is then placed last when sorting.

The order of levels follows their first occurrence unless `sorted = TRUE`.

Value

a character vector containing the unique levels across all inputs.

See Also

Other data.recode: `asBinary()`, `dummy()`, `mReplace()`, `nf()`, `recodeX()`, `revCode()`, `stringsAsFactors()`

Examples

```
x <- factor(c("A", "B"))
y <- c("B", "C")

combLevels(x, y)

# Sorted levels
combLevels(x, y, sorted = TRUE)

# Including NA as a level
x <- c("A", NA)
y <- c("B", NA)
```

```
combLevels(x, y, na = TRUE)
```

---

combN *Number of Combinations of a Set*

Description

Return the number of combinations with and without replacement and order.

Usage

```
combN(n, m, replace = FALSE, ordered = FALSE)
```

Arguments

n	number of elements from which to choose.
m	number of elements to choose. For combSet can m be a numeric vector too.
replace	logical; whether repetition of the same element is allowed. Defaults to FALSE.
ordered	logical. Does the order matter? Default is FALSE.

Value

a numeric value.

See Also

combn(), choose(), factorial(), vignette("Combinatorics", package = "bedrock")

Other combinatorics: combPairs(), combSet(), pairApply(), permn(), randGroupSplit(), sampleX()

Examples

```
n <- 5; m <- 2
combN(n, m, replace=TRUE, ordered=FALSE)
combN(n, m, replace=TRUE, ordered=TRUE)
combN(n, m, replace=FALSE, ordered=TRUE)
combN(n, m, replace=FALSE, ordered=FALSE)
```

combPairs *Get All Pairs Out of One or Two Sets of Elements*

Description

Returns all combinations of 2 out of the elements in x or x and y (if defined). Combinations of the same elements will be dropped (no replacing). The vector x need not contain unique values. Duplicate elements in x will result in duplicate pairs.

Usage

```
combPairs(x, y = NULL)
```

Arguments

x	a vector of elements, must contain at least 2 elements if y is NULL.
y	a vector of elements, need not be same dimension as x. If y is not NULL then all combination x and y are returned.

Details

If y = NULL then all combination of 2 out of x are returned.
If y is defined then all combinations of x and y are calculated.

Value

a data.frame with two columns X1 and X2 containing the pairwise combinations.

See Also

combn(), expand.grid(), outer(), lower.tri()

Other combinatorics: combN(), combSet(), pairApply(), permn(), randGroupSplit(), sampleX()

Examples

```
combPairs(letters[1:4])
combPairs(x = letters[1:4], y = LETTERS[1:2])

# get all pairs of combinations between factors and numerics out of a data.frame
combPairs(which(sapply(CO2, is.numeric)), which(sapply(CO2, is.factor)))
```

combSet *Samples for Combinations of a Set*

Description

Return the value sets of combinations.

Usage

```
combSet(x, m, replace = FALSE, ordered = FALSE, output = c("matrix", "list"))
```

Arguments

x	a vector of numeric values or characters. Character values need not be unique.
m	number of elements to choose. May be a vector.
replace	logical. Should repetition of the same element be allowed? Default is FALSE.
ordered	logical; whether order matters. Default is FALSE.
output	character string specifying the output representation. One of "matrix" (return combinations as a matrix, the default) or "list" (return combinations as a flat list).

Details

Depending on output, the result is returned either as:

- a matrix with one combination per row
- a flat list where each element represents one combination

If m contains more than one value, combinations are generated independently for each value of m.

Value

if output = "matrix":

- a matrix with one combination per row.
- if length(m) > 1, a list of matrices.

If output = "list":

- a flat list with one element per combination.

See Also

combn(), choose(), factorial(), vignette("Combinatorics")

Other combinatorics: combN(), combPairs(), pairApply(), permn(), randGroupSplit(), sampleX()

Examples

```
x <- letters[1:4]
m <- 2

# combinations with replacement
combSet(x, m, replace = TRUE, ordered = FALSE)

# ordered combinations with replacement
combSet(x, m, replace = TRUE, ordered = TRUE)

# ordered combinations without replacement
combSet(x, m, replace = FALSE, ordered = TRUE)

# unordered combinations without replacement
combSet(x, m, replace = FALSE, ordered = FALSE)

# return as flat list
x <- letters[1:5]

combSet(
  x = x,
  m = c(1, 3, 5),
  output = "list"
)
```

---

compareDataFrames *Compare Two Data Frames by Key Column*

Description

Compares two data frames row-by-row based on a key column, identifying rows present in only one of the two frames and columns that differ in matched rows.

Usage

```
compareDataFrames(x, y, key)
```

Arguments

x	a data frame.
y	a data frame to compare against x.
key	character string. Name of the column used as row identifier. Must be present in both x and y, with unique values.

Details

Only columns present in both data frames are compared. Rows are matched by the `key` column using `identical()` for element-wise comparison, so type differences (e.g., `integer` vs. `double`) will be flagged as differences.

The values of the `key` column must be unique in both data frames.

Value

a named list with four elements:

`identical` logical. `TRUE` if the two data frames are identical with respect to the common columns and key.

`onlyInX` data frame of rows whose key value appears in `x` but not in `y`.

`onlyInY` data frame of rows whose key value appears in `y` but not in `x`.

`diffs` data frame with columns named after the `key` argument (the key value) and `diffCols` (a list column of character vectors naming the differing columns for that key).

See Also

Other data.equal: `allDuplicated()`, `allIdentical()`

Examples

```
x <- data.frame(id = c("A", "B", "C"), v1 = 1:3, v2 = c(10, 20, 30))
y <- data.frame(id = c("A", "B", "D"), v1 = c(1L, 9L, 4L), v2 = c(10, 20, 40))
compareDataFrames(x, y, key = "id")
```

`completeColumns` *Identify Columns Without Missing Values*

Description

This function checks each element of a data frame or list-like object for missing values (`NA`) and identifies those that are completely observed, i.e., contain no missing entries.

Usage

```
completeColumns(x, output = c("names", "logical"))
```

Arguments

`x`	a data.frame or list-like object whose elements are checked for missing values.
`output`	character string specifying the output representation. One of `"names"` (return the names of the complete elements, the default) or `"logical"` (return a logical vector indicating completeness for each element).

Details

An element is considered *complete* if it contains zero missing values. Internally, the function uses `anyNA()` to detect missing values.

Value

if `output = "names"`, a character vector with the names of all complete elements.

If `output = "logical"`, a logical vector of length `length(x)`, where `TRUE` indicates that the corresponding element contains no missing values.

See Also

`anyNA()`, `is.na()`, `na.omit()`, `complete.cases()`

Other data.missing: `countCompCases()`

Examples

```
# Names of columns without missing values
completeColumns(airquality)

# Logical vector indicating completeness
completeColumns(airquality, output = "logical")
```

---

concepts *Concept Utilities for Package Documentation*

Description

Helper functions to inspect and analyse the use of `\\concept` tags within a package.

Usage

```
getConcepts(package, prefix = NULL)

conceptMap(package)

conceptAudit(package)
```

Arguments

`package`	character string. Name of the installed package.
`prefix`	optional character string. If given, `getConcepts()` returns only the concepts starting with it, with the prefix removed.

Details

These utilities extract concept metadata from Rd files and allow structured auditing of conceptual organisation inside a package.

Functions

- `getConcepts()` - Returns all unique concepts used in a package.
- `conceptMap()` - Returns a mapping of concepts to functions.
- `conceptAudit()` - Returns a summary table of concept usage.

The functions use `tools::Rd_db()` to parse Rd files and extract `\\concept` tags programmatically.

These tools are intended for package development, documentation consistency checks, and conceptual audits.

Value

`getConcepts` character vector of unique concept names, sorted

`conceptMap` named list mapping concepts to topics

`conceptAudit` data frame with the columns `concept` and `nTopics`, ordered by decreasing frequency

Examples

```
getConcepts("stats")
getConcepts("stats", prefix = "erf")
head(conceptMap("stats"))
head(conceptAudit("stats"))
```

---

countCompCases *Count Complete Cases*

Description

Return for each variable of a data frame the number of missing values and the complete cases to be expected if this variable would be omitted.

Usage

```
countCompCases(x)
```

Arguments

x	a data.frame containing the data.

Value

a list with three elements. The first gives the number of rows, the second the number of complete cases for the whole data frame. The third element `tab` contains the data for the single variables.

See Also

pharos::plotMiss, complete.cases(), is.na(), na.omit()

Other data.missing: completeColumns()

Examples

```
countCompCases(airquality)
```

---

crossProd *Cross Product of 3D Vectors or Matrices*

Description

Computes the cross product in three-dimensional space for vectors or matrices. For matrices, the operation can be applied row-wise or column-wise.

Usage

```
crossProd(x, y, orientation = c("rows", "cols"))
```

Arguments

x	a numeric or complex vector of length 3, or a matrix with one dimension of length 3.
y	a numeric or complex vector or matrix with the same dimensions as x.
orientation	character string specifying whether vectors are stored in rows or columns when matrices are supplied. Must be one of "rows" or "cols". Ignored if x and y are vectors.

Details

For vectors $x = (x_1, x_2, x_3)$ and $y = (y_1, y_2, y_3)$, the cross product is:

$$x \times y = (x_2 y_3 - x_3 y_2,\ x_3 y_1 - x_1 y_3,\ x_1 y_2 - x_2 y_1)$$

For matrix inputs:

- orientation = "rows": each row is treated as a vector (requires ncol(x) == 3)
- orientation = "cols": each column is treated as a vector (requires nrow(x) == 3)

Numeric and complex inputs can be mixed; standard R coercion rules apply.

Value

- a vector of length 3 if inputs are vectors.
- a matrix if matrices are supplied. Dimension names along the vector axis are propagated from x, the component axis is labelled x, y, z.

See Also

Other math.basic: `closest()`, `crossProdN()`, `dotProd()`, `roundTo()`, `unirootAll()`

Examples

```
# Vector case
crossProd(c(1,0,0), c(0,1,0))

# Row-wise
x <- matrix(c(1,0,0,
              0,1,0), ncol = 3, byrow = TRUE)
y <- matrix(c(0,1,0,
              0,0,1), ncol = 3, byrow = TRUE)
crossProd(x, y, "rows")

# Column-wise
x <- matrix(1:9, nrow = 3)
y <- matrix(9:1, nrow = 3)
crossProd(x, y, "cols")
```

`crossProdN` *Generalized Cross Product via Determinants*

Description

Computes a vector orthogonal to all rows of a matrix using a determinant-based construction. This generalizes the cross product to higher dimensions.

Usage

```
crossProdN(x)
```

Arguments

x	a numeric or complex vector of length 2, or a matrix of dimension $nx(n+1)$.

Details

For a matrix A with dimensions $nx(n+1)$, the result is a vector in R^{n+1} orthogonal to all rows of A. The components are given by:

$$v_i = (-1)^{i+1} \det(A_{-i})$$

where A_{-i} is the matrix obtained by removing the i-th column.

For a vector of length 2, the function returns the perpendicular vector $(a_2, -a_1)$, consistent with the formula above.

Internally, the function computes a nullspace vector using SVD (which is numerically stable also for ill-conditioned input) and rescales it to match the magnitude of the determinant-based generalized

cross product. For numeric input, the sign is chosen to reproduce the orientation of the determinant formula (and hence anticommutativity: swapping two rows of A flips the sign of the result). For complex input, where `det()` is not available, the sign is fixed by the convention that the first component with non-zero modulus has a positive real part.

Value

a numeric or complex vector of length $n + 1$.

See Also

Other math.basic: `closest()`, `crossProd()`, `dotProd()`, `roundTo()`, `unirootAll()`

Examples

```
# 2D case
crossProdN(c(1, 2))

# 3D case (standard cross product)
A <- matrix(c(1,0,0,
              0,1,0), nrow = 2, byrow = TRUE)
crossProdN(A)

# swapping rows flips the sign (anticommutativity)
crossProdN(A[2:1, ])
```

---

dataDescription *Extract Data Description from Excel File*

Description

Reads a documentation sheet from an Excel file and extracts variable descriptions and coding information.

Usage

```
dataDescription(file, sheet = "Description")
```

Arguments

`file`	character string. Path to the Excel file.
`sheet`	character string. Name of the documentation sheet. Default is `"Description"`.

Details

The function reads the specified sheet and trims trailing empty rows.

If a column named "Codes" is present, its contents are split by line breaks and returned as a list of codes per variable, keyed by the "Variable" column.

The Excel sheet is expected to contain at least:

- Variable names
- Descriptions
- Optional coding definitions

If the sheet does not exist or no additional sheets are present, the function returns NULL.

Value

a list with the following components:

- desctable: A data frame containing the description table.
- codes: A named list of code definitions per variable.

See Also

Other label.import: openDataObject()

Examples

```
fn <- system.file("extdata", "example.xlsx", package = "bedrock")

desc <- dataDescription(fn)
desc$desctable
desc$codes[["gender"]]
```

---

digitSum *Digit Sum for Integer Values*

Description

Computes the sum of digits for whole-numbered inputs. Negative values are handled by taking the absolute value.

Usage

```
digitSum(x)
```

Arguments

x	an integer vector, or a numeric vector of whole numbers.

Details

The function accepts integer vectors as well as doubles holding whole numbers (e.g. `124` and `124L` are both valid). Fractional values raise an error. Missing values (`NA`) are propagated.

Value

an integer vector containing the digit sums.

See Also

Other number.theory: `GCD-LCM`, `divisors()`, `factorize()`, `fibonacci()`, `isOdd()`, `isPrime()`, `primes()`

Examples

```
digitSum(124)
digitSum(c(10L, 99L, -1234L))
```

---

divisors *Calculate Divisors*

Description

Calculate the proper divisors of positive natural numbers.

Usage

```
divisors(x)
```

Arguments

`x`	vector of positive whole numbers for which the divisors are to be returned.

Details

Divisibility is a mathematical relationship between two integers. An integer is divisible by another integer if there is no remainder in the division. This function returns the *proper* divisors of `x`, i.e. all positive divisors excluding `x` itself. The number 11 is prime and has only the proper divisor 1, whereas the number 12 has the proper divisors 1, 2, 3, 4 and 6. In elementary number theory, the concept of divisibility is limited to natural numbers. The number of proper divisors can be determined with the function `length()`.

Value

an integer vector containing the proper divisors in ascending order if `x` is a single number, otherwise a named list of such vectors. A prime number yields `1`, and 1 itself yields `integer(0)` - its only divisor is 1, which is `x` itself and therefore not a proper one.

See Also

Other number.theory: GCD-LCM, digitSum(), factorize(), fibonacci(), isOdd(), isPrime(), primes()

Examples

```
divisors(786)

divisors(c(145, 786))

# the number of proper divisors
length(divisors(786))

# a prime has only one, and this one is at the integer limit
divisors(.Machine$integer.max)
```

---

dotProd *Dot Product of Vectors or Matrices*

Description

Computes the dot product between two numeric or complex vectors, or the column-wise dot products of two matrices with identical dimensions.

Usage

```
dotProd(x, y)
```

Arguments

x	a numeric or complex vector, or a numeric/complex matrix.
y	a numeric or complex vector, or a numeric/complex matrix with the same dimensions as x.

Details

For vectors x and y, the dot product is defined as:

$$\sum_i \overline{x_i} y_i$$

where $\overline{x_i}$ denotes the complex conjugate of x_i (for real input this is simply $\sum_i x_i y_i$).

For matrices, the dot product of each column of x with the corresponding column of y is returned.

Note that crossprod() does *not* conjugate its first argument for complex input, so it computes $t(X)Y$ rather than the Hermitian inner product; this function does conjugate.

Value

- a scalar if x and y are vectors.
- a numeric or complex vector containing the column-wise dot products if matrices are supplied.

See Also

crossprod()

Other math.basic: closest(), crossProd(), crossProdN(), roundTo(), unirootAll()

Examples

```
# Vector dot product
dotProd(c(1, 2, 3), c(4, 5, 6))

# Complex vectors (Hermitian inner product)
dotProd(c(1+1i, 2), c(3, 4-1i))

# Matrix (column-wise dot products)
x <- matrix(1:6, ncol = 2)
y <- matrix(6:1, ncol = 2)
dotProd(x, y)
```

---

dummy *Generate Dummy Codes for a Factor*

Description

Generate a matrix of dummy codes, also known as class indicators, for a factor or class vector.

Usage

```
dummy(
  x,
  method = c("treatment", "sum", "helmert", "poly", "full"),
  base = 1,
  levels = NULL
)
```

Arguments

x	factor or vector of classes.
method	character string specifying the contrast method. One of "treatment", "sum", "helmert", "poly", or "full". Abbreviations are accepted.
base	integer or character string specifying the baseline group. Only used for method = "treatment" (see Details).
levels	optional character vector specifying the possible levels of x. If NULL, levels are inferred by factor(x).

Details

The argument `method` controls the contrast coding. The option `"full"` returns one indicator column for each level of `x`. This full-rank coding is usually redundant for `lm()` and related modelling functions.

The `base` argument is only used by `method = "treatment"`. The other contrast types have no freely choosable baseline: `"sum"` implicitly uses the last level as reference, `"helmert"` contrasts each level against the preceding ones, and `"poly"` uses orthogonal polynomials.

Column names reflect the semantics of the coding: level names for `"treatment"` (without the baseline), `"full"` (all levels), `"sum"` (without the last level) and `"helmert"` (without the first level); `"poly"` keeps the standard degree labels (`.L`, `.Q`, ...).

Value

a matrix with dummy codes. The number of rows equals `length(x)`. For `method = "full"`, the number of columns equals the number of levels. Otherwise, the number of columns equals the number of levels minus one.

The returned matrix has an attribute `"base"` containing the baseline level for treatment coding, and `NA` otherwise.

See Also

`model.frame()`, `contrasts()`, `contr.treatment()`, `contr.sum()`, `contr.helmert()`, `contr.poly()`

Other data.recode: `asBinary()`, `combLevels()`, `mReplace()`, `nf()`, `recodeX()`, `revCode()`, `stringsAsFactors()`

Examples

```
x <- c("red", "blue", "green", "blue", "green", "red", "red", "blue")
dummy(x)
dummy(x, base = 2)
dummy(x, method = "sum")

y <- c("Max", "Max", "Max", "Max", "Max", "Bill", "Bill", "Bill")
dummy(y)
dummy(y, base = "Max")
dummy(y, base = "Max", method = "full")

# Revert full dummy coding
m <- dummy(y, method = "full")
apply(m, 1, function(z) colnames(m)[z == 1])

# Revert treatment dummy coding
m <- dummy(y)
apply(
  m,
  1,
  function(z) ifelse(sum(z) == 0, attr(m, "base"), colnames(m)[z == 1])
)
```

`extractArgs` *Extract Named Arguments from Dots with Defaults*

Description

Utility to extract a subset of arguments from a list (typically `list(...)`) and merge them with default values. Elements of `dots` override entries in `defaults` of the same name; explicit `NULL` values are preserved.

Usage

```
extractArgs(dots, defaults, validate = NULL, returnRest = FALSE)
```

Arguments

`dots`	named list of arguments (usually `list(...)`).
`defaults`	named list of default values.
`validate`	optional validation function, called with the merged argument list for its side effect. It should throw an error on invalid input; its return value is ignored.
`returnRest`	logical; if `TRUE`, a list with components `args` (the merged arguments) and `rest` (all elements of `dots` not matching a default, including unnamed ones) is returned.

Value

named list of extracted arguments, or a list with components `args` and `rest` if `returnRest = TRUE`.

See Also

Other pkg.args: `callIf()`, `getDotsArg()`, `mergeArgs()`, `recycle()`

Examples

```
dots <- list(col = "red", lwd = 2, 99)
extractArgs(dots, defaults = list(col = "black", lty = 1))

extractArgs(dots, defaults = list(col = "black", lty = 1),
            returnRest = TRUE)
```

`factorize` *Prime Factorization of Integers*

Description

Compute the prime factorization(s) of integer(s) `x`, using Pollard's rho algorithm with deterministic Miller-Rabin primality testing (64-bit, implemented in C++).

Usage

```
factorize(x)
```

Arguments

`x`	vector of positive whole numbers to factorize, not exceeding `2^53`.

Details

`x` must not exceed `2^53` (`9007199254740992`), the largest integer up to which every integer can be represented exactly. Larger integers can still be representable – every power of two is – but not all of them are, and above this bound `x` may already have been rounded by R before it reaches this function, so a factorization could silently be correct for a different number than the one entered – for such inputs, use the **gmp** package's `gmp::factorize()`, which represents arbitrarily large integers exactly (e.g. via `gmp::as.bigz()` or a string).

Value

a named `list()` of the same length as `x`, each element a 2-column matrix with column `"p"` the prime factors in increasing order and column `"m"` their respective exponents (or multiplicities), i.e., for a prime number `x`, the resulting matrix is `cbind(p = x, m = 1)`.

Each prime appears in exactly one row, so `prod(p^m)` returns `x` and `p` is strictly increasing. `x = 1` yields a matrix with zero rows: 1 is the empty product, and `prod(numeric(0))` is 1 accordingly.

See Also

Other number.theory: `GCD-LCM`, `digitSum()`, `divisors()`, `fibonacci()`, `isOdd()`, `isPrime()`, `primes()`

Examples

```
factorize(47)
factorize(seq(101, 120, by=2))

# the defining invariant
f <- factorize(360)[[1]]
f
prod(f[, "p"]^f[, "m"])
```

`fibonacci` *Fibonacci Numbers*

Description

Generate Fibonacci numbers. The Fibonacci numbers can also be calculated using the golden ratio `phi`, as demonstrated in the examples.

Usage

```
fibonacci(n)
```

Arguments

`n`	nonnegative integer (<= 78) or vector of such integers.

Details

Generates the `n`-th Fibonacci number, whereas `fibonacci(0) = 0`.
The golden ratio is defined as `phi = 0.5*(1+sqrt(5))`.

Values of `n` are limited to 78, as larger Fibonacci numbers exceed the range in which doubles represent integers exactly (`2^53`).

Value

an integer-valued numeric vector.

References

https://en.wikipedia.org/wiki/Fibonacci_number
https://mathworld.wolfram.com/GoldenRatio.html

See Also

Other number.theory: GCD-LCM, digitSum(), divisors(), factorize(), isOdd(), isPrime(), primes()

Examples

```
fibonacci(0)                            # 0
fibonacci(2)                            # 1
fibonacci(0:3)                          # 0 1 1 2
fibonacci(0:25)                         # ... 75025 121393

# Golden ratio = Fib(25)/ Fib(24)
f25 <- quot(fibonacci(24:25))           # 1.618033989
phi <- (sqrt(5) + 1)/2
abs(f25 - phi)                          # 7.945178e-11
```

```
# Fibonacci numbers without iteration
fibo <- function(n) {
  phi <- (sqrt(5) + 1)/2
  fib <- (phi^(n+1) - (1-phi)^(n+1)) / (2*phi - 1)
  round(fib)
}

fibo(30:33)                             # 1346269 2178309 3524578 5702887
```

`findDownload` *Locate a File in the Downloads Directory*

Description

Returns the full path to a file located in the user's Downloads directory.

Usage

```
findDownload(file)
```

Arguments

`file`	character string. Name of the file.

Details

The function resolves the path to the user's Downloads directory using an internal helper and appends `file`. It does not perform any downloading; it only locates files that already exist locally.

If the file does not exist, an error is thrown.

Value

a character string giving the full path to the file.

See Also

Other file.path: `buildPath()`, `isFilePath()`, `isUrl()`, `splitPath()`, `urlExists()`

Examples

```
## Not run:
# cannot be run automatically: reads the personal Downloads
# directory of the user, where no such file exists
findDownload("data.csv")

## End(Not run)
```

flags *Extract Dichotomous (Binary) Variables*

Description

Identify and extract dichotomous (binary) variables from a data frame or matrix using `isDichotomous()`.

Usage

```
flags(
  x,
  strict = FALSE,
  na.rm = FALSE,
  output = c("data", "names", "index", "logical")
)
```

Arguments

`x`	a data frame or matrix.
`strict`	logical. If `TRUE`, only variables with exactly two distinct values are considered dichotomous. If `FALSE` (default), variables with one or two distinct values are allowed.
`na.rm`	logical. Should missing values be ignored when checking for dichotomous variables? Default is `FALSE`.
`output`	character string specifying the output representation. One of `"data"` (subset of `x` containing only dichotomous variables, the default), `"names"` (names of dichotomous variables), `"index"` (column indices) or `"logical"` (logical vector indicating dichotomous variables).

Details

Variables with only missing values are not considered dichotomous when `na.rm = FALSE`.

When `na.rm = TRUE`, such variables are treated as empty vectors and are considered dichotomous only if `strict = FALSE`.

Internally, variables with indeterminate dichotomous status (i.e. `NA` returned by `isDichotomous()`) are treated as non-dichotomous for filtering purposes.

Value

depending on `output`:

- `"data"`: data frame or matrix.
- `"names"`: character vector.
- `"index"`: integer vector.
- `"logical"`: logical vector.

See Also

Other data.predicate: `isDichotomous()`, `isEuclid()`, `isLowCardinality()`, `isNumeric()`, `isWholeLike()`, `isZero()`, `nUnique()`

Examples

```
dat <- data.frame(
  a = c(0, 1, 1, 0),
  b = c(1, 2, 3, 4),
  c = c(TRUE, FALSE, TRUE, TRUE),
  d = c(NA, NA, NA, NA)
)

flags(dat)

# effect of na.rm
flags(dat, na.rm = TRUE)

# return variable names
flags(dat, output = "names")

# return column indices
flags(dat, output = "index")
```

---

funArgs *List All Arguments of a Function*

Description

Returns the formal arguments of a function together with their default values.

Usage

```
funArgs(
  fun,
  package = NULL,
  sorted = FALSE,
  output = c("data.frame", "list", "string")
)
```

Arguments

`fun`	function object or function name.
`package`	optional package name used to resolve `fun`.
`sorted`	logical; should arguments be sorted alphabetically? `...` is always kept last. Ignored when `output = "list"`.

output — character string specifying the output format:
- "data.frame" (default): return a data frame.
- "list": return a named list of formal arguments.
- "string": return a comma-separated character string of argument assignments.

