Package ‘str2str’

May 9, 2026

Type Package

Title Convert R Objects from One Structure to Another

Version 1.0.0

Description Offers a suite of functions for converting to and from (atomic) vectors, matrices, data.frames, and (3D+) arrays as well as lists of these objects. It is an alternative to the base R as.<str>.<method>() functions (e.g., as.data.frame.array()) that provides more useful and/or flexible restructuring of R objects. To do so, it only works with common structuring of R objects (e.g., data.frames with only atomic vector columns).

Depends R (>= 4.0.0), datasets, stats, utils, methods

Imports abind, checkmate, plyr, reshape

License GPL (>= 2)

Encoding UTF-8

RoxygenNote 7.2.3

NeedsCompilation no

Author David Disabato [aut, cre]

Maintainer David Disabato <ddisab01@gmail.com>

Repository CRAN

Date/Publication 2023-11-20 21:50:02 UTC

Contents

str2str-package 3
a2d 4
a2la 6
a2ld 7
a2lm 7
a2v 8
abind<- 9
all_diff 11

all_same . . . 12
append<- . . . 13
cat0 . . . 14
cbind<- . . . 15
cbind_fill . . . 17
cbind_fill_matrix . . . 18
codes . . . 20
d2a . . . 21
d2d . . . 23
d2ld . . . 25
d2lv . . . 26
d2m . . . 27
d2v . . . 28
dimlabels . . . 30
dimlabels<- . . . 31
e2l . . . 32
fct2v . . . 33
grab . . . 34
inbtw . . . 35
is.avector . . . 36
is.cnumeric . . . 37
is.colnames . . . 38
is.Date . . . 38
is.dummy . . . 39
is.empty . . . 40
is.names . . . 40
is.POSIXct . . . 41
is.POSIXlt . . . 42
is.row.names . . . 42
is.rownames . . . 43
is.whole . . . 44
Join . . . 44
la2a . . . 47
laynames . . . 48
ld2a . . . 49
ld2d . . . 50
ld2v . . . 52
lm2a . . . 54
lm2d . . . 55
lm2v . . . 57
lv2d . . . 59
lv2m . . . 61
lv2v . . . 63
m2d . . . 65
m2lv . . . 66
m2v . . . 67
ndim . . . 68
nlay . . . 69

not.colnames 70
not.names 70
not.row.names 71
not.rownames 71
order.custom 72
pick 73
rbind<- 75
sn 77
stack2 77
try_expr 79
try_fun 80
t_list 81
undim 83
undimlabel 83
undimname 84
unstack2 85
v2d 87
v2fct 89
v2frm 90
v2lv 91
v2m 92

Index **94**

`str2str-package` *Structure to Structure*

Description

`str2str` is a package for converting R objects to different structures. It focuses on four primary R objects: (atomic) vectors, matrices, data.frames, and arrays as well as lists of these objects. For example, converting a (atomic) vector to a data.frame (i.e., `v2d()`) or a list of (atomic) vectors to a matrix (i.e., `lv2m()`. The current version of the package does not have a function for every convertion (e.g., `a2m()`), but some additional convertion functions may be included in future versions if I find a use for them. The package was motivated by limitations of the base R `as.<str>.<method>` suite of functions and the `plyr` R package `**ply(.fun = NULL)` suite of functions for converting R objects to different structures. While those functions are often useful, there are times different conversions are desired or different naming schemes are desired. That is what this package offers R users. It also contains various utility functions for working with common R objects. For example, `is.colnames` and `ndim`.

Limitations

This packages does NOT handle the nuances of R objects. It is not for comprehensive restructuring of any version of R objects, but rather for restructuring commonly used versions of R objects. For example, the functions are not tested with the raw and complex typeof atomic vectors, list arrays, or data.frames containing non-atomic vector columns (e.g., matrix or list columns). The base R `as.<str>.<method>` functions allow for comprehensive restructuring of R objects; however, at the

cost of less convenient convertions for commonly used versions of R objects. The `str2str` package seeks to fill that gap in useability.

Abbreviations

See the table below

(atomic) vector

m **v**matrix

d data.frame

a (3D+) array

l list

el elements

nm names

uv unique values

lgl logical

int integer

dbl double

num numeric

chr character

fct factor

lvl levels

vrb variable

frm formula

fun function

rtn return

str structure

Author(s)

Maintainer: David Disabato <ddisab01@gmail.com>

`a2d` *(3D+) Array to Data-Frame*

Description

`a2d` converts a (3D+ array) to a data.frame. It allows you to specify a dimension of the array to be the columns. All other dimensions are variables in the data.frame. This is different than `as.data.frame.array` which converts the (3D+) array to a matrix first; although it is very similar to `as.data.frame.table` when `col` = 0.

Usage

```
a2d(a, col = 0, stringsAsFactors = FALSE, check = TRUE)
```

Arguments

a	3D+ array.
col	integer vector or character vector of length 1 specifing the dimension of a to have as columns in the return object. If an integer vector, col refers to the dimension number. If a character vector, col refers to the name of the dimension (i.e., dimlabel). The columns are in order of the dimnames for that dimension (not alphabetical order like reshape::cast). If 0 (default), then no dimension of the array are columns and the function becomes similar to as.data.frame.table.
stringsAsFactors	logical vector of length 1 specifying whether the variable dimensions should be factors of chracter vectors.
check	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether a is a (3D+) array. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Details

a2d is mostly a wrapper for reshape::melt.array (+ reshape::cast) that allows for the variable dimensions to be character vectors rather than factors.

Value

data.frame of a's elements. The colnames of the variable dimensions are the dimlabels in a. If there were no dimlabels in a, then each dimension is named after its number with an X in front. If col is not 0, then the rest of the colnames are the dimnames of that dimension from a. If col is 0, then the names of the single column with a's elements is "element".

Examples

```
a2d(HairEyeColor)
a2d(HairEyeColor, col = 1)
a2d(HairEyeColor, col = "Hair", stringsAsFactors = TRUE)
a2d(HairEyeColor, col = 2)
a2d(HairEyeColor, col = "Sex", stringsAsFactors = TRUE)
try_expr(a2d(as.matrix(attitude))) # error due to inputting a matrix. Instead use `m2d`.

# correlation array example from psych::corr.test(attitude[1:3])
# corr_test <- psych::corr.test(attitude[1:3])
# a <- lm2a(corr_test[c("r","se","t","p")])
r <- matrix(c(1.0000000, 0.8254176, 0.4261169, 0.8254176, 1.0000000, 0.5582882,
   0.4261169, 0.5582882, 1.0000000), nrow = 3, ncol = 3, byrow = FALSE)
se <- matrix(c(0.0000000, 0.1066848, 0.1709662, 0.1066848, 0.0000000, 0.1567886,
   0.1709662, 0.1567886, 0.0000000), nrow = 3, ncol = 3, byrow = FALSE)
t <- matrix(c(Inf, 7.736978, 2.492404, 7.736978, Inf, 3.560771,
```

```
   2.492404, 3.560771, Inf), nrow = 3, ncol = 3, byrow = FALSE)
p <- matrix(c(0.000000e+00, 1.987682e-08, 1.887702e-02, 5.963047e-08, 0.000000e+00,
  1.345519e-03, 0.018877022, 0.002691039, 0.000000000), nrow = 3, ncol = 3, byrow = FALSE)
a <- abind::abind(r, se, t, p, along = 3L)
dimnames(a) <- list(names(attitude[1:3]), names(attitude[1:3]), c("r","se","t","p"))
d <- a2d(a = a, col = 3)
```

---

a2la *(3D+) Array to List of (3D+) Arrays*

Description

a2la converts an (3D+) array to a list of (3D+) arrays. This function is a simple wrapper for asplit(x = a, MARGIN = along).

Usage

```
a2la(a, along = ndim(a), check = TRUE)
```

Arguments

a	(3D+) array
along	integerish vector of length 1 specifying the dimension to split the array along. Default is the last dimension of a.
check	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether a is a 3D+ array. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

list of arrays where each array is one dimension less than a and the names of the list are dimnames(a)[[along]].

Examples

```
# without dimnames
a <- abind::abind(HairEyeColor*1, HairEyeColor*2, HairEyeColor*3, along = 4L)
a2la(a)
# with dimnames
a <- abind::abind("one" = HairEyeColor*1, "two" = HairEyeColor*2,
   "three" = HairEyeColor*3, along = 4L)
a2la(a)
a2la(a, along = 1) # along = 1
```

a2ld — *3D Array to List of Data-Frames*

Description

a2ld converts a 3D array to a list of data.frames. This is a simple call to a2lm followed by m2d. The default is to convert the third dimension to the list dimension.

Usage

```
a2ld(a, along = 3L, stringsAsFactors = FALSE, check = TRUE)
```

Arguments

a	3D array.
along	integer vector of length 1 specifying the dimension to slice the array along. This dimension is converted to the list dimension. 1 = rows; 2 = columns; 3 = layers.
stringsAsFactors	logical vector of length 1 specifying whether character vectors should be converted to factors. Note, that if the array is character and stringsAsFactors = TRUE, then all columns in the returned list of data.frames will be factors.
check	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether a is a 3D array. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

list of data.frames - all with the same dimensions.

Examples

```
a2ld(HairEyeColor)
a2ld(HairEyeColor, along = 1)
try_expr(a2ld(mtcars)) # error b/c not a 3D array
```

a2lm — *(3D) Array to List of Matrices*

Description

a2lm converts a (3D) array to a list of matrices. This is a simple call to asplit with a default to convert the third dimension to a list dimension.

Usage

```
a2lm(a, along = 3L, check = TRUE)
```

Arguments

a	3D array.
along	integer vector of length 1 specifying the dimension to slice the array along. This dimension is converted to the list dimension. 1 = rows; 2 = columns; 3 = layers.
check	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether a is a 3D array. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

list of matrices - all with the same dimensions.

Examples

```
a2lm(HairEyeColor)
a2lm(HairEyeColor, along = 1)
try_expr(a2lm(mtcars)) # error b/c  not a 3D array
```

---

a2v *(3D+) Array to (Atomic) Vector*

Description

a2v converts a matrix to a (atomic) vector. The benefit of m2v over as.vector or c is that 1) the vector can be formed along rows any sequence of dimensions and 2) the dimnames from a can be used for the names of the returned vector.

Usage

```
a2v(a, along = ndim(a):1, use.dimnames = TRUE, sep = "_", check = TRUE)
```

Arguments

a	3D+ array.
along	numeric vector of length = ndim(a) that contains the integers 1:ndim(a) specifying the order which the array elements should be concatenated. For example, with a 3D array, 3:1 (default) specifies to split the array by layers first, then columns, and then rows. See examples.

`use.dimnames` logical vector of length 1 that specifies whether the dimnames of `a` should be used to create the names for the returned vector. If FALSE, the returned vector will have NULL names. If TRUE, then each element's name will be analogous to `paste(dimnames(a)[[1L]][i], dimnames(a)[[2L]][j], dimnames(a)[[3L]][k],..., sep = sep)`. If `a` does not have dimnames, then they will be replaced by dimension positions.

`sep` character vector of length 1 specifying the string that will separate the dimnames from each dimension in the naming scheme of the return object. Note, `sep` is not used if `use.dimnames` = FALSE.

`check` logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `a` is a 3D+ array. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

(atomic) vector of length = `length(a)` where the order of elements from `a` has been determined by `along` and the names determined by the `use.dimnames`, `dimnames(a)`, and `sep`.

Examples

```
a2v(HairEyeColor) # layers, then columns, then rows (default)
a2v(HairEyeColor, along = c(3,1,2)) # layers, then rows, then columns
a2v(HairEyeColor, along = 1:3) # rows, then columns, then layers
a2v(HairEyeColor, along = 1:3, use.dimnames = FALSE)
```

`abind<-` *Add array slices to 3D+ Array* ‘`abind<-`‘ *adds array slices to arrays as a side effect. It used the function* `abind` *in the* `abind` *package. The purpose of the function is to replace the need to use ary2 <- abind(ary1, mat1); ary3 <- rbind(ary2, mat2); ary4 <- rbind(ary3, mat3), etc. It allows you to specify the dimension you wish to bind* `along` *as well as the dimname you wish to bind* `after`. *Unlike* ‘`cbind<-`‘, ‘`rbind<-`‘, *and* ‘`append<-`‘, *it does not have overwriting functionality (I could not figure out how to code that); therefore, if* `value` *has some dimnames that are the same as those in* `a`, *it will NOT overwrite them and simply bind them to* `a`, *resulting in duplicate dimnames.*

Description

Some traditional R folks may find this function uncomfortable. R is famous for limiting side effects, except for a few notable exceptions (e.g., `` `[<-` `` and `` `names<-` ``). Part of the reason is that side effects can be computationally inefficient in R. The entire object often has to be re-constructed and re-saved to memory. For example, a more computationally efficient alternative to abind(ary) <- mat1; abind(ary) <- mat2; abind(ary) <- mat3 is ary1 <- do.call(what = abind, args = list(ary, mat1, mat2, mat3)). However, `` `abind<-` `` was not created for R programming use when computational efficiency is valued; it is created for R interactive use when user convenience is valued.

Usage

```
abind(
  a,
  along = ndim(a),
  after = dim(a)[along],
  dim.nm = NULL,
  overwrite = FALSE
) <- value
```

Arguments

a	3D+ array.
along	either an integer vector with length 1 or a character vector of length 1 specifying the dimension along which to bind value. If an integer vector, it is the position of the dimension. If a character vector it is the dimension with that dimlabel.
after	either an integer vector with length 1 or a character vector of length 1 specifying where to add value within the dimension specified by along. If an integer vector, it is the position within the dimension. If a character vector it is the dimname within the dimension. Similar to append, use 0L if you want the added array slice to be first.
dim.nm	character vector of length equal to ndim(value)[along] that specifies the dimnames of value once added to a as array slices. This is an optional argument that defaults to NULL where the pre-existing dimnames of value are used.
overwrite	not currently used, but there are plans to use it in future versions of the functions. Right now the only option is FALSE.
value	matrix or array to be added as slices to a. Must have ndim equal to ndim(a) or ndim(a) - 1L. Note, the dimensions have to match those in a. For example, if value is a matrix you want to bind along the third dimension of a, then dim(value) must be equal to dim(a)[1:2]. If not, you will get an error from abind::abind.

Value

Like other similar functions (e.g., `names<-` and `[<-`), `rbind<-` does not appear to have a return object. However, it technically does as a side effect. The argument data will have been changed such that value has been added as rows. If a traditional return object is desired, and no side effects, then it can be called like a traditional function: dat2 <- ‘rbind<-‘(dat1, value = add1).

Examples

```
# abind along the last dimension
# default `along` and `after`
HairEyeColor2 <- HairEyeColor
intersex_ary <- array(1:16, dim = c(4,4,1), dimnames = list(NULL, NULL, "Sex" = "Intersex"))
abind(HairEyeColor2) <- intersex_ary
print(HairEyeColor2)
# user-specified `along` and `after`
HairEyeColor2 <- HairEyeColor
```

```
intersex_ary <- array(1:16, dim = c(4,4,1), dimnames = list(NULL, NULL, "Sex" = "Intersex"))
abind(HairEyeColor2, along = "Sex", after = 0L) <- intersex_ary
print(HairEyeColor2)
# matrix as `value`
HairEyeColor2 <- HairEyeColor
intersex_mat <- matrix(1:16, nrow = 4, ncol = 4)
abind(HairEyeColor2, dim.nm = "Intersex") <- intersex_mat
print(HairEyeColor2)

# abind along the first dimension
# array as `value`
HairEyeColor2 <- HairEyeColor
auburn_ary <- array(1:8, dim = c(1,4,2), dimnames = list("Hair" = "Auburn", NULL, NULL))
abind(HairEyeColor2, along = 1L) <- auburn_ary
print(HairEyeColor2)
# matrix as `value`
HairEyeColor2 <- HairEyeColor
auburn_mat <- matrix(1:8, nrow = 4, ncol = 2) # rotate 90-degrees counter-clockwise in your mind
abind(HairEyeColor2, along = 1L, dim.nm = "Auburn") <- auburn_mat
print(HairEyeColor2)
# `after` in the middle
HairEyeColor2 <- HairEyeColor
auburn_mat <- matrix(1:8, nrow = 4, ncol = 2) # rotate 90-degrees counter-clockwise in your mind
abind(HairEyeColor2, along = 1L, after = 2L, dim.nm = "Auburn") <- auburn_mat
print(HairEyeColor2)

# abind along the second dimension
# array as `value`
HairEyeColor2 <- HairEyeColor
amber_ary <- array(1:8, dim = c(4,1,2), dimnames = list(NULL, "Eye" = "Amber", NULL))
abind(HairEyeColor2, along = 2L) <- amber_ary
print(HairEyeColor2)
# matrix as `value`
HairEyeColor2 <- HairEyeColor
amber_mat <- matrix(1:8, nrow = 4, ncol = 2)
abind(HairEyeColor2, along = 2L, dim.nm = "Amber") <- amber_mat
print(HairEyeColor2)
# `after` in the middle
HairEyeColor2 <- HairEyeColor
amber_mat <- matrix(1:8, nrow = 4, ncol = 2)
abind(HairEyeColor2, along = 2L, after = "Blue", dim.nm = "Amber") <- amber_mat
print(HairEyeColor2)
```

`all_diff` *Test if All Elements are Different*

Description

`all_diff` tests if all elements are different. The elements could be either from an atomic vector, list vector, or list. If x does not have any unique values (e.g., NULL), then FALSE is returned.

Usage

```
all_diff(x)
```

Arguments

x	atomic vector, list vector, or list.

Details

The machine precision of `all_diff` for numeric vectors is the same as `unique`. This can causes a problem for some floating-point numbers.

Value

logical vector of length 1 specifying whether all the elements in `x` are the same (TRUE) or not (FALSE).

Examples

```
all_diff(1:10)
all_diff(c(1:10, 10))
all_diff(c(1.0000000, 1.0000001, 0.9999999)) # machine precision good for most cases
all_diff(1) # works for vectors of length 1
```

---

all_same *Test if All Elements are the Same*

Description

`all_same` tests if all elements are the same. The elements could be either from an atomic vector, list vector, or list. If `x` does not have any unique values (e.g., NULL), then FALSE is returned.