Value

depending on output:

- "data.frame": a data frame with columns name and value.
- "list": a named list of formal arguments.
- "string": a character vector of length one.

See Also

formals(), args()

Other pkg.funinfo: auditNames(), funCalls(), funKeywords(), funList(), rdLabels(), rdTitle()

Examples

```
funArgs("combN")

funArgs("combN", output = "list")

funArgs("combN", output = "string")

cat(funArgs("combN", output = "string"))
```

---

funCalls *List Calls Used in Function*

Description

For screening purposes it can be useful to get a list of all function calls our function may depend on. funCalls() parses the function source and returns all found function calls grouped by their package.

Usage

```
funCalls(fun, package = NULL, sorted = FALSE)
```

Arguments

fun — the name of the function.

package — optional name of a package; if given, the result is filtered to source environments matching package.

sorted — logical; whether calls are sorted alphabetically. Defaults to FALSE.

Details

The source packages are resolved via `find()`, which only sees attached packages. Calls to functions from packages that are not on the search path are reported under `"<not found>"`.

Value

a list of character vectors with the function calls, grouped by the environment the called functions were found in.

Note

Based on code by Nicholas Cooper, adapted to conform to package standards.

See Also

`getParseData()`

Other pkg.funinfo: `auditNames()`, `funArgs()`, `funKeywords()`, `funList()`, `rdLabels()`, `rdTitle()`

Examples

```
funCalls("combN", package="bedrock")
```

`funKeywords` *List Keywords For R Manual Pages*

Description

List the keywords for specific R man pages or return a list of valid R keywords.

Usage

```
funKeywords(topic)
```

Arguments

`topic`	optional, object or man page topic.

Details

If `topic` is provided, return a list of the Keywords associated with `topic`. Otherwise, display the list of valid R Keywords from the R doc/Keywords file.

Value

if `topic` is missing, the R keywords documentation file is opened for display via `file.show()`, invisibly returning `NULL`. Otherwise, a character vector of topic names whose keywords match `topic`.

Note

Substantially based on the `keywords()` function from the **gtools** package by Gregory R. Warnes, with minor adaptations by the package author.

See Also

`help()`

Other pkg.funinfo: `auditNames()`, `funArgs()`, `funCalls()`, `funList()`, `rdLabels()`, `rdTitle()`

Examples

```
## Show all valid R Keywords
funKeywords()

## Show Keywords associated with the 'merge' function
funKeywords(merge)
funKeywords("merge")
```

---

funList *List Functions in a Package*

Description

List all the functions in a package.

Usage

```
funList(package, exported = TRUE)
```

Arguments

`package`	the name of the package.
`exported`	logical; whether only exported functions are listed. Defaults to `TRUE`.

Details

This is just a wrapper for the namespace inspection functions (as I always forgot how to do the trick). By default only the exported functions are returned; with `exported = FALSE` all functions defined in the package namespace are listed, including internal ones.

Value

a sorted character vector with the function names.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`ls()`, `ls.str()`, `lsf.str()`, `getNamespaceExports()`

Other pkg.funinfo: `auditNames()`, `funArgs()`, `funCalls()`, `funKeywords()`, `rdLabels()`, `rdTitle()`

Examples

```
funList("bedrock")
```

GCD-LCM *Greatest Common Divisor and Least Common Multiple*

Description

Calculates the greatest common divisor (GCD) and least common multiple (LCM) of all the values present in its arguments.

Usage

```
GCD(..., na.rm = FALSE)

LCM(..., na.rm = FALSE)
```

Arguments

`...`	integer or logical vectors.
`na.rm`	logical; whether missing values (including NaN) are removed.

Details

The computation is based on the Euclidean algorithm without using the extended version. The greatest common divisor for all numbers in the integer vector `x` will be computed (the multiple GCD). Negative values are allowed and enter via their absolute value; logical vectors are coerced to integer.

Value

a numeric (integer) value.

Zero

Zero behaves differently in the two functions, which is why they do not treat it the same way. For the greatest common divisor it is *neutral* - every number divides 0, so `GCD(0, a)` is `abs(a)` and zeros can simply be dropped. For the least common multiple it is *absorbing* - 0 is a multiple of every number and the smallest non-negative one, so `LCM(0, a)` is 0. `GCD(0, 0)` and `LCM(0, 0)` are both 0.

Note

The following relation is always true:

`n * m = GCD(n, m) * LCM(n, m)`

It also holds when one of the values is zero, and that is the shortest way to see why `LCM(0, 6)` has to be 0 rather than 6.

See Also

Other number.theory: `digitSum()`, `divisors()`, `factorize()`, `fibonacci()`, `isOdd()`, `isPrime()`, `primes()`

Examples

```
GCD(12, 10)
GCD(144, 233)    # Fibonacci numbers are relatively prime to each other

LCM(12, 10)
LCM(144, 233)    # = 144 * 233

# all elements will be flattened by unlist
GCD(2, 3, c(5, 7) * 11)
GCD(c(2*3, 3*5, 5*7))
LCM(c(2, 3, 5, 7) * 11)
LCM(2*3, 3*5, 5*7)

# zero is neutral for the GCD and absorbing for the LCM
GCD(0, 6)
LCM(0, 6)

# n * m == GCD(n, m) * LCM(n, m), zero included
GCD(0, 6) * LCM(0, 6)
```

`getDotsArg` *Get a Single Argument from Dots with Default*

Description

Lightweight helper to extract a single named argument from a list (typically `list(...)`). If the argument is not present, a default value is returned.

Usage

```
getDotsArg(dots, name, default = NULL)
```

Arguments

dots	named list (usually `list(...)`).
name	character string, argument name.
default	default value if argument not present.

Value

the value of the argument or `default`.

See Also

For extracting several arguments at once use `extractArgs()`.

Other pkg.args: `callIf()`, `extractArgs()`, `mergeArgs()`, `recycle()`

Examples

```
f <- function(...) {
  dots <- list(...)
  getDotsArg(dots, "col", default = "black")
}

f(col = "red", lwd = 2)
f(lwd = 2)
```

---

intervals *Interval Arithmetic*

Description

Functions for computing relationships between numeric intervals. All functions accept intervals as numeric vectors of length 2 or matrices with 2 columns (one interval per row). Unordered bounds are silently sorted; rows are recycled to equal length.

Usage

```
overlapSize(x, y)

overlaps(x, y)

distance(x, y)

x %overlaps% y
```

Arguments

x	a numeric vector of length 2 `c(lower, upper)`, or a numeric matrix with 2 columns where each row defines one interval.
y	a numeric vector of length 2 `c(lower, upper)`, or a numeric matrix with 2 columns where each row defines one interval.

Details

Intervals are treated as closed, i.e., $[a, b]$. Consequently:

- Two intervals sharing only a boundary point have `overlap` 0 but `overlaps` returns `TRUE`.
- `distance` returns 0 whenever intervals touch or overlap.
- The returned vector is always unnamed, whatever dimnames the inputs carry.

Value

`overlap` numeric vector of overlap lengths (0 if no overlap).

`overlaps` logical vector; `TRUE` if intervals share at least one point.

`distance` numeric vector of gap lengths between non-overlapping intervals (0 if overlapping or touching).

`%overlaps%` logical vector; operator wrapper for `overlaps()`.

See Also

Other data.interval: `between-operators`, `range-operators`

Examples

```
# overlap length
overlapSize(c(1, 5), c(3, 7))   # 2
overlapSize(c(1, 3), c(3, 5))   # 0 (boundary only)

# overlap check
overlaps(c(1, 5), c(3, 7))  # TRUE
overlaps(c(1, 3), c(3, 5))  # TRUE  (boundary counts)
overlaps(c(1, 2), c(3, 4))  # FALSE

# gap distance
distance(c(1, 2), c(4, 5))  # 2
distance(c(1, 5), c(3, 7))  # 0

# operator
c(1, 5) %overlaps% c(3, 7)  # TRUE

# vectorised (matrix input)
m <- matrix(c(1,3, 2,6, 5,8), ncol = 2, byrow = TRUE)
overlapSize(m, c(4, 7))
```

`isDichotomous` *Check Whether a Vector Is Dichotomous*

Description

Determines whether a vector contains at most two distinct values.

Usage

```
isDichotomous(x, strict = FALSE, na.rm = FALSE)
```

Arguments

`x`	a vector.
`strict`	logical. If `TRUE`, exactly two distinct values must be present. If `FALSE` (default), at most two distinct values are allowed.
`na.rm`	logical. If `TRUE`, missing values are removed before evaluation. If `FALSE` (default), the presence of `NA` results in `NA` (indeterminate status).

Value

`TRUE`, `FALSE`, or `NA` if the status cannot be determined because of missing values (see `na.rm`).

See Also

Other data.predicate: `flags()`, `isEuclid()`, `isLowCardinality()`, `isNumeric()`, `isWholeLike()`, `isZero()`, `nUnique()`

Examples

```
isDichotomous(c(0, 1, 1))
isDichotomous(c(1, 1, 1))
isDichotomous(c(1, 1, 1), strict = TRUE)
isDichotomous(c(0, 1, NA))              # NA
isDichotomous(c(0, 1, NA), na.rm = TRUE)
isDichotomous(c("A", "A", "B"))
isDichotomous(c("A", "A", "B", "C"))
isDichotomous(factor(c("A", "A", "B", "C")))
```

`isEuclid` *Test if a Distance Matrix Is Euclidean*

Description

Checks whether a distance matrix corresponds to Euclidean distances.

Usage

```
isEuclid(x, tol = 1e-07)
```

Arguments

`x`	an object of class `dist`.
`tol`	numeric tolerance for detecting negative eigenvalues, relative to the largest absolute eigenvalue.

Details

The test is based on the eigenvalues of the double-centered squared distance matrix $B = -\frac{1}{2}JD^2J$. A distance matrix is Euclidean if and only if B is positive semi-definite, i.e., all eigenvalues are non-negative (within numerical tolerance).

The tolerance is applied *relative* to the largest absolute eigenvalue, so that the test is invariant to rescaling of the distances. Note that this holds in both directions: the comparison below uses `max(abs(lambda))` without an absolute floor, so shrinking all distances by a constant factor cannot turn a non-Euclidean matrix into a Euclidean one.

The returned logical value carries additional diagnostic information as attributes:

- `eigenvalues`: Eigenvalues of the centered matrix
- `minEigenvalue`: Smallest eigenvalue
- `tol`: Tolerance used for the test

Value

a logical scalar. Returns `TRUE` if the distance matrix is (approximately) Euclidean, otherwise `FALSE`.

See Also

Other data.predicate: `flags()`, `isDichotomous()`, `isLowCardinality()`, `isNumeric()`, `isWholeLike()`, `isZero()`, `nUnique()`

Examples

```
d <- dist(matrix(rnorm(20), ncol = 2))
res <- isEuclid(d)
res

# Access diagnostics
attr(res, "eigenvalues")
attr(res, "minEigenvalue")
```

`isFilePath` *Check Whether a String Is a File Path*

Description

Returns `TRUE` if the given string looks like a local file path (absolute or relative, Unix/Windows style), `FALSE` otherwise. Convenience wrapper around the internal `.detectInputType()` helper.

Usage

```
isFilePath(x)
```

Arguments

`x`	`character(1)` - the string to test.

Value

`logical(1)` - `TRUE` if `x` is a file path, `FALSE` otherwise.

See Also

`isUrl()` for the complementary URL check.

Other file.path: `buildPath()`, `findDownload()`, `isUrl()`, `splitPath()`, `urlExists()`

Examples

```
isFilePath("/home/user/data/file.csv")   # TRUE
isFilePath("~/documents/report.pdf")     # TRUE
isFilePath("./relative/path/file.R")     # TRUE
isFilePath("../other/folder/data.rds")   # TRUE
isFilePath("C:/Users/Hans/file.xlsx")    # TRUE
isFilePath("https://example.com/f.csv")  # FALSE
```

isLowCardinality *Check for Low Cardinality*

Description

Checks whether x contains at most maxUnique unique, non-missing values. Unlike nUnique(), this stops counting as soon as the threshold is exceeded, which makes it considerably faster for large, high-cardinality vectors.

Usage

```
isLowCardinality(x, maxUnique = 12)
```

Arguments

x	a numeric or integer vector.
maxUnique	integer. The threshold up to which x is considered to have low cardinality. Defaults to 12.

Value

a logical of length one: TRUE if x has maxUnique or fewer unique, non-NA values, FALSE otherwise.

See Also

nUnique() for the uncapped count.

Other data.predicate: flags(), isDichotomous(), isEuclid(), isNumeric(), isWholeLike(), isZero(), nUnique()

Examples

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

isLowCardinality(1:100, maxUnique = 12)
```

isNA *Test for a Scalar Missing Value*

Description

Check whether an object is a single missing value (NA).

Usage

```
isNA(x)
```

Arguments

x	an object to be tested.

Details

This is a strict helper that returns TRUE only if x is an atomic vector of length one and equal to NA. In contrast to is.na(), which is vectorized, isNA is intended for scalar checks, e.g. in conditional statements.

This function differs from is.na() in that it:

- Only returns TRUE for length-one inputs
- Returns a single logical value (not vectorized)
- Works consistently across all NA types

Value

logical scalar. Returns TRUE if x is a single missing value (NA), and FALSE otherwise.

See Also

Other vector.na: coalesceX(), locf(), naIf(), naReplace()

Examples

```
isNA(NA)             # TRUE
isNA(NA_real_)       # TRUE
isNA(NA_integer_)    # TRUE

isNA(c(NA, NA))      # FALSE (length > 1)
isNA(NULL)           # FALSE
isNA(1)              # FALSE
isNA(c(1, NA))       # FALSE
```

---

isNumeric *Check Whether an Object Is a Valid Numeric Vector*

Description

Validates that an object is numeric and optionally satisfies additional structural constraints such as integer-valuedness or positivity.

Usage

```
isNumeric(
  x,
  isIntegerValued = FALSE,
  isPositive = FALSE,
  tol = sqrt(.Machine$double.eps),
  na.rm = FALSE
)
```

Arguments

x	an object to be tested.
isIntegerValued	logical. If TRUE, values must be whole-like (within tolerance). Uses isWholeLike() internally.
isPositive	logical. If TRUE, all values must be strictly greater than zero.
tol	numerical tolerance used when isIntegerValued = TRUE. Default is sqrt(.Machine$double.eps).
na.rm	logical. If TRUE, missing values are removed before validation. If FALSE (default) and x contains NA, the function returns FALSE.

Details

The function checks:

- Whether x is numeric.
- Whether all values are finite.
- Optional integer-like constraint via isWholeLike().
- Optional positivity constraint.

This function is intended for internal validation in statistical routines. Length validation is the responsibility of the caller and should be performed separately with an explicit length() check.

Value

a single logical value.

See Also

Other data.predicate: flags(), isDichotomous(), isEuclid(), isLowCardinality(), isWholeLike(), isZero(), nUnique()

Examples

```
isNumeric(c(1, 2, 3))
isNumeric(c(1, 2.1, 3), isIntegerValued = TRUE)
isNumeric(c(1, -2, 3), isPositive = TRUE)
isNumeric(c(1, NA), na.rm = TRUE)
```

isOdd *Test if Numbers Are Odd*

Description

Checks whether elements of a numeric vector are odd integers.

Usage

```
isOdd(x)
```

Arguments

x	a numeric vector.

Details

The function first checks whether values are finite integers. Non-integer values (e.g. 3.5), NA, NaN, or Inf return NA. A bare logical NA is accepted and treated as a missing numeric value.

Value

a logical vector of the same length as x. Returns TRUE for odd integers, FALSE for even integers, and NA for non-integer or non-finite values.

See Also

Other number.theory: GCD-LCM, digitSum(), divisors(), factorize(), fibonacci(), isPrime(), primes()

Examples

```
isOdd(1:5)
isOdd(c(2, 3, 4.5, NA, Inf))
```

isPrime *Test Whether Numbers Are Prime*

Description

Determines whether integer values are prime numbers.

Usage

```
isPrime(x)
```

Arguments

x	a numeric vector. Values must be finite whole numbers not exceeding `2^53`.

Details

This function is vectorized and returns a logical vector of the same length as the input.

Internally, a fast deterministic primality test for 64-bit integers is used.

Non-integer, negative, missing, or non-finite values result in `FALSE`: there the answer is known, it simply is not "prime".

Value

a logical vector indicating whether each element of `x` is a prime number, `NA` where `x` exceeds `2^53`.

Upper limit

Values above `2^53` (`9007199254740992`) return `NA` with a warning, because for them there is no answer to give. `2^53` is the largest integer up to which *every* integer is exactly representable; above it the representable integers thin out, so the value that reaches the test need not be the value that was entered: R parses `9007199254740997`, which is prime, as `9007199254740996`. Every representable double above `2^53` is even, so testing the neighbour would report `FALSE` for *every* prime beyond the bound – silently, and with no way for the caller to notice. For larger numbers, use `gmp::isprime()` with a `gmp::as.bigz()` or character input.

`factorize()` carries the same bound but rejects the input with an error instead. The difference is deliberate: `factorize()` answers one number per call element and can refuse the call, whereas a vectorized predicate should not let a single unrepresentable element discard the result for all the others.

See Also

Other number.theory: `GCD-LCM`, `digitSum()`, `divisors()`, `factorize()`, `fibonacci()`, `isOdd()`, `primes()`

Examples

```
isPrime(2)
isPrime(1:10)
isPrime(c(17, 18, 19))
```

isUrl *Check Whether a String Is a URL*

Description

Returns TRUE if the given string starts with a recognised URL scheme, FALSE otherwise. Convenience wrapper around the internal .detectInputType() helper.

Usage

```
isUrl(x)
```

Arguments

x	character(1) - the string to test.

Value

logical(1) - TRUE if x is a URL, FALSE otherwise.

See Also

For the complementary check on an existing path, see isFilePath().

Other file.path: buildPath(), findDownload(), isFilePath(), splitPath(), urlExists()

Examples

```
isUrl("https://example.com/data.csv")   # TRUE
isUrl("ftp://files.example.org/x.zip")  # TRUE
isUrl("s3://my-bucket/file.parquet")    # TRUE
isUrl("/home/user/file.csv")            # FALSE
isUrl("./script.R")                     # FALSE
```

isWholeLike *Test Whether Values Are (Nearly) Whole Numbers*

Description

Checks whether values are integer-like within a numerical tolerance. Works for numeric, integer, and complex vectors.

Usage

```
isWholeLike(
  x,
  all = TRUE,
  isNonNegative = FALSE,
  tol = sqrt(.Machine$double.eps),
  na.rm = FALSE
)
```

Arguments

x	a numeric, integer, or complex vector.
all	logical. If TRUE (default), returns a single logical indicating whether all elements are whole-like. If FALSE, returns a logical vector of the same length as x.
isNonNegative	logical. If TRUE, additionally requires values to be non-negative.
tol	numerical tolerance for comparing to the nearest integer. Default is sqrt(.Machine$double.eps).
na.rm	logical. If TRUE, missing values are removed before testing. If FALSE (default) and x contains NA, the result is FALSE.

Details

A value is considered whole-like if the absolute difference between the value and its nearest integer is smaller than tol.

For complex numbers, both real and imaginary parts must be whole-like; with isNonNegative = TRUE, both parts must additionally be non-negative.

Value

if all = TRUE, a single logical value. If all = FALSE, a logical vector.

See Also

Other data.predicate: flags(), isDichotomous(), isEuclid(), isLowCardinality(), isNumeric(), isZero(), nUnique()

Examples

```
isWholeLike(c(1, 2, 3))
isWholeLike(c(1, 2.0000001), tol = 1e-6)
isWholeLike(c(1, 2.5), all = FALSE)
isWholeLike(c(1, -2), isNonNegative = TRUE)
isWholeLike(1:5 + 0i)
```

isZero *Check a Vector For Being Zero*

Description

Test if x is zero. This is done by checking if the numeric value is below the machine tolerance.

Usage

```
isZero(x, tol = sqrt(.Machine$double.eps), na.rm = FALSE)
```

Arguments

x	a (non-empty) numeric or complex vector of data values.
tol	tolerance to be used.
na.rm	logical, indicating whether NA values should be stripped before the computation proceeds. Defaults to FALSE.

Value

logical vector of the same length as x (after optional NA removal). Non-numeric input yields all-FALSE.

References

Burns, P. (2011). *The R Inferno*. https://www.burns-stat.com/documents/books/the-r-inferno/

See Also

all.equal()

Other data.predicate: flags(), isDichotomous(), isEuclid(), isLowCardinality(), isNumeric(), isWholeLike(), nUnique()

Examples

```
# "... These are people who live in ignorance of the Floating Point Gods.
# These pagans expect [...] the following to be TRUE" (Burns, 2011):
(.1 - .3 / 3) == 0

# they might be helped by
isZero(.1 - .3 / 3)
```

`label` *Get or Set Object and Variable Labels*

Description

Retrieve or assign a label to an object, or to variables (columns) of a data frame.

Usage

```
label(x, vars = NULL)

label(x, vars = NULL) <- value
```

Arguments

`x`	an object. Typically an atomic vector or a data.frame.
`vars`	optional specification of variables (columns) in a data.frame. Can be: • `NULL`: operate on the object label (default). • `TRUE`: all columns. • numeric indices or character names of columns.
`value`	a character vector of labels, or `NULL` to remove them. For object labels, must be of length 1. For variable labels, must have length 1 or the same length as `vars`.

Details

For atomic objects, a single label can be stored as an attribute `"label"`. For data frames, a label can be assigned either to the whole dataset or to individual columns.

The function provides a unified interface for working with labels:

- `label(x)` returns the label of an object
- `label(x) <- "text"` sets the label of an object
- `label(x, vars = ...)` returns labels of selected variables
- `label(x, vars = ...) <- value` sets variable labels

Variable labels are stored as attribute `"label"` on each column. Assigning `NULL` removes the label(s).

Value

- getter: a character scalar (object label) or a named character vector (variable labels).
- setter: the modified object `x`.

See Also

Other label.attrs: `renameX()`, `setAttr-removeAttr-keepAttr`, `setNamesX()`

Examples

```
df <- data.frame(age = 1:3, sex = c("m", "f", "m"))

# Set dataset label
label(df) <- "Example dataset"
label(df)

# Set variable labels
label(df, vars = TRUE) <- c("Age in years", "Sex")
label(df, vars = TRUE)

# Set single variable label
label(df, vars = "age") <- "Age"
label(df, vars = "age")

# Remove variable labels
label(df, vars = TRUE) <- NULL

# Atomic vector
x <- 1:5
label(x) <- "Simple vector"
label(x)
```

---

linScale *Linearly Rescale Numeric Data*

Description

Performs a linear transformation of numeric data to a specified range. Each column of `x` is rescaled independently.

Usage

```
linScale(x, low = NULL, high = NULL, newLow = 0, newHigh = 1)
```

Arguments

`x`	a numeric vector, matrix or data frame.
`low`, `high`	optional numeric vectors specifying the lower and upper bounds of the original scale. If `NULL`, the column-wise minima and maxima of `x` are used.
`newLow`, `newHigh`	numeric vectors specifying the target range. Defaults to `0` and `1`.

Details

The transformation is defined as:

$$x_{scaled} = \frac{x - low}{high - low} \cdot (newHigh - newLow) + newLow$$

Constant columns (where high == low) are mapped to newLow.

If low and high are supplied, values of x outside [low, high] are extrapolated linearly and are not clipped to the target range.

Value

an object of the same shape as x: a numeric vector for vector input, otherwise a numeric matrix with the same dimensions, where each column is linearly rescaled to the interval [newLow, newHigh].

See Also

scale(), DescToolsX::scaleX

Other math.transform: logit(), percentRank(), rankX(), winsorize()

Examples

```
x <- matrix(1:10, ncol = 2)

# default scaling to [0,1]
linScale(x)

# custom range
linScale(x, newLow = -1, newHigh = 1)

# using predefined bounds
linScale(x, low = 1, high = 10)
```

---

locf *Last Observation Carried Forward*

Description

In longitudinal studies it's common that individuals drop out before all responses can be obtained. Measurements obtained before the individual dropped out can be used to impute the unknown measurement(s). The last observation carried forward method is one way to impute values for the missing observations. For the last observation carried forward (LOCF) approach the missing values are replaced by the last observed value of that variable for each individual regardless of when it occurred.

Usage

```
locf(x)
```

Arguments

x	a vector, a data.frame or a matrix containing NAs.

Details

`locf()` replaces NAs with the most recent non-NA prior to it.

The function will replace all NAs found in a vector with the last earlier value not being NA. In data frames and matrices each column is treated separately, so that values are never carried across column boundaries. Factors are supported and keep their levels and ordering.

It should be noted, that the last observation carried forward approach may result in biased estimates and may underestimate the variability.

Value

an object of the same type and dimension as `x`.

Note

Based on code by Daniel Wollschlaeger, adapted to conform to package standards; multi-column, data-frame, and factor support added by the package author.

See Also

See also the package **Hmisc** for less coarse imputation functions.

Other vector.na: `coalesceX()`, `isNA()`, `naIf()`, `naReplace()`

Examples

```
d.frm <- data.frame(
  day=rep(c("mon", "tue", "wed", "thu", "fri", "sat", "sun"), 4)
, val=rep(c(runif(5), rep(NA,2)), 4) )

d.frm$locf <- locf( d.frm$val )
d.frm
```

---

logit *Logit Transformation and Its Inverse*

Description

Computes the logit transformation and its inverse for values defined on a finite interval $[min, max]$.

Usage

```
logit(x, min = 0, max = 1, eps = .Machine$double.eps, warn = FALSE)

logitInv(x, min = 0, max = 1)
```

Arguments

x	numeric vector. For logit(), values are interpreted relative to the interval $[min, max]$. For logitInv(), x can be any real number.
min	lower bound of the interval. Must be finite.
max	upper bound of the interval. Must be finite and greater than min.
eps	small positive value used to clamp probabilities away from 0 and 1 for numerical stability in logit(). Default: .Machine$double.eps.
warn	logical; if TRUE, a warning is issued when values are effectively clamped because they fall outside $(eps, 1 - eps)$ after rescaling. Default: FALSE.

Details

The logit() function maps values from $[min, max]$ to the real line $(-\infty, \infty)$. The inverse transformation logitInv() maps real-valued inputs back to $[min, max]$.

The logit transformation is defined as:

$$\text{logit}(x) = \log\left(\frac{p}{1-p}\right)$$

where

$$p = \frac{x - min}{max - min}.$$

For numerical stability, p is clamped to $[eps, 1 - eps]$ before applying the transformation. This prevents returning -Inf or Inf for values exactly equal to min or max, or slightly outside the interval due to floating point error.

If warn = TRUE, a warning is issued when such clamping occurs.

The inverse transformation is given by:

$$x = min + (max - min) \cdot \frac{1}{1 + e^{-z}}$$

where z is the input to logitInv().

Note that logitInv() does not perform clamping. This asymmetry is intentional: plogis() is well-defined for all real inputs, so no stabilization is required.

Value

a numeric vector of the same length as x.

See Also

qlogis(), plogis()

Other math.transform: linScale(), percentRank(), rankX(), winsorize()

Examples

```
x <- seq(0, 1, length.out = 5)
z <- logit(x)
logitInv(z)

# Boundary values are clamped internally:
# 0 -> eps, 1 -> 1 - eps
logit(c(0, 0.5, 1))

# With warn = TRUE, clamping at the boundaries triggers a warning
logit(c(0, 0.5, 1), warn = TRUE)

# Values strictly outside the interval also trigger a warning
logit(c(-0.1, 0.5, 1.1), warn = TRUE)

# Custom interval
x <- seq(10, 20, length.out = 5)
z <- logit(x, min = 10, max = 20)
logitInv(z, min = 10, max = 20)
```

---

long-wide-reshape *Reshape Between Long and Wide Format*

Description

Reshape data between long and wide format using a grouping variable.

Usage

```
toLong(x, varNames = NULL, includeRowNames = FALSE)

toWide(x, groups, by = NULL, varNames = NULL)
```

Arguments

x	object to reshape. For `toLong()`, a matrix, table, data frame, or list. For `toWide()`, a vector.
varNames	optional character vector of column names for the result.
includeRowNames	logical. If `TRUE`, append a column containing the row names of `x` when reshaping to long format.
groups	grouping vector used to define the columns in the wide result.
by	optional vector used to align values row-wise when reshaping to wide format. If `NULL`, values are aligned by their order within each group.

Details

`toLong()` expects `x` to be a matrix, table, data frame, or list and reshapes it to a long data frame representation. `toWide()` expects a vector `x` and a grouping vector `groups`, and reshapes the values into one column per group.

Value

a reshaped object of class `data.frame`.

See Also

`reshape()`, `stack()`, `unstack()`

Other data.reshape: `collapseTable()`, `splitAt()`, `splitX()`, `untable()`

Examples

```
d.x <- read.table(header = TRUE, text = "
AA BB CC DD EE FF GG
7.9 18.1 13.3 6.2 9.3 8.3 10.6
9.8 14.0 13.6 7.9 2.9 9.1 13.0
6.4 17.4 16.0 10.9 8.6 11.7 17.5
")

toLong(d.x)

# to wide by row order
toWide(PlantGrowth$weight, PlantGrowth$group)

# to wide aligned by key
set.seed(41)
PlantGrowth$nr <- c(sample(12, 10), sample(12, 10), sample(12, 10))
toWide(PlantGrowth$weight, PlantGrowth$group, by = PlantGrowth$nr)
```

---

mergeArgs *Merge Default Arguments with User Overrides*

Description

Helper used to merge defaults with user arguments, remove forbidden argument names, and optionally warn if forbidden arguments were supplied.

Usage

```
mergeArgs(defaults, user, forbidden = NULL, warn = TRUE)
```

Arguments

defaults	named list of default arguments.
user	named list of user-supplied arguments, or NULL.
forbidden	character vector of argument names that are not allowed.
warn	logical; whether to issue a warning if forbidden arguments are removed.

Details

User values override defaults of the same name. Unlike modifyList(), elements with the value NULL are preserved (so that an explicit NULL can be passed on as an argument value instead of silently deleting the entry).

Value

a named list of merged arguments.

See Also

modifyList()

Other pkg.args: callIf(), extractArgs(), getDotsArg(), recycle()

Examples

```
mergeArgs(list(col = "black", lty = 1), list(col = "red"))

# explicit NULL survives the merge
mergeArgs(list(col = "black"), list(col = NULL))
```

---

mGsub *Multiple String Substitution*

Description

Replaces multiple substrings in a character vector simultaneously, avoiding the cascade problem where an earlier replacement becomes the target of a later one. Internally uses temporary unique tokens as an intermediate step.

Usage

```
mGsub(x, patterns, replacements)
```

Arguments

x	a character vector in which substitutions are performed.
patterns	a character vector of substrings to search for (fixed = TRUE).
replacements	a character vector of replacement strings, in the same order as patterns.

Details

Patterns are processed in the given order. For overlapping patterns (e.g. "AB" and "A"), list the longer pattern first, otherwise the shorter one consumes its characters before the longer one is considered.

Value

a character vector of the same length as x.

See Also

mReplace() for exact whole-element replacement.

Other string.transform: char-ascii-conversion, strSplitToCol(), strSplitToDummy()

Examples

```
mGsub(c("foo bar", "bar foo"), c("foo", "bar"), c("bar", "foo"))
# [1] "bar foo" "foo bar"

# Without simultaneous replacement this would yield "foo foo"
# with sequential gsub().

x <- c("A", "B", "AB", "BA")
mGsub(x, patterns = c("A", "B"), replacements = c("BX", "CY"))
```

---

midx *Midpoints of a Numeric Vector*

Description

Compute the midpoints between consecutive elements of a numeric vector. This is useful, for example, when positioning labels in stacked bar plots.

Usage

```
midx(x, inclZero = FALSE, cumulate = FALSE)
```

Arguments

x	a numeric vector.
inclZero	logical. If TRUE, a zero is prepended to x before computing midpoints. In this case, the first midpoint equals x[1] / 2. Default is FALSE.
cumulate	logical. If TRUE, returns the cumulative sum of the midpoints. Default is FALSE.

Details

The midpoints are defined as:

$$m_i = \frac{x_i + x_{i+1}}{2}$$

When `inclZero = TRUE`, the computation is performed on `c(0, x)`.

Value

a numeric vector of length `length(x) - 1` (or `length(x)` if `inclZero = TRUE`) containing the midpoints. Returns an empty numeric vector if fewer than two values are available.

See Also

Other vector.window: `moveAvg()`, `quot()`

Examples

```
x <- c(1, 3, 6, 7)

midx(x)
midx(x, inclZero = TRUE)
midx(x, inclZero = TRUE, cumulate = TRUE)

# Example: label positions in a stacked bar plot
tab <- matrix(c(401,216,221,254,259,169), nrow = 2, byrow = TRUE)
b <- barplot(tab, beside = FALSE, horiz = TRUE)

xpos <- t(apply(tab, 2, midx, inclZero = TRUE, cumulate = TRUE))
text(x = xpos, y = b, labels = t(tab), col = "red")
```

---

moveAvg *Moving Average*

Description

Computes a simple moving average (running mean) of a numeric vector or time series.

Usage

```
moveAvg(
  x,
  order,
  align = c("center", "left", "right"),
  endrule = c("NA", "keep", "constant", "trim")
)
```

Arguments

x	a univariate numeric vector or ts object. Matrices and multi-column objects are not supported.
order	a single positive integer giving the window width. Must satisfy 1 <= order <= length(x).
align	a character string controlling how the window is positioned relative to each output value: "center" default. The window is centred on the current observation. For odd order the window is symmetric; for even order see Details. "left" the window starts at the current observation and extends to the right. "right" the window ends at the current observation and extends to the left.
endrule	a character string indicating how boundary values (where a full window is unavailable) are handled: "NA" default. Boundary values are left as NA. "keep" boundary values are taken from the original x. "constant" boundary values are filled with the nearest computed moving-average value. "trim" boundary values are computed from all available observations in a progressively smaller window.

Details

The core computation uses cumulative sums for O(n) efficiency:

$$\bar{x}_i = \frac{1}{k} \sum_j x_{i+j}$$

where the summation range depends on align.

Even-order windows and center alignment

For even order, centering is ambiguous. This implementation averages the two adjacent right-aligned windows of width order, which is the convention used by forecast::ma().

Boundary handling (endrule = "trim")

At the boundaries the window is contracted to include only the available observations. For center alignment with even order, the boundary window width at position i is $i + \lfloor order/2 \rfloor$.

Missing values

NA in x propagates through cumsum() and will produce NA in all moving-average values whose window contains that observation. There is no na.rm option; pre-filter with x[!is.na(x)] if needed (note this changes index positions).

Value

a vector of the same length and class as x, with NA at boundary positions unless endrule specifies otherwise.

See Also

zoo::rollmean(), forecast::ma(), runmed()

Other vector.window: midx(), quot()

Examples

```
moveAvg(AirPassengers, order = 5)
moveAvg(AirPassengers, order = 5, endrule = "trim")
moveAvg(AirPassengers, order = 4, align = "right", endrule = "constant")
```

---

mReplace *Replace Multiple Values in a Vector*

Description

Replaces elements of a character vector based on a lookup defined by two parallel vectors. Each element exactly matching a pattern is replaced with the corresponding replacement.

Usage

```
mReplace(x, patterns, replacements)
```

Arguments

x	a character vector whose elements are to be replaced.
patterns	a character vector of values to search for.
replacements	a character vector of replacement values, in the same order as patterns.

Value

a character vector of the same length as x, with matching elements replaced. Non-matching elements are returned unchanged.

See Also

mGsub() for substring replacement.

Other data.recode: asBinary(), combLevels(), dummy(), nf(), recodeX(), revCode(), stringsAsFactors()

Examples

```
mReplace(c("a", "b", "c", "d"), c("a", "c"), c("A", "C"))
# [1] "A" "b" "C" "d"
```

multMerge *Merge Multiple Data Frames*

Description

Merge multiple data frames by row names, or do other versions of database join operations.

Usage

```
multMerge(..., all.x = TRUE, all.y = TRUE, by = NULL)
```

Arguments

...	data frames to be coerced to one.
all.x	logical; if TRUE, then extra rows will be added to the output, one for each row in x that has no matching row in y. These rows will have NAs in those columns that are usually filled with values from y. The default is TRUE, so that non-matching rows are kept and padded with NAs (full outer join).
all.y	logical; analogous to all.x.
by	column used for merging, if this is not defined rownames will be used by default. The column must be included in all the provided data frames and its values must be unique within each data frame. Note that the restored key column is of type character.

Value

a data frame. The rows are sorted according to the appearance of previously unobserved rownames. So the rownames appearing in the first data frame are first, then the rownames in the second data frame, which have no corespondence in the first data frame and so on. The columns are the remaining columns in x1 and then those in x2 and then those in x3. The result has the row names resulting from the merge.

See Also

merge()

Other data.append: appendEnum(), appendRowNames(), appendX()

Examples

```
x1 <- setNamesX(data.frame(v = letters[1:6], w = 1:6),
                rownames = LETTERS[1:6])
x2 <- setNamesX(data.frame(v = letters[2:4], ww = 11:13),
                rownames = LETTERS[2:4])
x3 <- setNamesX(data.frame(v = letters[c(1, 3, 5, 7, 10)], wwww = 22:26),
                rownames = LETTERS[c(1, 3, 5, 7, 10)])

# the default merges on the row names and returns their union,
```

```
# with NA wherever a frame has no such row
multMerge(x1, x2, x3)

# v is not a key in the call above and is simply carried along from
# each frame; here it becomes the key instead
multMerge(x1, x2, x3, by = "v")
```

---

naIf *Replace Values with NA*

Description

Replaces specified values in a vector with NA, in the manner of SQL's NULLIF. This is the complementary operation to coalesceX().

Usage

```
naIf(x, values)
```

Arguments

x	a vector.
values	values to be replaced by NA.

Value

a vector of the same type as x.

See Also

Other vector.na: coalesceX(), isNA(), locf(), naReplace()

Examples

```
naIf(c(1, 2, 99, 3, 99), 99)
naIf(c("a", "b", "n/a", ""), c("n/a", ""))
```

naReplace *Replace NA Values*

Description

Replaces NA values in a vector or factor with a specified value.

Usage

```
naReplace(x, value)

## Default S3 method:
naReplace(x, value)

## S3 method for class 'factor'
naReplace(x, value)
```

Arguments

x	a vector or factor.
value	the replacement value. For factors, a single character string.

Details

For factors (including ordered factors), value is appended as a new level at the last position if it is not already present. If value is an existing level, the missing values are simply filled with it.

Value

an object of the same class as x with NA values replaced by value.

See Also

Other vector.na: coalesceX(), isNA(), locf(), naIf()

Examples

```
# default: numeric vector
naReplace(c(1, NA, 3), 0)

# character vector
naReplace(c("a", NA, "c"), "missing")

# unordered factor
naReplace(factor(c("a", "b", NA)), "missing")

# ordered factor: the new level is appended at the end
naReplace(factor(c("low", "high", NA), levels = c("low", "high"),
```

```
                ordered = TRUE), "unknown")
```

---

nf *Convert to Numeric via Factor*

Description

Converts an object to numeric by first coercing it to a factor and then to numeric. This is useful whenever a categorical or character variable needs a purely numeric stand-in – for example, as input to functions that require numeric data (distance calculations, correlation matrices, some modelling routines), or to obtain a compact, deterministic small-integer code for an ordinal variable by passing an explicit `levels` order.

Usage

```
nf(x, ...)
```

Arguments

`x`	a vector to be converted.
`...`	additional arguments passed to `factor()`.

Details

This function is a shorthand for `as.numeric(factor(x, ...))`.

Note that the resulting numeric values correspond to the internal factor levels, not the original numeric values. In particular, for character vectors holding numbers the codes follow the alphabetical level order (see the last example) – use `as.numeric(as.character(x))` to recover the values themselves.

Value

a numeric vector corresponding to the integer codes of the factor levels.

See Also

`factor()`, `as.numeric()`

Other data.recode: `asBinary()`, `combLevels()`, `dummy()`, `mReplace()`, `recodeX()`, `revCode()`, `stringsAsFactors()`

Examples

```
nf(c("a", "b", "a"))
nf(c("low", "medium", "high"), levels = c("low", "medium", "high"))

# caution: codes, not values
nf(c("10", "2"))    # 1 2, not 10 2
```

numeric-conversions *Convert Numbers Between Bases*

Description

Vectorized conversion between positional numeral systems (bases 2-36), plus Roman-numeral parsing. The convenience wrappers cover the most common cases; `baseToBase()` handles any combination of bases.

Usage

```
hexToDec(x)

decToHex(x)

octToDec(x)

decToOct(x)

binToDec(x)

decToBin(x)

romanToInt(x)

baseToBase(x, from, to, width = NULL)
```

Arguments

`x`	a vector of numbers or character strings representing values in the input base. For `baseToBase()` a numeric `x` is accepted only when `from = 10`. `NA` propagates to the output.
`from`	a single integer in [2, 36] specifying the input base (`baseToBase()` only).
`to`	a single integer in [2, 36] specifying the output base (`baseToBase()` only).
`width`	a single non-negative integer or `NULL` (default). When given, output strings are left-padded with zeros to at least `width` characters (`baseToBase()` only).

Value

a vector of the same length as `x`:

- `binToDec()`, `octToDec()`, `hexToDec()`, `romanToInt()` - integer or numeric vector.
- `decToHex()` - object of class `hexmode()`.
- `decToOct()` - numeric vector (octal digit string coerced to numeric).
- `decToBin()`, `baseToBase()` - character vector (uppercase digits).

`NA` input always produces `NA` output.

Convenience wrappers

All specialist functions are thin wrappers around `baseToBase()`:

Function	Equivalent call	Returns
`binToDec(x)`	`baseToBase(x, 2, 10)`	integer
`decToBin(x)`	`baseToBase(x, 10, 2)`	character
`octToDec(x)`	`baseToBase(x, 8, 10)`	integer
`decToOct(x)`	`baseToBase(x, 10, 8)`	numeric (octal digits)
`hexToDec(x)`	`baseToBase(x, 16, 10)`	integer
`decToHex(x)`	`baseToBase(x, 10, 16)`	`hexmode`

`hexToDec()` additionally strips a leading # from CSS-style colour strings.

Roman numerals

`romanToInt()` converts Roman numeral strings (e.g. `"XIV"`) to integers. Input is trimmed and upper-cased before parsing; invalid strings return `NA`. See also base R's `as.roman()` for the reverse direction.

Platform limits

`baseToBase()` uses `strtoi()` internally, which operates on `long int`. On 32-bit platforms values above $2^{31} - 1$ may silently return `NA`. `decToBin()` applies the same cap explicitly (values > `536870911` become `NA`).

See Also

`strtoi()`, `as.hexmode()`, `as.octmode()`, `as.roman()`

Examples

```
# binary
decToBin(c(0, 1, 17, 255))
binToDec(c("0", "1", "10001", "11111111"))

# octal
decToOct(c(8, 64, 255))
octToDec(c(10, 100, 377))

# hexadecimal  (CSS colour strings are also accepted by hexToDec)
decToHex(c(0, 255, 65535))
hexToDec(c("FF", "ff", "#1A2B3C"))

# Roman numerals
romanToInt(c("I", "IV", "XIV", "MCMXCIX"))   # 1, 4, 14, 1999
romanToInt("invalid")                          # NA

# baseToBase: general case
baseToBase("FF",       from = 16, to = 10)   # 255
```

```
baseToBase("255",      from = 10, to = 16)   # "FF"
baseToBase("11111111", from =  2, to = 10)   # 255
baseToBase("1A3F",     from = 16, to =  2)   # binary expansion
baseToBase("Z9",       from = 36, to = 10)   # base-36 -> decimal

# fixed-width padding (useful for bit-pattern alignment)
baseToBase(c(0, 7, 255), from = 10, to = 2, width = 8)

# vectorized over x
baseToBase(c("A", "B", "FF"), from = 16, to = 10)
```

---

nUnique *Count Unique Values*

Description

Returns the number of unique vector elements.

Usage

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

Arguments

x	a vector.
na.rm	logical. Should missing values (NA) be removed before counting unique values? Defaults to FALSE.

Value

an integer of length one.

See Also

nlevels(), isLowCardinality() to check whether x has at most a given number of unique values, without counting all of them first.

Other data.predicate: flags(), isDichotomous(), isEuclid(), isLowCardinality(), isNumeric(), isWholeLike(), isZero()

Examples

```
nUnique(c(1, 1, 2, 3))

nUnique(c(1, 1, 2, NA))

nUnique(c(1, 1, 2, NA), na.rm = TRUE)
```

nz *Extract Non-Zero Values*

Description

Returns all non-zero elements of a vector. Zeroness is determined by `isZero()`, i.e. within a numerical tolerance.

Usage

```
nz(x, tol = sqrt(.Machine$double.eps))
```

Arguments

`x`	a numeric vector.
`tol`	tolerance passed to `isZero()`.

Details

`NA` elements are not considered zero and are retained in the result.

Value

a vector containing only the non-zero elements of `x`.

See Also

`isZero()`

Other vector.utils: `unwhich()`

Examples

```
nz(c(0, 1, 2, 0, 3))
nz(c(1e-20, 1, NA))
```

`openDataObject` *Load Excel Data with Metadata (Codes and Labels)*

Description

Downloads an Excel file from a remote server and imports it as a data frame. Optionally processes a documentation sheet to assign variable labels and convert variables into factors with labeled levels.

Usage

```
openDataObject(name, url = NULL, doc = NULL, ...)
```

Arguments

`name`	character string. File name including extension (e.g. `"data.xlsx"`).
`url`	character string. Base URL where the file is located. Defaults to `https://raw.githubusercontent.com` (see https://github.com/AndriSignorell/Teaching/)
`doc`	list or `NA`. Defines the structure of the documentation sheet. If `NULL`, the function tries to detect a sheet named `"Description"`. If `NA`, no metadata processing is performed.
`...`	additional arguments passed to `readxl::read_excel()`.

Details

The function downloads the Excel file to a temporary location and reads the first sheet as the main dataset.

If a documentation sheet is available, it is expected to contain columns such as:

- Variable name
- Description (label)
- Codes (e.g. "1=Male | 2=Female")
- Scale ("nominal", "ordinal", etc.)

Variables with scale `"nominal"` or `"ordinal"` are converted to factors. Data values without a matching entry in the codes column become `NA`.

Value

a `data.frame` containing the imported data. If metadata is available:

- variables may be converted to factors (nominal/ordinal).
- factor levels are labeled using provided codes.
- variable labels are assigned using `label()`.

See Also

Other label.import: `dataDescription()`

Examples

```
# the file is downloaded from a remote repository, so the examples
# need an internet connection and fail gracefully without one

# load the dataset with automatic metadata detection
try(openDataObject("beauty.xlsx"))

# load it without metadata processing
try(openDataObject("beauty.xlsx", doc = NA))
```

---

pairApply *Pairwise Calculations*

Description

Implements a logic to run pairwise calculations on the columns of a data.frame or a matrix.

Usage

```
pairApply(x, FUN = NULL, ..., symmetric = FALSE)
```

Arguments

x	a list, a data.frame or a matrix with columns to be processed pairwise.
FUN	a function (or the name of a function) to be calculated. It is assumed, that the first 2 arguments denominate x and y, and that it returns a single numeric value.
...	the dots are passed to FUN.
symmetric	logical. Does the function yield the same result for FUN(x, y) and FUN(y, x)? If TRUE just the lower triangular matrix is calculated and mirrored. Default is FALSE.

Details

This code is based on the logic of `cor()` and extended for asymmetric functions. Cell `[i, j]` of the result contains `FUN(x[[i]], x[[j]], ...)`, so the first argument of `FUN` corresponds to the row variable and the second to the column variable.

Value

a matrix with the results of FUN.

See Also

`outer()`, `pairwise.table()`

Other combinatorics: `combN()`, `combPairs()`, `combSet()`, `permn()`, `randGroupSplit()`, `sampleX()`

Examples

```
# build a dataset
set.seed(1)
d.sub <- transform(
  data.frame(
    X1 = rnorm(n <- 300),
    X3 = rnorm(n)),
  X2 = 0.8*X1 + rnorm(n),
  X4 = 0.5*X3 + rnorm(n)
  )

pairApply(d.sub, FUN = cor, method="spearman")

# user defined functions are ok as well
pairApply(d.sub,
  FUN = function(x,y)
    wilcox.test(as.numeric(x), as.numeric(y))$p.value, symmetric=TRUE)
```

---

parseSasDatalines *Parse SAS DATALINES/CARDS blocks into a data.frame*

Description

A parser for simple SAS dataline command texts. A `data.frame` is being built with the column-names listed in the input section.

Usage

```
parseSasDatalines(x, validateNames = FALSE)
```

Arguments

`x`	a single character string containing a SAS DATA step with a DATALINES, CARDS, or CARDS4 block.
`validateNames`	logical. If `TRUE` (default `FALSE`), emits a warning when the dataset name violates SAS naming rules.

Details

The SAS function `DATA` is designed for quickly creating a dataset from scratch. The whole step normally consists out of the `DATA` part defining the name of the dataset, an `INPUT` line declaring the variables and a `DATALINES` command followed by the values.
The default delimiter used to separate the different variables is a space (thus each variable should be one word). The $ after the variable name indicates that the variable preceding contain character values and not numeric values. Without specific instructions, SAS assumes that variables are numeric. The function will fail, if it encounters a character in the place of an expected numeric value.

Each new row in datalines will create a corresponding unique row in the dataset. Notice that a ; is not needed after every row, rather it is included at the end of the entire data step.

More complex command structures, i.e. other delimiters (dlm), in the INPUT-section are not (yet) supported.

Only free-format (list) input is supported. The following SAS features are intentionally rejected with an informative error:

- Column pointers (@, @)
- Column input (e.g. var 1-10)
- Formatted input (:)

Character values must not contain spaces or quotes; scan-based parsing splits on whitespace and does not handle quoted strings.

Value

a data.frame with column names taken from the INPUT statement. The attribute sas_dataset_name carries the DATA step name. For DATA _NULL_ the data is still parsed and returned; the caller decides what to do with it (matching SAS semantics). SAS missing-value markers (.) are converted to NA.

See Also

Other file.io: pdfManual(), peekFile(), readDownload()

Examples

```
sas_code <- "
  data mydata;
    input name $ age score;
  datalines;
  Alice 30 95.5
  Bob   25 88.0
  ;
"
df <- parseSasDatalines(sas_code)
```

---

pdfManual *Open CRAN PDF Manual of a Package*

Description

PDF versions of the manual are usually not included as vignettes in R packages. Still this format is convenient for reading and doing full text search.
This function creates the appropriate link to the pdf file on CRAN and opens the pdf manual in a browser window.

Usage

```
pdfManual(package)
```

Arguments

package	package name (symbol or character).

Details

A warning (not an error) is issued if the package is not installed locally, as the manual may well exist on CRAN anyway.

Value

the URL of the PDF manual, invisibly. Called for its side effect of opening the browser.

See Also

browseURL()

Other file.io: parseSasDatalines(), peekFile(), readDownload()

Examples

```
# opens a browser window, hence only run in an interactive session
if (interactive()) {
  pdfManual(DescToolsX)
  pdfManual("bedrock")
}
```

---

peekFile *Preview a Delimited Text File*

Description

Read the first n data rows of a delimited text file and return the result as a base R data.frame (a kind of head() for files).

Usage