Usage

```
all_same(x)
```

Arguments

x	atomic vector, list vector, or list.

Details

The machine precision of `all_same` for numeric vectors is the same as `unique`. This can causes a problem for some floating-point numbers.

Value

logical vector of length 1 specifying whether all the elements in `x` are the same (TRUE) or not (FALSE).

Examples

```
all_same(rep.int("a", times = 10))
all_same(rep.int(1, times = 10))
all_same(c(1.0000000, 1.0000001, 0.9999999)) # machine precision good for most cases
all_same(1) # works for vectors of length 1
```

`append<-` *Add Elements to Vectors*

Description

`` `append<-` `` adds elements to vectors as a side effect. The purpose of the function is to replace the need to use vec2 <- append(vec1, add1); vec3 <- append(vec2, add2); vec4 <- append(vec3, add3), etc. It functions similarly to `` `[<-.default` ``, but allows you to specify the location of the elements similar to `append` (vs. `c`).

Usage

```
append(x, after = length(x), nm = NULL, overwrite = TRUE) <- value
```

Arguments

`x`	atomic vector, list vector, or list.
`after`	either an integer vector with length 1 or a character vector of length 1 specifying where to add `value`. If an integer vector, it is the position of an element. If a character vector, it is the element with that name. Similar to `append`, use 0L if you want the added elements to be first.
`nm`	character vector of length equal to the `length(value)` that specifies the names of `value` once added to `x` as elements. This is an optional argument that defaults to NULL where the pre-existing names of `value` are used.
`overwrite`	logical vector of length 1 specifying whether elements from `value` or `nm` should overwrite elements in `x` with the same names. Note, if `overwrite` = FALSE, repeat names are possible similar to `append`.
`value`	vector of the same typeof as `x` to be added as elements to `x`. Note that for atomic vectors, if more complex elements are added, then the return object will be typeof the most complex element in `x` and `value`.

Details

Some traditional R folks may find this function uncomfortable. R is famous for limiting side effects, except for a few notable exceptions (e.g., `` `[<-` `` and `` `names<-` ``). Part of the reason is that side effects can be computationally inefficient in R. The entire object often has to be re-constructed and re-saved to memory. For example, a more computationally efficient alternative to append(vec) <- add1; append(vec) <- add2; append(vec) <- add3 is vec1 <- do.call(what = c, args = list(dat, add1, add2, add3)). However, `` `append<-` `` was not created for R programming use when computational efficiency is valued; it was created for R interactive use when user convenience is valued.

Value

Like other similar functions (e.g., `names<-` and `[<-`), it does not appear to have a return object. However, it technically does as a side effect. The argument x will have been changed such that value has been added as elements. If a traditional return object is desired, and no side effects, then it can be called like a traditional function: vec2 <- ‘append<-‘(vec1, value = add1).

Examples

```
# no names
x <- letters
append(x) <- LETTERS
append(x, after = match("z", table = x)) <- "case_switch" # with the position
   # of the added value specified

# ya names
y <- setNames(object = letters, nm = LETTERS)
append(y) <- c("ONE" = 1, "TWO" = 2, "THREE" = 3) # with names provided by `value`
tmp <- 1:(length(y) - 3)
y <- y[tmp] # if I put a () inside of a [], Roxygen doesn't like it
append(y, nm = c("ONE","TWO","THREE")) <- c(1,2,3) # with names specified by `nm`
append(y, after = "Z", nm = "ZERO") <- "0" # using name to provide `after`

# using overwrite
append(y, overwrite = TRUE) <- c("ONE" = "one","TWO" = "two", "THREE" = "three")
append(y, overwrite = FALSE) <- c("ONE" = "one","TWO" = "two", "THREE" = "three")
```

`cat0` *Concatenate and Print with No Separator*

Description

`cat0` concatenates and prints objects without any separator. `cat0` is to `cat` as `paste0` is to `paste`. It also allows you to specify line breaks before (`n.before`) and after (`n.after`) the the printing of the concatenated R objects. `cat0` function can be useful in conjunction with `sink` for quick and dirty exporting of results.

Usage

```
cat0(
  ...,
  n.before = 1L,
  n.after = 1L,
  file = "",
  fill = FALSE,
  labels = NULL,
  append = FALSE
)
```

Arguments

`...`	one or more R objects. See details of `cat` for types of objects allowed.
`n.before`	integer vector of length 1 specifying how many line breaks to have before printing the concatenated R objects.
`n.after`	integer vector of length 1 specifying how many line breaks to have after printing the concatenated R objects.
`file`	A connection or a character string naming the file to print to. If "" (default), `cat0` prints to the standard output connection - the console - unless redirected by `sink`.
`fill`	A logical or (positive) numeric vector of length 1 controlling how the output is broken into successive lines. If FALSE (default), new line breaks are only created explicity by `"\n"` bring called. If TRUE, the output is broken into lines with print width equal to the option "width" (`options()[["width"]]`). If a (positive) number, then the output is broken after width of that length.
`labels`	A character vector of labels for the lines printed. Ignored if `fill` = FALSE.
`append`	A logical vector of length 1. Only used if the argument `file` is the name of a file (and not a connection). If TRUE, output will be appended to the existing file. If FALSE, output will overwrite the contents of the file.

Value

nothing as the function only prints and does not return an R object.

Examples

```
cat0(names(attitude))
cat0("MODEL COEFFICIENTS:", coef(lm(rating ~ critical + advance, data = attitude)),
   n.before = 0, n.after = 2)
```

---

cbind<- *Add Columns to Data Objects*

Description

`` `cbind<-` `` adds columns to data objects as a side effect. The purpose of the function is to replace the need to use dat2 <- cbind(dat1, add1); dat3 <- cbind(dat2, add2); dat4 <- cbind(dat3, add3), etc. For data.frames, it functions similarly to `` `[<-.data.frame` ``, but allows you to specify the location of the columns similar to `append` (vs. `c`) and overwrite columns with the same colnames. For matrices, it offers more novel functionality since `` `[<-.matrix` `` does not exist.

Usage

```
cbind(data, after = ncol(data), col.nm = NULL, overwrite = TRUE) <- value
```

Arguments

`data`	data.frame or matrix of data.
`after`	either an integer vector with length 1 or a character vector of length 1 specifying where to add `value`. If an integer vector, it is the position of a column. If a character vector it is the column with that name. Similar to `append`, use 0L if you want the added columns to be first.
`col.nm`	character vector of length equal to `NCOL(value)` that specifies the colnames of `value` once added to `data` as columns. This is an optional argument that defaults to NULL where the pre-existing colnames of `value` are used.
`overwrite`	logical vector of length 1 specifying whether columns from `value` or `col.nm` should overwrite columns in `data` with the same colnames. Note, if `overwrite` = FALSE, repeat colnames are possible similar to `cbind`.
`value`	data.frame, matrix, or atomic vector to be added as columns to `data`. If a data.frame or matrix, it must have the same nrow as `data`. If an atomic vector, it must have length equal to nrow of `data`. Note, if it is an atomic vector and `col.nm` is NULL, then the name of the added column will be "value".

Details

Some traditional R folks may find this function uncomfortable. R is famous for limiting side effects, except for a few notable exceptions (e.g., `` `[<-` `` and `` `names<-` ``). Part of the reason is that side effects can be computationally inefficient in R. The entire object often has to be re-constructed and re-saved to memory. For example, a more computationally efficient alternative to cbind(dat) <- add1; cbind(dat) <- add2; cbind(dat) <- add3 is dat1 <- do.call(what = cbind, args = list(dat, add1, add2, add3)). However, `` `cbind<-` `` was not created for R programming use when computational efficiency is valued; it is created for R interactive use when user convenience is valued.

Similar to `` `cbind` ``, `` `cbind<-` `` works with both data.frames and matrices. This is because `` `cbind` `` is a generic function with a default method that works with matrices and a data.frame method that works with data.frames. Similar to `` `cbind` ``, if colnames of `value` are not given and `col.nm` is left NULL, then the colnames of the return object are automatically created and can be dissatisfying.

Value

Like other similar functions (e.g., `` `names<-` `` and `` `[<-` ``), `` `cbind<-` `` does not appear to have a return object. However, it technically does as a side effect. The argument `data` will have been changed such that `value` has been added as columns. If a traditional return object is desired, and no side effects, then it can be called like a traditional function: dat2 <- ‘cbind<-‘(dat1, value = add1).

Examples

```
attitude2 <- attitude
cbind(attitude2) <- rowMeans(attitude2) # defaults to colnames = "value"
attitude2["value"] <- NULL
cbind(attitude2, col.nm = "mean") <- rowMeans(attitude2) # colnames specified by `col.nm`
attitude2["mean"] <- NULL
cbind(attitude2, after = "privileges", col.nm = c("mean","sum")) <-
   cbind(rowMeans(attitude2), rowSums(attitude2)) # `value` can be a matrix
attitude2[c("mean","sum")] <- NULL
```

```
attitude2 <- `cbind<-`(data = attitude2, value = rowMeans(attitude2)) # traditional call
attitude2["value"] <- NULL
cbind(attitude2, after = "privileges", col.nm = "mean") <-
   rowMeans(attitude2) # `data` can be a matrix
cbind(attitude2) <- data.frame("mean" = rep.int(x = "mean", times = 30L)) # overwrite = TRUE
cbind(attitude2, overwrite = FALSE) <-
   data.frame("mean" = rep.int(x = "mean", times = 30L)) # overwrite = FALSE
cbind(attitude2) <- data.frame("mean" = rep.int(x = "MEAN", times = 30L),
   "sum" = rep.int(x = "SUM", times = 30L)) # will overwrite only the first "mean" column
   # then will append the remaining columns
```

`cbind_fill` *Bind DataFrames Along Columns - Filling in Missing Rows with NA*

Description

`cbind_fill` binds together matrix-like objects by columns. The input objects will all internally be converted to data.frames by the generic function `as.data.frame`. When some objects do not contain rows that are present in other objects, NAs will be added to fill up the returned combined data.frame. If a matrix doesn't have rownames, the row number is used. Note that this means that a row with name "1" is merged with the first row of a matrix without name and so on. The returned matrix will always have row names. Colnames are ignored.

Usage

```
cbind_fill(...)
```

Arguments

`...`	any combination of data.frames, matrices, or atomic vectors input as separate arguments or a list.

Details

`cbind_fill` ensures each object has unique colnames and then calls `Join(by = "0")`. It is intended to be the column version of `plyr::rbind.fill`; it differs by allowing inputs to be matrices or vectors in addition to data.frames.

Value

data.frame created by combining all the objects input together. It will always have rownames. If colnames are not provided to the matrix-like objects, the returned colnames can be rather esoteric since default colnaming will be revised to ensure each colname is unique. If `...` is a list of vectors, then the colnames will be the names of the list.

See Also

`cbind_fill_matrix` `rbind.fill`

Examples

```
# standard use
A <- data.frame("first" = 1:2, "second" = 3:4)
B <- data.frame("third" = 6:8, "fourth" = 9:11)
print(A)
print(B)
cbind_fill(A, B)

# help with unstack()
row_keep <- sample(1:nrow(InsectSprays), size = 36)
InsectSprays2 <- InsectSprays[row_keep, ]
unstacked <- unstack(InsectSprays2)
cbind_fill(unstacked)

# using rownames for binding
rownames(A) <- c("one", "two")
rownames(B) <- c("three","two","one")
print(A)
print(B)
cbind_fill(A, B)

# matrices as input
A <- matrix(1:4, 2)
B <- matrix(6:11, 3)
print(A)
print(B)
cbind_fill(A, B)

# atomic vector input
A <- data.frame("first" = 1:2, "second" = 3:4)
B <- data.frame("third" = 6:8, "fourth" = 9:11)
C <- c(12,13,14,15)
D <- c(16,17,18,19)
cbind_fill(A, B, C, D)

# same as plyr::rbind.fill, it doesn't handles well some inputs with custom rownames
# and others with default rownames
rownames(A) <- c("one", "two")
print(A)
print(B)
cbind_fill(A, B)
```

`cbind_fill_matrix` *Bind Matrices Along Columns - Filling in Missing Rows with NA*

Description

`cbind_fill_matrix` binds together matrix-like objects by columns. The input objects will all internally be converted to matrices by the generic function `as.matrix`. When some objects do not

contain rows that are present in other objects, NAs will be added to fill up the returned combined matrix. If a matrix doesn't have rownames, the row number is used. Note that this means that a row with name "1" is merged with the first row of a matrix without name and so on. The returned matrix will always have row names. Colnames are ignored.

Usage

```
cbind_fill_matrix(...)
```

Arguments

...	any combination of matrices, data.frames, or atomic vectors input as separate arguments or a list.

Details

`cbind_fill_matrix` is `t.default` + `plyr::rbind.fill.matrix` + `t.default` and is based on the code within `plyr::rbind.fill.matrix`.

Value

matrix created by combining all the objects input together. It will always have rownames. It will only have colnames if `...` is a list of vectors, in which the colnames will be the names of the list.

See Also

`cbind_fill` `rbind.fill.matrix`

Examples

```
# standard use
A <- matrix(1:4, 2)
B <- matrix(6:11, 3)
print(A)
print(B)
cbind_fill_matrix(A, B)

# help with unstack()
row_keep <- sample(1:nrow(InsectSprays), size = 36)
InsectSprays2 <- InsectSprays[row_keep, ]
unstacked <- unstack(InsectSprays2)
cbind_fill_matrix(unstacked)

# using rownames for binding
rownames(A) <- c("one", "two")
rownames(B) <- c("three","two","one")
print(A)
print(B)
cbind_fill_matrix(A, B)

# data.frame input
C <- data.frame("V1" = c(12,13,14,15), row.names = c("one","two","three","four"))
```

```
print(C)
cbind_fill_matrix(A, B, C)

# atomic vector input
A <- matrix(1:4, 2)
B <- matrix(6:11, 3)
C <- c(12,13,14,15)
cbind_fill_matrix(A, B, C)

# same as plyr::rbind.fill.matrix, cbind_fill_matrix doesn't like some input
# with dimnames and others without...
rownames(A) <- c("one", "two")
print(A)
print(B)
cbind_fill_matrix(A, B)
```

`codes` *Integer Codes of Factor Levels*

Description

`codes` returns the integer codes for each factor level from a factor.

Usage

```
codes(fct)
```

Arguments

`fct`	factor.

Value

integer vector with length = `length(levels(fct))`, elements = integer codes of `fct` and names = `levels(fct)`.

Examples

```
codes(state.region)
codes(iris$"Species")
```

d2a *Data-Frame to (3D+) Array or Matrix*

Description

d2a converts a data.frame to a (3D+) array or matrix. This function assumes the data.frame contains 2+ variable dimensions, which will correspond to the returned arrays/matrix dimensions. One or multiple variables can contain the elements of the returned array (only one variable can contain the elements for returning a matrix). In the case of multiple variables, they will be binded as the last dimension in the returned array with dimnames equal to the variable names.

Usage

```
d2a(d, dim.nm = names(d)[-ncol(d)], rtn.dim.lab = "el_nm", check = TRUE)
```

Arguments

d	data.frame with at least 3 columns, where 2+ columns are variable dimensions and 1+ columns contain the to-be returned array/matrix elements.
dim.nm	character vector of 2+ length specifying the colnames in d that contain the variable dimensions. These do not need to be factors or character vectors. Note, all columns in d other than dim.nm are assumed to be element columns.
rtn.dim.lab	character vector of length 1 specifying the dimlabel to use for the last dimension in the returned array when there are multiple element columns in d. Note, that NA will be converted to "NA" and NULL will return an error. If you don't want any dimlabel to show, "" is probably the best option. If there is only one element column in d, this argument is ignored by d2a.
check	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether d is a data.frame. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Details

d2a is a wrapper for reshape::cast with the addition of reordering the dimnames by position, which sorts the dimnames by the position they first appear in the variable dimensions of the data.frame (reshape::cast sorts all the dimnames alphabetically).

Value

(3D+) array or matrix formed from the dimensions d[dim.nm] with dimlabels = dim.nm (and rtn.dim.lab if there are multiple element columns). The dimnames are the unique elements d[dim.nm] and are ordered by position (rather than alphabetically), which allow for conversions back to the original array after a call to a2d() or matrix after a call to m2d().

Examples

```
# 3D array
print(HairEyeColor)
d <- reshape::melt.array(HairEyeColor)
a <- reshape::cast(d, Hair ~ Eye ~ Sex)
identical(a, unclass(HairEyeColor)) # not the same as HairEyeColor
d <- a2d(HairEyeColor)
a <- d2a(d, dim.nm = c("Hair","Eye","Sex"))
identical(a, unclass(HairEyeColor)) # yes the same as HairEyeColor

# matrix
attitude_mat <- d2m(attitude)
d <- m2d(attitude_mat, col = 0)
m <- d2a(d)
identical(m, attitude_mat) # yes the same as attitude_mat

# correlation data.frame example for p-values using psych::corr.test(attitude[1:3])
# corr_test <- psych::corr.test(attitude)
# a <- lm2a(corr_test[c("r","se","t","p")])
r <- matrix(c(1.0000000, 0.8254176, 0.4261169, 0.8254176, 1.0000000, 0.5582882,
   0.4261169, 0.5582882, 1.0000000), nrow = 3, ncol = 3, byrow = FALSE)
se <- matrix(c(0.0000000, 0.1066848, 0.1709662, 0.1066848, 0.0000000, 0.1567886,
   0.1709662, 0.1567886, 0.0000000), nrow = 3, ncol = 3, byrow = FALSE)
t <- matrix(c(Inf, 7.736978, 2.492404, 7.736978, Inf, 3.560771,
   2.492404, 3.560771, Inf), nrow = 3, ncol = 3, byrow = FALSE)
p <- matrix(c(0.000000e+00, 1.987682e-08, 1.887702e-02, 5.963047e-08, 0.000000e+00,
  1.345519e-03, 0.018877022, 0.002691039, 0.000000000), nrow = 3, ncol = 3, byrow = FALSE)
a <- abind::abind(r, se, t, p, along = 3L)
dimnames(a) <- list(names(attitude[1:3]), names(attitude[1:3]), c("r","se","t","p"))
d <- a2d(a = a, col = 3)
a2 <- d2a(d = d, dim.nm = c("X1","X2"))
all.equal(a, a2) # dimlabels differ
dimnames(a2) <- unname(dimnames(a2))
all.equal(a, a2) # now it is true

# correlation data.frame example for confidence intervals using psych::corr.test(attitude[1:3])
# corr_test <- psych::corr.test(attitude[1:3])
# d <- corr_test[["ci"]][c("r","p","lower","upper")]
# cbind(d, after = 0L) <- reshape::colsplit(row.names(d), split = "-", names = c("X1","X2"))
# tmp <- d[c("X2","X1","r","p","lower","upper")]
# d2 <- plyr::rename(tmp, c("X1" = "X2", "X2" = "X1"))
# short_nm <- unique(c(fct2v(d[["X1"]]), fct2v(d[["X2"]])))
# d3 <- data.frame("X1" = short_nm, "X2" = short_nm,
#    "r" = NA_real_, "p" = NA_real_, "lower" = NA_real_, "upper" = NA_real_)
# d_all <- ld2d(ld = list(d, d2, d3), rtn.listnames.nm = NULL, rtn.rownames.nm = NULL)
d_all <- data.frame(
   "X1" = c("ratng","ratng","cmpln","cmpln","prvlg","prvlg","ratng","cmpln","prvlg"),
   "X2" = c("cmpln","prvlg","prvlg","ratng","ratng","cmpln","ratng","cmpln","prvlg"),
   "r" = c(0.8254176, 0.4261169, 0.5582882, 0.8254176, 0.4261169, 0.5582882, NA, NA, NA),
   "p" = c(1.987682e-08, 1.887702e-02, 1.345519e-03, 1.987682e-08,
      1.887702e-02, 1.345519e-03, NA, NA, NA),
   "lower" = c(0.66201277, 0.07778967, 0.24787510, 0.66201277, 0.07778967,
```

```
      0.24787510, NA, NA, NA),
   "upper" = c(0.9139139, 0.6817292, 0.7647418, 0.9139139, 0.6817292,
      0.7647418, NA, NA, NA)
)
tmp <- d2a(d = d_all, dim.nm = c("X1","X2"), rtn.dim.lab = "stat")
short_nm <- c("ratng","cmpln","prvlg")
dim_names <- list(short_nm, short_nm, c("r","p","lower","upper"))
a <- do.call(what = `[`, args = c(list(tmp), dim_names))
print(a)
```

---

d2d *Data-Frame to Data-Frame (e.g., factors to character vectors)*

Description

d2d converts a data.frame to a modified version of the data.frame. It is used to convert factors, character vectors, and logical vectors to different classes/types (e.g., factors to character vectors).

Usage

```
d2d(
  d,
  fct = "chr",
  chr = "chr",
  lgl = "int",
  order.lvl = "alphanum",
  decreasing = FALSE,
  na.lvl = FALSE,
  check = TRUE
)
```

Arguments

d	data.frame.
fct	character vector of length 1 specifying what factors should be converted to. There are three options: 1) "chr" for converting to character vectors (i.e., factor labels), 2) "int" for converting to integer vectors (i.e., factor codes), or 3) "fct" for keeping the factor as is without any changes.
chr	character vector of length 1 specifying what character vectors should be converted to. There are three options: 1) "fct" for converting to factors (i.e., elements will be factor labels), 2) "int" for converting to integer vectors (i.e., factor codes after first converting to a factor), or 3) "chr" for keeping the character vectors as is without any changes.
lgl	character vector of length 1 specifying what logical vectors should be converted to. There are four options: 1) "fct" for converting to factors (i.e., "TRUE" and "FALSE" will be factor labels), 2) "chr" for converting to character vectors (i.e.,

elements will be "TRUE" and "FALSE"), 3) "int" for converting to integer vectors (i.e., TRUE = 1; FALSE = 0), and 4) "lgl" for keeping the logical vectors as is without any changes.

`order.lvl` character vector of length 1 specifying how you want to order the levels of the factor. The options are "alphanum", which sorts the levels alphanumerically (with NA last); "position", which sorts the levels by the position the level first appears; "frequency", which sorts the levels by their frequency. If any frequencies are tied, then the ties are sorted alphanumerically (with NA last).

`decreasing` logical vector of length 1 specifying whether the ordering of the levels should be decreasing (TRUE) rather than increasing (FALSE).

`na.lvl` logical vector of length 1 specifying if NA should be considered a level.

`check` logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `d` is a data.frame. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Details

`d2d` internally uses the `fct2v` and `v2fct` functions. See them or more details about how column conversions work.

Value

data.frame with the same dim and dimnames as `d`, but with potentially altered columns which were factors, character vectors, and/or integer vectors.

Examples

```
dat <- data.frame(
"lgl_1" = c(TRUE, FALSE, NA),
"lgl_2" = c(FALSE, TRUE, NA),
"int_1" = c(1L, NA, 2L),
"int_2" = c(2L, NA, 1L),
"dbl_1" = c(1.1, NA, 2.2),
"dbl_2" = c(2.2, NA, 1.1),
"chr_1" = c(NA, "a","b"),
"chr_2" = c(NA, "b","a"),
"fct_1" = factor(c(NA, "one","two")),
"fct_2" = factor(c(NA, "two","one"))
)
str(dat)
x <- d2d(dat); str(x) # default
x <- d2d(dat, fct = "fct", chr = "fct", lgl = "fct"); str(x) # all to factors
x <- d2d(dat, fct = "int", chr = "int"); str(x) # all to integers
```

d2ld *Data-Frame to List of Data-Frames*

Description

d2ld converts a data.frame to a list of data.frames. This is a simple call to split.data.frame splitting the data.frame up by groups.

Usage

```
d2ld(
  d,
  by,
  keep.by = TRUE,
  drop = FALSE,
  sep = ".",
  lex.order = FALSE,
  check = TRUE
)
```

Arguments

d	data.frame.
by	character vector of colnames specifying the groups to split the data.frame up by. Can be multiple colnames, which implicitly calls interaction.
keep.by	logical vector of length 1 specifying whether the by columns should be kept in the list of data.frames (TRUE) or removed (FALSE).
drop	logical vector of length 1 specifying whether unused groups from the by columns should be dropped (TRUE) or kept (FALSE). This only applies when there are multiple by columns. drop = FALSE can then result in some data.frames with nrow = 0. See interaction for details.
sep	character vector of length 1 specifying the string used to separate the group names. Only applicable with multiple by columns. See interaction for details.
lex.order	logical vector of length 1 specifying the order of the data.frames in the list based on the groups in the by columns. This only applies when there are multiple by columns. See interaction for details.
check	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether d is a data.frame and by are colnames of d. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

list of data.frames split by the groups specified in the by columns. The list names are the group names (with sep if there are multiple by columns).

Examples

```
# one grouping variable
d2ld(d = mtcars, by = "vs")
d2ld(d = mtcars, by = "gear")

# two grouping variables
d2ld(d = mtcars, by = c("vs","gear"))
d2ld(d = mtcars, by = c("vs","gear"), lex.order = TRUE)

# keep.by argument
d2ld(d = mtcars, by = "vs", keep.by = FALSE)
d2ld(d = mtcars, by = "gear", keep.by = FALSE)
d2ld(d = mtcars, by = c("vs","gear"), keep.by = FALSE)
```

`d2lv` *Data-Frame to List of (Atomic) Vectors*

Description

`d2lv` converts a data.frame to a list of (atomic) vectors. This function is really only worthwhile when `along` = 1 since when `along` = 2, the function is essentially `as.list.data.frame(d)`.

Usage

```
d2lv(d, along, check = TRUE)
```

Arguments

`d`	data.frame.
`along`	numeric vector of length 1 specifying which dimension to slice the data.frame along. If 1, then the data.frame is sliced by rows. If 2, then the data.frame is sliced by columns.
`check`	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `d` is a data.frame. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

list of (atomic) vectors. If `along` = 1, then the names are the rownames of `d` and the vectors are rows from `d`. If `along` = 2, then the names are the colnames of `d` and the vector are columns from `d`. Note, the vectors always have the same length as `nrow(d)`.

Examples

```
d2lv(mtcars, along = 1)
d2lv(mtcars, along = 2)
d2lv(CO2, along = 1) # all vectors converted to typeof character
d2lv(CO2, along = 2) # each column stays its own typeof (or class for factors)
# check = FALSE
try_expr(d2lv(mtcars, along = 3, check = FALSE)) # less informative error message
try_expr(d2lv(mtcars, along = 3, check = TRUE)) # more informative error message
```

`d2m` *Data-Frame to Matrix*

Description

`d2m` converts a data.frame to a matrix. The user can specify how to convert factors, character vectors, and integer vectors in the data.frame through the internal use of the `d2d` function. After the call to `d2d`, `d2m` simply calls `as.matrix.data.frame(rownames.force = TRUE)`, which will return a matrix of the most complex typeof of any column in the data.frame (most complex to least complex: character, double, integer, logical). Therefore, if any factors or character vectors are left in the data.frame, it will return a character matrix. On the other side of things, if all columns in the data.frame are logical, then it will return a logical matrix. However, if every column in the data.frame is logical except for one factor or character vector, then it will return a character matrix. (If you have a data.frame where 2 columns are the matrix dimnames and one column is the matrix elements, then use `d2a()`).

Usage

```
d2m(
  d,
  fct = "chr",
  chr = "chr",
  lgl = "int",
  order.lvl = "alphanum",
  decreasing = FALSE,
  na.lvl = FALSE,
  check = TRUE
)
```

Arguments

`d`	data.frame.
`fct`	character vector of length 1 specifying what factors should be converted to. There are three options: 1) "chr" for converting to character vectors (i.e., factor labels), 2) "int" for converting to integer vectors (i.e., factor codes), or 3) "fct" for keeping the factor as is without any changes.

`chr` character vector of length 1 specifying what character vectors should be converted to. There are three options: 1) "fct" for converting to factors (i.e., elements will be factor labels), 2) "int" for converting to integer vectors (i.e., factor codes after first converting to a factor), or 3) "chr" for keeping the character vectors as is without any changes.

`lgl` character vector of length 1 specifying what logical vectors should be converted to. There are four options: 1) "fct" for converting to factors (i.e., "TRUE" and "FALSE" will be factor labels), 2) "chr" for converting to character vectors (i.e., elements will be "TRUE" and "FALSE"), 3) "int" for converting to integer vectors (i.e., TRUE = 1; FALSE = 0), and 4) "lgl" for keeping the logical vectors as is without any changes.

`order.lvl` character vector of length 1 specifying how you want to order the levels of the factor. The options are "alphanum", which sorts the levels alphanumerically (with NA last); "position", which sorts the levels by the position the level first appears; "frequency", which sorts the levels by their frequency. If any frequencies are tied, then the ties are sorted alphanumerically (with NA last).

`decreasing` logical vector of length 1 specifying whether the ordering of the levels should be decreasing (TRUE) rather than increasing (FALSE).

`na.lvl` logical vector of length 1 specifying if NA should be considered a level.

`check` logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `d` is a data.frame. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

matrix with the same dim and dimnames as `d`. After applying the factor, character vector, and/or integer vector conversions through `d2d`, the matrix will have typeof = most complex typeof of any column in the modified data.frame.

Examples

```
x <- d2m(mtcars); str(x)
dat <- as.data.frame(CO2)
x <- d2m(dat); str(x)
x <- d2m(dat, fct = "int"); str(x)
```

`d2v` *Data-Frame to (Atomic) Vector*

Description

`d2v` converts a data.frame to a matrix. The user can specify how to convert factors, character vectors, and integer vectors in the data.frame through the internal use of the `d2d` function. After the call to `d2d`, the data.frame is simplied to an atomic vector, which will return a vector of the most complex typeof of any column in the data.frame (most complex to least complex: character, double, integer,

logical). Therefore, if any factors or character vectors are left in the data.frame, it will return a character vector. On the other side of things, if all columns in the data.frame are logical, then it will return a logical vector. However, if every column in the data.frame is logical except for one factor or character vector, then it will return a character vector.

Usage

```
d2v(
  d,
  along = 2,
  use.dimnames = TRUE,
  sep = "_",
  fct = "chr",
  chr = "chr",
  lgl = "int",
  order.lvl = "alphanum",
  decreasing = FALSE,
  na.lvl = FALSE,
  check = TRUE
)
```

Arguments

`d`	data.frame.
`along`	numeric vector of length one that is equal to either 1 or 2. 1 means that `d` is split along rows (i.e., dimension 1) and then concatenated. 2 means that `d` is split along columns (i.e., dimension 2) and then concatenated.
`use.dimnames`	logical vector of length 1 that specifies whether the dimnames of `d` should be used to create the names for the returned vector. If FALSE, the returned vector will have NULL names. If TRUE, see details of `m2v`.
`sep`	character vector of length 1 specifying the string that will separate the rownames and colnames in the naming scheme of the returned vector. Note, `sep` is not used if `use.dimnames` = FALSE.
`fct`	character vector of length 1 specifying what factors should be converted to. There are three options: 1) "chr" for converting to character vectors (i.e., factor labels), 2) "int" for converting to integer vectors (i.e., factor codes), or 3) "fct" for keeping the factor as is without any changes.
`chr`	character vector of length 1 specifying what character vectors should be converted to. There are three options: 1) "fct" for converting to factors (i.e., elements will be factor labels), 2) "int" for converting to integer vectors (i.e., factor codes after first converting to a factor), or 3) "chr" for keeping the character vectors as is without any changes.
`lgl`	character vector of length 1 specifying what logical vectors should be converted to. There are four options: 1) "fct" for converting to factors (i.e., "TRUE" and "FALSE" will be factor labels), 2) "chr" for converting to character vectors (i.e., elements will be "TRUE" and "FALSE"), 3) "int" for converting to integer vectors (i.e., TRUE = 1; FALSE = 0), and 4) "lgl" for keeping the logical vectors as is without any changes.

order.lvl	character vector of length 1 specifying how you want to order the levels of the factor. The options are "alphanum", which sorts the levels alphanumerically (with NA last); "position", which sorts the levels by the position the level first appears; "frequency", which sorts the levels by their frequency. If any frequencies are tied, then the ties are sorted alphanumerically (with NA last).
decreasing	logical vector of length 1 specifying whether the ordering of the levels should be decreasing (TRUE) rather than increasing (FALSE).
na.lvl	logical vector of length 1 specifying if NA should be considered a level.
check	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether d is a data.frame. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

(atomic) vector with elements from d. If d had one row, then the names of the return object are names(d). If d has one column, then the names of the return object are row.names(d).

Examples

```
# general data.frame
d2v(mtcars) # default
d2v(d = mtcars, along = 1) # concatenate along rows
d2v(d = mtcars, sep = ".") # change the sep of the rownames(d) and colnames(d)
d2v(d = mtcars, use.dimnames = FALSE) # return object has no names
# one row/column data.frame
one_row <- mtcars[1,, drop = FALSE]
d2v(one_row)
one_col <- mtcars[, 1, drop = FALSE]
d2v(one_col)
one_all <- mtcars[1,1, drop = FALSE]
d2v(one_all)
d2v(one_all, use.dimnames = FALSE)
```

---

dimlabels *Dimension labels (i.e., names of dimnames)*

Description

dimlabels returns the the dimension labels (i.e., names of dimnames) of an object. This is most useful for arrays, which can have anywhere from 1 to 1000+ dimensions.

Usage

```
dimlabels(x)
```

Arguments

x	object that has dimensions (e.g., array).

Details

dimlabels is a very simple function that is simply names(dimnames(x)).

Value

character vector of length = ndim(x) specifying the dimension labels (i.e., names of dimnames) of x. If x does not have any dimensions, or has dimensions but no dimension labels, then NULL is returned.

Examples

```
dimlabels(state.region)
dimlabels(attitude)
dimlabels(HairEyeColor)
```

---

dimlabels<- *Add Elements to Vectors*

Description

`dimlabels<-` adds elements to vectors as a side effect. The purpose of the function is to replace names(dimnames(x)) with a single function call.

Usage

```
dimlabels(x) <- value
```

Arguments

x	array or any object with dimnames. The object may or may not already have dimlabels.
value	character vector of dimlabels to be added to x. If `dimlabels<-` is used on its own, then the length of value must be the same as ndim. If `dimlabels<-` is used in confunction with the subsetting function `[[<-` or `[<-`, then the length of values should be equal to the length of dimlabels after from the subsetting. This is the same way `names<-` works.

Value

Like other similar functions (e.g., `names<-` and `[<-`), it does not appear to have a return object. However, it technically does as a side effect. The argument x will have been changed such that value has been added as dimlabels. If a traditional return object is desired, and no side effects, then it can be called like a traditional function: obj2 <- ‘dimlabels<-‘(x = obj, value = dimlab).

Examples

```
a <- array(c(letters, NA), dim = c(3,3,3),
   dimnames = replicate(3, expr = 1:3, simplify = FALSE))
dimlabels(a) <- c("first","second","third")
dimlabels(a)[[2]] <- c("2nd")
dimlabels(a)[c(1,3)] <- c("1st","3rd")
print(a)
```

---

e2l *Environment to List*

Description

e2l converts an environment to a list. The function assumes you don't want *all* objects in an environment and uses pick to determine which objects you want included. If you want all objects in an environment, then use grab(x = objects(envir, all.names = TRUE), envir).