```
peekFile(file, n = 10, ..., output = c("data.frame", "tibble"))
```

Arguments

file	character string specifying the file name.
n	integer specifying the number of data rows to read, defaults to 10.
...	additional arguments passed to readr::read_delim(), e.g. delim or skip. The arguments n_max and show_col_types are managed internally and will be ignored if supplied; guess_max defaults to n but may be overridden.
output	character, either "data.frame" (default) or "tibble", determining the class of the returned object. Conversion to data.frame is done by toBaseR(). The argument can be abbreviated. Note that it must be given as a named argument, as it follows the dots.

Details

This function is intended for quickly inspecting large text files, including compressed files supported by readr::read_delim().

Column types are guessed from the previewed rows only (the default guess_max equals n). If early rows are not representative, supply a larger guess_max via the dots.

Value

a data.frame or a tibble (according to output) containing the first n data rows of the file.

See Also

readr::read_delim(), toBaseR(), head(),

Other file.io: parseSasDatalines(), pdfManual(), readDownload()

Examples

```
# a small file to look into
fn <- tempfile(fileext = ".csv")
write.csv(iris, fn, row.names = FALSE)

if (requireNamespace("readr", quietly = TRUE)) {

  peekFile(fn, delim = ",")

  # unrepresentative early rows: guess types over more lines
  peekFile(fn, n = 5, delim = ",", guess_max = 150)
}

unlink(fn)
```

percentRank *Percent Rank of a Numeric Vector*

Description

Computes the percent rank of each element in a numeric vector. The percent rank is defined as:

$$(rank(x) - 1)/(n - 1)$$

where n is the number of non-missing observations.

Usage

```
percentRank(x)
```

Arguments

x	a numeric (or comparable) vector.

Details

This corresponds to the definition used in SQL and `dplyr::percent_rank()`.

The smallest value in `x` receives a percent rank of 0, and the largest value receives a percent rank of 1 (if there are at least two non-missing values).

Ties are handled using `ties.method = "min"` via `rankX()`, meaning tied values receive the same minimal rank.

Missing values (`NA`) are preserved in the output.

If `x` contains fewer than two non-missing values, all results are `NA`.

Value

a numeric vector of the same length as `x`, containing values between 0 and 1.

See Also

`rank()`, `rankX()`

Other math.transform: `linScale()`, `logit()`, `rankX()`, `winsorize()`

Examples

```
x <- c(10, 20, 20, 30)
percentRank(x)

# With ties
x <- c(1, 2, 2, 3)
percentRank(x)
```

```
# With missing values
x <- c(3, NA, 1, 2)
percentRank(x)

# Single non-missing value
percentRank(c(5, NA, NA))
```

---

permn *Set of Permutations*

Description

Returns all distinct permutations of a vector. Repeated values in x are treated as indistinguishable, so duplicated permutations are not returned.

Usage

```
permn(x, sorted = FALSE)
```

Arguments

x	atomic vector. Missing values are not supported.
sorted	logical scalar. If TRUE, the result matrix is sorted using sortX(). Default is FALSE.

Value

a matrix containing all distinct permutations of x, one permutation per row.

See Also

combn(), factorial()

Other combinatorics: combN(), combPairs(), combSet(), pairApply(), randGroupSplit(), sampleX()

Examples

```
permn(letters[2:5])
permn(2:5)

# repeated elements are handled as indistinguishable
permn(c("a", "b", "c", "a"))
```

`Pizza` *Pizza Delivery Data (Extended)*

Description

An extended artificial dataset inspired by a similar dataset pizza.sav in *Arbeitsbuch zur deskriptiven und induktiven Statistik* by Toutenburg et al. The data describe a pizza delivery service in London serving three areas, each record being one order and its associated characteristics.

Usage

```
Pizza
```

Format

A data frame with 1209 observations on 22 variables:

index integer, index of the record, complete by construction.

date date of the delivery.

week numeric, the week of the year.

weekday numeric, the day of the week.

area factor with the levels `Brent`, `Camden` and `Westminster`.

count integer, the number of pizzas delivered.

rebate logical, `TRUE` if a rebate was given.

price numeric, the total price of the pizzas delivered.

operator factor with three levels, the operator taking the order.

driver factor with seven levels, the driver delivering the order.

delivery_min numeric, the delivery time in minutes.

temperature numeric, the temperature in degrees Celsius on delivery.

wine_ordered integer, `1` if wine was ordered, `0` if not.

wine_delivered integer, `1` if wine was delivered, `0` if not.

wrongpizza logical, `TRUE` if a wrong pizza was delivered.

quality ordered factor with the levels `low < medium < high`, the quality of the pizza on delivery.

vegetarian integer, `1` if the order was vegetarian, `0` if not.

nps numeric, the Net Promoter Score from 1 to 10, an ordinal customer rating.

complaint integer, `1` if a complaint was filed, `0` if not.

style character, the type of pizza, e.g. `italian`, `american`, `gourmet` or `vegan`.

channel character, the order channel, `app`, `web` or `phone`.

tip numeric, the tip in monetary units.

Details

Compared to the original dataset, this extended version includes additional behavioural and outcome variables such as customer satisfaction, Net Promoter Score (NPS), complaints, dietary choices and tipping behaviour. These variables are generated using probabilistic models to resemble realistic business data, including noise, imperfect relationships and heterogeneous customer behaviour.

The dataset is designed to be realistically complex. It contains the data types commonly met in practice: numerics, integers, factors, ordered factors, logicals, characters and dates. Missing values occur both systematically and at random, in every variable except `index`.

The variable `nps` is a simulated Net Promoter Score from 1 to 10, calibrated to resemble realistic customer feedback distributions, including asymmetric lower-tail behaviour.

The variable `complaint` is generated using a probabilistic model depending on delivery time, order correctness and additional noise, ensuring that complaints are not deterministically linked to single factors.

The variable `tip` is based on a percentage of the order price and is influenced by customer satisfaction (`nps`), delivery performance and driver-specific effects. Tips are zero for complaints or very low satisfaction, and otherwise increase monotonically with customer satisfaction while retaining stochastic variation.

Overall, the dataset is designed to provide a realistic benchmark for statistical modelling, including classification (binary and ordinal), regression and performance evaluation, e.g. ROC curves and AUC with confidence intervals.

Every variable carries a `label` attribute with its description, so that the labels can be used in tables and plots without repeating them in the code.

Source

Simulated data.

References

Toutenburg H, Schomaker M, Wissmann M, Heumann C (2009): *Arbeitsbuch zur deskriptiven und induktiven Statistik* Springer, Berlin Heidelberg.

See Also

Other datasets: `Cards`, `Roulette`, `Tarot`, `readCourseData()`

Examples

```
str(bedrock::Pizza)

summary(bedrock::Pizza$delivery_min)
table(bedrock::Pizza$area, bedrock::Pizza$channel)

# the missing values are part of the design
colSums(is.na(bedrock::Pizza))
```

`precision` *Precision, Decimal Places and Fractional Part of a Numeric Value*

Description

Four small utilities for the written form of a number, as opposed to its value.

Usage

```
nDec(x)

maxDec(x)

prec(x)

frac(x)
```

Arguments

`x`	a numeric vector, or a character vector of numbers as written.

Details

`nDec()` returns the number of decimal places of every element.
`maxDec()` returns the largest of those numbers.
`prec()` returns the precision, the smallest positional value of the last significant digit found in `x` (e.g. 0.001 for 3.142).
`frac()` returns the fractional part.

`nDec()` and `maxDec()` count what is printed: the input is converted with `as.character()`, an exponent is discarded, and the digits behind the last decimal separator are counted. A number that R chooses to print in scientific notation therefore has no decimal places, `nDec(1e-300)` is `0`, and trailing zeros of a numeric are gone before counting, as `1.50` and `1.5` are the same number. Pass the values as character strings to count them as written.

Where R switches to scientific notation is R's decision, not this function's, and it has moved between versions: up to R 4.2 `as.character()` followed `options(scipen=)`, since R 4.3 it writes the shortest representation that reads back as the same number. A value near that switch, such as `0.00001`, may therefore count five decimals or none, depending on the R version. Pass it as a character string to fix the count.

Both a period and a comma are accepted as the decimal separator of a character input, the last one in the string deciding, so that a thousands separator does not distort the count. Numeric input always arrives with a period, whatever `getOption("OutDec")` says.

`maxDec(x)` is the maximum of `nDec(x)`, missing values removed, and `0` when nothing is left to count.

`prec()` works on the value rather than on its written form and reports the position of the last significant digit across the whole vector, not one value per element. For input that is exact in decimal it is `10^-maxDec(x)`.

`frac()` discards the sign, the fractional part of `-1.25` being `0.25`, as the sign belongs to the integer part of the number. To read the decimals as an integer, scale and round the result, `round(1e4 * frac(x))` for the first four of them.

Value

- `nDec()`: an integer vector of the same length as `x`; `NA` elements yield `NA`.
- `maxDec()`: a single integer value, `0` if `x` has no non-missing element with decimals.
- `prec()`: a single numeric value, the finest precision found across all (non-missing) elements of `x`. Returns 1 if all values are zero and `NA` if no non-missing values are left.
- `frac()`: a numeric vector of the same length as `x`.

See Also

`format.info()`, `as.integer()`, `trunc()`

Examples

```
x <- rnorm(5)*100
x
frac(x)

# the first four decimal digits, as an integer
round(1e4 * frac(x))

# the sign belongs to the integer part
frac(c(-1.25, 1.25))
## [1] 0.25 0.25

nDec(c(1.25, 1.8, 12.0, 1.00000))
## [1] 2 1 0 0

# the same numbers, summarised
maxDec(c(1.25, 1.8, 12.0, 1.00000))
## [1] 2

x <- c("0.0000", "0", "159.283", "1.45e+10", "1.4599E+10" )
nDec(x)
prec(as.numeric(x))

# trailing zeros survive in a character input, but not in a numeric one
nDec("1.500")
## [1] 3
nDec(1.500)
## [1] 1
```

`primes` *Generate Prime Numbers up to Given Limits*

Description

Computes all prime numbers less than or equal to each value in `n`.

Usage

```
primes(n)
```

Arguments

`n`	a numeric vector of positive whole numbers, none exceeding 100,000,000.

Details

The function is vectorized over `n`. For a single value, the primes are returned as an integer vector; for several values, a named list is returned, with names corresponding to the input values.

Value

an integer vector containing the prime numbers less than or equal to `n` in ascending order if `n` is a single number, otherwise a named list of such vectors.

Upper limit

`n` may not exceed 100,000,000. The limit is a practical one, not a limit of the type: the sieve of Eratosthenes needs one bit per candidate and one integer per prime found, which at 100 million comes to roughly 12.5 MB for the sieve, 23 MB for the 5,761,455 primes, and a peak below about 70 MB once the copy into R is counted. At `.Machine$integer.max` the same three figures are 268 MB, 105,097,565 primes for 420 MB, and a peak beyond a gigabyte - which is why the integer limit is not a sensible bound here. A substantially larger range would call for a segmented sieve rather than for a larger allocation.

See Also

Other number.theory: `GCD-LCM`, `digitSum()`, `divisors()`, `factorize()`, `fibonacci()`, `isOdd()`, `isPrime()`

Examples

```
primes(10)
primes(c(5, 10))

# the number of primes below a limit
length(primes(1e6))
```

printCharMatrix *Pretty-print a character matrix with alignment, spacing and column splitting*

Description

Prints a character matrix to the console with configurable alignment, column spacing, optional row/column names, optional **cli**-based styling, and automatic splitting into column blocks if the output exceeds the console width.

Usage

```
printCharMatrix(
  m,
  align = "right",
  sep = 2,
  showRowNames = TRUE,
  showColNames = TRUE,
  useCliStyle = FALSE,
  width = getOption("width")
)
```

Arguments

m	a matrix (or object coercible to a matrix) containing values that will be converted to character for display.
align	character vector specifying alignment of cell contents, either "right" (default) or "left". A single value is recycled across all columns; alternatively a vector of length ncol(m) sets the alignment per column.
sep	integer. Number of spaces between columns. Default is 2.
showRowNames	logical. Should row names be printed? Default is TRUE.
showColNames	logical. Should column names be printed? Default is TRUE.
useCliStyle	logical. If TRUE, column names and row names are styled using cli::style_bold(). Default is FALSE.
width	integer. Maximum output width (in characters). Defaults to getOption("width"). If the table exceeds this width, it is split into column blocks that are printed one after another.

Details

The function formats all entries as character strings and computes column widths dynamically. NA entries are shown as "NA". If the full table does not fit into the specified width, it is split column-wise into multiple blocks (cell contents themselves are never wrapped). In this case, row names and column headers are repeated for each block.

If a single column is wider than `width`, that column is printed on its own and the requested `width` is deliberately exceeded, since a column cannot be split further.

Alignment is applied per column, and spacing between columns is controlled via `sep`. The function is designed as a lightweight alternative to `print.data.frame()` with more control over formatting, making it suitable for reporting outputs in packages.

Value

invisibly returns `NULL`. The formatted table is printed to the console.

See Also

Other data.print: `columnWrap()`

Examples

```
m <- matrix(c(
  "50.575","50.543","45.207",
  "49.900","51.400","44.300",
  "5.106","8.192","10.197"
), nrow = 3, byrow = TRUE)

rownames(m) <- c("mean","median","sd")
colnames(m) <- c("Brent","Camden","Westminster")

# Default (right-aligned)
printCharMatrix(m)

# Left-aligned with custom spacing
printCharMatrix(m, align = "left", sep = 4)

# With CLI styling (requires cli package)
if (requireNamespace("cli", quietly = TRUE)) {
  printCharMatrix(m, useCliStyle = TRUE)
}

# Force splitting into column blocks by reducing width
printCharMatrix(m, width = 20)
```

---

ptInPoly *Point-in-Polygon Test (Angle Summation, Rcpp)*

Description

Determines whether points lie inside a polygon. Points located exactly on polygon edges or vertices are treated as **inside**.

Usage

```
ptInPoly(x, y, polyX, polyY)
```

Arguments

x	numeric vector of x-coordinates of the query points.
y	numeric vector of y-coordinates of the query points.
polyX	numeric vector of x-coordinates of the polygon vertices.
polyY	numeric vector of y-coordinates of the polygon vertices.

Details

The function uses a numerically stable angle summation algorithm implemented in C++ via Rcpp.

- The polygon is implicitly closed (last vertex connects to the first).
- The polygon is assumed to be non-self-intersecting.
- Numerical robustness is ensured via epsilon-based comparisons.

Value

an integer vector of length `length(x)`:

0 point is outside the polygon.

1 point is inside the polygon or on its boundary.

Examples

```
# Define a square
px <- c(0, 1, 1, 0)
py <- c(0, 0, 1, 1)

# Query points
x <- c(0.5, 1.5, 0, 0.5)
y <- c(0.5, 0.5, 0, 1)

ptInPoly(x, y, px, py)
```

---

quot *Lagged Quotients*

Description

Returns suitably lagged and iterated quotients.

Usage

```
quot(x, lag = 1L, quotients = 1L, ...)
```

Arguments

x	a numeric vector or matrix containing the values to be used for calculating the quotients.
lag	an integer indicating which lag to use.
quotients	an integer indicating the order of the quotient.
...	further arguments to be passed to or from methods.

Details

NA()'s propagate.

Value

if x is a vector of length n and quotients = 1, then the computed result is equal to the successive quotients x[(1+lag):n] / x[1:(n-lag)].

If quotients is larger than one this algorithm is applied recursively to x. Note that the returned value is a vector which is shorter than x.

If x is a matrix then the division operations are carried out on each column separately.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

diff()

Other vector.window: midx(), moveAvg()

Examples

```
quot(1:10, 2)
quot(1:10, 2, 2)
x <- cumprod(cumprod(1:10))
quot(x, lag = 2)
quot(x, quotients = 2)
```

randGroupSplit *Randomly Split a Vector into Groups of Given Sizes*

Description

Randomly assigns the elements of a vector x into groups with predefined sizes given by groupSizes. The grouping is performed without replacement and each element is assigned to exactly one group.

Usage

```
randGroupSplit(x, groupSizes)
```

Arguments

x	a vector containing the elements to be split into groups.
groupSizes	an integer vector specifying the sizes of the groups. The sum of groupSizes must equal length(x). If the vector is named, the names are used as group names in the result.

Details

This function is useful for random group assignments, for example in teaching settings, simulations, or experimental designs where groups of unequal sizes are required. It uses sample(), so results can be made reproducible with set.seed().

Value

a list of vectors, where each element corresponds to one group. The length of the list equals length(groupSizes).

See Also

Other combinatorics: combN(), combPairs(), combSet(), pairApply(), permn(), sampleX()

Examples

```
# Split letters into 3 groups of sizes 4, 3, and 5
set.seed(123)
randGroupSplit(LETTERS[1:12], groupSizes = c(4, 3, 5))

# named groups
randGroupSplit(LETTERS[1:7], groupSizes = c(treat = 4, ctrl = 3))
```

range-operators *Select a Range Between Two Elements*

Description

Returns the elements of `x` from the first occurrence of `rng[1]` up to an occurrence of `rng[2]`. The two operators differ in which occurrence of the end value terminates the range, analogous to lazy and greedy quantifiers in regular expressions:

Usage

```
x %:% rng

x %::% rng
```

Arguments

`x`	a vector.
`rng`	a vector of length 2: `c(from, to)`. May contain `NA` to match missing values in `x`.

Details

- `%:%` (lazy): up to the *first* occurrence of `rng[2]`.
- `%::%` (greedy): up to the *last* occurrence of `rng[2]`.

Value

a subset of `x`, from the first occurrence of `rng[1]` to the first (`%:%`) or last (`%::%`) occurrence of `rng[2]`.

See Also

Other data.interval: `between-operators`, `intervals`

Examples

```
letters %:% c("c", "g")

x <- c("a", "b", "c", "d", "c", "e", "f", "c")
x %:%  c("c", "e")
x %::% c("c", "c")     # greedy: up to the last "c"

# select a column range by name
colnames(mtcars) %:% c("hp", "vs")
```

rankX *Fast Ranking with Extended Tie Handling*

Description

Computes ranks for vectors or multiple inputs using a fast implementation based on `data.table::frankv`. Supports additional tie-handling methods such as `"dense"` and multi-column ranking via `...`.

Usage

```
rankX(
  ...,
  decreasing = FALSE,
  na.last = TRUE,
  ties.method = c("average", "first", "last", "random", "max", "min", "dense")
)
```

Arguments

`...`	one or more vectors to be ranked. If multiple vectors are provided, they are ranked lexicographically (like `order`). All inputs must have the same length.
`decreasing`	logical; if `TRUE`, larger values receive smaller ranks (i.e., ranking in descending order). When ranking multiple inputs, a logical vector may be given to control the direction per input.
`na.last`	logical or `"keep"`; determines the placement of `NA` values. Passed to `data.table::frankv`.
`ties.method`	character string specifying how ties are handled. One of: `"average"` average of the ranks for tied values (default). `"first"` ranks assigned in order of appearance. `"last"` ranks assigned in reverse order of appearance. `"random"` ranks assigned at random. `"max"` maximum rank for tied values. `"min"` minimum rank for tied values. `"dense"` like `"min"`, but ranks are consecutive integers without gaps.

Details

This function is a fast alternative to `rank()`, powered by `data.table::frankv`. It extends base functionality by:

- Supporting dense ranking (`ties.method = "dense"`)
- Allowing multiple input vectors for lexicographic ranking
- Providing improved performance for large datasets

When multiple inputs are supplied, ranking is performed jointly, similar to:

```
order(x1, x2, ...)
```

Value

an integer or numeric vector of ranks with the same length as the input.

See Also

`rank()`, `data.table::frankv()`

Other math.transform: `linScale()`, `logit()`, `percentRank()`, `winsorize()`

Examples

```
x <- c(10, 20, 20, 30)

# Basic ranking
rankX(x)

# Dense ranking
rankX(x, ties.method = "dense")

# Descending order
rankX(x, decreasing = TRUE)

# Handling NA values
x2 <- c(3, NA, 1, 2)
rankX(x2, na.last = "keep")

# Multi-column ranking
a <- c(1, 1, 2, 2)
b <- c(2, 1, 2, 1)
rankX(a, b)
```

---

rBetaShape *Generate Beta-Distributed Random Values by Shape*

Description

Generates beta-distributed random values using predefined distributional shapes and transforms them to a specified interval.

Usage

```
rBetaShape(
  n,
  shape = c("norm", "left", "right", "unif", "u", "j", "inv-j"),
  bounds = c(0, 1)
)
```

Arguments

n	non-negative integer giving the number of values to generate.
shape	distributional shape: either one of the predefined names listed under **Details**, or a numeric vector of length 2 giving shape1 and shape2 directly.
bounds	numeric vector containing the lower and upper bound.

Details

The following predefined shapes and beta parameters are available:

Shape	shape1	shape2	Description
"norm"	5.0	5.0	symmetric and bell-shaped
"left"	5.0	2.0	left-skewed with values concentrated near the upper bound
"right"	2.0	5.0	right-skewed with values concentrated near the lower bound
"unif"	1.0	1.0	uniform
"u"	0.5	0.5	U-shaped with values concentrated near both bounds
"j"	2.0	0.5	J-shaped with values concentrated near the upper bound
"inv-j"	0.5	2.0	inverse J-shaped with values concentrated near the lower bound

Note that "left" and "right" name the direction of the *skew*, i.e. of the long tail, so "right" places the bulk of the values near the lower bound. This is the standard convention, but it is the opposite of what the names suggest at first reading - and unrelated to the meaning of "left"/"right" in the sides argument of the interval functions, where they name the side carrying the finite bound.

The "norm" shape is symmetric and bell-shaped but is not a normal distribution. Unlike the normal distribution, all generated values are bounded.

Values from the standard beta distribution on the interval $[0, 1]$ are transformed to the interval specified by bounds as

$$a + (b - a)X$$

where a and b are the lower and upper bounds, respectively.

Value

a numeric vector of length n with values within bounds.

Random number generation

The values are drawn with rbeta() and therefore depend on the state of R's global random number generator. No seed is set internally; call set.seed() beforehand, or wrap the call in withSeed(), for reproducible results.

See Also

rbeta(), runif()

Other random.numbers: rSum21()

Examples

```
set.seed(42)

x <- rBetaShape(
  1000,
  shape = "right",
  bounds = c(10, 90)
)

summary(x)
range(x)

# shape parameters can also be given directly
rBetaShape(5, shape = c(3, 1.5), bounds = c(0, 100))
```

`rdLabels` *Extract Variable Labels from Rd Documentation*

Description

Reads the variable descriptions out of the `\describe` section of a documented dataset and returns them as a named character vector, the names being the variable names. This turns documentation that already exists into labels usable in tables, plots and codebooks, instead of maintaining the same descriptions a second time in the code.

Usage

```
rdLabels(dataName, package)
```

Arguments

`dataName`	character string, the name of the dataset.
`package`	character string, the name of the package holding the dataset.

Details

The Rd database is read with `tools::Rd_db()` and searched recursively for the first `\describe` section, from which all `\item{var}{description}` entries are taken. Only that first section is read: on a page documenting more than one dataset, the labels of the first one are returned.

Descriptions are returned as written in the Rd file, with whitespace and line breaks collapsed to single spaces. Rd markup inside a description, such as `\code{}` or `\eqn{}`, contributes its content without the surrounding command.

The package must be installed, as the documentation is read from the installed Rd database rather than from the sources.

Value

a named character vector of variable descriptions, the names being the variable names.

See Also

`tools::Rd_db()`

Other pkg.funinfo: `auditNames()`, `funArgs()`, `funCalls()`, `funKeywords()`, `funList()`, `rdTitle()`

Examples

```
# the labels of a documented dataset, taken from the \describe
# section of its help page
rdLabels("Pizza", "bedrock")
```

---

rdTitle *Extract the Title from an Rd Help File*

Description

Searches all '`.Rd`' files in a package's '`man/`' directory for a given topic (matched against `\\alias` entries) and returns its `\\title` string.

Usage

```
rdTitle(topic, man = "man")
```

Arguments

`topic`	a single character string giving the topic (function name or alias) to look up.
`man`	a single character string giving the path to the directory containing '`.Rd`' files. Defaults to `"man"`, i.e. the '`man/`' subdirectory of the current working directory.

Value

a single character string with the title, trimmed of leading and trailing whitespace. Stops with an error if `topic` is not found.

See Also

`tools::parse_Rd()`

Other pkg.funinfo: `auditNames()`, `funArgs()`, `funCalls()`, `funKeywords()`, `funList()`, `rdLabels()`

Examples

```
# a minimal man/ directory to search in
man <- file.path(tempdir(), "man")
dir.create(man, showWarnings = FALSE)

writeLines(c("\\\\name{foo}", "\\\\alias{foo}", "\\\\alias{bar}",
             "\\\\title{A Minimal Help Page}",
             "\\\\description{Nothing to see here.}"),
           file.path(man, "foo.Rd"))

rdTitle("foo", man = man)
rdTitle("bar", man = man)          # aliases are matched as well
rdTitle("nothing", man = man)      # NA

unlink(man, recursive = TRUE)
```

---

readCourseData *Load Course Dataset from Server*

Description

Downloads and loads a dataset from predefined course servers or a user-defined URL.

Usage

```
readCourseData(name, url = NULL, header = TRUE, sep = ";", ...)
```

Arguments

name	character string. File name including extension (e.g. "data.csv").
url	optional character string. Base URL where the file is located. If NULL, default course repositories are searched.
header	logical. Whether the file contains a header row. Passed to read.table().
sep	character. Field separator used in the file. Default is ";".
...	additional arguments passed to the underlying import functions such as read.table() or openDataObject().

Details

If no url is provided, the function searches for the file in the following locations (see https://github.com/AndriSignorell/Teaching):

- https://raw.githubusercontent.com/AndriSignorell/Teaching/main/book/
- https://raw.githubusercontent.com/AndriSignorell/Teaching/main/data/

The first location where the file exists is used.

File type handling:

- `.xls`, `.xlsx`: loaded via `openDataObject()`
- other files: loaded via `read.table()`

Value

a data frame or object returned by the respective import function:

- for text files: a `data.frame`.
- for Excel files: an object returned by `openDataObject()`.

See Also

Other datasets: `Cards`, `Pizza`, `Roulette`, `Tarot`

Examples