Usage

```
e2l(
  e = sys.frame(),
  val,
  pat = FALSE,
  not = FALSE,
  fixed = FALSE,
  sorted = FALSE,
  check = TRUE
)
```

Arguments

e	environment to pull the objects from. Default is the global environment.
val	character vector specifying which objects from e will be extracted. If pat = FALSE (default), then val can have length > 1, and exact matching will be done via is.element (essentially match). If pat = TRUE, then val has to be a character vector of length 1 and partial matching will be done via grepl with the option of regular expressions if fixed = FALSE (default).
pat	logical vector of length 1 specifying whether val should refer to exact matching (FALSE) via is.element (essentially match) or partial matching (TRUE) and/or use of regular expressions via grepl. See details for a brief description of some common symbols and help(regex) for more.
not	logical vector of length 1 specifying whether val indicates values that should be retained (FALSE) or removed (TRUE).
fixed	logical vector of length 1 specifying whether val refers to values as is (TRUE) or a regular expression (FALSE). Only used if pat = TRUE.

sorted	logical vector of length 1 specifying whether the objects should be sorted alphanumerically. If FALSE, the objects are usually in the order they were initially created, but not always (see help(objects)).
check	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether e is an environment. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

list with object contents from environment e with names as the object names.

Examples

```
model_1 <- lm(v2frm(names(attitude)), data = attitude)
model_2 <- lm(v2frm(names(mtcars)), data = mtcars)
model_3 <- lm(v2frm(names(airquality)), data = airquality)
e2l(val = "model_", pat = TRUE)
```

---

fct2v *Factor to (Atomic) Vector*

Description

fct2v converts a factor to an (atomic) vector. It allows the user to specify whether they want the factor to always return a character vector (simplify = TRUE), simplified if possible (simplify = FALSE), or just return the integer codes (codes = TRUE).

Usage

```
fct2v(fct, simplify = TRUE, codes = FALSE, check = TRUE)
```

Arguments

fct	factor.
simplify	logical vector of length 1 specifying whether R should attempt to simplify fct to typeof simplier than character (e.g., logical, integer, double). If FALSE, a character vector is always returned.
codes	logical vector of length 1 specifying whether the integer codes of fct should be returned. If codes = TRUE, then simplify is ignored.
check	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether fct is a factor. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Details

When `simplify = TRUE`, `fct2v` uses `type.convert` to try to simplify the factor. Note, missing values are assumed to be "NA" and decimals are assumed to be "."; however, "L" after a number is not interpreted as an integer specifier.

Value

(atomic) vector of the same length as `fct`. If `codes` = TRUE, then the returned vector is typeof integer containing the underlying factor codes. If `codes` = FALSE and `simplify` = FALSE, then the returned vector is typeof character containing the factor levels. If `codes` = FALSE, and `simplify` = TRUE, then the returned vector is the simpliest typeof possible without having to coerce any elements to NA. For example, if `fct` contains all integer numerals (e.g., "1", "2", "3", etc), then it will be converted to an integer vector. See examples.

Examples

```
fct2v(state.region)
fct2v(fct = factor(c("7.00001","8.54321","9.99999"))) # double
fct2v(fct = factor(c("7","8","9")), simplify = FALSE) # character
fct2v(fct = factor(c("7","8","9")), simplify = TRUE) # integer
fct2v(fct = factor(c("7","8","9")), codes = TRUE) # integer codes
fct2v(fct = factor(c("7L","8L","9L")),
   simplify = TRUE) # does not understand "L" for integers
```

---

grab — `grab` *extracts the contents of objects in an environment based on their object names as a character vector. The object contents are stored to a list where the names are the object names.*

Description

`grab` extracts the contents of objects in an environment based on their object names as a character vector. The object contents are stored to a list where the names are the object names.

Usage

```
grab(x, envir = sys.frame())
```

Arguments

`x`	character vector providing the exact names of objects in the environment `envir`.
`envir`	environment to pull the objects from. Default is the global environment.

Value

list of objects with names `x`.

Examples

```
grab(x = c("attitude","mtcars","airquality"))
grab(x = c("mean.default","mean.Date","mean.difftime"))
```

`inbtw` *Elements Inbetween Values Within a (Atomic) Vector*

Description

`inbtw` extracts all elements inbetween (by position) two specific elements of a (atomic) vector. This can be useful when working with rownames and colnames since `seq` does not work with names. Primary for character vectors but can be used with other typeof.

Usage

```
inbtw(x, from, to, left = TRUE, right = TRUE)
```

Arguments

`x`	atomic vector.
`from`	vector of length 1 specifying the element to start with on the left.
`to`	vector of length 1 specifying the element to end with on the right.
`left`	logical vector of length 1 specifying whether the leftmost element, `from`, should be included in the return object.
`right`	logical vector of length 1 specifying whether the rightmost element, `to`, should be included in the return object.

Details

An error is returned if either `from` or `to` don't appear in `x` or appear more than once in `x`.

Value

vector of the same type as `x` that only includes elements in `x` inbetween (by position) `from` and `to`, which may or may not include `from` and `to` themselves, depending on `left` and `right`, respectively.

Examples

```
# character vector
row_names <- inbtw(x = row.names(LifeCycleSavings), from = "China", to = "Peru")
LifeCycleSavings[row_names, ] # default use
row_names <- inbtw(x = row.names(LifeCycleSavings), from = "China", to = "Peru",
   right = FALSE, left = FALSE)
LifeCycleSavings[row_names, ] # use with right and left arguments FALSE
try_expr(inbtw(x = row.names(LifeCycleSavings), from = "china",
   to = "peru")) # error due to `from` and `to` not appearing in `x`
try_expr(inbtw(x = rep.int(x = row.names(LifeCycleSavings), times = 2), from = "China",
```

```
   to = "Peru")) # error due to `from` and `to` appearing more than once in `x`
# numeric vector
vec <- sample(x = 150:199, size = 50)
inbtw(x = vec, from = 150, to = 199)
# list vector (error)
lst <- list(FALSE, 3L, 9.87, "abc", factor("lvl"))
try_expr(inbtw(x = lst, from = 3L, to = "abc")) # error because `lst` is a
   # list vector and not an atomic vector
```

`is.avector` *Test for an Atomic Vector*

Description

`is.avector` returns whether an object is an atomic vector with typeof character, logical, integer, or double.

Usage

```
is.avector(x, attr.ok = TRUE, fct.ok = TRUE)
```

Arguments

`x`	object whose structure is desired to be tested.
`attr.ok`	logical vector with length 1 specifying whether non-core attributes are allowed in `x`. Core attributes are 1) "names", 2) "dim", 3) "dimnames", 4) "levels", and 5) "class". Therefore, `attr.ok` refers to attributes *other* than these 5.
`fct.ok`	logical vector with length 1 specifying whether factors are allowed.

Details

`is.avector` is simply a logical "and" of `is.atomic` and `is.vector`.

Value

logical vector with length 1 specifying whether `x` is an atomic vector. If `attr.ok` is TRUE then non-core attributes are allowed (e.g., "value.labels"). If `fct.ok` is TRUE then factors are allowed.

Examples

```
# normal use
is.avector(x = c(1,2,3))
is.avector(x = c("one" = 1, "two" = 2, "three" = 3)) # names are always okay
is.avector(x = array(c(1,2,3))) # returns false for arrays
is.avector(x = list(1,2,3)) # returns false for lists

# non-core attributes
x <- structure(.Data = c(1,2,3), "names" = c("one","two","three"),
```

```
   "value.labels" = c("woman","man","non-binary"))
attributes(x)
is.avector(x)
is.avector(x, attr.ok = FALSE)

# factors
x <- factor(c(1,2,3), labels = c("one","two","three"))
is.avector(x)
is.avector(x, fct.ok = FALSE)
```

`is.cnumeric` *Test for Character Numbers*

Description

`is.cnumeric` returns whether an object is a character vector with all number strings.

Usage

```
is.cnumeric(x, warn = FALSE)
```

Arguments

`x`	object whose structure is desired to be tested.
`warn`	logical vector with length 1 specifying whether warnings should be printed due to coercing a character vector that is not all number strings (i.e., one reason the return object could be ‘FALSE‘).

Details

`is.cnumeric` is useful for ensuring that converting a character vector to a numeric vector is safe (i.e., won't introduce NAs).

Value

logical vector with length 1 specifying whether ‘`x`‘ is a character vector with all number strings.

Examples

```
is.cnumeric(x = c("1","2","3")) # returns TRUE
is.cnumeric(x = c("1","number","3")) # returns FALSE
is.cnumeric(x = c("1","number","3"), warn = TRUE) # includes the warning
is.cnumeric(x = c(1,2,3)) # returns false because not a character vector
```

is.colnames *Test for ‘colnames‘*

Description

`is.colnames` returns whether elements of a character vector are colnames of an object.

Usage

```
is.colnames(nm, x)
```

Arguments

`nm`	character vector.
`x`	object whose colnames are desired to be tested.

Details

If the object does not have any colnames, then the function will return ‘FALSE‘ for each element of the character vector.

Value

TRUE for every element of ‘nm‘ that is a colname of x and FALSE otherwise. The structure is a logical vector with length = length(‘nm‘) and names = ‘nm‘. See details for special cases.

Examples

```
data("mtcars")
is.colnames(x = as.matrix(mtcars), nm = c("MPG","mpg"))
```

is.Date *Test for a Date object*

Description

`is.Date` returns whether an object is a Date object (aka has class = "Date").

Usage

```
is.Date(x)
```

Arguments

`x`	an object.

Value

TRUE is x has class "Date" and FALSE if x does not have class "Date".

Examples

```
date <- as.Date("2021-05-24", format = "%Y-%m-%d") # as.Date.character
is.Date(date)
class(date) <- append(class(date), "extra_class")
is.Date(date) # classes other than Date are allowed
is.Date(list(date)) # returns FALSE
```

---

is.dummy *Test for a Dummy Variable*

Description

is.dummy returns whether a numeric vector is a dummy variable, meaning all elements one of two observed values (or missing values). Depending on the argument any.values, the two observed values are required to be 0 and 1 or any values.

Usage

```
is.dummy(x, any.values = FALSE)
```

Arguments

x	atomic vector.
any.values	logical vector of length 1 specifying whether the two observed values need to be 0 or 1 (FALSE) or can be any values (TRUE).

Value

TRUE if ‘x‘ is a dummy variable; FALSE otherwise.

Examples

```
# any.values = FALSE (default)
is.dummy(mtcars$"am") # TRUE
is.dummy(c(mtcars$"am", NA, NaN)) # works with missing values
is.dummy(c(as.integer(mtcars$"am"), NA, NaN)) # works with typeof integer
x <- ifelse(mtcars$"am" == 1, yes = 2, no = 1)
is.dummy(x) # FALSE

# any.values = TRUE
is.dummy(x, any.values = TRUE) # TRUE
is.dummy(c(x, NA), any.values = TRUE) # work with missing values
is.dummy(c(as.character(x), NA), any.values = TRUE) # work with typeof character
is.dummy(mtcars$"gear") # FALSE for nominal variables with more than 2 levels
```

is.empty *Test for Empty Characters*

Description

`is.empty` returns whether elements of a character vector are empty (i.e., "").

Usage

```
is.empty(x, trim = FALSE)
```

Arguments

`x`	character vector.
`trim`	logical vector with a single element specifying whether white spaces should be trimmed from the character vector. See `trimws`.

Value

TRUE for every element of ‘x‘ that is empty (i.e., "") and FALSE otherwise. The structure is a logical vector with length = length(‘x‘) and names = names(‘x‘).

Examples

```
v <- c("1", " ", "")
is.empty(v)
```

is.names *Test for ‘names‘*

Description

`is.names` returns whether elements of a character vector are names of an object.

Usage

```
is.names(nm, x)
```

Arguments

`nm`	character vector.
`x`	object whose names are desired to be tested.

Details

If the object does not have any names, then the function will return ‘FALSE‘ for each element ‘nm‘.

Value

TRUE for every element of ‘nm‘ that is a name of ‘x‘ and FALSE otherwise. The structure is a logical vector with length = length(‘nm‘) and names = ‘nm‘. See details for special cases.

Examples

```
v <- setNames(object = letters, nm = LETTERS)
is.names(x = v, nm = c("A","a"))
data("mtcars")
is.names(x = mtcars, nm = c("MPG","mpg"))
```

---

is.POSIXct *Test for a POSIXct object*

Description

`is.POSIXct` returns whether an object is a POSIXct object (aka has class = "POSIXct").

Usage

```
is.POSIXct(x)
```

Arguments

x	an object.

Value

TRUE is x has class "POSIXct" and FALSE if x does not have class "POSIXct".

Examples

```
tmp <- as.POSIXlt("2021-05-24 21:49:11", tz = "America/New_York",
   format = "%Y-%m-%d %H:%M:%OS") # as.POSIXlt.character
time <- as.POSIXct(tmp) # as.POSIXct.POSIXlt
is.POSIXct(time)
class(time) <- append(class(time), "extra_class")
is.POSIXct(time) # classes other than POSIXct are allowed
is.POSIXct(list(time)) # returns FALSE
```

is.POSIXlt — *Test for a POSIXlt object*

Description

`is.POSIXlt` returns whether an object is a POSIXlt object (aka has class = "POSIXlt").

Usage

```
is.POSIXlt(x)
```

Arguments

`x`	an object.

Value

TRUE is `x` has class "POSIXlt" and FALSE if `x` does not have class "POSIXlt".

Examples

```
time <- as.POSIXlt("2021-05-24 21:49:11", tz = "America/New_York",
   format = "%Y-%m-%d %H:%M:%OS") # as.POSIXlt.character
is.POSIXlt(time)
class(time) <- append(class(time), "extra_class")
is.POSIXlt(time) # classes other than POSIXlt are allowed
is.POSIXlt(list(time)) # returns FALSE
```

is.row.names — *Test for ‘row.names‘*

Description

`is.row.names` returns whether elements of a character vector are row.names of an object.

Usage

```
is.row.names(nm, x)
```

Arguments

`nm`	character vector.
`x`	object whose row.names are desired to be tested.

Details

If the object does not have any row.names, then the function will return ‘FALSE‘ for each element of the character vector. As a reminder, `row.names` does not respond to a manually added "row.names" attribute (e.g., to a vector). If this is tried, then `is.row.names` will return ‘FALSE‘ for each element ‘nm‘.

Value

TRUE for every element of ‘nm‘ that is a row.name of x and FALSE otherwise. The structure is a logical vector with length = length(‘nm‘) and names = ‘nm‘. See details for special cases.

Examples

```
data("mtcars")
is.row.names(x = mtcars, nm = c("Mazda RX4","mazda RX4"))
```

`is.rownames` *Test for ‘rownames‘*

Description

`is.rownames` returns whether elements of a character vector are rownames of an object.

Usage

```
is.rownames(nm, x)
```

Arguments

`nm`	character vector.
`x`	object whose rownames are desired to be tested.

Details

If the object does not have any rownames, then the function will return ‘FALSE‘ for each element of the character vector.

Value

TRUE for every element of ‘nm‘ that is a rowname of x and FALSE otherwise. The structure is a logical vector with length = length(‘nm‘) and names = ‘nm‘. See details for special cases.

Examples

```
data("mtcars")
is.rownames(x = as.matrix(mtcars), nm = c("Mazda RX4","mazda RX4"))
```

is.whole — *Test for Whole Numbers*

Description

`is.whole` returns whether elements of a numeric vector are whole numbers.

Usage

```
is.whole(x, tol = .Machine[["double.eps"]])
```

Arguments

`x`	numeric vector.
`tol`	tolerance allowed for double floating point numbers. This is always a positive number. The default is based on the numerical characteristics of the machine that R is running on. See `.Machine`.

Value

TRUE for every element of ‘x‘ that is a whole number and FALSE otherwise. The structure is a logical vector with length = length(‘x‘) and names = names(‘x‘).

Examples

```
v <- c(1.0, 1L, 1.1)
is.whole(v)
```

Join — *Join (or Merge) a List of Data-frames*

Description

`Join` merges a list of data.frames into a single data.frame. It is a looped version of `plyr::join` that allows you to merge more than 2 data.frames in the same function call. It is different from `plyr::join_all` because it allows you to join by the row.names.

Usage

```
Join(
  data.list,
  by,
  type = "full",
  match = "all",
  rownamesAsColumn = FALSE,
  rtn.rownames.nm = "row_names"
)
```

Arguments

`data.list` list of data.frames of data.

`by` character vector specifying what colnames to merge `data.list` by. It can include "0" which specifies the rownames of `data.list`. If you are merging by rownames, then you can only merge by rownames and not other columns as well. This is because rownames, by definition, have all unique values. Note, it is assumed that no data.frame in `data.list` has a colname of "0", otherwise unexpected results are possible. If `by` is NULL, then all common columns will be used for merging. This is not recommended as it can result in `Join` merging different data.frames in `data.list` by different columns.

`type` character vector of length 1 specifying the type of merge. Options are the following: 1. "full" = all rows from any of the data.frames in `data.list`, 2. "left" = only rows from the first data.frame in `data.list`: `data.list[[1L]]`), 3. "right" = only rows from the last data.frame in `data.list`: `data.list[[length(data.list)]]`, 4. "inner" = only rows present in each and every of the data.frames in `data.list`. See `join`.

`match` character vector of length 1 specifying whether merged elements should be repeated in each row of the return object when duplicate values exist on the `by` columns. If "all", the merged elements will only appear in every row of the return object with repeated values. If "first", only the merged elements will only appear in the first row of the return object with subsequent rows containing NAs. See `join`.

`rownamesAsColumn` logical vector of length 1 specifying whether the original rownames in `data.list` should be a column in the return object. If TRUE, the rownames are a column and the returned data.frame has default row.names 1:nrow. If FALSE, the returned data.frame has rownames from the merging.

`rtn.rownames.nm` character vector of length 1 specifying what the names of the rownames column should be in the return object. The `rtn.rownames.nm` argument is only used if `rownamesAsColumn` = TRUE.

Details

`Join` is a polished rendition of `Reduce(f = plyr::join, x = data.list)`. A future version of the function might allow for the `init` and `right` arguments from `Reduce`.

Value

data.frame of all uniquely colnamed columns from `data.list` with the rows included specified by `type` and rownames specified by `rownamesAsColumn`. Similar to `plyr::join`, `Join` returns the rows in the same order as they appeared in `data.list`.