```
# the files are fetched from a remote repository, so the examples
# need an internet connection and fail gracefully without one

# load from the default repositories
try(readCourseData("fullmoon.xlsx"))

# load from a custom URL
try(readCourseData("mydata.csv", url = "https://example.com/data/"))
```

---

readDownload *Read a File from the Downloads Directory*

Description

Reads a file from the Downloads directory and returns it as a data frame. The file type is automatically detected from the extension.

Usage

```
readDownload(file, ..., output = c("data.frame", "tibble"))
```

Arguments

`file`	character string specifying the name of the file.
`...`	additional arguments passed to the underlying read function, e.g. `sheet` for Excel files or `delim` for text files.
`output`	character, either `"data.frame"` (default) or `"tibble"`, determining the class of the returned object. Conversion to `data.frame` is done by `toBaseR()`. The argument can be abbreviated. Note that it must be given as a named argument, as it follows the dots.

Details

This is a convenience wrapper combining `findDownload()` with common file readers:

- Excel files (`.xls`, `.xlsx`) via `readxl::read_excel`
- CSV files via `readr::read_csv`
- TSV files via `readr::read_tsv`
- Text files (`.txt`) via `readr::read_delim`, which guesses the delimiter from the file content

For the readr-based formats the column specification message is suppressed by default; supply `show_col_types = TRUE` to restore it. By default, the result is converted to a base R `data.frame`.

Value

a `data.frame` or a tibble, according to `output`.

See Also

`findDownload()`, `toBaseR()`, `readxl::read_excel()`, `readr::read_csv()`

Other file.io: `parseSasDatalines()`, `pdfManual()`, `peekFile()`

Examples

```
## Not run:
# cannot be run automatically: reads the personal Downloads
# directory of the user, where no such files exist

# read an Excel file
readDownload("data.xlsx")

# read a CSV file
readDownload("data.csv")

# keep the tibble output
readDownload("data.csv", output = "tibble")

## End(Not run)
```

---

recodeX *Recode a Variable*

Description

Combining or rearranging a factor can be tedious if it has many levels. `recodeX()` supports this step by accepting a direct definition of new levels by enumerating old levelnames as argument and adding an `"elseLevel"` option. If new levels are given as integer values they will be translated in the according levels.

Usage

```
recodeX(
  x,
  ...,
  keep = NULL,
  elseLevel = NA,
  ref = NULL,
  useEmpty = FALSE,
  num = FALSE
)
```

Arguments

x	the factor whose levels are to be altered. If x is character it will be factorized (using factor defaults) but returned as character again.
...	the old levels (combined by c() if there are several) named with the new level: newlevel_a = c("old_a", "old_b"), newlevel_b = c("old_c", "old_d") See examples.
keep	vector of levels that should be left untouched.
elseLevel	the value for levels, which are not matched by newlevel list. If this is set to NULL, the elseLevels will be left unchanged. If set to NA (default) non matched levels will be set to NA.
ref	the reference level, typically a string.
useEmpty	logical. Defines how a new level, which can't be found in x, should be handled. Should it be left in the level's list or be dropped? The default is FALSE, which drops empty levels.
num	logical. If set to TRUE the result will be numeric. This is useful if you want to recode strings to specific numeric values.

Value

the factor having the new levels applied.
if x was a character vector, the result will also be character.

See Also

factor(), levels(), relevel(), reorder()
There's another solution for this problem in the package **car**.

Other data.recode: asBinary(), combLevels(), dummy(), mReplace(), nf(), revCode(), stringsAsFactors()

Examples

```
set.seed(1984)
x <- factor(sample(1:15, 20, replace=TRUE))
levels(x) <- paste("old", levels(x), sep="_")
```

```
y <- recodeX(x,
             "new_1"   = c("old_1","old_4","old_5"),
             "new_2"   = c("old_6","old_10","old_11"),
             "new_3"   = c("old_12","old_13"),
             elseLevel = "other")
data.frame(x=x, y=y)

# Coding NAs, NA is recoded to new_1
x[5:6] <- NA
x <- x[1:7]

data.frame(
  x,
  RecodeNA = recodeX(x,
                     "new_1"   = c("old_4","old_8", NA),
                     elseLevel = "other"),

  # NAs remain unaffected, unless specified to be processed
  NoRecodeNA = recodeX(x,
                       "new_1"   = c("old_4","old_8"),
                       elseLevel = "other")
)

# keep some levels, collapse others and reset the reference level
ff <- factor(c("apple","pear","banana","kiwi",
               "mango","peach","grape","plum"))
recodeX(ff,
       stone=c("peach", "plum"),
       keep=c("apple","banana"),
       elseLevel = "other", ref="stone")


x <- factor(letters[1:6])

z1 <- recodeX(x, AB=c("a","b"), CD=c("c","d"), elseLevel="none of these")
z2 <- recodeX(x, AB=c("a","b"), CD=c("c","d"), elseLevel=NA)
z3 <- recodeX(x, AB=c("a","b"), CD=c("c","d"), elseLevel=NULL)
z4 <- recodeX(x, AB=c("a","b"), GH=c("g","h"), elseLevel=NA, useEmpty=TRUE)
z5 <- recodeX(x, AB=c("a","b"), GH=c("g","h"), elseLevel=NA, useEmpty=FALSE)

data.frame(z1, z2, z3, z4, z5)

lapply(data.frame(z1, z2, z3, z4, z5), levels)

# empty level GH exists in z4...
table(z4, useNA="ifany")
# and is dropped in z5
table(z5, useNA="ifany")

# use integers to define the groups to collapse
set.seed(1972)
(likert <- factor(sample(1:10, size=15, replace=TRUE),
                  levels=1:10, labels=gettextf("(%s)", 1:10)))
```

```
recodeX(likert, det=1:6, pas=7:8, pro=9:10)

# or directly turned to numeric
recodeX(likert, "1"=1:6, "2"=7:8, "5"=9:10, num=TRUE)
```

`recycle` *Recycle a List of Elements*

Description

This function recycles all supplied elements to the maximal dimension.

Usage

```
recycle(..., maxDim = NULL, strict = FALSE)
```

Arguments

`...`	a number of vectors of elements.
`maxDim`	defines the maximal dimension, if set to `NULL` (default) the maximal dimension of the list.
`strict`	logical, if `TRUE` each element must have length 1 or `maxDim`, so that no partial recycling (or truncation) can occur. Default is `FALSE`.

Details

If `maxDim` is smaller than the length of an element, that element is truncated to the first `maxDim` values. Zero-length elements are recycled to `NA` vectors of length `maxDim`. Both situations are rejected when `strict = TRUE`.

Value

a list of the supplied elements
`attr(,"maxdim")` contains the maximal dimension of the recycled list.

See Also

`rep()`, `replicate()`

Other pkg.args: `callIf()`, `extractArgs()`, `getDotsArg()`, `mergeArgs()`

Examples

```
recycle(x=1:5, y=1, s=letters[1:2])

z <- recycle(x=letters[1:5], n=2:3, sep=c("-"," "))
sapply(1:attr(z, "maxdim"), function(i) paste(rep(z$x[i], times=z$n[i]),
                                              collapse=z$sep[i]))
```

renameX *Rename Elements of a Named Object*

Description

Renames selected elements of a named object by specifying old-to-new name mappings. Works on any R object that supports `names()`, including vectors, lists, data frames, and matrices. For matrix-like objects, `rownames` and `colnames` can be targeted via the `which` argument.

Usage

```
renameX(
  x,
  ...,
  on = c("names", "rownames", "colnames"),
  useGsub = FALSE,
  fixed = TRUE,
  warn = TRUE
)
```

Arguments

`x`	a named object. Any type that supports `names()`, `rownames()`, or `colnames()`, e.g. a vector, list, data frame, or matrix.
`...`	name mappings of the form `old = "new"`, a single function to apply to all names (e.g. `toupper`), or unnamed character strings applied positionally (see Details).
`on`	character scalar specifying which names to operate on. One of `"names"` (default), `"rownames"`, or `"colnames"`. Partial matching is supported.
`useGsub`	logical scalar. If `TRUE`, each mapping is applied as a `gsub()` pattern substitution across all current names rather than an exact replacement. Default is `FALSE`.
`fixed`	logical scalar. Passed to `gsub()` when `useGsub = TRUE`. If `TRUE` (default), patterns are treated as fixed strings rather than regular expressions.
`warn`	logical scalar. If `TRUE` (default), a warning is issued when one or more old names supplied in `...` are not found in the targeted names of `x`. Only relevant in exact mode (`useGsub = FALSE`).

Details

The function supports three modes:

Exact mode (`useGsub = FALSE`, **default**) Names are matched exactly via `match()`. Each element of `...` must be a named scalar character string of the form `old = "new"`. Unmatched old names trigger a warning when `warn = TRUE`.

Pattern mode (`useGsub = TRUE`) Each mapping is treated as a `gsub()` substitution applied in sequence to *all* current names. The left-hand side is the pattern, the right-hand side is the replacement. The `fixed` argument is forwarded to `gsub()`.

Function mode If a single function is passed in ..., it is applied to all current names. Useful for bulk transformations such as `toupper`, `tolower`, or `make.names`.

When ... contains unnamed character elements, the names are assigned positionally: the first element replaces `names(x)[1]`, the second `names(x)[2]`, and so on.

Value

the object `x` with updated names; all other attributes are preserved.

See Also

`names()`, `setNames()`

Other label.attrs: `label()`, `setAttr-removeAttr-keepAttr`, `setNamesX()`

Examples

```
x <- c(a = 1, b = 2, c = 3)

# Exact mode: rename by old = "new" pairs
renameX(x, a = "alpha", c = "gamma")

# Positional mode: replaces names(x)[1:2]
renameX(x, "alpha", "beta")

# Function mode: apply a function to all names
renameX(x, toupper)
renameX(x, tolower)

# Data frame columns
d <- data.frame(foo = 1:3, bar = 4:6)
renameX(d, foo = "x", bar = "y")

# Pattern mode: strip a common prefix
y <- c(v_mean = 1, v_sd = 2, v_n = 3)
renameX(y, v_ = "", useGsub = TRUE)

# Pattern mode with regex (fixed = FALSE)
renameX(y, `^v_` = "", useGsub = TRUE, fixed = FALSE)

# Matrix: rename colnames selectively
m <- matrix(1:6, nrow = 2,
            dimnames = list(c("row_a", "row_b"), c("col_x", "col_y", "col_z")))
renameX(m, col_x = "alpha", on = "colnames")

# Matrix: uppercase all rownames via function mode
renameX(m, toupper, on = "rownames")

# Matrix: rename rownames via gsub
renameX(m, `row_` = "", useGsub = TRUE, fixed = FALSE, on = "rownames")
```

`resolveContingency` *Resolve a Contingency Table*

Description

Brings a two-way classification into one canonical shape, no matter whether it arrives as a ready-made contingency table or as two classification variables. The function validates the counts, drops the incomplete observations and reports the table together with its dimensions, so that association measures, tests of independence and agreement statistics can share one entry point instead of each repeating the same preparation.

Usage

```
resolveContingency(
  x,
  y = NULL,
  square = FALSE,
  integerCounts = TRUE,
  dataName = NULL
)
```

Arguments

`x`	a contingency table or matrix of counts, or a factor or vector of classifications.
`y`	an optional factor or vector of classifications, of the same length as `x`. Required unless `x` is a table, ignored when it is.
`square`	logical, whether a square contingency table is required, defaults to `FALSE`. For two classification variables, both are tabulated over the union of their levels (see Details).
`integerCounts`	logical, whether non-integer counts should be reported with a warning, defaults to `TRUE`.
`dataName`	optional character string used as the `dataName` entry of the result. If `NULL` (default), it is derived from the unevaluated arguments. That name only reflects what `resolveContingency()` itself sees: a function calling it internally should build its own name from `substitute()` at its own call site and pass it through here, as it would otherwise report its own formal argument names, typically `"x and y"`, instead of the names the end user typed.

Details

Any two-dimensional object is taken as a contingency table and used as it is, which covers a matrix as well as a `table()` or `xtabs()` object; a data frame of counts is coerced with `as.matrix()`. Its entries must be numeric, non-negative and finite; non-integer counts are reported with a warning unless `integerCounts` is set to `FALSE`, as they occur legitimately in weighted or expected tables. An array of any other number of dimensions is an error, rather than being flattened into a classification variable.

Two classification variables are cross-tabulated instead. Observations missing in either variable are dropped, both variables are then coerced to factors, which drops the levels that no longer occur, and at least two levels must remain on each side.

Whichever way the table arrives, it must have at least two rows and two columns: a one-way table carries no association to measure and is rejected rather than passed on to a caller that cannot use it.

`square` is meant for the statistics that compare two ratings of the same items, such as the tests of marginal homogeneity or the agreement measures. For two classification variables it tabulates both over the union of their observed levels (those of `x` first), so the axes carry the same categories in the same order and a category used by only one rating becomes a row or column of zeros: `x` using A, B and `y` using B, C give a 3 x 3 table over A, B, C instead of rows A, B against columns B, C. For a ready-made table it only guarantees as many columns as rows: whether the two axes really carry the same categories cannot be checked on a table that may have no `dimnames` at all, and remains the responsibility of the caller.

Value

a list containing:

table the contingency table.

n the total sample size, the sum of all counts.

r integer, the number of rows.

c integer, the number of columns.

dataName character description of the input, for use as the `data.name` of an `htest` object.

See Also

`table()`, `resolveGroups()`, `resolveFormula()`

Other data.resolve: `resolveFormula()`, `resolveGroups()`

Examples

```
# from an existing contingency table
tab <- matrix(c(10, 5, 3, 12), nrow = 2,
              dimnames = list(c("A", "B"), c("yes", "no")))
str(resolveContingency(tab))

# from two classification variables
set.seed(1)
x <- sample(c("low", "high"), 100, replace = TRUE)
y <- sample(c("yes", "no"), 100, replace = TRUE)
resolveContingency(x, y)$table

# a caller passes the name it sees at its own call site
myTest <- function(x, y) {
  r <- resolveContingency(x, y,
                          dataName = paste(deparse1(substitute(x)), "and",
                                           deparse1(substitute(y))))
  r$dataName
}
```

```
myTest(x, y)
## [1] "x and y"
```

`resolveFormula` *Parse and Classify a Model Formula*

Description

Parses a model formula, builds the model frame and classifies the resulting design into one of seven dependency structures. The pieces of the design are returned under a fixed set of names, so that every function offering a formula interface can share one entry point instead of re-implementing the parsing, the `subset` handling and the distinction between a grouping factor, a numeric predictor and a blocking variable.

`resolveFormulaFromCall()` is the entry point for a function offering a formula interface: called from its body, it forwards the caller's `formula`, `data` and `subset` to `resolveFormula()` exactly as they were written, so that `subset` keeps its base R semantics (see Details).

Usage

```
resolveFormula(
  formula,
  data,
  subset,
  na.action = na.pass,
  allowed = c("one-sample", "two-sample-independent", "two-sample-dependent",
   "n-sample-independent", "n-sample-dependent", "numeric-numeric", "regression")
)

resolveFormulaFromCall(allowed, na.action = na.pass)
```

Arguments

`formula`	a two-sided model formula. Supported forms are: `y ~ 1` one-sample design. `Pair(x, y) ~ 1` two-sample dependent (paired). `Pair()` constructs a two-column matrix of paired observations. `y ~ g` two-sample or n-sample independent group comparison. `y ~ a:b` independent group comparison of the cells of several grouping variables, combined into one grouping factor. `y ~ a + b` is not a grouped design: it is `regression` if allowed, else an error. `y ~ x`, `x` **numeric** numeric-numeric (correlation, simple regression). `y ~ x1 + x2 + ...` general regression. `y ~ trt | block` n-sample dependent (blocked design).
`data`	an optional data frame containing the variables in `formula`. A matrix is coerced to a data frame.

`subset` an optional expression indicating the observations to use, evaluated in `data` as in `model.frame()` (`subset = len > 10`), or an index vector. See Details.

`na.action` a function specifying how missing values are handled, defaults to `na.pass()`.

`allowed` a character vector restricting which design types are accepted, any combination of `"one-sample"`, `"two-sample-independent"`, `"two-sample-dependent"`, `"n-sample-independent"`, `"n-sample-dependent"`, `"numeric-numeric"` and `"regression"`. The values are matched exactly, an unknown one is an error rather than being ignored. A further error is raised if the detected type is not among the allowed ones. Defaults to all types.

Details

Design types

`one-sample` `y ~ 1`, as in the one-sample t-test or the one-sample Wilcoxon test.

`two-sample-independent` `y ~ g` with two groups, as in the two-sample t-test or the Wilcoxon rank-sum test.

`two-sample-dependent` `Pair(x, y) ~ 1`, as in the paired t-test or the Wilcoxon signed-rank test.

`n-sample-independent` `y ~ g` with more than two groups, as in the analysis of variance or the Kruskal-Wallis test.

`n-sample-dependent` `y ~ trt | block`, as in a repeated-measures analysis of variance or the Friedman test.

`numeric-numeric` `y ~ x` with a numeric right-hand side, as in correlation or simple regression.

`regression` a general regression formula with one or more predictors.

Type detection

The type follows from the shape of the formula and from the class of the right-hand side variable, not from `allowed`. `allowed` only decides whether the detected type is accepted, with four exceptions worth knowing.

1. A grouping factor carrying a single level is reported as `one-sample` if that type is allowed.
2. A two-group design is reported as `n-sample-independent` if `"two-sample-independent"` is not among the allowed types. Together with the previous rule this lets a caller that treats every group count alike allow one type only.
3. `allowed = "regression"` on its own forces the `regression` type for every formula, including `y ~ 1` and `y ~ g`, with the exception of the blocked syntax `y ~ trt | block`, which is always `n-sample-dependent` and therefore an error then. This is the entry point for model-fitting callers, which interpret the right-hand side themselves. Duplicates in `allowed` are ignored, so `c("regression", "regression")` behaves the same.
4. Otherwise `regression` is reported only for more than one right-hand side variable. With `allowed` containing both `"regression"` and `"numeric-numeric"`, `y ~ x` is therefore `numeric-numeric` while `y ~ x1 + x2` is `regression`.

Cells of several grouping variables

A right-hand side consisting of a single interaction term, `y ~ a:b` (or `y ~ a:b:c`), is the explicit request for the cells of these variables. They are combined into one grouping factor via `interaction()`,

with levels such as "OJ:0.5"; numeric components are treated as categorical. The design is then classified like y ~ g, by the number of non-empty cells.

Unlike boxplot(), y ~ a + b is *not* read as cells: additive terms are not an interaction, as in lm(). Such a formula, like y ~ a * b, is regression if that type is allowed, and an error otherwise. If "regression" is allowed, it also takes precedence over the cell reading of y ~ a:b, which a model-fitting caller interprets as an interaction term.

The distinction rests on the terms of the formula: the model frame holds the variables a and b in both cases and cannot tell the two apart.

offset() is only accepted for the regression design and an error otherwise: an offset column is part of the model frame without being a term, and would pass for a grouping variable or a numeric predictor.

Field naming contract (binding across all types)

- response is present for every type and always holds the left-hand side of the formula. It is the one field a caller can rely on without branching on type.
- x is an alias of response for the types that are conventionally described in terms of a sample rather than a model (one-sample, two-sample-independent, n-sample-independent, numeric-numeric). The exception is two-sample-dependent: there response is the whole Pair() matrix, while x and y are its first and second column.
- group is reserved for a categorical, factor-coercible variable of length n (the full sample) that splits the response into groups. It is never pre-split and never used for a continuous variable. x and group have an identical shape for two-sample-independent and for n-sample-independent, so that a caller can use split(rx, rgroup) uniformly, without branching on the number of groups. For y ~ a:b, group holds the combined cell factor.
- predictor is used for a continuous, numeric right-hand side variable (numeric-numeric), never group.
- treatment is used for the explanatory variable of a blocked design (n-sample-dependent), as distinct from block, the stratification factor. Neither is ever called group.
- y, where present, is a convenience field only, holding the second group of a two-sample design or the second paired vector. It is never needed for correct use: x and group (or x and predictor, or treatment and block) are always sufficient and are the canonical access path.
- rows is present for every type and holds the positions of the retained observations in the original data, after subset and after na.action. It is the handle for synchronising an external vector (an ordering variable, weights) with the model frame: z <- z[r$rows]. It is NULL in the rare case where the row names of the model frame cannot be matched back, e.g. when a numeric subset selects a row twice.
- terms is returned for the regression type. Build the design matrix from it, model.matrix(r$terms, r$mf), never from the original formula: the columns of a model frame are named after the deparsed expressions ("log(x)"), so re-evaluating the formula against the model frame fails for every transformed term.

Missing values

Missing values are left to na.action and are not touched otherwise, so with the default na.pass() they reach the caller untouched. The one exception is the grouping factor of an independent design, where empty and missing levels are dropped before the groups are counted. A grouping variable

that is missing throughout leaves no level at all and is an error. A cell of `y ~ a:b` is missing as soon as one of its components is.

Rows removed by `na.action` are recorded in `attr(r$mf, "na.action")`, but those indices are relative to the already subsetted frame. To align an external vector with the model frame use `rows`, which accounts for `subset` and `na.action` at once.

subset handling

`subset` is evaluated as in base R: it is taken unevaluated, like `lm()` does, by handing this function's own call on to `model.frame()`, which evaluates the expression in `data`, with `environment(formula)` as the enclosure. `resolveFormula(y ~ g, df, subset = g == "A")` therefore works exactly like `boxplot(y ~ g, df, subset = g == "A")`, and a quoted expression fails in both.

A wrapper function forwards its arguments the same way base R does (see `boxplot.formula()`): it rebuilds its own call and evaluates it in its caller's frame, so that `subset` reaches `resolveFormula()` unevaluated. `resolveFormulaFromCall()` does exactly this and is the entry point to use in a formula method:

```
myFun <- function(formula, data, subset, na.action = na.omit, ...) {
  r <- resolveFormulaFromCall(
         allowed   = c("two-sample-independent", "n-sample-independent"),
         na.action = na.action)
  ...
}
```

Passing a captured expression on by value (`resolveFormula(formula, data, subset = substitute(subset))`) does not work, just as it does not for `model.frame()` itself.

Return components by type

Every return value starts with `type`, `mf`, `rows` and `response`, in that order, followed by the design-specific components below, and ends with `dataName`:

`one-sample` `x`

`two-sample-independent` `x`, `group`, `y` (convenience: the second group)

`two-sample-dependent` `x`, `y`

`n-sample-independent` `x`, `group`

`n-sample-dependent` `treatment`, `block`

`numeric-numeric` `x`, `predictor`

`regression` `terms`

resolveFormulaFromCall()

Rebuilds the call of the function it is called from - via `match.call()` against that function's definition, which works for an S3 method as well

- keeps its `formula`, `data` and `subset` arguments, and evaluates `resolveFormula()` with them in the frame the calling function was called from. This is the forwarding pattern of `boxplot.formula()` and `lm()`, written once:

```
plotBox.formula <- function(formula, data, subset, na.action = na.omit, ...) {
  r <- bedrock::resolveFormulaFromCall(
         allowed   = c("two-sample-independent", "n-sample-independent"),
         na.action = na.action)
  ...
}
```

`allowed` is passed on as given (omitted: all types). `na.action` should be the calling function's own `na.action` value; it is always passed on, so that the caller's default (typically `na.omit()`) applies rather than the `na.pass()` default of `resolveFormula()`.

Requirements for the calling function: its arguments must be named `formula`, `data` and `subset`, and `resolveFormulaFromCall()` must be called directly in its body, not from a nested helper function, since the call is looked up one frame up.

Value

a named list containing at least:

type character, one of the design types listed above.

mf the `model.frame()` the design was read from.

rows integer, the positions of the retained observations in the original data, or `NULL` if they cannot be determined.

response the left-hand side of the formula.

dataName character, the deparsed formula, for use as the `data.name` of an `htest` object.

plus the design-specific components described under Details.

See Also

`model.frame()`, `Pair()`, `resolveGroups()`

Other data.resolve: `resolveContingency()`, `resolveGroups()`

Examples

```
set.seed(1)
df <- data.frame(
  y   = rnorm(30, 50, 10),
  g2  = rep(c("A", "B"), 15),
  g3  = rep(c("A", "B", "C"), 10),
  trt = rep(c("T1", "T2", "T3"), 10),
  blk = rep(1:10, 3)
)

# one-sample
resolveFormula(y ~ 1, data = df)$type
## [1] "one-sample"

# two-sample independent: x and group have full length, the same shape
# as for more than two groups
```

```
r2 <- resolveFormula(y ~ g2, data = df,
                     allowed = c("two-sample-independent",
                                 "n-sample-independent"))
r2$type
## [1] "two-sample-independent"
length(r2$x) == length(r2$group)
## [1] TRUE

# n-sample independent
resolveFormula(y ~ g3, data = df,
               allowed = "n-sample-independent")$type
## [1] "n-sample-independent"

# cells of two grouping variables: a:b, not a + b
r3 <- resolveFormula(y ~ g2:g3, data = df,
                     allowed = "n-sample-independent")
levels(r3$group)
## [1] "A:A" "B:A" "A:B" "B:B" "A:C" "B:C"
try(resolveFormula(y ~ g2 + g3, data = df,
                   allowed = "n-sample-independent"))

# two-sample dependent (paired)
df2 <- data.frame(pre = rnorm(15, 50, 10), post = rnorm(15, 55, 10))
resolveFormula(Pair(pre, post) ~ 1, data = df2,
               allowed = c("one-sample",
                           "two-sample-dependent"))$type
## [1] "two-sample-dependent"

# n-sample dependent (blocked): treatment, not group
r4 <- resolveFormula(y ~ trt | blk, data = df,
                     allowed = "n-sample-dependent")
names(r4)
## [1] "type" "mf" "rows" "response" "treatment" "block" "dataName"