See Also

`join_all` `join` `merge`

Examples

```
# by column
mtcars1 <- mtcars
mtcars1$"id" <- row.names(mtcars)
mtcars2 <- data.frame("id" = mtcars1$"id", "forward" = 1:32)
mtcars3 <- data.frame("id" = mtcars1$"id", "backward" = 32:1)
mtcars_list <- list(mtcars1, mtcars2, mtcars3)
by_column <- Join(data.list = mtcars_list, by = "id")
by_column2 <- Join(data.list = mtcars_list, by = "id", rownamesAsColumn = TRUE)
by_column3 <- Join(data.list = mtcars_list, by = NULL)

# by rownames
mtcars1 <- mtcars
mtcars2 <- data.frame("forward" = 1:32, row.names = row.names(mtcars))
mtcars3 <- data.frame("backward" = 32:1, row.names = row.names(mtcars))
by_rownm <- Join(data.list = list(mtcars1, mtcars2, mtcars3), by = "0")
by_rownm2 <- Join(data.list = list(mtcars1, mtcars2, mtcars3), by = "0",
   rownamesAsColumn = TRUE)
identical(x = by_column[names(by_column) != "id"],
   y = by_rownm) # same as converting rownames to a column in the data
identical(x = by_column2[names(by_column2) != "id"],
   y = by_rownm2) # same as converting rownames to a column in the data

# inserted NAs (by columns)
mtcars1 <- mtcars[1:4]
mtcars2 <- setNames(obj = as.data.frame(scale(x = mtcars1[-1],
   center = TRUE, scale = FALSE)), nm = paste0(names(mtcars1[-1]), "_c"))
mtcars3 <- setNames(obj = as.data.frame(scale(x = mtcars1[-1],
   center = FALSE, scale = TRUE)), nm = paste0(names(mtcars1[-1]), "_s"))
tmp <- lapply(X = list(mtcars1, mtcars2, mtcars3), FUN = function(dat)
   dat[sample(x = row.names(dat), size = 10), ])
mtcars_list <- lapply(X = tmp, FUN = reshape::namerows)
by_column_NA <- Join(data.list = mtcars_list, by = "id") # join by row.names
by_column_NA2 <- Join(data.list = mtcars_list, by = "id", rownamesAsColumn = TRUE)
identical(x = row.names(by_column_NA), # rownames from any data.frame are retained
   y = Reduce(f = union, x = lapply(X = mtcars_list, FUN = row.names)))

# inserted NAs (by rownames)
mtcars1 <- mtcars[1:4]
mtcars2 <- setNames(obj = as.data.frame(scale(x = mtcars1, center = TRUE, scale = FALSE)),
   nm = paste0(names(mtcars1), "_c"))
mtcars3 <- setNames(obj = as.data.frame(scale(x = mtcars1, center = FALSE, scale = TRUE)),
   nm = paste0(names(mtcars1), "_s"))
mtcars_list <- lapply(X = list(mtcars1, mtcars2, mtcars3), FUN = function(dat)
   dat[sample(x = row.names(dat), size = 10), ])
by_rownm_NA <- Join(data.list = mtcars_list, by = "0") # join by row.names
by_rownm_NA2 <- Join(data.list = mtcars_list, by = "0", rownamesAsColumn = TRUE)
identical(x = row.names(by_rownm_NA), # rownames from any data.frame are retained
   y = Reduce(f = union, x = lapply(X = mtcars_list, FUN = row.names)))

# types of joins
Join(data.list = mtcars_list, by = "0", type = "left") # only rows included in mtcars1
```

```
Join(data.list = mtcars_list, by = "0", type = "right") # only rows included in mtcars3
Join(data.list = mtcars_list, by = "0", type = "inner") # only rows included in
   # all 3 data.frames (might be empty due to random chance from sample() call)

# errors returned
tmp <- str2str::try_expr(
   Join(data.list = list(mtcars, as.matrix(mtcars), as.matrix(mtcars)))
)
print(tmp[["error"]]) # "The elements with the following positions in
   # `data.list` are not data.frames: 2 , 3"
tmp <- str2str::try_expr(
   Join(data.list = replicate(n = 3, mtcars, simplify = FALSE), by = 0)
)
print(tmp[["error"]]) # "Assertion on 'by' failed: Must be of type
   # 'character' (or 'NULL'), not 'double'."
tmp <- str2str::try_expr(
   Join(data.list = replicate(n = 3, mtcars, simplify = FALSE), by = c("0","mpg"))
)
print(tmp[["error"]]) # "If '0' is a value in `by`, then it must be the
   # only value and `by` must be length 1."
tmp <- str2str::try_expr(
   Join(data.list = list(attitude, attitude, mtcars), by = "mpg")
)
print(tmp[["error"]]) # "The data.frames associated with the following positions in
   # `data.list` do not contain the `by` columns: 1 , 2"
```

`la2a` *List of (3D+) Arrays to (3D+) Array*

Description

`la2a` converts a list of (3D+) arrays to a one dimension larger (3D+) array where the list dimension becomes the additional dimension of the array. `la2a` is a simple wrapper function for `abind::abind`. If you have a list of matrices, then use `lm2a`.

Usage

```
la2a(la, dim.order = 1:(ndim(la[[1]]) + 1L), dimlab.list = NULL, check = TRUE)
```

Arguments

`la`	list of 3D+ arrays which each have the same dimensions.
`dim.order`	integer vector of length = `ndim(la[[1]]) + 1L` specifying the order of dimensions for the returned array. The default is `1:(ndim(la[[1]]) + 1L)` which means the arrays within `la` maintain their dimensions and the list dimension is appended as the last dimension.
`dimlab.list`	character vector of length 1 specifying the dimlabel for the list dimension.

check — logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether la is a list of 3D+ arrays. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

3D+ array where the list elements of la is now a dimension. The order of the dimensions is determined by the argument dim.order. The dimnames of the returned array is determined by the dimnames in la[[1]] and names(la).

Examples

```
la <- list("one" = HairEyeColor, "two" = HairEyeColor*2, "three" = HairEyeColor*3)
la2a(la) # default
la2a(la, dimlab.list = "Multiple")
la2a(la, dim.order = c(4,3,1,2))
la2a(la, dim.order = c(4,3,1,2), dimlab.list = "Multiple")
```

---

laynames — *Names of the Layers (the Third Dimension)*

Description

laynames returns the names of the layers - the third dimension - of an array. If the object does not have a third dimension (e.g., matrix), then the function will return NULL. If the object does not have any dimensions (e.g., atomic vector), then the function will also return NULL.

Usage

```
laynames(x)
```

Arguments

x — array.

Details

R does not have standard terminology for the third dimension. There are several common terms people use including "height" and "page". I personally prefer "layer" as it makes sense whether the user visualizes the third dimension as going into/ontop a desk or into/ontop a wall.

Value

Names of the layers (the third dimension) of x. The structure is a character vector with length = nlay(x). See details for special cases.

Examples

```
laynames(HairEyeColor)
a <- array(data = NA, dim = c(6,7,8,9))
laynames(a)
laynames(c(1,2,3))
```

`ld2a` *List of Data-Frames to a 3D Array*

Description

`ld2a` converts a list of data.frames to a 3D array. The data.frames must have the same dimensions.

Usage

```
ld2a(
  ld,
  dim.order = c(1, 2, 3),
  dimlab.list = NULL,
  fct = "chr",
  chr = "chr",
  lgl = "int",
  order.lvl = "alphanum",
  decreasing = FALSE,
  na.lvl = FALSE,
  check = TRUE
)
```

Arguments

`ld`	list of data.frames that all have the same dimensions.
`dim.order`	integer vector of length 3 specifying the order of dimensions for the returned array. The default is `c(1,2,3)` which means the rows of the data.frames in `ld` is the first dimension (i.e., rows), the columns of the data.frames in `ld` is the second dimension (i.e., columns), and the list elements of `ld` is the third dimension (i.e., layers).
`dimlab.list`	character vector of length 1 specifying the dimlabel for the list dimension.
`fct`	character vector of length 1 specifying what factors should be converted to. There are three options: 1) "chr" for converting to character vectors (i.e., factor labels), 2) "int" for converting to integer vectors (i.e., factor codes), or 3) "fct" for keeping the factor as is without any changes.
`chr`	character vector of length 1 specifying what character vectors should be converted to. There are three options: 1) "fct" for converting to factors (i.e., elements will be factor labels), 2) "int" for converting to integer vectors (i.e., factor codes after first converting to a factor), or 3) "chr" for keeping the character vectors as is without any changes.

`lgl`	character vector of length 1 specifying what logical vectors should be converted to. There are four options: 1) "fct" for converting to factors (i.e., "TRUE" and "FALSE" will be factor labels), 2) "chr" for converting to character vectors (i.e., elements will be "TRUE" and "FALSE"), 3) "int" for converting to integer vectors (i.e., TRUE = 1; FALSE = 0), and 4) "lgl" for keeping the logical vectors as is without any changes.
`order.lvl`	character vector of length 1 specifying how you want to order the levels of the factor. The options are "alphanum", which sorts the levels alphanumerically (with NA last); "position", which sorts the levels by the position the level first appears; "frequency", which sorts the levels by their frequency. If any frequencies are tied, then the ties are sorted alphanumerically (with NA last).
`decreasing`	logical vector of length 1 specifying whether the ordering of the levels should be decreasing (TRUE) rather than increasing (FALSE).
`na.lvl`	logical vector of length 1 specifying if NA should be considered a level.
`check`	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `ld` is a list of data.frames. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Details

If the columns of the data.frames in `ld` are not all the same typeof, then the return object is coerced to the most complex type of any data.frame column (e.g., character > double > integer > logical). See `unlist` for details about the hierarchy of object types.

Value

3D array with all the elements from `ld` organized into dimensions specified by `dim.order`.

Examples

```
ld <- list("first" = BOD, "second" = BOD*2, "third" = BOD*3)
ld2a(ld)
ld <- list("cars" = cars, "mtcars" = mtcars)
try_expr(ld2a(ld)) # error
```

`ld2d` *List of Data-Frames to Data-Frame*

Description

`ld2d` converts a list of data.frames to a data.frame. The function is primarily for rbinding a list of data.frames (`along = 1`). An option to cbind the list of data.frames is included (`along = 2`), but is just a call to `data.frame(ld, stringsAsFactors = stringsAsFactors, check.names = check.names)`.

Usage

```
ld2d(
  ld,
  along = 1,
  fill = FALSE,
  rtn.listnames.nm = "list_names",
  rtn.rownames.nm = "row_names",
  stringsAsFactors = FALSE,
  check.names = FALSE,
  check = TRUE
)
```

Arguments

`ld` list of data.frames.

`along` integer vector of length 1 specifying which dimension the data.frames from `ld` should be binded along: 1 is for rows and 2 is for columns.

`fill` logical vector of length 1 specifying whether to fill in missing values for any data.frames from `ld` that do not have all the columns. At this time, `fill` is only available for rbinding and only used if `along` = 1.

`rtn.listnames.nm` character of length 1 specifying what the name of the column containing the names/positions of `ld` should be in the returned data.frame. If NULL, then no column is created for the names/positions of `ld` in the returned data.frame.

`rtn.rownames.nm` character of length 1 specifying what the name of the column containing the rownames of `ld`'s data.frames should be in the returned data.frame. If NULL, then no column is created for the rownames of `ld`'s data.frames in the returned data.frame.

`stringsAsFactors` logical vector of length 1 specifying whether character columns from `ld` should be converted to factors. Only available and used if `fill` = FALSE.

`check.names` logical vector of length 1 specifying whether the colnames of the returned data.frame should be checked for duplicates and made unique. Only used if for cbinding with `along` = 2.

`check` logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `ld` is a list of data.frames. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

data.frame with the rows (if `along` = 1) or columns (if `along` = 2) of `ld` binded together.

Examples

```
# without listnames and default rownames
```

```
ld <- list(BOD*1, BOD*2, BOD*3)
ld2d(ld)
# with listnames and default rownames
names(ld) <- LETTERS[1:3]
ld2d(ld)
# without listnames and custom rownames
ld <- lapply(unname(ld), FUN = `row.names<-`, letters[1:6])
ld2d(ld)
# with listnames and custom rownames
ld <- setNames(ld, LETTERS[1:3])
ld2d(ld)
# can handle same named columns in different positions
ld <- list(BOD*1, rev(BOD*2), rev(BOD*3))
ld2d(ld)
# can handle some columns being absent with fill = TRUE
ld[[2]]$"demand" <- NULL
try_expr(ld2d(ld, fill = FALSE)) # error
ld2d(ld, fill = TRUE) # NAs added
# along = 2 for cbinding
ld2d(ld, along = 2) # does not check/rename for double colnames
ld2d(ld, along = 2, check.names = TRUE) # makes unique colnames
ld2d(setNames(ld, nm = c("One","Two","Three")), along = 2,
   check.names = TRUE) # does not add prefixes from list names
```

---

ld2v *List of Data-Frames to (Atomic) Vector*

Description

ld2v converts a list of data.frames to a (atomic) vector. This function is a combination of d2v and lv2v. This function can be useful in conjunction with the boot::boot function when wanting to generate a statistic function that returns an atomic vector.

Usage

```
ld2v(
  ld,
  along = 2,
  use.listnames = TRUE,
  use.dimnames = TRUE,
  sep = "_",
  fct = "chr",
  chr = "chr",
  lgl = "int",
  order.lvl = "alphanum",
  decreasing = FALSE,
  na.lvl = FALSE,
  check = TRUE
)
```

Arguments

`ld`	list of data.frames. They do NOT have to have the same dimensions.
`along`	numeric vector of length one that is equal to either 1 or 2. 1 means that each data.frame in `ld` is split along rows (i.e., dimension 1) and then concatenated. 2 means that each data.frame in `ld` is split along columns (i.e., dimension 2) and then concatenated.
`use.listnames`	logical vector of length 1 specifying whether the returned vector should have names based on the list the element came from. If `ld` does not have names, `use.listnames` = TRUE will have the list positions serve as the list names (e.g., "1", "2", "3", etc.)
`use.dimnames`	logical vector of length 1 specifying whether the returned vector should have names based on the dimnames of the data.frame the element came from. If a data.frame within `ld` does not have dimnames, `use.dimnames` = TRUE will have the dimension positions serve as the dimnames (e.g., "1", "2", "3", etc.)
`sep`	character vector of length 1 specifying the string used to separate the listnames and dimnames from each other when creating the names of the returned vector.
`fct`	character vector of length 1 specifying what factors should be converted to. There are three options: 1) "chr" for converting to character vectors (i.e., factor labels), 2) "int" for converting to integer vectors (i.e., factor codes), or 3) "fct" for keeping the factor as is without any changes.
`chr`	character vector of length 1 specifying what character vectors should be converted to. There are three options: 1) "fct" for converting to factors (i.e., elements will be factor labels), 2) "int" for converting to integer vectors (i.e., factor codes after first converting to a factor), or 3) "chr" for keeping the character vectors as is without any changes.
`lgl`	character vector of length 1 specifying what logical vectors should be converted to. There are four options: 1) "fct" for converting to factors (i.e., "TRUE" and "FALSE" will be factor labels), 2) "chr" for converting to character vectors (i.e., elements will be "TRUE" and "FALSE"), 3) "int" for converting to integer vectors (i.e., TRUE = 1; FALSE = 0), and 4) "lgl" for keeping the logical vectors as is without any changes.
`order.lvl`	character vector of length 1 specifying how you want to order the levels of the factor. The options are "alphanum", which sorts the levels alphanumerically (with NA last); "position", which sorts the levels by the position the level first appears; "frequency", which sorts the levels by their frequency. If any frequencies are tied, then the ties are sorted alphanumerically (with NA last).
`decreasing`	logical vector of length 1 specifying whether the ordering of the levels should be decreasing (TRUE) rather than increasing (FALSE).
`na.lvl`	logical vector of length 1 specifying if NA should be considered a level.
`check`	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `ld` is a list of data.frames. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Details

When `use.listnames` and `use.dimnames` are both TRUE (default), the returned vector elements the following naming scheme: "[listname][sep][rowname][sep][colname]".

If the columns of the data.frames in `ld` are not all the same typeof, then the return object is coerced to the most complex type of any data.frame column (e.g., character > double > integer > logical). See `unlist` for details about the hierarchy of object types.

Value

(atomic) vector with an element for each element from `ld`.

Examples

```
ld <- list("cars" = cars, "mtcars" = mtcars)
# use.listnames = TRUE & use.dimnames = TRUE
ld2v(ld) # the first part of the name is the list names followed by the dimnames
# use.listnames = FALSE & use.dimnames = TRUE
ld2v(ld, use.listnames = FALSE) # only dimnames used,
   # which can result in repeat names
# use.listnames = TRUE & use.dimnames = FALSE
ld2v(ld, use.dimnames = FALSE) # listnames and vector position without any
   # reference to matrix dimensions
# use.listnames = FALSE & use.dimnames = FALSE
ld2v(ld, use.listnames = FALSE, use.dimnames = FALSE) # no names at all
# when list does not have names
ld <- replicate(n = 3, expr = attitude, simplify = FALSE)
ld2v(ld) # the first digit of the names is the list position and
   # the subsequent digits are the matrix dimnames
ld2v(ld, use.listnames = FALSE) # only dimnames used,
   # which can result in repeat names
```

`lm2a` *List of Matrices to 3D Array*

Description

`lm2a` converts a list of matrices to a 3D array where the list dimension becomes the third dimension of the array (layers). `lm2a` is a simple wrapper function for `abind::abind`.

Usage

```
lm2a(lm, dim.order = c(1, 2, 3), dimlab.list = NULL, check = TRUE)
```

Arguments

`lm`	list of matrices which each have the same dimensions.
`dim.order`	integer vector of length 3 specifying the order of dimensions for the returned array. The default is `c(1,2,3)` which means the rows of the matrices in `lm` is the first dimension (i.e., rows), the columns of the matrices in `lm` is the second dimension (i.e., columns), and the list elements of `lm` is the third dimension (i.e., layers).
`dimlab.list`	character vector of length 1 specifying the dimlabel for the list dimension.
`check`	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `lm` is a list of matrices. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

3D array where the list elements of `lm` is now a dimension. The order of the dimensions is determined by the argument `dim.order` with dimnames specified by names(`lm`). The dimnames of the returned array is determined by the dimnames in `lm[[1]]` and `names(lm)`.