# numeric-numeric: predictor, not group
df3 <- data.frame(y = rnorm(20), x = rnorm(20))
r5 <- resolveFormula(y ~ x, data = df3, allowed = "numeric-numeric")
is.numeric(r5$predictor)
## [1] TRUE

# regression: build the design matrix from 'terms', not from the formula
r6 <- resolveFormula(y ~ log(abs(x)) + I(x^2), data = df3,
                     allowed = "regression")
colnames(model.matrix(r6$terms, r6$mf))

# subset, as in base R
resolveFormula(y ~ g3, data = df, subset = g3 != "C",
               allowed = "two-sample-independent")$type
## [1] "two-sample-independent"
```

`resolveGroups` *Resolve Grouped Data*

Description

Brings grouped data into one canonical shape, no matter which of the two usual interfaces the caller was given: a response vector together with a grouping variable, or a list holding one vector per group. The function validates the input, drops missing values, builds the grouping factor and returns the commonly needed group information, providing a shared entry point for hypothesis tests, summaries, effect-size calculations and plotting functions.

Usage

```
resolveGroups(x, groups)
```

Arguments

`x`	a numeric vector of observations, or a list of numeric vectors, one per group. A data frame is a list and is accepted as one, every column being taken as a group.
`groups`	a grouping variable, a vector of the same length as `x`, coerced to a factor. Ignored with a warning when `x` is a list.

Details

The two input forms are treated as equivalent. For a list, every element is taken as one group and `groups` is ignored with a warning. For a vector, `groups` is coerced to a factor after the incomplete observations have been removed, so that empty levels are dropped. The levels of an input that already is a factor keep their order, everything else is ordered as `factor()` produces it.

The names of a list become the group labels and their order becomes the order of the levels. As these labels end up in printed results, they must be complete and unique; a partially or ambiguously named list is an error rather than being silently renamed. A list without any names is labelled `"1"`, `"2"`, and so on.

A data frame is a list of columns and is resolved as one, which covers the common case of one group per column. Its column names are complete and unique by construction and are used as the group labels. Note that the groups of a data frame all have the same length before the missing values are removed, so a ragged design has to be padded with `NA` or passed as a plain list.

Missing values are removed in both cases, but along different rules. From a list every `NA` and `NaN` in a group is dropped, from a vector those observations are dropped where either the response or the grouping variable is missing. What remains must leave at least two groups, each of them non-empty.

The returned `dataName` is built from the unevaluated arguments and is meant to be passed on to the `data.name` element of an `htest` object. It reads as `"x and g"` for a vector with a grouping variable and as the deparsed expression itself for a list.

Value

a list containing:

x numeric vector of the observations, missing values removed. For a list input the groups follow each other in the order of the list.

groups factor of the same length as `x` holding the group membership.

n integer, the total number of observations.

k integer, the number of groups.

groupSizes named integer vector of the group sample sizes, in the order of the levels.

groupNames character vector of the group labels, the levels of `groups`.

dataName character description of the input, for use as the `data.name` of an `htest` object.

See Also

Other data.resolve: `resolveContingency()`, `resolveFormula()`

Examples

```
# vector + grouping variable
set.seed(1)
x <- rnorm(30)
g <- rep(c("a", "b", "c"), each = 10)
str(resolveGroups(x, g))

# list of group-specific vectors, the names become the labels
resolveGroups(list(a = rnorm(10), b = rnorm(12), c = rnorm(8)))[c("k", "groupSizes")]

# both interfaces lead to the same result
identical(resolveGroups(x, g)$groupSizes,
          resolveGroups(split(x, g))$groupSizes)

# a data frame is resolved column by column
resolveGroups(data.frame(ctrl = c(1, 2, 3), treat = c(4, 5, NA)))$groupSizes
```

---

revCode *Reverse Coding of Variables*

Description

Reverses the coding of a vector. Supports numeric, logical, and factor inputs:

- **Numeric**: Transforms values using `min + max - x`
- **Logical**: Flips TRUE/FALSE
- **Factor**: Reverses the order of levels

Usage

```
revCode(x, min = NULL, max = NULL, na.rm = FALSE)
```

Arguments

x	a vector (numeric, logical, or factor).
min	optional numeric minimum. Must be provided together with max. If NULL (default), the observed minimum of x is used.
max	optional numeric maximum. Must be provided together with min. If NULL (default), the observed maximum of x is used.
na.rm	logical; whether to ignore NAs when computing the range (numeric only). If FALSE and NAs are present, a warning is issued and NA is returned for all values. Default is FALSE.

Value

a vector of the same type and length as x, with reversed coding.

Errors

Throws an error if all values are NA, if only one of min/max is provided, if min > max, or if x is not numeric, logical, or factor. A warning is issued if values of x lie outside an explicitly provided [min, max] range.

See Also

Other data.recode: asBinary(), combLevels(), dummy(), mReplace(), nf(), recodeX(), stringsAsFactors()

Examples

```
# Numeric
revCode(c(1, 2, 3, 4, 5))

# Numeric with explicit range (e.g., Likert scale)
revCode(c(1, 2, 3, 4, 5), min = 1, max = 5)

# Numeric with NAs
revCode(c(1, 2, NA, 4, 5), na.rm = TRUE)

# Logical
revCode(c(TRUE, FALSE, TRUE))

# Factor
x <- factor(c("low", "medium", "high"), ordered = TRUE)
revCode(x)
```

revX *Reverse the Order of Elements*

Description

Returns a reversed version of its argument. Where `rev()` treats every object as one long vector, `revX()` reverses the order along the dimensions of a multidimensional object, so that a matrix, table, array or data frame comes back with its rows, its columns, or both, in the opposite order and its `dimnames` moved along with the data. Which dimensions are turned around is chosen with `margin`.

Usage

```
revX(x, ...)

## Default S3 method:
revX(x, margin = 1L, ...)

## S3 method for class 'array'
revX(x, margin = seq_along(dim(x)), ...)

## S3 method for class 'matrix'
revX(x, margin = seq_along(dim(x)), ...)

## S3 method for class 'table'
revX(x, margin = seq_along(dim(x)), ...)

## S3 method for class 'data.frame'
revX(x, margin = 1:2, ...)
```

Arguments

`x`	a vector, matrix, table, array or data frame to be reversed.
`...`	further arguments, passed on to the method dispatched on. This is how `margin` is handed over; arguments beyond it are ignored with a warning.
`margin`	the dimensions to reverse, `1` for the rows, `2` for the columns, and so on, each at most once. Defaults to all dimensions of `x`, which for a vector means `1`.

Details

A vector has one dimension and is simply reversed, as by `rev()`, with `margin = 1` accepted so that calling code need not know whether its argument has dimensions. For everything else, `margin` names the dimensions to be reversed, `1` for the rows, `2` for the columns, and so on for the higher dimensions of an array; the default reverses all of them. The values in the object are not rearranged relative to their labels: reversing an object twice along the same margin returns the original.

`margin` names each dimension at most once; repeating one says nothing and is refused, as is any value outside the dimensions of `x`.

The additional arguments of the generic are the way `margin` reaches the methods. Anything else is ignored with a warning, rather than being dropped silently although it was meant to change the result.

Value

an object of the same class and dimensions as `x`, with the order of the elements along `margin` reversed.

See Also

`rev()`, `order()`, `sort()`, `seq()`

Other data.order: `binaryTree()`, `sortX()`

Examples

```
tab <- matrix(c(1, 11, 111,
                2, 22, 222,
                3, 33, 333),
              byrow=TRUE, nrow=3,
              dimnames=list(mar1=1:3, mar2=c("a","b","c")))

revX(tab, margin=1)
revX(tab, margin=2)

# reverse both dimensions
revX(tab, margin=c(1, 2))

# the dimnames travel with the data, so this is not a transposition
revX(tab, margin=c(1, 2))["3", "a"] == tab["3", "a"]
## [1] TRUE

# reverse a 3-dimensional array
aa <- array(c(tab, 2 * tab), dim = c(3, 3, 2),
            dimnames = c(dimnames(tab), list(mar3 = c("A", "Z"))))

# reverse rows
revX(aa, 1)
# reverse columns
revX(aa, 2)
# reverse the third dimension
revX(aa, 3)

# reverse all dimensions
revX(aa)
# same as
revX(aa, margin = 1:3)

# data frames are reversed by rows, by columns or both
d <- data.frame(a = 1:3, b = 4:6)
revX(d, 1)
revX(d, 2)
```

`Roulette` *European Roulette Wheel*

Description

The numbers on a single-zero (European) Roulette wheel and their associated properties: colour, betting categories and the traditional sectors of the wheel. Each row represents one of the 37 numbers (0–36).

Usage

```
Roulette
```

Format

A data frame with 37 rows and 7 variables:

num integer, the number in the pocket (0–36).

col factor, colour of the pocket: `red`, `black` or `green`.

parity factor, `even` or `odd`.

highlow factor, `low` (1–18) or `high` (19–36).

dozens factor, dozen on the table: `1` (1–12), `2` (13–24), `3` (25–36).

column factor, column on the table: `1`, `2` or `3`, counted from the one containing 1.

pocketrange factor, sector of the wheel: `jeu zero`, `voisins du zero`, `tiers du cylindre` or `orphelins`.

Details

The dataset can be used for teaching probability and categorical data analysis, as well as for simulating Roulette betting strategies.

The rows are ordered as the pockets follow each other on the wheel, starting at zero, and not by `num`. The sectors in `pocketrange` are therefore contiguous blocks of rows.

Zero takes part in none of the even/odd, high/low, dozen and column bets, so these variables are `NA` for zero. Note that this follows the rules of the game and not arithmetic, where zero would count as even.

The seven numbers of the *jeu zero* are part of the *voisins du zero* in the usual reading of the terms. As every number appears exactly once here, they are reported as a separate level and `voisins du zero` covers the remaining ten numbers of that sector.

Accents are dropped in the factor levels (`voisins du zero` for *voisins du zéro*).

Source

The standard layout of a single-zero Roulette wheel.

See Also

Other datasets: Cards, Pizza, Tarot, readCourseData()

Examples

```
head(Roulette)

table(Roulette$col)
table(Roulette$parity, Roulette$highlow, useNA = "ifany")

# the sectors of the wheel are blocks of neighbouring pockets
table(Roulette$pocketrange)
```

---

roundTo *Round to a Multiple*

Description

Rounds the values of a numeric vector to the nearest multiple of a given step width. Where round() is tied to multiples of a power of ten, roundTo() accepts an arbitrary step, so that prices can be rounded to the nearest 5 cents, durations to the nearest quarter of an hour or axis limits to the nearest 250. The direction of the rounding is controlled by FUN, which allows rounding to the nearest, upwards, downwards or towards zero with the same interface.

Usage

```
roundTo(x, multiple = 1, FUN = round)
```

Arguments

x	numeric. The values to be rounded.
multiple	numeric. The step width to whose multiples the values are to be rounded, defaults to 1. Must be finite and positive and either a single value or as long as x.
FUN	the rounding function applied to x / multiple. Typically one of round() (default), trunc(), ceiling() or floor(). Other functions accepting and returning a numeric vector can be used as well.

Details

There are several functions in base R to convert to integers. round() rounds to the nearest integer or to any number of digits. Using a negative number of digits rounds to a power of ten, so that round(x, -3) rounds to thousands. Each of trunc(), floor() and ceiling() rounds in a fixed direction, towards zero, down and up respectively. round() is documented to round half to even, so round(2.5) is 2.

roundTo() evaluates FUN(x / multiple) * multiple. With the default FUN = round a value lying exactly halfway between two multiples is therefore rounded to the one with the even quotient: roundTo(1, 2) is 0 and roundTo(3, 2) is 4. Setting FUN = ceiling always rounds up, FUN = floor always rounds down and FUN = trunc always towards zero (see the examples for a comparison).

Ties are rare in practice, as most decimal fractions have no exact binary representation. 1.3 / 0.2 is marginally smaller than 6.5 in double precision, so roundTo(1.3, 0.2) returns 1.2 and not the 1.4 that the rule for ties would suggest. Results for a fractional multiple are likewise only accurate to within representation error, which is why roundTo(x, 0.05) may still print more than two decimal places.

A single multiple is used for all the values in x. A vector of step widths is applied elementwise and must then be exactly as long as x, so that a length mismatch is reported as an error instead of being recycled silently.

Value

a numeric vector of the rounded values, as long as x. NAs in x are returned as NA.

See Also

round(), trunc(), ceiling(), floor()

Other math.basic: closest(), crossProd(), crossProdN(), dotProd(), unirootAll()

Examples

```
roundTo(10, 3)     # rounds 10 to the nearest multiple of 3 (9)
roundTo(-10, 3)    # rounds -10 to the nearest multiple of 3 (-9)

roundTo(1.3, 0.2)  # rounds 1.3 to the nearest multiple of 0.2 (1.2)
roundTo(-1.3, 0.2) # rounds -1.3 to the nearest multiple of 0.2 (-1.2)

# prices to the nearest 5 cents
roundTo(c(1.02, 1.03, 12.375), 0.05)

# a step width for every value
roundTo(c(1.23, 123, 1234), c(0.05, 10, 100))

# any other length is an error, the values are not recycled
try(roundTo(1:6, c(2, 3)))

# round down
roundTo(c(1, -1) * 1.2335, 0.05, floor)
roundTo(c(1, -1) * 1233.5, 100, floor)

# round up
roundTo(c(1, -1) * 1.2335, 0.05, ceiling)
roundTo(c(1, -1) * 1233.5, 100, ceiling)

# round towards zero
roundTo(c(1, -1) * 1.2335, 0.05, trunc)
roundTo(c(1, -1) * 1233.5, 100, trunc)
```

```
# the four directions side by side
x <- c(-1.5, -1.3, 1.3, 1.5)
cbind(x       = x,
      round   = roundTo(x, 0.2, FUN = round),
      trunc   = roundTo(x, 0.2, FUN = trunc),
      ceiling = roundTo(x, 0.2, FUN = ceiling),
      floor   = roundTo(x, 0.2, FUN = floor)
)

# note how the ties in the first column are resolved to even multiples
x <- -5:5
cbind(x       = x,
      round   = roundTo(x, 2, FUN = round),
      trunc   = roundTo(x, 2, FUN = trunc),
      ceiling = roundTo(x, 2, FUN = ceiling),
      floor   = roundTo(x, 2, FUN = floor)
)
```

`rSum21` *Random Numbers Summing to 1*

Description

Generates a vector of random proportions that sum exactly to 1.

Usage

```
rSum21(size, digits = NULL)
```

Arguments

`size`	integer. The number of values to generate.
`digits`	integer. If not `NULL` (default), the values are rounded to this number of decimal places while preserving the sum of 1.

Details

The values are drawn from a uniform distribution and normalized by their sum. If `digits` is given, the values are rounded and the rounding error is assigned to the largest element, which is then rounded again to the requested precision. Note that for very coarse rounding the exact-sum guarantee may not be attainable at the given precision.

Value

a numeric vector of length `size` summing to 1.

See Also

Other random.numbers: rBetaShape()

Examples

```
x <- rSum21(5)
sum(x)

x <- rSum21(5, digits = 2)
sum(x)
```

---

sampleX *Random Samples and Permutations*

Description

sampleX takes a sample of the specified size from the elements of x, with or without replacement. It does the same as sample() and additionally offers an interface for data frames, where rows are sampled.

Usage

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

## S3 method for class 'data.frame'
sampleX(x, size = nrow(x), replace = FALSE, prob = NULL)

## Default S3 method:
sampleX(x, size, replace = FALSE, prob = NULL)
```

Arguments

x	either a vector of one or more elements from which to choose, or a positive integer, or a data frame whose rows are to be sampled.
size	a non-negative integer giving the number of items (or rows) to choose. If missing, it defaults to the number of elements of x (resp. nrow(x) for data frames), yielding a random permutation.
replace	logical; whether sampling is performed with replacement.
prob	a vector of probability weights for obtaining the elements (or rows) being sampled.

Value

sampled elements in the same structure as x; for data frames, a data frame containing the sampled rows.

See Also

sample()

Other combinatorics: combN(), combPairs(), combSet(), pairApply(), permn(), randGroupSplit()

Examples

```
sampleX(1:10, size = 5)

# random permutation, like sample(x)
sampleX(1:10)

# sample rows of a data frame
sampleX(mtcars, size = 5)
```

---

setAttr-removeAttr-keepAttr

Set and Remove Object Attributes

Description

Convenience helpers to add, remove, or selectively retain attributes of an object.

Usage

```
setAttr(x, attrNames, attrValues)

removeAttr(x, attrNames = NULL)

keepAttr(x, attrNames)
```

Arguments

x	object to modify.
attrNames	character vector of attribute names.
attrValues	values for the attributes (only for setting). For a single attribute name, attrValues is taken as the value itself (which may be a vector). For several names, supply one value per name; use a list for non-scalar or mixed-type values.

Value

modified object.

See Also

setNames(), unname()

Other label.attrs: label(), renameX(), setNamesX()

Examples

```
x <- runif(10)

x <- setAttr(
  x,
  attrNames = c("some_attr", "other_attr"),
  attrValues = c("First attribute", "Second attribute")
)

# a single attribute can take a vector value
setAttr(1:10, "dim", c(2, 5))

# several non-scalar values via list
setAttr(1:10, c("dim", "myattr"), list(c(2, 5), "abc"))

# remove single attribute
removeAttr(x, "other_attr")

# remove all attributes
removeAttr(x)

# keep only selected attributes, remove all others
r.lm <- lm(Fertility ~ ., swiss)
keepAttr(r.lm$terms, "class")
```

---

setLength *Set the length of a vector, padding or truncating as needed*

Description

Extends `x` to length `n` by appending `fill`, or truncates it to length `n` if it is already longer – like `length(x) <- n`, but with a configurable fill value instead of `NA`.

Usage

```
setLength(x, n, fill = NA)
```

Arguments

`x`	a vector.
`n`	target length, a single non-negative whole number.
`fill`	value used for newly added elements when `x` is extended (default `NA`).

Value

`x`, of length `n`.

See Also

Other vector.reshape: `trim()`, `vRot()`, `vShift()`

Examples

```
setLength(LETTERS[1:3], 5)
setLength(LETTERS[1:3], 2)
setLength(1:4, 6, fill = 0)
```

`setNamesX` *Set the Names in an Object*

Description

This is a convenience function that sets the names of an object and returns it including the new names. It is most useful at the end of a function definition where one is creating the object to be returned and would prefer not to store it under a name just that the names can be assigned. In addition to the function `setNames()` in base R the user can decide, whether rownames, colnames or simply the names are to be set.

Usage

```
setNamesX(x, ...)
```

Arguments

`x`	an object for which a names attribute will be meaningful.
`...`	the names to be assigned to the object. This should be a character vector of names named `dimnames`, `rownames`, `colnames` or `names`. Setting `rownames=NULL` would remove existing rownames. All kind of names can be changed at the same time. Default would be `names`. Abbreviations are supported.

Details

A name of length one is recycled to the required extent, which is handy for blanking out names with `""`. Names of any other length must match the extent of the object exactly; a deviating length is reported as an error rather than being recycled silently, as an unexpected length is almost always a miscalculation upstream and duplicated names are hard to debug later on.

Value

an object of the same sort as object with the new names assigned.

See Also

`setNames()`

Other label.attrs: `label()`, `renameX()`, `setAttr-removeAttr-keepAttr`

Examples

```
setNamesX(1:5, names=letters[1:5])

# the default, if no argument names are provided, is "names"
setNamesX(1:5, letters[1:5])

# rownames and columnnames can be set at the same time
setNamesX(matrix(c(1:12), nrow=4),
         rownames=LETTERS[11:14], colnames=c("perc", "lci", "uci"))

# a single name is recycled, so this sets all the names to an empty string
setNamesX(diag(6), rownames="", colnames="")

# any other length must fit, a mismatch is an error
try(setNamesX(matrix(c(1:12), nrow=4), colnames=c("perc", "lci")))

# setting dimnames works as well
tab <- setNamesX(
  as.table(rbind(c(84,43), c(10,92))),
    dimnames= list(
       dipstick=c("positive","negative"),
       culture=c("positive","negative")))
```

`sortX` *Sort Vectors, Matrices, Tables, and Data Frames*

Description

`sortX` extends the base `sort()` function by providing a consistent interface for sorting not only vectors, but also matrices, tables, and data frames. For 2-dimensional objects, rows are sorted based on one or more columns.

Usage

```
sortX(x, ...)

## Default S3 method:
sortX(
  x,
  decreasing = FALSE,
  na.last = NA,
  method = c("default", "mixed"),
  factorsAsCharacter = TRUE,
  ...
)
```

```
## S3 method for class 'table'
sortX(
  x,
  by = NULL,
  decreasing = FALSE,
  na.last = TRUE,
  method = c("default", "mixed"),
  factorsAsCharacter = TRUE,
  ...
)

## S3 method for class 'matrix'
sortX(
  x,
  by = NULL,
  decreasing = FALSE,
  na.last = TRUE,
  method = c("default", "mixed"),
  factorsAsCharacter = TRUE,
  ...
)

## S3 method for class 'data.frame'
sortX(
  x,
  by = NULL,
  decreasing = FALSE,
  na.last = TRUE,
  method = c("default", "mixed"),
  factorsAsCharacter = TRUE,
  ...
)
```

Arguments

x	a numeric, complex, character or logical vector, factor, matrix, table, or data frame to be sorted.
...	further arguments passed to sort() in sortX.default.
decreasing	logical scalar or vector. Should the sort be in decreasing order? For 2-dimensional objects a vector of the same length as by may be supplied to control the direction per column; a scalar is recycled.
na.last	logical or NA. Should missing values be placed last (TRUE), first (FALSE), or removed (NA)? See order().
method	sorting method. Either "default" (base R behavior) or "mixed" for natural sorting of character data (e.g. "A2" < "A10").
factorsAsCharacter	logical. If TRUE (default), factors are converted to character before sorting so

that labels are used instead of level codes. Set to FALSE to sort by level order (useful for ordered factors).

by integer or character vector specifying the columns to sort by, and their priority (first element = primary key). Column names and positive integer indices (1:ncol(x)) refer to columns. The special value 0L (integer zero, always numeric) sorts by row names. For table and matrix objects, ncol(x) + 1L sorts by row marginal sums. This argument is not available for sortX.default. Default: NULL (all columns, left to right).

Details

By default, sorting follows the behavior of base R. In addition, method = "mixed" enables natural ("human-friendly") sorting of character data, e.g. "A2" < "A10".

For method = "mixed", sorting is applied column-wise using .orderMixed(). Each column's tokens are ordered independently (numeric runs numerically, text runs lexicographically) before the results are combined via stable right-to-left ordering.

The sort order for factors depends on factorsAsCharacter: if TRUE (default), factors are sorted by their labels (alphabetically or by natural order when method = "mixed"); if FALSE, they are sorted by their level order, which is appropriate for ordered factors but may be unintuitive for unordered ones.

Value

the sorted object, of the same class as x.

See Also

sort(), order()

Other data.order: binaryTree(), revX()

Examples

```
set.seed(3)
d.frm <- iris[sample(nrow(iris), 10),
              c("Species", "Sepal.Length", "Sepal.Width")]

# Vector sorting
sortX(d.frm[, 1])

# Data frame: sort by column name
sortX(d.frm, by = "Species")
sortX(d.frm, by = c("Species", "Sepal.Length"))

# Data frame: sort by column index
sortX(d.frm, by = c(1L, 2L))

# Decreasing order (per-column control)
sortX(d.frm, by = c("Species", "Sepal.Length"),
      decreasing = c(FALSE, TRUE))
```

```
# Natural sorting of character vectors
x <- c("A1", "A10", "A2")
sortX(x, method = "mixed")

# Factor: sort by label (default) vs. level order
sortX(d.frm, by = "Species")                          # by label
sortX(d.frm, by = "Species", factorsAsCharacter = FALSE)  # by level

# Tables: sort by column 2 descending
tab <- HairEyeColor[, , 1]
sortX(tab, by = 2L, decreasing = TRUE)

# Tables: sort by marginal row sums
sortX(tab, by = ncol(tab) + 1L, decreasing = TRUE)

# Sort by row names (always pass 0 as integer)
sortX(tab, by = 0L)
```

`splitAt` *Split a Vector at Given Positions*

Description

Splits a vector into consecutive segments at specified positions.

Usage

```
splitAt(x, pos)
```

Arguments

`x`	a vector to be split.
`pos`	an integer vector of positions at which to split `x`. Positions refer to indices in `x` where a new segment should start.

Details

The function splits `x` into consecutive chunks defined by `pos`. Internally, positions are sorted, duplicates are removed, and values that would not produce a non-empty segment (`pos < 2` or `pos > length(x)`) are ignored. Empty segments are never returned.

Each element of the returned list corresponds to a contiguous subset of `x`. The first segment always starts at position 1.

Value

a list of vectors, each representing a segment of `x`.

See Also

split()

Other data.reshape: collapseTable(), long-wide-reshape, splitX(), untable()

Examples

```
x <- 1:10

# split at positions 4 and 7
splitAt(x, c(4, 7))

# unsorted and duplicate positions are handled
splitAt(x, c(7, 4, 4, 20))
```

---

splitPath *Split a File Path into Its Components*

Description

Splits a file path into its components such as directory, file name, and extension. The function is OS-aware and works on both Windows and Unix-like systems.

Usage

```
splitPath(path, lastIsFile = NULL)
```

Arguments

path	a character vector of file paths.
lastIsFile	logical; if TRUE, the last component of path is treated as a file name. If FALSE, it is treated as part of the directory path. If NULL (default), the function determines this automatically based on whether the path ends with a path separator.

Details

The function uses basename() and dirname() for platform-independent path handling. File name and extension are extracted using tools::file_path_sans_ext() and tools::file_ext().

If lastIsFile = FALSE, the path is treated as a directory and file-related components (fullfilename, filename, extension) are returned as NA.