Examples

```
lm <- asplit(HairEyeColor, MARGIN = 3L)
lm2a(lm) # default
lm2a(lm, dimlab.list = "Sex")
lm2a(lm, dim.order = c(3,1,2))
lm2a(lm, dim.order = c(3,1,2), dimlab.list = "Sex")
```

`lm2d` *List of Matrices to Data-Frame*

Description

`lm2d` converts a list of matrices to a data.frame. The function is primarily for rbinding a list of matrices (`along` = 1). An option to cbind the list of matrices is included (`along` = 2), but is just a call to `data.frame(lapply(lm, m2d), stringsAsFactors = stringsAsFactors, check.names = check.names)`.

Usage

```
lm2d(
  lm,
  along = 1,
  fill = FALSE,
  rtn.listnames.nm = "list_names",
  rtn.rownames.nm = "row_names",
  stringsAsFactors = FALSE,
```

```
  check.names = FALSE,
  check = TRUE
)
```

Arguments

`lm` list of matrices.

`along` numeric vector of length 1 specifying which dimension the matrices from `lm` should be binded along: 1 is for rows and 2 is for columns.

`fill` logical vector of length 1 specifying whether to fill in missing values for any matrices from `lm` that do not have all the columns. At this time, `fill` is only available for rbinding and only used if `along` = 1.

`rtn.listnames.nm` character of length 1 specifying what the name of the column containing the names/positions of `lm` should be in the returned data.frame. If NULL, then no column is created for the names/positions of `lm` in the returned data.frame.

`rtn.rownames.nm` character of length 1 specifying what the name of the column containing the names/positions of the rows within `lm`'s matrices should be in the returned data.frame. If NULL, then no column is created for the rownames of `lm`'s matrices in the returned data.frame.

`stringsAsFactors` logical vector of length 1 specifying whether character columns from `lm` should be converted to factors. Note, that is a matrix is character, then `stringsAsFactors` would apply to all columns.

`check.names` logical vector of length 1 specifying whether the colnames of the returned data.frame should be checked for duplicates and made unique. Only used if for cbinding with `along` = 2.

`check` logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `lm` is a list of matrices. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Details

Another way to convert a list of matrices to a data.frame is to convert the list dimension, row dimension, and column dimension in the list of matrices all to variable dimensions in the data.frame. If this is desired, call `a2d(lm2a(lm))` instead of `lm2d`.

Value

data.frame with the rows (if `along` = 1) or columns (if `along` = 2) of `lm` binded together.

Examples

```
# list names and rownames
lm <- asplit(HairEyeColor, MARGIN = 3L)
lm2d(lm) # default
```

```
lm2d(lm, rtn.listnames.nm = "Sex", rtn.rownames.nm = "Hair")
# no list names
lm2 <- `names<-`(lm, value = NULL)
lm2d(lm2)
lm2d(lm2, rtn.listnames.nm = NULL)
# no rownames too
lm3 <- lapply(lm2, `rownames<-`, value = NULL)
lm2d(lm3)
lm2d(lm3, rtn.rownames.nm = NULL)
lm2d(lm3, rtn.listnames.nm = NULL, rtn.rownames.nm = NULL)
# cbinding as columns
lm2d(lm3, along = 2)
lm2d(lm3, along = 2, check.names = TRUE)
```

`lm2v` *List of Matrices to (Atomic) Vector*

Description

`lm2v` converts a list of matrices to a (atomic) vector. This function is a combination of `m2v` and `lv2v`. This function can be useful in conjunction with the `boot::boot` function when wanting to generate a `statistic` function that returns an atomic vector.

Usage

```
lm2v(
  lm,
  along = 2,
  use.listnames = TRUE,
  use.dimnames = TRUE,
  sep = "_",
  check = TRUE
)
```

Arguments

`lm`	list of matrices. They do NOT have to be the same typeof or have the same dimensions.
`along`	numeric vector of length one that is equal to either 1 or 2. 1 means that each matrix in `lm` is split along rows (i.e., dimension 1) and then concatenated. 2 means that each matrix in `lm` is split along columns (i.e., dimension 2) and then concatenated.
`use.listnames`	logical vector of length 1 specifying whether the returned vector should have names based on the list the element came from. If `lm` does not have names, `use.listnames` = TRUE will have the list positions serve as the list names (e.g., "1", "2", "3", etc.)

`use.dimnames` logical vector of length 1 specifying whether the returned vector should have named based on the dimnames of the matrix the element came from. If a matrix within `lm` does not have dimnames, `use.dimnames` = TRUE will have the dimension positions serve as the dimnames (e.g., "1", "2", "3", etc.)

`sep` character vector of length 1 specifying the string used to separate the listnames and dimnames from each other when creating the names of the returned vector.

`check` logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `lm` is a list of matrices. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Details

When `list.names` and `use.dimnames` are both TRUE (default), the returned vector elements the following naming scheme: "[listname][sep][rowname][sep][colname]".

If the matrices in `lm` are not all the same typeof, then the return object is coerced to the most complex type of any matrix (e.g., character > double > integer > logical). See `unlist` for details about the hierarchy of object types.

Value

(atomic) vector with an element for each element from ‘lm‘.

Examples

```
lm <- list("numeric" = data.matrix(npk), "character" = as.matrix(npk))
# use.listnames = TRUE & use.dimnames = TRUE
lm2v(lm) # the first part of the name is the list names followed by the dimnames
# use.listnames = FALSE & use.dimnames = TRUE
lm2v(lm, use.listnames = FALSE) # only dimnames used,
   # which can result in repeat names
# use.listnames = TRUE & use.dimnames = FALSE
lm2v(lm, use.dimnames = FALSE) # listnames and vector position without any
   # reference to matrix dimensions
# use.listnames = FALSE & use.dimnames = FALSE
lm2v(lm, use.listnames = FALSE, use.dimnames = FALSE) # no names at all
# when list does not have names
lm <- replicate(n = 3, expr = as.matrix(attitude, rownames.force = TRUE), simplify = FALSE)
lm2v(lm) # the first digit of the names is the list position and
   # the subsequent digits are the matrix dimnames
lm2v(lm, use.listnames = FALSE) # no listnames; only dimnames used,
   # which can result in repeat names
```

lv2d *List of (atomic) vectors to Data-Frame*

Description

lv2d converts a list of (atomic) vectors to a data.frame. This function is similar to as.data.frame.list, but allows for more flexibility in how the data.frame will be structured (e.g., rowwise), while simplifying the dimension naming process.

Usage

```
lv2d(
  lv,
  along,
  fill = FALSE,
  risky = FALSE,
  stringsAsFactors = FALSE,
  check = TRUE
)
```

Arguments

lv	list of (atomic) vectors.
along	numeric vector of length 1 specifying either 1 for binding along rows (i.e., each list element is a row) or 2 for binding along columns (i.e., each list element in a column).
fill	logical vector of length 1 specifying whether 1) to allow the vectors in lv to have different lengths, names, or both, 2) to bind by the names of the vectors within lv rather than by their positions (unless no names are present in which case positions are used), and 3) fill in any missing values in the return object with NA.
risky	logical vector of length 1 specifying whether to use list2DF rather than data.frame when along = 2 and fill = TRUE. If either along = 1 or fill = FALSE, this argument is not used.
stringsAsFactors	logical vector of length 1 specifying whether character vectors should be coerced to factors. See default.stringsAsFactors.
check	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether lv is a list of atomic vectors. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Details

If fill = FALSE, `lv2d` uses a combination of `do.call` and `rbind` if `along` = 1 or `do.call` and `cbind` if `along` = 2. rownames and colnames of the returned data.frame are determined by the names of `lv` and the names of the first vector within `lv`. If either are NULL, then the positions are used as the dimension names. If fill = FALSE, then an error is returned ff the vectors in `lv` do not all have the same length. If fill = FALSE, there is no check to ensure the elements within each `lv` vector have the same names in the same order. The names are taken from the first vector in `lv`, and it is assumed those names and their order apply to each vector in `lv`. Essentially, if fill = FALSE, `lv` binds the vectors by positions and not names.

If fill = TRUE, `lv2d` uses `plyr::rbind.fill` if `along` = 1 or `plyr::join_all` by the vector names if `along` = 2. If fill = TRUE, `lv2d` binds the vectors by by names (and by positions if no names are present). Depending on what the user wants, fill = FALSE or TRUE could be safer. If the user wants an error returned when any vectors within `lv` have different lengths, then fill = FALSE should be used. If the user wants to bind by names rather than position, then fill = TRUE should be used.

Value

data.frame with the elements of ‘lv‘ either as rows or columns and dimnames determined along the names of ‘lv‘ and ‘lv‘[[1]].

Examples

```
# 1) `lv` has names; vectors have names
lv <- setNames(object = lapply(X = letters, FUN = setNames, nm = "alphabet"), nm = LETTERS)
lv2d(lv, along = 1)
lv2d(lv, along = 2)
lv2d(lv, along = 2, stringsAsFactors = TRUE)

# 2) `lv` has names; no vector names
lv <- setNames(object = as.list(letters), nm = LETTERS)
lv2d(lv, along = 1)
lv2d(lv, along = 2)

# 3) no `lv` names; vector have names
lv <- lapply(X = letters, FUN = setNames, nm = "alphabet")
lv2d(lv, along = 1)
lv2d(lv, along = 2)

# 4) no `lv` names; no vector names
lv <- as.list.default(letters)
lv2d(lv, along = 1)
lv2d(lv, along = 2)

# we want vectors combined along rows
lv <- lapply(X = unclass(mtcars), FUN = `names<-`, value = row.names(mtcars))
rbind(lv) # not what we want (array list)
rbind.data.frame(lv) # also not what we want (combined along cols)
do.call(what = rbind.data.frame, args = lv) # doesn't have useful dimnames
lv2d(lv, along = 1) # finally what we want
```

```
# fill = TRUE
tmp <- lapply(X = unclass(mtcars), FUN = `names<-`, value = row.names(mtcars))
lv <- lapply(X = tmp, FUN = function(v) v[-(sample(x = seq_along(v), size = 9))])
lv2d(lv, along = 1L, fill = TRUE) # NA for missing values in any given row
tmp <- lapply(X = unclass(as.data.frame(t(mtcars))), FUN = `names<-`, value = names(mtcars))
lv <- lapply(X = tmp, FUN = function(v) v[-(sample(x = seq_along(v), size = 3))])
lv2d(lv, along = 2L, fill = TRUE) # NA for missing values in any given column

# actual use case
lv <- lapply(X = sn(1:30), FUN = function(i)
   coef(lm(v2frm(names(attitude)), data = attitude[-i, ])))
lv2d(lv, along = 2) # coefs in a data.frame

# when vectors have named elements in different positions use fill = TRUE
lv <- list("row_1" = c("col_A" = "col_A1", "col_B" = "col_B1", "col_C" = "col_C1"),
"row_2" = c("col_B" = "col_B2", "col_C" = "col_C2", "col_A" = "col_A2"),
"row_3" = c("col_C" = "col_C3", "col_A" = "col_A3", "col_B" = "col_B3"))
lv2d(lv, along = 1, fill = FALSE) # probably not what you want (See details)
lv2d(lv, along = 1, fill = TRUE) # what we want

# when you have a list with only one vector
lv <- list("A" = c("one" = 1, "two" = 2, "three" = 3))
x <- lv2m(lv, along = 1, fill = FALSE)
y <- lv2m(lv, along = 1, fill = TRUE)
identical(x, y)
```

`lv2m` *List of (atomic) Vectors to Matrix*

Description

`lv2m` converts a list of (atomic) vectors to a matrix. This function is similar to a hypothetical `as.matrix.list` method if it existed. Note, if the vectors are not all the same typeof, then the matrix will have the most complex typeof any vector in `lv`.

Usage

```
lv2m(lv, along, fill = FALSE, check = TRUE)
```

Arguments

`lv`	list of (atomic) vectors.
`along`	numeric vector of length 1 specifying either 1 for binding along rows (i.e., each list element is a row) and 2 for binding along columns (i.e., each list element in a column).
`fill`	logical vector of length 1 specifying whether 1) to allow the vectors in `lv` to have different lengths, names, or both, 2) to bind by the names of the vectors within `lv` rather than by their positions (unless no names are present in which

case positions are used), and 3) fill in any missing values in the return object with NA.

`check` logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `lv` is a list of atomic vectors. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Details

If fill = FALSE, `lv2m` uses a combination of `do.call` and `rbind` if `along` = 1 or `do.call` and `cbind` if `along` = 2. rownames and colnames of the returned data.frame are determined by the names of `lv` and the names of the first vector within `lv`. If either are NULL, then the positions are used as the dimension names. If fill = FALSE, then an error is returned ff the vectors in `lv` do not all have the same length. If fill = FALSE, there is no check to ensure the elements within each `lv` vector have the same names in the same order. The names are taken from the first vector in `lv`, and it is assumed those names and their order apply to each vector in `lv`. Essentially, if fill = FALSE, `lv` binds the vectors by positions and not names.

If fill = TRUE, `lv2m` uses `plyr::rbind.fill.matrix` if `along` = 1 or `plyr::rbind.fill.matrix` and `t.default` if `along` = 2. If fill = TRUE, `lv2d` binds the vectors by by names (and by positions if no names are present). Depending on what the user wants, fill = FALSE or TRUE could be safer. If the user wants an error returned when any vectors within `lv` have different lengths, then fill = FALSE should be used. If the user wants to bind by names rather than position, then fill = TRUE should be used.

Value

matrix with the elements of `lv` either as rows or columns and dimnames determined by the names of `lv` and `lv[[1]]`. The typeof is determined by the highest typeof in the elements of `lv` (i.e., highest to lowest: character > double > integer > logical).

Examples

```
# 1) `lv` has names; vectors have names
lv <- setNames(object = lapply(X = letters, FUN = setNames, nm = "alphabet"), nm = LETTERS)
lv2m(lv, along = 1)
lv2m(lv, along = 2)

# 2) `lv` has names; no vector names
lv <- setNames(object = as.list(letters), nm = LETTERS)
lv2m(lv, along = 1)
lv2m(lv, along = 2)

# 3) no `lv` names; vector have names
lv <- lapply(X = letters, FUN = setNames, nm = "alphabet")
lv2m(lv, along = 1)
lv2m(lv, along = 2)

# 4) no `lv` names; no vector names
lv <- as.list.default(letters)
lv2m(lv, along = 1)
```

```
lv2m(lv, along = 2)

# actual use case (sort of)
lv <- lapply(X = asplit(x = as.matrix(attitude), MARGIN = 1),
   FUN = undim) # need undim since asplit returns 1D arrays
cbind(lv) # not what we want
do.call(what = cbind, args = lv) # doesn't have useful dimnames
lv2m(lv, along = 2) # finally what we want

# when vectors have named elements in different positions
lv <- list("row_1" = c("col_A" = "col_A1", "col_B" = "col_B1", "col_C" = "col_C1"),
   "row_2" = c("col_B" = "col_B2", "col_C" = "col_C2", "col_A" = "col_A2"),
   "row_3" = c("col_C" = "col_C3", "col_A" = "col_A3", "col_B" = "col_B3"))
lv2m(lv, along = 1, fill = FALSE) # probably not what you want
lv2m(lv, along = 1, fill = TRUE) # what you want (See details)

# when you have a list with only one vector
lv <- list("A" = c("one" = 1, "two" = 2, "three" = 3))
x <- lv2m(lv, along = 1, fill = FALSE)
y <- lv2m(lv, along = 1, fill = TRUE)
identical(x, y)
```

`lv2v` *List of (atomic) Vectors to (atomic) Vector*

Description

`lv2v` converts a list of (atomic) vectors to an (atomic) vector. `lv2v` is simply a wrapper function for `unlist` that allows for more control over the names of the returned (atomic) vector.

Usage

```
lv2v(lv, use.listnames = TRUE, use.vecnames = TRUE, sep = "_", check = TRUE)
```

Arguments

`lv`	list of (atomic) vectors.
`use.listnames`	logical vector of length 1 specifying whether the names of `lv` should be used to construct names for the returned vector. If `lv` does not have names and `use.listnames` = TRUE, then the list positions will be used as names (i.e., "1","2","3", etc.).
`use.vecnames`	logical vector of length 1 specifying whether the names of each vector within `lv` should be used to construct names for the returned vector. If any vectors within `lv` do not have names and `use.vecnames` = TRUE, then the vector positions will be used as names (i.e., "1","2","3", etc.).
`sep`	character vector of length 1 specifying what string to use to separate list names from vector element names. Only used if `use.listnames` = TRUE.

check — logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether lv is a list of atomic vectors. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Details

There are four different scenarios. Each scenario is given as well as the structure of the returned object when both use.listnames and use.vecnames are TRUE (default): 1) if both lv and its vectors have names, then the names of the return object will be a pasted combination of the lv element's name and the vector element's name separated by sep; 2) if only lv has names and its vectors do not, then the names of the returned vector will be a pasted combination of the lv element's name and the vector element's position separated by sep; 3) if the vectors have names and lv does not, then the names of the returned vector will be a pasted combination of the lv positions and vector names separated by sep; 4) if both lv and its vectors do not have names, then the names of the returned vector will be the pasted combination of the lv positions and vector element's positions separated by sep.

If you want to convert a list of vectors where each vector has length = 1 and the list has names, then you probably want to specify use.vecnames = FALSE. This will replicate the functionality of unlist(lv). See the last example.

If you want have a list of vectors where each vector has length > 1 and you want to convert it to a list vector (i.e., all vectors with length = 1), then you can combine lv2v with v2lv via v2lv(lv2v(v)).

Value

atomic vector with length = sum of the lengths of the atomic vectors in lv and typeof = the highest typeof present in the atomic vectors in lv (i.e., from high to low: character > double > integer > logical). See the argument use.listnames for how names are created.