Value

a list with the following components (each a vector of the same length as `path`):

normpath normalized path as returned by `normalizePath()`.

drive drive letter on Windows systems (e.g., `"C:"`), otherwise `NA`.

dirname directory path without drive letter, including trailing separator.

fullfilename full file name including extension (if applicable).

fullpath full directory path including drive letter and trailing separator.

filename file name without extension.

extension file extension without leading dot.

See Also

`basename()`, `dirname()`, `tools::file_ext()`, `tools::file_path_sans_ext()`

Other file.path: `buildPath()`, `findDownload()`, `isFilePath()`, `isUrl()`, `urlExists()`

Examples

```
splitPath("C:/temp/file.txt")

splitPath("/home/user/data.csv")

# treat as directory
splitPath("/home/user/folder/", lastIsFile = FALSE)
```

---

splitX *Split Data into Groups (Extended Interface)*

Description

Splits a vector or object into groups defined by a factor or grouping variables. This is a wrapper around `split()` with an additional formula interface.

Usage

```
splitX(x, ...)

## Default S3 method:
splitX(x, f, drop = FALSE, ...)

## S3 method for class 'formula'
splitX(formula, data, subset, na.action, drop = FALSE, ...)
```

Arguments

x	object to be split (typically a vector).
...	further arguments passed to split().
f	a factor or list of factors defining the groups (default method).
drop	logical; if TRUE, unused factor levels are dropped.
formula	a formula of the form y ~ group or y ~ g1 + g2 specifying the variable to split (y) and the grouping variables.
data	a data frame containing the variables in the formula.
subset	optional logical expression indicating rows to include.
na.action	a function specifying how missing values are handled, passed to model.frame() (e.g., na.omit()).

Details

splitX extends split() by providing:

- an S3 interface
- a formula method for convenient specification of variables
- support for multiple grouping variables via formula

The formula interface evaluates a model.frame() and splits the response variable by one or more grouping variables.

If multiple grouping variables are provided, the data are split by their interaction (similar to split(..., interaction(...))).

Value

a list of subsets of x, grouped according to f or the grouping variables in the formula.

See Also

Other data.reshape: collapseTable(), long-wide-reshape, splitAt(), untable()

Examples

```
# Default usage
x <- 1:10
g <- rep(letters[1:2], each = 5)
splitX(x, g)

# Formula interface
df <- data.frame(
  y = rnorm(10),
  g1 = rep(letters[1:2], each = 5),
  g2 = rep(1:2, times = 5)
)
```

```
splitX(y ~ g1, data = df)
splitX(y ~ g1 + g2, data = df)
```

`stringsAsFactors` *Convert Character Columns to Factors*

Description

Strings as factors have recently been downgraded in base R's `data.frame()` function. However, it is still usually a good idea to encode string variables as factors. This function helps to convert some or all columns of a data.frame to factors.

Usage

```
stringsAsFactors(x, columns = NULL)
```

Arguments

`x`	the data.frame.
`columns`	names or indexes of the columns to be converted; negative values can be used to omit columns.

Value

the given data.frame including the converted factors.

See Also

Other data.recode: `asBinary()`, `combLevels()`, `dummy()`, `mReplace()`, `nf()`, `recodeX()`, `revCode()`

Examples

```
d.dat <- data.frame(char_x = LETTERS[1:5],
                    char_y = LETTERS[6:10],
                    n = 1:5)

# all character columns
str(stringsAsFactors(d.dat))
# only char_y
str(stringsAsFactors(d.dat, columns = "char_y"))
# only char_x
str(stringsAsFactors(d.dat, columns = "char_x"))
# all character columns except the second one ("char_y")
str(stringsAsFactors(d.dat, columns = -2))
```

strSplitToCol *Split Strings into Multiple Columns*

Description

Splits character vectors into multiple columns based on a delimiter. Each element of x is split using strsplit(), and the resulting parts are expanded into separate columns.

Usage

```
strSplitToCol(x, split = " ", fixed = TRUE, naForm = "", colNames = NULL)
```

Arguments

x	a character vector or a data frame of character columns to be split. Each element (or column) is processed separately.
split	character string specifying the delimiter for splitting. Passed to strsplit().
fixed	logical; if TRUE, split is used as a fixed string. Otherwise, it is treated as a regular expression.
naForm	character value used to replace missing elements created by unequal split lengths.
colNames	optional character vector specifying column names for the resulting data frame. Recycled if necessary.

Details

All rows are padded to the same number of columns per input element. Missing values are filled with naForm.

For each element (or column) in x, the function:

1. Splits each entry using strsplit()
2. Determines the maximum number of split parts
3. Pads shorter splits with naForm
4. Combines results into a matrix via rbind()

The final result is a data frame where each original element or column contributes one or more columns depending on the number of splits.

An attribute "cols" is attached, indicating the number of columns generated for each element of x.

Value

a data frame containing the split components of x. Additional attribute:

- cols: integer vector with number of columns per input element.

See Also

Other string.transform: char-ascii-conversion, mGsub(), strSplitToDummy()

Examples

```
x <- c("A B C", "D E", "F")
strSplitToCol(x)

# Custom delimiter
x <- c("A|B|C", "D|E", "F")
strSplitToCol(x, split = "|")

# Multiple columns
df <- data.frame(
  a = c("x y", "z"),
  b = c("1 2 3", "4 5"),
  stringsAsFactors = FALSE
)
strSplitToCol(df)
```

`strSplitToDummy` *Split a Character Vector into a Dummy Matrix*

Description

Splits a character vector of delimited tokens into a binary dummy data.frame where each unique token becomes a column.

Usage

```
strSplitToDummy(x, split = ",", trim = TRUE, na.action = na.pass, ...)
```

Arguments

`x`	a character vector with delimited tokens.
`split`	a character string to use as delimiter. Default is `","`.
`trim`	logical. If `TRUE` (default), whitespace is trimmed from each token after splitting.
`na.action`	a function to handle `NA` values. Accepted values are `na.pass()` (default), `na.omit()`, `na.exclude()`, and `na.fail()`.
	`na.pass` NAs are kept as all-zero rows (default).
	`na.omit` rows with NAs are silently removed.
	`na.exclude` like `na.omit` but the indices of removed rows are stored in a `"na.action"` attribute.
	`na.fail` an error is raised if any `NA` is present.
`...`	additional arguments passed to `strsplit()`.

Value

a `data.frame` with one row per element of `x` and one column per unique token. Values are `0L` or `1L`. Column names are the token values as-is and may not be syntactically valid R identifiers. The attribute `"tokens"` contains the sorted vector of unique tokens.

See Also

strsplit(), na.omit()

Other string.transform: char-ascii-conversion, mGsub(), strSplitToCol()

Examples

```
dat <- data.frame(id = 1:5,
                  txt = c("A,C,D", "A", "B,C", "D", "D,E"))

# default: NA passed through as zero row
strSplitToDummy(dat$txt)

# with an NA in the input
x_na <- c("A,B", "B,C", NA, "A")

# na.pass: NA becomes an all-zero row (default)
strSplitToDummy(x_na, na.action = na.pass)

# na.omit: NA rows are silently dropped
strSplitToDummy(x_na, na.action = na.omit)

# na.exclude: like na.omit but NA indices stored in attribute
res <- strSplitToDummy(x_na, na.action = na.exclude)
attr(res, "na.action")

# na.fail: error if any NA present
tryCatch(
  strSplitToDummy(x_na, na.action = na.fail),
  error = function(e) conditionMessage(e)
)
```

---

strX *Extended str() with numbered variables*

Description

Wrapper around str() that optionally numbers variables in lists and data frames. Useful for large objects where variables should be referenced by position.

Usage

```
strX(object, ..., enumerate = TRUE, recursive = FALSE, strict.width = "cut")
```

Arguments

object	any R object.
...	additional arguments passed to str().

`enumerate`	logical; whether variables or elements are numbered. Default is `TRUE`.
`recursive`	logical; whether nested list elements are also numbered. Default is `FALSE`.
`strict.width`	character string passed to `str()`. Default is `"cut"`.

Details

By default, only top-level elements are numbered. Recursive numbering of nested list elements can be enabled with `recursive = TRUE`; nested elements then get hierarchical labels such as `2.1`, which give their position (`x[[2]][[1]]`), and the top-level numbers stay the same in both views.

Value

invisibly returns the character vector produced by `str()`.

See Also

`str()`

Examples

```
# Data frame
strX(mtcars)

# Nested list
x <- list(
  a = 1,
  b = list(
    c = 2,
    d = 3
  )
)

strX(x)

# Recursive numbering
strX(x, recursive = TRUE)
```

Tarot *Tarot Cards dataset*

Description

A dataset representing a standard Tarot deck, including both Major and Minor Arcana Cards. Each row corresponds to a single card with attributes describing its type, suit and rank.

Usage

```
Tarot
```

Format

A data frame with 78 observations and 6 variables:

card Name of the Tarot card.

rank Raw rank of the card as character.

suit Suit of the card (wand, coin, cup, sword, or trump).

arcana Type of arcana (minor or major).

rank_minor Ordered factor indicating the rank within the minor arcana (NA for major arcana).

rank_major Integer indicating the rank within the major arcana (0–21, NA for minor arcana).

Details

The dataset is designed for teaching, simulation and modelling purposes, illustrating how heterogeneous ordinal structures can be represented in a statistically consistent way.

The Tarot deck consists of 56 Minor Arcana Cards and 22 Major Arcana Cards. Since both groups follow different ranking systems, two separate variables are provided: `rank_minor` for the ordered structure within the minor arcana and `rank_major` for the numeric ordering of the major arcana.

This separation avoids mixing incompatible ordinal scales and makes the dataset suitable for statistical modelling and machine learning applications.

Source

Simulated data.

See Also

Other datasets: `Cards`, `Pizza`, `Roulette`, `readCourseData()`

Examples

```
head(Tarot)
table(Tarot$arcana)
summary(Tarot$rank_minor)
```

---

toBaseR *Back to Basics with Tibbles and data.tables*

Description

Sometimes we might wish for the old days be back and want to work with familiar objects. This function helps to convert `tibbles` and `data.tables` to plain `data.frames` as smoothly as possible.

Usage

```
toBaseR(x, ...)

## S3 method for class 'tbl_df'
toBaseR(x, ...)

## S3 method for class 'data.table'
toBaseR(x, ...)

## S3 method for class 'haven_labelled'
toBaseR(x, ...)

## Default S3 method:
toBaseR(x, ...)
```

Arguments

x	the object to be converted.
...	arguments passed on.

Details

For a `data.table` the key (`"sorted"`), secondary indices (`"index"`) and the `".internal.selfref"` pointer are dropped, and a deep copy is returned, so that later modifications by reference on the original cannot leak into the result. Columns of class `haven_labelled` are converted to factors in both cases.

Value

converted object.

See Also

Other data.coerce: `as.array.xtabs()`, `type-aliases`

Examples

```
# a tibble is rolled back to a plain data.frame
if (requireNamespace("tibble", quietly = TRUE)) {
  tbl <- tibble::as_tibble(head(iris))
  class(toBaseR(tbl))
}

# same for a keyed data.table
dt <- data.table::as.data.table(head(iris))
data.table::setkey(dt, Species)
df <- toBaseR(dt)
class(df)
attr(df, "sorted")
```

```
# an object without a method is returned unchanged, with a warning
x <- suppressWarnings(toBaseR(1:3))
identical(x, 1:3)


# labelled data from other statistical packages: needs 'haven' and
# an internet connection, hence the try()
if (requireNamespace("haven", quietly = TRUE)) {
  url <- "http://www.stata.com/videos13/data/webclass.dta"
  d.webclass <- try(toBaseR(haven::read_dta(url)))
}
```

`trim` *Trim a Vector*

Description

Clean data by means of trimming, i.e., by omitting outlying observations.

Usage

```
trim(x, trim = 0.1, na.rm = FALSE)
```

Arguments

`x`	a numeric vector to be trimmed.
`trim`	the fraction (0 to 0.5) of observations to be trimmed from each end of `x`. Values of trim outside that range (and < 1) are taken as the nearest endpoint. If `trim` is set to a value >1 it's interpreted as the number of elements to be cut off at each tail of `x`.
`na.rm`	a logical value indicating whether `NA` values should be stripped before the computation proceeds.

Details

A symmetrically trimmed vector `x` with a fraction of trim observations (resp. the given number) deleted from each end will be returned. If `trim` is set to a value ≥ 0.5 or to an integer value $\geq n/2$ then the result will be `NA`. The same applies if `x` contains `NA`s and `na.rm` is `FALSE`.

Value

the trimmed vector `x`. The indices of the trimmed values will be attached as attribute named `"trim"`.

Note

This function is basically an excerpt from the base function `mean()`, which allows the vector `x` to be trimmed before calculating the mean.

See Also

Other vector.reshape: `setLength()`, `vRot()`, `vShift()`

Examples

```
## generate data
set.seed(1234)     # for reproducibility
x <- rnorm(10)     # standard normal
x[1] <- x[1] * 10  # introduce outlier

## Trim data
x
trim(x, trim=0.1)

## Trim fixed number, say cut the 3 extreme elements from each end
trim(x, trim=3)

## check function
s <- sample(10:20)
s.tr <- trim(s, trim = 2)
setequal(c(s[attr(s.tr, "trim")], s.tr), s)
```

---

type-aliases *Type Coercion Shortcuts*

Description

Concise aliases for common base R coercion functions. `num()`, `int()`, `chr()` are direct wrappers around `as.numeric()`, `as.integer()`, and `as.character()`. `nchr()` handles the common pitfall of coercing factors to numeric. `bin()` converts any two-valued vector to logical.

Usage

```
num(x, ...)

int(x, ...)

chr(x, ...)

nchr(x)

bin(x, ...)
```

Arguments

`x`	a vector. For `bin()`, exactly two unique non-`NA` values are required.
`...`	further arguments passed to the underlying base function (`as.numeric`, `as.integer`, `as.character`, or `asBinary()`).

Details

`num(x, ...)` equivalent to `as.numeric(x)`.

`int(x, ...)` equivalent to `as.integer(x)`.

`chr(x, ...)` equivalent to `as.character(x)`.

`nchr(x)` shortcut for `as.numeric(as.character(x))`. Avoids the trap of `as.numeric(factor)` returning internal integer codes instead of the label values.

`bin(x, ...)` converts a two-valued vector (character, factor, integer, or numeric) to logical. Mapping follows `factor()` level order: the *first* level becomes `FALSE`, the *second* `TRUE`. To reverse, use `!bin(x)`.

Value

a vector of the target type and the same length as `x`.

See Also

`nf()`, `asBinary()`

Other data.coerce: `as.array.xtabs()`, `toBaseR()`

Examples

```
num("3.14")
int(3.9)                              # truncates, does not round
chr(1:3)
nchr(factor(c("1.5", "2.0", "1.5"))) # correct: 1.5 2.0 1.5
as.numeric(factor(c("1.5", "2.0")))  # wrong:   1   2

bin(c(0L, 1L, 0L, 1L))
bin(c("no", "yes", "no"))            # "no" -> FALSE, "yes" -> TRUE
!bin(c("no", "yes", "no"))           # reversed
bin(factor(c("m", "w", "m")))        # "m" -> FALSE, "w" -> TRUE
```

---

unirootAll *Find multiple roots of a function within an interval*

Description

Searches a numeric interval for all roots (zeros) of a function `f` by subdividing it into `n` sub-intervals, detecting sign changes, and refining each candidate with `uniroot()`.

Usage

```
unirootAll(
  f,
  interval,
  lower = min(interval),
  upper = max(interval),
  tol = .Machine$double.eps^0.5,
  maxiter = 1000,
  n = 100,
  ...
)
```

Arguments

f	a function for which roots are sought. Must accept a numeric first argument; additional arguments are passed via ....
interval	a numeric vector of length 2 specifying the search interval. Either interval or both lower and upper must be supplied.
lower	lower bound of the search interval. Default: min(interval).
upper	upper bound of the search interval. Default: max(interval).
tol	convergence tolerance passed to uniroot(), and also used as the threshold for (i) treating grid-point values as exact zeros and (ii) collapsing near-duplicate roots. Default: .Machine$double.eps^0.5.
maxiter	maximum number of iterations for uniroot(). Default: 1000.
n	number of sub-intervals used for the initial grid search. Increase n if roots may be close together or the function oscillates rapidly. Default: 100.
...	additional arguments passed to f.

Details

The function f is called as f(x, ...) where x is a numeric vector. If f does not accept vector input, it is called element-wise via sapply.

Grid points at which |f(x)| < tol are returned directly as roots. Sign changes are detected using sign(), which avoids numerical overflow that can occur with product-based approaches. Non-finite function values are silently ignored when detecting sign changes. If uniroot fails on a sub-interval, that interval is skipped with a warning rather than aborting the entire search.

Limitations: Roots within the same sub-interval of width (upper - lower) / n may be missed. Roots of even multiplicity that do not produce a sign change will not be found unless they happen to fall on a grid point. A warning is issued if no roots are found at all despite finite function values being present.

Value

a numeric vector of roots found in [lower, upper], sorted in ascending order. Returns numeric(0) if no roots are found.

See Also

stats::uniroot() for the underlying single-root solver.

Other math.basic: closest(), crossProd(), crossProdN(), dotProd(), roundTo()

Examples

```
f <- function(x) cos(2 * x)^3
roots <- unirootAll(f, c(0, 10))
stopifnot(all(abs(f(roots)) < 1e-6))

# Non-vectorized function
g <- Vectorize(function(x) integrate(function(t) t^x, 0, 1)$value - 0.5)
unirootAll(g, c(0.1, 5))
```

---

untable *Recover Original Data From Contingency Table*

Description

Recreates the data.frame out of a contingency table x.

Usage

```
untable(x, ...)

## S3 method for class 'data.frame'
untable(x, freq = "Freq", rowNames = NULL, ...)

## Default S3 method:
untable(x, dimnames = NULL, type = NULL, rowNames = NULL, colNames = NULL, ...)
```

Arguments

x	a numeric vector, a matrix, a table or a data.frame. If x is a vector, a matrix or a table it is interpreted as frequencies which are to be inflated to the original list. If x is a data.frame it is interpreted as a table in frequency form (containing one or more factors and a frequency variable).
...	further arguments passed to or from functions (not used here).
freq	character, the name of the frequency variable in case x is a data.frame.
rowNames	a names vector for the rownames of the resulting data.frame If set to NULL (default) the names will be defined according to the table's dimnames.
dimnames	the dimension names of x to be used for expanding. Can be used to expand a weight vector to its original values. If set to NULL (default) the dimnames of x will be used.

type	defines the data type generated. This allows to directly define factors or ordered factors, but also numeric values. See examples.
colNames	a names vector for the colnames of the resulting data.frame If set to NULL (default) the names will be defined according to the table's dimnames.

Details

For x being a vector this reduces to `rep(..., n)` with n as vector (which is not supported by `rep()`). NAs in the table will be treated as 0 without raising an error.

Value

a data.frame with the detailed data (even if x was a 1-dimensional table).

See Also

`expand.grid()`, `rep()`, `gl()`, `xtabs()`

Other data.reshape: `collapseTable()`, `long-wide-reshape`, `splitAt()`, `splitX()`

Examples

```
d.titanic <- untable(Titanic)
str(d.titanic)

# ... not the same as:
data.frame(Titanic)


tab <- table(set1=sample(letters[1:5], size=40, replace=TRUE),
             set2=sample(letters[11:15], size=40, replace=TRUE))
untable(tab)


# return a numeric vector by setting type and coerce to a vector by [,]
untable(c(6,2,2), type="as.numeric")[,]


# how to produce the original list based on frequencies, given as a data.frame
d.freq <- data.frame(xtabs(Freq ~ Sex + Survived, data=Titanic))

# a data list with each individual
d.data <- untable( xtabs(c(1364, 126, 367, 344) ~ .,
             expand.grid(levels(d.freq$Sex),levels(d.freq$Survived))))
head(d.data)

# expand a weights vector
untable(c(1,4,5), dimnames=list(c("Zurich","Berlin","London")))

# and the same with a numeric vector
untable(c(1,4,5), dimnames=list(c(5,10,15)), type="as.numeric")[,]
# ... which again is nothing else than
```

```
rep(times=c(1,4,5), x=c(5,10,15))

# the data.frame interface
d.freq <- data.frame(f1=c("A","A","B","B"), f2=c("C","D","C","D"), Freq=c(1,2,3,4))
untable(d.freq)
```

---

unwhich *Inverse Which*

Description

Reconstructs the TRUE positions from the index vector returned by which(), producing a logical vector of length n. Note that this is not a perfect inverse: which() discards NA and FALSE positions, so the original vector cannot be fully recovered.

Usage

```
unwhich(
  idx,
  n = if (length(idx) > 0L && !anyNA(idx) && all(idx > 0L)) max(idx) else 0L,
  useNames = TRUE
)
```

Arguments

idx	a vector of non-zero whole-number indices. Positive values mark TRUE positions; negative values mark FALSE positions (all others become TRUE). As in base R, positive and negative indices must not be mixed. Duplicate indices are allowed and result in a single TRUE (or FALSE) at that position.
n	a single non-negative whole number giving the length of the result. For positive idx, defaults to max(idx); for negative or empty idx, defaults to 0L. Must not be less than max(abs(idx)).
useNames	logical. If TRUE (default) *and* idx has names, those names are attached to the corresponding TRUE positions of the result; all other positions receive an empty string. If FALSE or idx is unnamed, the result has no names. Ignored for negative indices.

Details

Negative indices follow standard R semantics: unwhich(-2, 5) returns a vector with TRUE everywhere *except* position 2. Positive and negative indices must not be mixed.

Value

a logical vector of length n.

Note

The positive-index construction (`rv[indices] <- TRUE` with name propagation) is based on code by Nick Sabbe; negative-index handling and input validation are original additions.

References

Sabbe, N. (2012). Inverse of `which`. https://stackoverflow.com/questions/7659833/inverse-of-which

See Also

`which()`

Other vector.utils: `nz()`

Examples

```
ll <- c(TRUE, FALSE, TRUE, NA, FALSE, FALSE, TRUE)
names(ll) <- letters[seq_along(ll)]
i <- which(ll)

# reconstruct TRUE positions (names preserved on TRUE positions)
unwhich(i, length(ll))

# without names
unwhich(i, length(ll), useNames = FALSE)

# negative index: TRUE everywhere except position 2
unwhich(-2, 5)

# empty index -> all-FALSE vector
unwhich(integer(0), n = 5L)
```

---

urlExists *Check if a File Exists at a URL*

Description

Performs an HTTP request to determine whether a resource exists at a given URL. Uses a `HEAD` request by default and falls back to `GET` if necessary.

Usage

```
urlExists(url, timeout = 5)
```

Arguments

`url`	character string. The full URL to check.
`timeout`	numeric. Timeout in seconds for the HTTP request. Default is 5.

Details

The function first sends an HTTP `HEAD` request to minimize data transfer. If the server responds with a status indicating that `HEAD` itself is not supported (403, 405, 501), a `GET` request is attempted as a fallback. A plain 404 is taken at face value, so that non-existing files do not trigger a second request.

If the request fails (e.g., due to network issues or invalid URLs), the function returns `FALSE` and stores the error message as an attribute.

Value

logical value indicating whether the resource exists (`TRUE`) or not (`FALSE`).

The returned value has additional attributes:

- `status`: HTTP status code returned by the server (e.g. 200, 404).
- `error`: error message (if a request error occurred).

See Also

Other file.path: `buildPath()`, `findDownload()`, `isFilePath()`, `isUrl()`, `splitPath()`

Examples

```
# needs an internet connection; an unreachable host is reported
# through the attributes rather than by an error
urlExists("https://www.example.com/data.csv")

res <- urlExists("https://invalid-url.test/file.csv")
attr(res, "status")
attr(res, "error")
```

---

vRot *Rotate a vector*

Description

Rotates a vector cyclically to the right by k positions. Negative values of k rotate to the left.

Usage

```
vRot(x, k = 1L)
```

Arguments

x	a vector.
k	integer. Number of positions to rotate (default = 1).

Details

The rotation is cyclic, meaning elements shifted off one end reappear on the other.

Value

a vector of the same length as x, rotated cyclically.

See Also

Other vector.reshape: setLength(), trim(), vShift()

Examples

```
vRot(1:5, 2)
# 4 5 1 2 3

vRot(1:5, -1)
# 2 3 4 5 1
```

---

vShift *Shift a vector with NA padding*

Description

Shifts a vector to the left or right by k positions. Vacated positions are filled with NA.

Usage

```
vShift(x, k = 1L)
```

Arguments

x	a vector.
k	integer. Number of positions to shift. Positive values shift to the right, negative values to the left.

Details

Unlike vRot(), this function does not wrap elements around. Elements shifted beyond the vector bounds are discarded.

Value

a vector of the same length as x, shifted with NA padding.

See Also

Other vector.reshape: setLength(), trim(), vRot()

Examples

```
vShift(1:5, 2)
# NA NA 1 2 3

vShift(1:5, -2)
# 3 4 5 NA NA

vShift(1:5, 10)
# NA NA NA NA NA
```

`winsorize` *Winsorize a Numeric Vector*

Description

Winsorization replaces extreme values in a numeric vector by less extreme, predefined bounds. Values below a lower limit are set to that limit, and values above an upper limit are set to that upper limit.

Usage

```
winsorize(x, limits = quantile(x, probs = c(0.05, 0.95), na.rm = TRUE))
```

Arguments

`x`	a numeric vector to be winsorized.
`limits`	a numeric vector of length two specifying the lower and upper winsorization limits. Defaults to the 5% and 95% quantiles of `x` with `na.rm = TRUE`.

Details

By default, the limits are defined as the 5% and 95% quantiles of the data. Missing values are ignored when computing quantiles and are preserved in the output.

Formally, the winsorized vector $g(x)$ is defined as:

$$g(x) = \begin{cases} l & \text{if } x \leq l \\ x & \text{if } l < x < u \\ u & \text{if } x \geq u \end{cases}$$

where l and u denote the lower and upper bounds.

The argument `limits` allows full control over the limits. It can be:

- A numeric vector of length two specifying fixed bounds
- The result of a call to `quantile()` (e.g. with custom `type`)

Winsorization is commonly used in robust statistics to reduce the influence of outliers. In some cases, it can be beneficial to standardize the data (e.g., using `scale()`) before applying winsorization.

Value

a numeric vector of the same length as `x`, where:

- values below the lower limit are replaced by the lower limit.
- values above the upper limit are replaced by the upper limit.
- missing values remain unchanged.

See Also

`DescToolsX::scaleX()`, `robustHD::winsorize()`

Other math.transform: `linScale()`, `logit()`, `percentRank()`, `rankX()`

Examples

```
set.seed(9128)
x <- c(rnorm(10), NA, -100, 100)

# Default winsorization (5% / 95% quantiles)
winsorize(x)

# Winsorization using fixed bounds
winsorize(x, limits = c(-10, 10))

# Custom quantile definition
winsorize(x, limits = quantile(x, c(0.1, 0.9), type = 1, na.rm = TRUE))

# One-sided winsorization
winsorize(x, limits = c(-Inf, 2))  # upper bound only
winsorize(x, limits = c(-2, Inf)) # lower bound only
```

---

withSeed *Evaluate an expression under a temporary random seed*

Description

Sets the random seed for the duration of `expr` and restores the caller's random state afterwards. This makes a single result reproducible without hijacking the random stream of the surrounding script: two calls with the same seed give the same result, and whatever is drawn after them is unaffected by either.

Usage

```
withSeed(seed, expr)
```

Arguments

seed	a single number, or NULL to leave the random stream untouched and simply evaluate expr
expr	the expression to evaluate; evaluated lazily, in the caller's environment

Details

The plain idiom set.seed(s); expr lacks the second half. In a script that generates a series of random objects, seeding one of them shifts every draw that follows, so results that were correct before the seed was added silently change.

Value

the value of expr

Examples

```
set.seed(1)
a <- runif(1)

set.seed(1)
withSeed(99, runif(1))     # unrelated draw in between
identical(runif(1), a)     # the stream continued as if it never happened

identical(withSeed(7, runif(3)), withSeed(7, runif(3)))
```

Index

∗ **append**
appendEnum, 7
appendRowNames, 9
appendX, 10
multMerge, 88
∗ **attribute**
label, 76
renameX, 126
setAttr-removeAttr-keepAttr, 146
setNamesX, 148
∗ **binary**
asBinary, 13
flags, 56
isDichotomous, 65
∗ **cardinality**
isLowCardinality, 68
nUnique, 94
∗ **categorization**
combLevels, 36
dummy, 50
nf, 91
randGroupSplit, 113
recodeX, 122
resolveGroups, 136
revCode, 137
splitX, 154
stringsAsFactors, 156
∗ **combinatorics**
combN, 37
combPairs, 38
combSet, 39
pairApply, 97
permn, 103
randGroupSplit, 113
sampleX, 145
∗ **comparison**
allDuplicated, 5
allIdentical, 6
compareDataFrames, 40
∗ **data-inspection**
allIdentical, 6
compareDataFrames, 40
completeColumns, 41
countCompCases, 43
flags, 56
isNumeric, 69
isWholeLike, 73
isZero, 75
naIf, 89
nUnique, 94
nz, 95
peekFile, 100
setLength, 147
splitAt, 152
strX, 159
unwhich, 169
∗ **data-resolution**
resolveContingency, 128
resolveFormula, 130
resolveGroups, 136
∗ **data.append**
appendEnum, 7
appendRowNames, 9
appendX, 10
multMerge, 88
∗ **data.coerce**
as.array.xtabs, 12
toBaseR, 161
type-aliases, 164
∗ **data.equal**
allDuplicated, 5
allIdentical, 6
compareDataFrames, 40
∗ **data.interval**
between-operators, 17
intervals, 63
range-operators, 114
∗ **data.missing**

completeColumns, 41
countCompCases, 43
∗ **data.order**
binaryTree, 20
revX, 139
sortX, 149
∗ **data.predicate**
flags, 56
isDichotomous, 65
isEuclid, 66
isLowCardinality, 68
isNumeric, 69
isWholeLike, 73
isZero, 75
nUnique, 94
∗ **data.print**
columnWrap, 35
printCharMatrix, 109
∗ **data.recode**
asBinary, 13
combLevels, 36
dummy, 50
mReplace, 87
nf, 91
recodeX, 122
revCode, 137
stringsAsFactors, 156
∗ **data.reshape**
collapseTable, 33
long-wide-reshape, 81
splitAt, 152
splitX, 154
untable, 167
∗ **data.resolve**
resolveContingency, 128
resolveFormula, 130
resolveGroups, 136
∗ **datasets**
Cards, 23
dataDescription, 46
Pizza, 104
readCourseData, 120
Roulette, 141
Tarot, 160
∗ **date.format**
asCDateFmt, 14
∗ **dummy-coding**
dummy, 50
strSplitToDummy, 158
∗ **encoding**
asBinary, 13
char-ascii-conversion, 24
numeric-conversions, 92
∗ **file.io**
findDownload, 55
openDataObject, 96
parseSasDatalines, 98
pdfManual, 99
peekFile, 100
readCourseData, 120
readDownload, 121
∗ **file.path**
buildPath, 21
findDownload, 55
isFilePath, 67
isUrl, 73
splitPath, 153
urlExists, 170
∗ **formatting**
asCDateFmt, 14
columnWrap, 35
printCharMatrix, 109
∗ **geometry**
isEuclid, 66
ptInPoly, 110
∗ **imputation**
coalesceX, 32
locf, 78
naReplace, 90
∗ **indexing**
unwhich, 169
∗ **introspection**
auditNames, 16
concepts, 42
extractArgs, 52
funArgs, 57
funCalls, 58
funKeywords, 59
funList, 60
getDotsArg, 62
mergeArgs, 82
pdfManual, 99
rdLabels, 118
rdTitle, 119
recycle, 125
strX, 159

∗ **label.attrs**
label, 76
renameX, 126
setAttr-removeAttr-keepAttr, 146
setNamesX, 148
∗ **label.import**
dataDescription, 46
openDataObject, 96
∗ **label**
dataDescription, 46
label, 76
openDataObject, 96
rdLabels, 118
renameX, 126
setNamesX, 148
∗ **linear-algebra**
crossProd, 44
crossProdN, 45
dotProd, 49
∗ **math.basic**
closest, 30
crossProd, 44
crossProdN, 45
dotProd, 49
roundTo, 142
unirootAll, 165
∗ **math.geometry**
ptInPoly, 110
∗ **math.precision**
precision, 106
∗ **math.transform**
linScale, 77
logit, 79
percentRank, 102
rankX, 115
winsorize, 173
∗ **missing-value**
coalesceX, 32
completeColumns, 41
countCompCases, 43
isNA, 68
naIf, 89
naReplace, 90
∗ **moving-window**
midx, 84
moveAvg, 85
quot, 111
∗ **number-theory**
combN, 37
combPairs, 38
digitSum, 47
divisors, 48
factorize, 53
fibonacci, 54
GCD-LCM, 61
isOdd, 71
isPrime, 71
permn, 103
primes, 108
∗ **number.baseconv**
numeric-conversions, 92
∗ **number.theory**
digitSum, 47
divisors, 48
factorize, 53
fibonacci, 54
GCD-LCM, 61
isOdd, 71
isPrime, 71
primes, 108
∗ **numerical-methods**
closest, 30
crossProd, 44
crossProdN, 45
digitSum, 47
divisors, 48
dotProd, 49
factorize, 53
fibonacci, 54
GCD-LCM, 61
logit, 79
midx, 84
precision, 106
primes, 108
quot, 111
roundTo, 142
unirootAll, 165
∗ **ordering**
binaryTree, 20
revX, 139
sortX, 149
∗ **outlier-detection**
trim, 163
winsorize, 173
∗ **overlap**
intervals, 63

∗ **path-handling**
buildPath, 21
findDownload, 55
isFilePath, 67
isUrl, 73
splitPath, 153
urlExists, 170
∗ **pkg.args**
callIf, 22
extractArgs, 52
getDotsArg, 62
mergeArgs, 82
recycle, 125
∗ **pkg.funinfo**
auditNames, 16
funArgs, 57
funCalls, 58
funKeywords, 59
funList, 60
rdLabels, 118
rdTitle, 119
∗ **pkg.introspection**
concepts, 42
∗ **precision**
precision, 106
∗ **probability**
Cards, 23
Roulette, 141
∗ **programming**
auditNames, 16
callIf, 22
extractArgs, 52
funArgs, 57
funCalls, 58
funKeywords, 59
funList, 60
getDotsArg, 62
mergeArgs, 82
pairApply, 97
rdTitle, 119
recycle, 125
resolveFormula, 130
setAttr-removeAttr-keepAttr, 146
∗ **random.numbers**
rBetaShape, 116
rSum21, 144
∗ **random**
rSum21, 144
withSeed, 174
∗ **range**
between-operators, 17
intervals, 63
range-operators, 114
unirootAll, 165
∗ **recoding**
combLevels, 36
mGsub, 83
mReplace, 87
recodeX, 122
revCode, 137
∗ **reshape**
collapseTable, 33
long-wide-reshape, 81
revX, 139
setLength, 147
sortX, 149
splitAt, 152
splitX, 154
strSplitToCol, 157
trim, 163
untable, 167
vRot, 171
vShift, 172
∗ **sampling**
combSet, 39
randGroupSplit, 113
rBetaShape, 116
rSum21, 144
sampleX, 145
∗ **shift**
vRot, 171
vShift, 172
∗ **simulation**
Cards, 23
Pizza, 104
Roulette, 141
Tarot, 160
∗ **standardization**
linScale, 77
∗ **string-manipulation**
buildPath, 21
columnWrap, 35
mGsub, 83
mReplace, 87
parseSasDatalines, 98
splitPath, 153

strSplitToCol, 157
strSplitToDummy, 158
∗ **string.transform**
char-ascii-conversion, 24
mGsub, 83
strSplitToCol, 157
strSplitToDummy, 158
∗ **table**
appendEnum, 7
appendRowNames, 9
appendX, 10
as.array.xtabs, 12
collapseTable, 33
long-wide-reshape, 81
multMerge, 88
printCharMatrix, 109
readDownload, 121
resolveContingency, 128
∗ **ties**
allDuplicated, 5
rankX, 115
∗ **time-series**
locf, 78
∗ **transformation**
linScale, 77
logit, 79
percentRank, 102
rankX, 115
winsorize, 173
∗ **type-coercion**
as.array.xtabs, 12
nf, 91
stringsAsFactors, 156
toBaseR, 161
type-aliases, 164
∗ **type-test**
isDichotomous, 65
isEuclid, 66
isFilePath, 67
isLowCardinality, 68
isNA, 68
isNumeric, 69
isOdd, 71
isPrime, 71
isUrl, 73
isWholeLike, 73
isZero, 75
ptInPoly, 110
urlExists, 170
∗ **utils**
withSeed, 174
∗ **vector.na**
coalesceX, 32
isNA, 68
locf, 78
naIf, 89
naReplace, 90
∗ **vector.reshape**
setLength, 147
trim, 163
vRot, 171
vShift, 172
∗ **vector.utils**
nz, 95
unwhich, 169
∗ **vector.window**
midx, 84
moveAvg, 85
quot, 111
%)[% *(between-operators)*, 17
%::% *(range-operators)*, 114
%:% *(range-operators)*, 114
%[)% *(between-operators)*, 17
%[]% *(between-operators)*, 17
%][% *(between-operators)*, 17
%overlaps% *(intervals)*, 63

all.equal(), *75*
allDuplicated, 5
allDuplicated(), *7*, *41*
allIdentical, 6
allIdentical(), *6*, *41*
anyNA(), *42*
append(), *8–10*
appendEnum, 7
appendEnum(), *9*, *10*, *88*
appendRowNames, 9
appendRowNames(), *8*, *10*, *88*
appendX, 10
appendX(), *8*, *9*, *88*
applySides, 11
args(), *58*
as.array.xtabs, 12
as.array.xtabs(), *162*, *165*
as.character(), *106*
as.hexmode(), *93*
as.integer(), *107*

as.matrix(), *128*
as.matrix.xtabs *(*as.array.xtabs*)*, 12
as.numeric(), *91*
as.octmode(), *93*
as.roman(), *93*
asBinary, 13
asBinary(), *36*, *51*, *87*, *91*, *123*, *138*, *156*, *164*, *165*
asCDateFmt, 14
asciiToChar *(*char-ascii-conversion*)*, 24
auditNames, 16
auditNames(), *58–61*, *119*

basename(), *153*, *154*
baseToBase *(*numeric-conversions*)*, 92
between-operators, 17
bin *(*type-aliases*)*, 164
binaryTree, 20
binaryTree(), *140*, *151*
binToDec *(*numeric-conversions*)*, 92
boxplot(), *132*
boxplot.formula(), *133*
browseURL(), *100*
buildPath, 21
buildPath(), *55*, *67*, *73*, *154*, *171*

callIf, 22
callIf(), *52*, *63*, *83*, *125*
Cards, 23, *105*, *121*, *142*, *161*
ceiling(), *142*, *143*
char-ascii-conversion, 24
charToAscii *(*char-ascii-conversion*)*, 24
charToRaw(), *25*
checkConfLevel, 26
checkConfLevel(), *12*, *27–29*
checkCount, 27
checkCount(), *26*, *28*, *29*
checkFlag, 28
checkFlag(), *12*, *26*, *27*, *29*
checkString, 29
checkString(), *26–28*
choose(), *37*, *39*
chr *(*type-aliases*)*, 164
closest, 30
closest(), *45*, *46*, *50*, *143*, *167*
coalesceX, 32
coalesceX(), *69*, *79*, *89*, *90*
collapseTable, 33
collapseTable(), *82*, *153*, *155*, *168*
columnWrap, 35
columnWrap(), *110*
combLevels, 36
combLevels(), *14*, *51*, *87*, *91*, *123*, *138*, *156*
combN, 37
combN(), *38*, *39*, *97*, *103*, *113*, *146*
combn(), *37–39*, *103*
combPairs, 38
combPairs(), *37*, *39*, *97*, *103*, *113*, *146*
combSet, 39
combSet(), *37*, *38*, *97*, *103*, *113*, *146*
compareDataFrames, 40
compareDataFrames(), *6*, *7*
complete.cases(), *42*, *44*
completeColumns, 41
completeColumns(), *44*
conceptAudit *(*concepts*)*, 42
conceptMap *(*concepts*)*, 42
concepts, 42
contr.helmert(), *51*
contr.poly(), *51*
contr.sum(), *51*
contr.treatment(), *51*
contrasts(), *51*
countCompCases, 43
countCompCases(), *42*
crossProd, 44
crossProd(), *31*, *46*, *50*, *143*, *167*
crossprod(), *49*, *50*
crossProdN, 45
crossProdN(), *31*, *45*, *50*, *143*, *167*

data.frame(), *156*
data.table::frankv(), *116*
dataDescription, 46
dataDescription(), *96*
decToBin *(*numeric-conversions*)*, 92
decToHex *(*numeric-conversions*)*, 92
decToOct *(*numeric-conversions*)*, 92
diff(), *112*
digitSum, 47
digitSum(), *49*, *53*, *54*, *62*, *71*, *72*, *108*
dirname(), *153*, *154*
distance *(*intervals*)*, 63
distance(), *18*
divisors, 48
divisors(), *48*, *53*, *54*, *62*, *71*, *72*, *108*
dotProd, 49
dotProd(), *31*, *45*, *46*, *143*, *167*

dummy, 50
dummy(), *14*, *36*, *87*, *91*, *123*, *138*, *156*
duplicated(), *5, 6*

expand.grid(), *38*, *168*
extractArgs, 52
extractArgs(), *23*, *63*, *83*, *125*

factor(), *91*, *123*, *136*, *165*
factorial(), *37*, *39*, *103*
factorize, 53
factorize(), *48, 49*, *54*, *62*, *71, 72*, *108*
fibonacci, 54
fibonacci(), *48, 49*, *53*, *62*, *71, 72*, *108*
file.path(), *21*
file.show(), *59*
find(), *59*
findDownload, 55
findDownload(), *21*, *67*, *73*, *122*, *154*, *171*
flags, 56
flags(), *65, 66*, *68*, *70*, *74, 75*, *94*
floor(), *142, 143*
formals(), *58*
format.info(), *107*
frac *(*precision*)*, 106
funArgs, 57
funArgs(), *17*, *59–61*, *119*
funCalls, 58
funCalls(), *17*, *58*, *60, 61*, *119*
funKeywords, 59
funKeywords(), *17*, *58, 59*, *61*, *119*
funList, 60
funList(), *17*, *58–60*, *119*

GCD *(*GCD-LCM*)*, 61
GCD-LCM, 61
getConcepts *(*concepts*)*, 42
getDotsArg, 62
getDotsArg(), *23*, *52*, *83*, *125*
getNamespaceExports(), *61*
getParseData(), *59*
gl(), *168*
gsub(), *126*

head(), *100, 101*
help(), *60*
hexmode(), *92*
hexToDec *(*numeric-conversions*)*, 92

identical(), *6, 7*, *41*
if(), *19*
ifelse(), *19*
int *(*type-aliases*)*, 164
interaction(), *131*
intervals, *19*, 63, *114*
is.finite(), *32*
is.na(), *32*, *42*, *44*, *69*
isDichotomous, 65
isDichotomous(), *57*, *66*, *68*, *70*, *74, 75*, *94*
isEuclid, 66
isEuclid(), *57*, *65*, *68*, *70*, *74, 75*, *94*
isFilePath, 67
isFilePath(), *21*, *55*, *73*, *154*, *171*
isLowCardinality, 68
isLowCardinality(), *57*, *65, 66*, *70*, *74, 75*, *94*
isNA, 68
isNA(), *32*, *79*, *89, 90*
isNumeric, 69
isNumeric(), *57*, *65, 66*, *68*, *74, 75*, *94*
isOdd, 71
isOdd(), *48, 49*, *53, 54*, *62*, *72*, *108*
isPrime, 71
isPrime(), *48, 49*, *53, 54*, *62*, *71*, *108*
isUrl, 73
isUrl(), *21*, *55*, *67*, *154*, *171*
isWholeLike, 73
isWholeLike(), *57*, *65, 66*, *68*, *70*, *75*, *94*
isZero, 75
isZero(), *57*, *65, 66*, *68*, *70*, *74*, *94, 95*

keepAttr *(*setAttr-removeAttr-keepAttr*)*, 146

label, 76
label(), *127*, *146*, *148*
label<- *(*label*)*, 76
LCM *(*GCD-LCM*)*, 61
length(), *48*
levels(), *123*
linScale, 77
linScale(), *80*, *102*, *116*, *174*
list(), *53*
lm(), *132, 133*
locf, 78
locf(), *32*, *69*, *89, 90*
logit, 79
logit(), *78*, *102*, *116*, *174*
logitInv *(*logit*)*, 79

long-wide-reshape, 81
lower.tri(), *38*
ls(), *61*
ls.str(), *61*
lsf.str(), *61*

match(), *126*
match.arg(), *11*
match.call(), *133*
maxDec *(*precision*)*, 106
mean(), *163*
merge(), *88*
mergeArgs, 82
mergeArgs(), *23*, *52*, *63*, *125*
mGsub, 83
mGsub(), *25*, *87*, *157*, *159*
midx, 84
midx(), *87*, *112*
model.frame(), *51*, *131*, *133, 134*, *155*
modifyList(), *22*, *83*
moveAvg, 85
moveAvg(), *85*, *112*
mReplace, 87
mReplace(), *14*, *36*, *51*, *84*, *91*, *123*, *138*, *156*
multMerge, 88
multMerge(), *8–10*

NA(), *112*
na.exclude(), *158*
na.fail(), *158*
na.omit(), *42*, *44*, *134*, *155*, *158, 159*
na.pass(), *131, 132*, *134*, *158*
naIf, 89
naIf(), *32*, *69*, *79*, *90*
names(), *126, 127*
naReplace, 90
naReplace(), *32*, *69*, *79*, *89*
nchr *(*type-aliases*)*, 164
nDec *(*precision*)*, 106
nf, 91
nf(), *14*, *36*, *51*, *87*, *123*, *138*, *156*, *165*
nlevels(), *94*
normalizePath(), *21*, *154*
num *(*type-aliases*)*, 164
numeric-conversions, 92
nUnique, 94
nUnique(), *57*, *65, 66*, *68*, *70*, *74, 75*
nz, 95
nz(), *170*

octToDec *(*numeric-conversions*)*, 92
offset(), *132*
openDataObject, 96
openDataObject(), *47*
order(), *140*, *150, 151*
outer(), *38*, *97*
overlaps *(*intervals*)*, 63
overlapSize *(*intervals*)*, 63
overlapSize(), *18*

p.adjust.methods, *17*
Pair(), *130*, *132*, *134*
pairApply, 97
pairApply(), *37–39*, *103*, *113*, *146*
pairwise.table(), *97*
parseSasDatalines, 98
parseSasDatalines(), *100, 101*, *122*
pdfManual, 99
pdfManual(), *99*, *101*, *122*
peekFile, 100
peekFile(), *99, 100*, *122*
percentRank, 102
percentRank(), *78*, *80*, *116*, *174*
permn, 103
permn(), *37–39*, *97*, *113*, *146*
Pizza, *24*, 104, *121*, *142*, *161*
plogis(), *80*
prec *(*precision*)*, 106
precision, 106
primes, 108
primes(), *48, 49*, *53, 54*, *62*, *71, 72*
printCharMatrix, 109
printCharMatrix(), *35*
ptInPoly, 110

qlogis(), *80*
quantile(), *173*
quot, 111
quot(), *85*, *87*

randGroupSplit, 113
randGroupSplit(), *37–39*, *97*, *103*, *146*
range-operators, 114
rank(), *102*, *115, 116*
rankX, 115
rankX(), *78*, *80*, *102*, *174*
rawToChar(), *25*
rbeta(), *117*
rBetaShape, 116

rBetaShape(), *145*
rbind(), *157*
rdLabels, 118
rdLabels(), *17*, *58–61*, *119*
rdTitle, 119
rdTitle(), *17*, *58–61*, *119*
readCourseData, 120
readCourseData(), *24*, *105*, *142*, *161*
readDownload, 121
readDownload(), *99–101*
readr::read_csv(), *122*
readr::read_delim(), *101*
readxl::read_excel(), *122*
recodeX, 122
recodeX(), *14*, *36*, *51*, *87*, *91*, *138*, *156*
recycle, 125
recycle(), *23*, *52*, *63*, *83*
relevel(), *123*
removeAttr *(*setAttr-removeAttr-keepAttr*)*, 146
renameX, 126
renameX(), *76*, *146*, *148*
reorder(), *123*
rep(), *125*, *168*
replicate(), *125*
reshape(), *82*
resolveContingency, 128
resolveContingency(), *134*, *137*
resolveFormula, 130
resolveFormula(), *129*, *137*
resolveFormulaFromCall *(*resolveFormula*)*, 130
resolveGroups, 136
resolveGroups(), *129*, *134*
rev(), *139*, *140*
revCode, 137
revCode(), *14*, *36*, *51*, *87*, *91*, *123*, *156*
revX, 139
revX(), *20*, *151*
romanToInt *(*numeric-conversions*)*, 92
Roulette, *24*, *105*, *121*, 141, *161*
round(), *142*, *143*
roundTo, 142
roundTo(), *31*, *45*, *46*, *50*, *167*
rSum21, 144
rSum21(), *117*
runif(), *117*
runmed(), *87*

sample(), *113*, *145*, *146*
sampleX, 145
sampleX(), *37–39*, *97*, *103*, *113*
scale(), *78*, *173*
seq(), *140*
set.seed(), *16*, *113*, *117*
setAttr *(*setAttr-removeAttr-keepAttr*)*, 146
setAttr-removeAttr-keepAttr, 146
setLength, 147
setLength(), *164*, *172*
setNames(), *127*, *146*, *148*
setNamesX, 148
setNamesX(), *76*, *127*, *146*
sort(), *140*, *149–151*
sortX, 149
sortX(), *20*, *103*, *140*
split(), *6*, *153–155*
splitAt, 152
splitAt(), *34*, *82*, *155*, *168*
splitPath, 153
splitPath(), *21*, *55*, *67*, *73*, *171*
splitX, 154
splitX(), *34*, *82*, *153*, *168*
stack(), *82*
stats::uniroot(), *167*
str(), *159*, *160*
stringsAsFactors, 156
stringsAsFactors(), *14*, *36*, *51*, *87*, *91*, *123*, *138*
strsplit(), *157–159*
strSplitToCol, 157
strSplitToCol(), *25*, *84*, *159*
strSplitToDummy, 158
strSplitToDummy(), *25*, *84*, *157*
strtoi(), *93*
strwrap(), *35*
strX, 159
substitute(), *128*

table(), *6*, *128*, *129*
Tarot, *24*, *105*, *121*, *142*, 160
toBaseR, 161
toBaseR(), *13*, *101*, *121*, *122*, *165*
toLong *(*long-wide-reshape*)*, 81
tools::file_ext(), *153*, *154*
tools::file_path_sans_ext(), *153*, *154*

tools::parse_Rd(), *119*
tools::Rd_db(), *43*, *118*, *119*
toWide *(*long-wide-reshape*)*, 81
trim, 163
trim(), *148*, *172*
trunc(), *107*, *142*, *143*
type-aliases, 164

unique(), *6*
uniroot(), *165*, *166*
unirootAll, 165
unirootAll(), *31*, *45*, *46*, *50*, *143*
unname(), *146*
unstack(), *82*
untable, 167
untable(), *34*, *82*, *153*, *155*
unwhich, 169
unwhich(), *95*
urlExists, 170
urlExists(), *21*, *55*, *67*, *73*, *154*

vRot, 171
vRot(), *148*, *164*, *172*
vShift, 172
vShift(), *148*, *164*, *172*

which(), *31*, *169*, *170*
winsorize, 173
winsorize(), *78*, *80*, *102*, *116*
withSeed, 174
withSeed(), *117*

xtabs(), *128*, *168*