Examples

```
# 1) both `lv` and its atomic vectors have names
lv <- setNames(object = Map(object = 1:26, nm = letters, f = setNames), nm = LETTERS)
lv2v(lv, use.listnames = TRUE, use.vecnames = TRUE)
lv2v(lv, use.listnames = FALSE, use.vecnames = TRUE)
lv2v(lv, use.listnames = TRUE, use.vecnames = FALSE)
lv2v(lv, use.listnames = FALSE, use.vecnames = FALSE)
# 2) only `lv` has names
lv <- list("lower" = letters, "upper" = LETTERS)
lv2v(lv, use.listnames = TRUE, use.vecnames = TRUE)
lv2v(lv, use.listnames = FALSE, use.vecnames = TRUE)
lv2v(lv, use.listnames = TRUE, use.vecnames = FALSE) # FYI - results in repeat names
lv2v(lv, use.listnames = FALSE, use.vecnames = FALSE)
# 3) only the atomic vectors have names
lv <- list(setNames(object = 1:26, nm = letters), setNames(object = 1:26, nm = LETTERS))
lv2v(lv, use.listnames = TRUE, use.vecnames = TRUE)
lv2v(lv, use.listnames = FALSE, use.vecnames = TRUE)
lv2v(lv, use.listnames = TRUE, use.vecnames = FALSE)
lv2v(lv, use.listnames = FALSE, use.vecnames = FALSE)
# 4) neither `lv` nor its atomic vectors have names
```

```
lv <- as.list(1:26)
lv2v(lv, use.listnames = TRUE, use.vecnames = TRUE)
lv2v(lv, use.listnames = FALSE, use.vecnames = TRUE) # FYI - results in repeat names
lv2v(lv, use.listnames = TRUE, use.vecnames = FALSE)
lv2v(lv, use.listnames = FALSE, use.vecnames = FALSE)
# common use case for when vectors are all length 1 and list has names
lv <- setNames(as.list(letters), nm = LETTERS)
lv2v(lv, use.listnames = TRUE, use.vecnames = TRUE)
lv2v(lv, use.listnames = FALSE, use.vecnames = TRUE)
lv2v(lv, use.listnames = TRUE, use.vecnames = FALSE) # FYI - probably what you want
lv2v(lv, use.listnames = FALSE, use.vecnames = FALSE)
identical(unlist(lv), lv2v(lv, use.vecnames = FALSE)) # identical to unlist()
```

`m2d` *Matrix to Data-Frame*

Description

`m2d` converts a matrix to a data.frame. The benefit of `m2d` over `as.data.frame.matrix` is that it provides the `col` argument, which allows the columns of the data.frame to be the columns of the matrix (i.e., `col = 2`), the rows of the matrix (i.e., `col = 1`), or the expanded matrix (i.e., `col = 0`).

Usage

```
m2d(m, col = 2, stringsAsFactors = FALSE, check = TRUE)
```

Arguments

`m` matrix

`col` numeric vector of length 1 that is equal to either 0, 1, or 2. `col` specifies what dimension from `m` should be the columns of the returned data.frame. If `col = 2`, then the columns of `m` (i.e., dimension 2) are the columns of the returned data.frame. If `col = 1`, then the rows of `m` (i.e., dimension 1) are the columns of the returned data.frame. If `col = 0`, neither of the `m` dimensions are the columns and instead the matrix is expanded by `reshape::melt.array` such that in the returned data.frame the first column is `rownames(m)`, the second column is `colnames(m`, and the third column is the elements of `m`. If any `dimnames(m)` are NULL, then they are replaced with the positions of the dimensions.

`stringsAsFactors` logical vector of length 1 specifying whether any resulting character columns in the return object should be factors. If `m` is a character matrix and `stringsAsFactors` = TRUE, then all columns in the returned data.frame will be factors. If `col` = 0 and `stringsAsFactors` = TRUE, then the first two columns in the returned data.frame specifying `dimnames(m)` will be factors.

`check` logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `m` is a matrix. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

data.frame with rownames and colnames specified by `dimnames(m)` and `col`. If `col = 0`, then the rownames are default (i.e., "1","2","3", etc.) and the colnames are the following: the first two columns are `names(dimnames(m))` (if NULL they are "rownames" and "colnames", respectively) and the third is "element".

Examples

```
mtcars2 <- as.matrix(mtcars, rownames.force = TRUE) # to make sure dimnames stay in the example
m2d(mtcars2) # default
m2d(m = mtcars2, col = 1) # data.frame columns are matrix rownames
m2d(m = mtcars2, col = 0) # data.frame columns are the entire matrix
mat <- cbind(lower = letters, upper = LETTERS)
m2d(mat)
m2d(mat, stringsAsFactors = TRUE)
m2d(mat, col = 0)
m2d(mat, col = 0, stringsAsFactors = TRUE)
```

---

m2lv *Matrix to List of (Atomic) Vectors*

Description

`m2lv` converts a matrix to a list of (atomic) vectors. This is useful since there is no `as.list.matrix` method. When rownames and/or colnames are NULL, they are replaced by their position numerals so that the dimension information is retained.

Usage

```
m2lv(m, along, check = TRUE)
```

Arguments

`m`	matrix (i.e., array with 2 dimensions).
`along`	numeric vector of length 1 specifying which dimension to slice the matrix along. If 1, then the matrix is sliced by rows. If 2, then the matrix is sliced by columns.
`check`	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `m` is a matrix. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

list of (atomic) vectors. If `along` = 1, then the names are the rownames of `m` and the vectors are rows from `m`. If `along` = 2, then the names are the colnames of `m` and the vector are columns from `m`. Note, the vectors always have the same length as `nrow(m)`.

Examples

```
m2lv(VADeaths, along = 1)
m2lv(VADeaths, along = 2)
m2lv(m = as.matrix(x = attitude, rownames.force = TRUE), along = 1)
m2lv(m = as.matrix(x = attitude, rownames.force = TRUE), along = 2)
m2lv(m = as.matrix(x = unname(attitude), rownames.force = FALSE),
   along = 1) # dimnames created as position numerals
m2lv(m = as.matrix(x = unname(attitude), rownames.force = FALSE),
   along = 2) # dimnames created as position numerals
# check = FALSE
try_expr(m2lv(VADeaths, along = 3, check = FALSE)) # less informative error message
try_expr(m2lv(VADeaths, along = 3, check = TRUE)) # more informative error message
```

---

m2v *Matrix to (Atomic) Vector*

Description

m2v converts a matrix to a (atomic) vector. The benefit of m2v over as.vector or c is that 1) the vector can be formed along rows as well as columns and 2) the dimnames from m can be used for the names of the returned vector.

Usage

```
m2v(m, along = 2, use.dimnames = TRUE, sep = "_", check = TRUE)
```

Arguments

m	matrix
along	numeric vector of length one that is equal to either 1 or 2. 1 means that m is split along rows (i.e., dimension 1) and then concatenated. 2 means that m is split along columns (i.e., dimension 2) and then concatenated.
use.dimnames	logical vector of length 1 that specifies whether the dimnames of m should be used to create the names for the returned vector. If FALSE, the returned vector will have NULL names. If TRUE, see details.
sep	character vector of length 1 specifying the string that will separate the rownames and colnames in the naming scheme of the return object. Note, sep is not used if use.dimnames = FALSE.
check	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether m is a matrix. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Details

If `use.dimnames` = TRUE, then each element's name will be analogous to `paste(rownames(m)[i], colnames(m)[j], sep = sep)`. If `m` does not have rownames and/or colnames, then they will be replaced by dimension positions. This is also true when `m` has only one row *and* one column. The exception is when `m` has either a single row *or* single column. In these cases, only the non-single dimension's names will be used. If `m` has one row, then the names of the returned vector will be `colnames(m)`. If `m` has one column, then the names of the returned vector will be `rownames(m)`. Again, if `m` does not have rownames and/or colnames, then they will be replaced by dimension positions.

Value

(atomic) vector of length = `length(m)` where the order of elements from `m` has been determined by `along` and the names determined by the `use.dimnames`, `dimnames(m)`, and `sep`. See details for when `use.dimnames` = TRUE.

Examples

```
# general matrix
mtcars2 <- as.matrix(mtcars, rownames.force = TRUE) # to make sure dimnames stay in the example
m2v(mtcars2) # default
m2v(m = mtcars2, along = 1) # concatenate along rows
m2v(m = mtcars2, sep = ".") # change the sep of the rownames(m) and colnames(m)
m2v(m = `dimnames<-`(mtcars2, list(NULL, NULL))) # use dimension positions as dimnames
m2v(m = mtcars2, use.dimnames = FALSE) # return object has no names
# one row/column matrix
one_row <- mtcars2[1,, drop = FALSE]
m2v(one_row)
one_col <- mtcars2[, 1, drop = FALSE]
m2v(one_col)
one_all <- mtcars2[1,1, drop = FALSE]
m2v(one_all)
m2v(one_all, use.dimnames = FALSE)
```

`ndim` *Number of Object Dimensions*

Description

`ndim` returns the number of dimensions an object has. This is most useful for arrays, which can have anywhere from 1 to 1000+ dimensions.

Usage

```
ndim(x)
```

Arguments

`x`	object that has dimensions (e.g., array).

Details

`ndim` is a very simple function that is simply `length(dim(x))`.

Value

integer vector of length 1 specifying the number of dimensions in `x`. If `x` does not have any dimensions, then O is returned.

Examples

```
ndim(state.region)
ndim(attitude)
ndim(HairEyeColor)
```

---

nlay — *Number of Layers (the Third Dimension)*

Description

`nlay` returns the number of layers - the third dimension - of an array. If the object does not have a third dimension (e.g., matrix), then the function will return NA with typeof = integer. If the object does not have any dimensions (e.g., atomic vector), then the function will return NULL.

Usage

```
nlay(x)
```

Arguments

`x`	array.

Details

R does not have standard terminology for the third dimension. There are several common terms people use including "height" and "page". I personally prefer "layer" as it makes sense whether the user visualizes the third dimension as going into/ontop a desk or into/ontop a wall.

Value

The number of layers (the third dimension) of `x`. The structure is an integer vector with length = 1. See details for special cases.

Examples

```
nlay(HairEyeColor)
a <- array(data = NA, dim = c(6,7,8,9))
nlay(a)
```

not.colnames *Identify Elements That are Not Colnames*

Description

`not.colnames` identifies which elements from `nm` are not colnames of `x`. If all elements are colnames, then a character vector of length 0 is returned.

Usage

```
not.colnames(x, nm)
```

Arguments

`x`	object with a colnames attribute
`nm`	character vector specifying the elements to test as colnames of `x`.

Value

character vector containing the elements of `nm` that are not colnames of `x`.

Examples

```
not.colnames(x = as.matrix(mtcars), nm = c("MPG","mpg"))
```

not.names *Identify Elements That are Not Names*

Description

`not.names` identifies which elements from `nm` are not names of `x`. If all elements are names, then a character vector of length 0 is returned.

Usage

```
not.names(x, nm)
```

Arguments

`x`	object with a names attribute
`nm`	character vector specifying the elements to test as names of `x`.

Value

character vector containing the elements of `nm` that are not names of `x`.

Examples

```
v <- setNames(object = letters, nm = LETTERS)
not.names(x = v, nm = c("A","a"))
data("mtcars")
not.names(x = mtcars, nm = c("MPG","mpg"))
not.names(x = mtcars, names(mtcars)) # returns a character vector of length 0
```

---

not.row.names *Identify Elements That are Not Row.names*

Description

`not.row.names` identifies which elements from `nm` are not row.names of `x`. If all elements are row.names, then a character vector of length 0 is returned.

Usage

```
not.row.names(x, nm)
```

Arguments

`x`	object with a row.names attribute
`nm`	character vector specifying the elements to test as row.names of `x`.

Value

character vector containing the elements of `nm` that are not row.names of `x`.

Examples

```
not.row.names(x = mtcars, nm = c("Mazda RX4","mazda RX4"))
```

---

not.rownames *Identify Elements That are Not Rownames*

Description

`not.rownames` identifies which elements from `nm` are not rownames of `x`. If all elements are rownames, then a character vector of length 0 is returned.

Usage

```
not.rownames(x, nm)
```

Arguments

x	object with a rownames attribute
nm	character vector specifying the elements to test as rownames of x.

Value

character vector containing the elements of nm that are not rownames of x.

Examples

```
not.rownames(x = as.matrix(mtcars), nm = c("Mazda RX4","mazda RX4"))
```

---

order.custom *Custom Order Permutation*

Description

order.custom creates the order of the positions in the atomic vectors from X that would cause the atomic vectors from X to be sorted according to the atomic vectors from ORD. This is analogus to the order function, but instead of doing default sorting (e.g., 1, 2, 3, etc. or "A", "B", "C", etc.), the sorting is customized by ORD. order.custom does custom ordering by converting each atomic vector from X to an ordered factor and then default sorting the ordered factors.

Usage

```
order.custom(X, ORD, na.last = FALSE, decreasing = FALSE)
```

Arguments

X	list of atomic vectors parellel matched with the atomic vectors in X specifying the elements to be ordered. Can also be a single atomic vector, which will internally be converted to a list with one element.
ORD	list of atomic vectors that do NOT have to be the same length specifying the order of the unique values for sorting. Can also be a single atomic vector, which will internally be converted to a list with one element.
na.last	logical vector of length 1 specifying whether missing values should be put last (TRUE), first (FALSE), or removed (NA).
decreasing	logical vector of length 1 specifying whether the sorting should start with the first element of the atomic vectors within ORD and proceed forward (FALSE) or the last element of the atomic vectors within ORD and proceed backwards (TRUE).

Details

Note, that the atomic vectors within X are always forward sequenced; if backward sequence is desired, then the user should call rev on both the input to X and ORD. This is analogous to reversing the order of the atomic vectors given to ... within order.

Value

integer vector of length = `X[[1]]` (after converting `X` to a list with one element is need be) providing the revised order of the atomic vectors within `X` that sorts them according to `ORD`.

Examples

```
# character vector
x <- esoph[["tobgp"]]
order.custom(X = x, ORD = c("20-29","10-19","30+","0-9g/day"))
x[order.custom(X = x, ORD = c("20-29","10-19","30+","0-9g/day"))] # returns character
esoph[order.custom(X = x, ORD = c("20-29","10-19","30+","0-9g/day")), ]
# order by position
sort(state.region)
x <- as.character(state.region)
order.custom(X = x, ORD = unique(x))
x[order.custom(X = x, ORD = unique(x))]
# numeric vector
y <- esoph[["ncases"]]
order.custom(X = y, ORD = c(6,5,4,3,2,1,0,17,8,9))
y[order.custom(X = y, ORD = c(6,5,4,3,2,1,0,17,8,9))] # returns numeric
esoph[order.custom(X = y, ORD = c(6,5,4,3,2,1,0,17,8,9)), ]
   # some unique values not provided in `ORD` (appended at the end and sorted by
   # where they appear in the dataset)
y <- esoph[["ncases"]]
order.custom(X = y, ORD = c(6,5,4,3,2,1,0))
y[order.custom(X = y, ORD = c(6,5,4,3,2,1,0))] # returns numeric
esoph[order.custom(X = y, ORD = c(6,5,4,3,2,1,0)), ]
# multiple vectors
z <- esoph[c("agegp","alcgp","tobgp")]
ord <- order.custom(X = z, ORD = list(
   "agegp" = c("45-54","55-64","35-44","65-74","25-34","75+"),
   "alcgp" = c("40-79","80-119","0-39g/day","120+"),
   "tobgp" = c("10-19","20-29","0-9g/day","30+")))
esoph[ord, ]
```

---

pick — *Extract Elements From a (Atomic) Vector*

Description

`pick` extracts the elements from a (atomic) vector that meet certain criteria: 1) using exact values or regular expressions (`pat`), 2) inclusion vs. exclusion of the value/expression (`not`), 3) based on elements or names (`nm`). Primarily for character vectors, but can be used with other typeof.

Usage

```
pick(x, val, pat = FALSE, not = FALSE, nm = FALSE, fixed = FALSE)
```

Arguments

x	atomic vector or an object with names (e.g., data.frame) if nm = TRUE.
val	atomic vector specifying which elements of x will be extracted. If pat = FALSE (default), then val should be an atomic vector of the same typeof as x, can have length > 1, and exact matching will be done via is.element (essentially match). If pat = TRUE, then val has to be a character vector of length 1 and partial matching will be done via grepl with the option of regular expressions if fixed = FALSE (default). Note, if nm = TRUE, then val should refer to names of x to determine which elements of x should be extracted.
pat	logical vector of length 1 specifying whether val should refer to exact matching (FALSE) via is.element (essentially match) or partial matching (TRUE) and/or use of regular expressions via grepl. See details for a brief description of some common symbols and help(regex) for more.
not	logical vector of length 1 specifying whether val indicates values that should be retained (FALSE) or removed (TRUE).
nm	logical vector of length 1 specifying whether val refers to the names of x (TRUE) rather than the elements of x themselves (FALSE).
fixed	logical vector of length 1 specifying whether val refers to values as is (TRUE) or a regular expression (FALSE). Only used if pat = TRUE.

Details

pick allows for 8 different ways to extract elements from a (atomic) vector created by the 2x2x2 combination of logical arguments pat, not, and nm. When pat = FALSE (default), pick uses is.element (essentially match) and requires exact matching of val in x. When pat = TRUE, pick uses grepl and allows for partial matching of val in x and/or regular expressions if fixed = FALSE (default).

When dealing with regular expressions via pat = TRUE and fixed = FALSE, certain symbols within val are not interpreted as literal characters and instead have special meanings. Some of the most commonly used symbols are . = any character, "|" = logical or, "^" = starts with, "\n" = new line, "\t" = tab.

Value

a subset of x that only includes the elements which meet the criteria specified by the function call.

Examples

```
# pedagogical cases
chr <- setNames(object = c("one","two","three","four","five"), nm = as.character(1:5))
# 1) pat = FALSE, not = FALSE, nm = FALSE
pick(x = chr, val = c("one","five"), pat = FALSE, not = FALSE, nm = FALSE)
# 2) pat = FALSE, not = FALSE, nm = TRUE
pick(x = chr, val = c("1","5"), pat = FALSE, not = FALSE, nm = TRUE)
# 3) pat = FALSE, not = TRUE, nm = FALSE
pick(x = chr, val = c("two","three","four"), pat = FALSE, not = TRUE, nm = FALSE)
# 4) pat = FALSE, not = TRUE, nm = TRUE
pick(x = chr, val = c("2","3","4"), pat = FALSE, not = TRUE, nm = TRUE)
```

```
# 5) pat = TRUE, not = FALSE, nm = FALSE
pick(x = chr, val = "n|v", pat = TRUE, not = FALSE, nm = FALSE)
# 6) pat = TRUE, not = FALSE, nm = TRUE
pick(x = chr, val = "1|5", pat = TRUE, not = FALSE, nm = TRUE)
# 7) pat = TRUE, not = TRUE, nm = FALSE
pick(x = chr, val = "t|r", pat = TRUE, not = TRUE, nm = FALSE)
# 8) pat = TRUE, not = TRUE, nm = TRUE
pick(x = chr, val = c("2|3|4"), pat = TRUE, not = TRUE, nm = TRUE)
datasets <- data()[["results"]][, "Item"]
# actual use cases
pick(x = datasets, val = c("attitude","mtcars","airquality"),
   not = TRUE) # all but the three most common datasets used in `str2str` package examples
pick(x = datasets, val = "state", pat = TRUE) # only datasets that contain "state"
pick(x = datasets, val = "state.*state", pat = TRUE) # only datasets that have
   # "state" twice in their name
pick(x = datasets, val = "US|UK", pat = TRUE) # only datasets that contain
   # "US" or "UK"
pick(x = datasets, val = "^US|^UK", pat = TRUE) # only datasets that start with
   # "US" or "UK"
pick(x = datasets, val = "k.*o|o.*k", pat = TRUE) # only datasets containing both
   # "k" and "o"
```

`rbind<-` *Add Rows to Data Objects*

Description

`` `rbind<-` `` adds rows to data objects as a side effect. The purpose of the function is to replace the need to use dat2 <- rbind(dat1, add1); dat3 <- rbind(dat2, add2); dat4 <- rbind(dat3, add3), etc. For data.frames, it functions similarly to `` `[<-.data.frame` ``, but allows you to specify the location of the rows similar to `append` (vs. `c`) and overwrite rows with the same rownames. For matrices, it offers more novel functionality since `` `[<-.matrix` `` does not exist.

Usage

```
rbind(data, after = nrow(data), row.nm = NULL, overwrite = TRUE) <- value
```

Arguments

`data`	data.frame or matrix of data.
`after`	either an integer vector with length 1 or a character vector of length 1 specifying where to add `value`. If an integer vector, it is the position of a row. If a character vector it is the row with that name. Similar to `append`, use 0L if you want the added rows to be first.
`row.nm`	character vector of length equal to `NROW(value)` that specifies the rownames of `value` once added to `data` as columns. This is an optional argument that defaults to NULL where the pre-existing rownames of `value` are used.

`overwrite`	logical vector of length 1 specifying whether rows from `value` or `row.nm` should overwrite rows in `data` with the same rownames. Note, if `overwrite` = FALSE, R will prevent repeat rownames by adding "1" to the end of the repeat rownames similar to `rbind`.
`value`	data.frame, matrix, or atomic vector to be added as rows to `data`. If a data.frame or matrix, it must have the same ncol as `data`. If an atomic vector, it must have length equal to ncol of `data`.

Details

Some traditional R folks may find this function uncomfortable. R is famous for limiting side effects, except for a few notable exceptions (e.g., `` `[<-` `` and `` `names<-` ``). Part of the reason is that side effects can be computationally inefficient in R. The entire object often has to be re-constructed and re-saved to memory. For example, a more computationally efficient alternative to rbind(dat) <- add1; rbind(dat) <- add2; rbind(dat) <- add3 is dat1 <- do.call(what = rbind, args = list(dat, add1, add2, add3)). However, `` `rbind<-` `` was not created for R programming use when computational efficiency is valued; it is created for R interactive use when user convenience is valued.

Similar to `` `rbind` ``, `` `rbind<-` `` works with both data.frames and matrices. This is because `` `rbind` `` is a generic function with a default method that works with matrices and a data.frame method that works with data.frames. Similar to `` `rbind` ``, if rownames of `value` are not given and `row.nm` is left NULL, then the rownames of the return object are automatically created and can be dissatisfying.

Value

Like other similar functions (e.g., `` `names<-` `` and `` `[<-` ``), `` `rbind<-` `` does not appear to have a return object. However, it technically does as a side effect. The argument `data` will have been changed such that `value` has been added as rows. If a traditional return object is desired, and no side effects, then it can be called like a traditional function: dat2 <- ‘rbind<-‘(dat1, value = add1).

Examples

```
attitude2 <- attitude
rbind(attitude2) <- colMeans(attitude2) # defaults to rownames = as.character(nrow(`data`) + 1)
attitude2 <- attitude2[!(`%in%`(x = row.names(attitude2), table = "31")), ] # logical subset
rbind(attitude2, row.nm = "mean") <- colMeans(attitude2)
attitude2 <- attitude2[-1*(match(x = "mean", table = row.names(attitude2))), ] # position subset
rbind(attitude2, after = "10", row.nm = c("mean","sum")) <-
   rbind(colMeans(attitude2), colSums(attitude2)) # `value` as a matrix
attitude2 <- attitude2[grep(pattern = "mean|sum", x = row.names(attitude2),
   invert = TRUE), ] # rownames subset
attitude2 <- `rbind<-`(data = attitude2, value = colMeans(attitude2)) # traditional call
attitude2 <- as.matrix(attitude, rownames.force = TRUE) # as.matrix.data.frame
rbind(attitude2, after = "10", row.nm = "mean") <- colMeans(attitude2) # `data` as a matrix
# using overwrite
mtcars2 <- mtcars
rownames(mtcars2)
add <- mtcars[c("Mazda RX4","Mazda RX4 Wag","Datsun 710"), ]*11
rbind(mtcars2, overwrite = TRUE) <- add
mtcars2 <- mtcars
rbind(mtcars2, overwrite = FALSE) <- add
```

sn — *Set a Vector's Names as its Elements*

Description

sn sets a vector's names as its elements. It is a simple utility function equal to `setNames(x, nm = as.character(x))`. This is particularly useful when using `lapply` and you want the return object to have `X` as its names.

Usage

```
sn(x)
```

Arguments

x	atomic or list vector.

Value

x with the elements of x as its names.

Examples

```
sn(1:10)
sn(c("one","two","three"))
```

stack2 — *Stack one Set of Variables from Wide to Long*

Description

stack2 converts one set of variables in a data.frame from wide to long format. (If you want to convert *multiple* sets of variables from wide to long, see `reshape`.) It is a modified version of `stack` that 1) adds a column for the rownames, 2) returns character vectors rather than factors, 3) can return additional (repeated) columns, and 4) can order by rownames original positions rather than the variable names being stacked call order.

Usage

```
stack2(
  data,
  select.nm,
  keep.nm = pick(x = names(data), val = select.nm, not = TRUE),
  rtn.el.nm = "el",
  rtn.vrbnames.nm = "vrb_names",
  rtn.rownames.nm = "row_names",
```

```
  order.by.rownames = TRUE,
  stringsAsFactors = FALSE
)
```

Arguments

`data` data.frame of data.

`select.nm` character vector of colnames from `data` specifying the variables to be stacked.

`keep.nm` optional argument containing a character vector of colnames from `data` specifying the additional columns to be included in the return object. These columns are repeated down the data.frame as they are not stacked together. The default is the inclusion of all other columns in `data` other than `select.nm`. If NULL, then no other columns will be included.

`rtn.el.nm` character vector of length 1 specifying the name of the column in the return object that corresponds to the elements of the stacked variables.

`rtn.vrbnames.nm` character vector of length 1 specifying the name of the column in the return object that corresponds to the names of the stacked variables.

`rtn.rownames.nm` character vector of length 1 specifying the name of the column in the return object that corresponds to the rownames.

`order.by.rownames` logical vector of length 1 specifying whether the returned data.frame should be ordered by the positions of the rownames (TRUE) or by the positions of the names of the stacked variables (i.e., `select.nm`). Note, the ordering is by the *positions*, not by alphabetical order. If that is desired, convert the rownames to a (id) column and use `reshape::melt.data.frame`.

`stringsAsFactors` logical vector of length 1 specifying whether the `rtn.vrbnames.nm` and `rtn.rownames.nm` columns should be converted to factors. Note, the factor levels are ordered by positions and not alphabetically (see `v2fct`).

Details

`stack2` is also very similar to `reshape::melt.data.frame`. The differences are that it 1) adds a column for the rownames, 2) returns character vectors rather than factors, and 3) can order by rownames original positions rather than the variable names being stacked call order.

Value

data.frame with nrow = `nrow(data) * length(` `select.nm` `)` from stacking the elements of `data[select.nm]` on top of one another. The first column is the rownames with name `rtn.rownames.nm`, the second column is the names of the stacked variables with name `rtn.vrbnames.nm`, the third column is the stacked elements with name `rtn.el.nm`, and the additional columns are those specified by `keep.nm`.

See Also

`unstack2` `stack` `melt.data.frame`

Examples

```
# general
stack2(data = mtcars, select.nm = c("disp","hp","drat","wt","qsec"),
   keep.nm = c("vs","am"))
stack2(data = mtcars, select.nm = c("disp","hp","drat","wt","qsec"),
   keep.nm = c("vs","am"), rtn.el.nm = "rating", rtn.vrbnames.nm = "item",
   rtn.rownames.nm = "row_names") # change the return object colnames
stack2(data = mtcars, select.nm = c("disp","hp","drat","wt","qsec"),
   keep.nm = pick(x = names(mtcars), val = c("disp","hp","drat","wt","qsec"),
   not = TRUE)) # include all columns from `data` in the return object

# keep options
stack2(data = mtcars, select.nm = c("mpg","cyl","disp")
   ) # default = keep all other variables in `data`
stack2(data = mtcars, select.nm = c("mpg","cyl","disp"), keep = c("gear","carb")
   ) # character vector = keep only specified variables in `data`
stack2(data = mtcars, select.nm = c("mpg","cyl","disp"), keep = NULL,
   ) # NULL = keep no other variables in `data`

# compare to utils:::stack.data.frame and reshape::melt.data.frame
ChickWeight2 <- as.data.frame(datasets::ChickWeight)
ChickWeight2$"Diet" <- as.integer(ChickWeight2$"Diet")
x <- stack(x = ChickWeight2, select = c("weight","Diet")) # does not allow
   # keeping additional columns
y <- reshape::melt(data = ChickWeight2, measure.vars = c("weight","Diet"),
   id.nm = c("Chick","Time"), variable_name = "vrb_names") # does not include
   # rownames and not ordered by rownames
z <- stack2(data = ChickWeight2, select.nm = c("weight","Diet"),
   keep.nm = c("Chick","Time"))
head(x); head(y); head(z)
```

`try_expr` *Add Try to Expression*

Description

`try_expr` evaluates an expression `expr` and returns a list with three elements: 1) return object, 2) warning message, 3) error message. This can be useful when you want to evaluate an expression and are not sure if it will result in a warning and/or error and don't want R to stop if an error does arise.

Usage

```
try_expr(expr, output.class = NULL)
```

Arguments

`expr`	expression
`output.class`	character vector of length 1 specifying the class you want the returned object of `try_expr` to be. The default is NULL for no class.

Details

This function is heavily based on the following StackOverflow post: https://stackoverflow.com/questions/4948361/how-do-i-save-warnings-and-errors-as-output-from-a-function.

Value

list object with three elements: "result" = 1) return object of `expr`, "warning" = warning message, "error" = error message. When an element is not relevant (e.g., no errors), then that element is NULL.

Examples

```
# apply to log()
try_expr(log(1))
try_expr(log(0))
try_expr(log(-1))
try_expr(log("a"))
# return a list where NULL if an error or warning appears
lapply(X = list("positive" = 1, "zero" = 0, "negative" = -1,"letter" = "a"),
   FUN = function(x) {
      log_try <- try_expr(log(x))
      result <- log_try[["result"]]
      warning <- log_try[["warning"]]
      error <- log_try[["error"]]
      if (!(is.null(error))) return(NULL)
      if (!(is.null(warning))) return(NULL)
      return(result)
})
```

`try_fun` *Add Try to Function*

Description

`try_fun` creates a version of the function `fun` that evaluates the function and then returns a list with three elements: 1) return object, 2) warning message, 3) error message. This can be useful when you want to apply a function and are not sure if it will result in a warning and/or error and don't want R to stop if an error does arise.

Usage

```
try_fun(fun, output.class = paste0(deparse(substitute(fun)), ".try"))
```

Arguments

`fun`	function

`output.class` character vector of length 1 specifying the class you want the result from a call to the returned function to be. Note, if `fun` is an annonymous function, then the default will probably not work due to the character limitations of deparsing a function. You can always put down NULL for no class, which will always work with annonymous functions.

Details

This function is heavily based on the following StackOverflow post: https://stackoverflow.com/questions/4948361/how-do-i-save-warnings-and-errors-as-output-from-a-function.

Value

function that returns a list object with three elements: "result" = 1) return object of `fun`, "warning" = warning message, "error" = error message. When an element is not relevant (e.g., no errors), then that element is NULL.

Examples

```
# apply to log()
log.try <- try_fun(log)
log.try(1)
log.try(0)
log.try(-1)
log.try("a")
# return a list where NULL if an error or warning appears
lapply(X = list("positive" = 1, "zero" = 0, "negative" = -1,"letter" = "a"),
   FUN = function(x) {
      log_try <- log.try(x)
      result <- log_try[["result"]]
      warning <- log_try[["warning"]]
      error <- log_try[["error"]]
      if (!(is.null(error))) return(NULL)
      if (!(is.null(warning))) return(NULL)
      return(result)
})
```

`t_list` *Transpose a List*

Description

`t_list` transposes a list, similar to what `t.default` does for matrices. `t_list` assumes the structure of each `x` element is the same. Tests are done to ensure the lengths and names are the same for each `x` element. The returned list has list elements in the same order as in `x[[1]]`.

Usage

```
t_list(x, rtn.atomic = FALSE)
```

Arguments

x	list where each element has the same structure.
rtn.atomic	logical vector of length 1 specifying whether the returned list should be a list of atomic vectors (TRUE) rather than a list of lists (FALSE).

Details

If any element within x has no names (NULL), then the transposition is done based on positions. If all element within x have the same names, then the transposition is done based on those names.

Value

list where each element is from those in x[[1]] and each element of the returned object has a subelement for each element in x.

Examples

```
# modeling example
iris_bySpecies <- split(x = iris, f = iris$"Species")
lmObj_bySpecies <- lapply(X = iris_bySpecies, FUN = function(dat) {
   lm(Sepal.Length ~ Petal.Width, data = dat)})
lmEl_bySpecies <- t_list(lmObj_bySpecies)
summary(lmObj_bySpecies); summary(lmEl_bySpecies)
summary.default(lmEl_bySpecies[[1]]); summary.default(lmEl_bySpecies[[2]])

# no names
lmObj_bySpecies2 <- unname(lapply(X = lmObj_bySpecies, FUN = unname))
lmEl_bySpecies2 <- t_list(lmObj_bySpecies2)
summary(lmObj_bySpecies2); summary(lmEl_bySpecies2)
summary.default(lmEl_bySpecies2[[1]]); summary.default(lmEl_bySpecies2[[2]])
all(unlist(Map(name = lmEl_bySpecies, nameless = lmEl_bySpecies2,
   f = function(name, nameless) all.equal(unname(name), nameless)))) # is everything
   # but the names the same?

# atomic vector example
x <- list("A" = c("a"=1,"b"=2,"c"=3),"B" = c("a"=1,"b"=2,"c"=3),
   "C" = c("a"=1,"b"=2,"c"=3))
t_list(x, rtn.atomic = TRUE)

# names in different positions
x <- list("A" = c("a"=1,"b"=2,"c"=3),"B" = c("b"=2,"a"=1,"c"=3),
   "C" = c("c"=3,"b"=2,"a"=1))
t_list(x, rtn.atomic = TRUE)

# no names
x <- list(c(1,2,3), c(1,2,3), c(1,2,3))
t_list(x, rtn.atomic = TRUE)

# lists with a single element
x <- list("A" = c("a"=1,"b"=2,"c"=3))
t_list(lmObj_bySpecies[1])
```

undim — *Undimension an Object*

Description

undim removes all dimensions from an object. This is particularly useful for simplifying 1D arrays where the dimnames from the array are used for the returned object. Although the function can also be used when dimensions were temporarily (or erroneously) given to an object.

Usage

```
undim(x)
```

Arguments

x	object with dimensions (usually an array of some kind)

Value

x without any dimensions. If x is an array, then the return object will be an atomic vector. If x is a 1D array, then the returned vector will have names = the 1D dimnames.

Examples

```
a <- array(NA, dim = 1, dimnames = list("A"))
v <- undim(a)
str(a); str(v)
```

undimlabel — *Undimlabel an Object*

Description

undimname removes dimlabels from an object. This function is to allow for removing dimlabels from only certain dimensions specified by dims.

Usage

```
undimlabel(x, dims = seq_along(dim(x)))
```

Arguments

x	object with dimlabels (usually an array of some kind)
dims	integer vector of dimension positions or character vector of dimlabels specifying the dimensions for which dimlabels should be removed. Defaults to all dimensions.

Value

x without any dimlabels for the dimensions specified by dims. Consistent with how base R handles removed dimlabels, the removed dimlabels are converted to NA. If all dimlabels are removed, then the dimlabels are empty (aka NULL).

Examples

```
# matrix
m <- array(rep.int(NA, times = 4), dim = c(2,2),
   dimnames = list("lower" = c("a","b"),"UPPER" = c("A","B")))
dimlabels(m)
m2 <- undimlabel(m) # remove dimlabels from both dimensions
dimlabels(m2)
m3 <- undimlabel(m, dims = 1) # remove dimlabels from only the first dimenion
dimlabels(m3)
m4 <- undimlabel(m, dims = "lower")
dimlabels(m4)
all.equal(m3, m4) # same return object
# array
a <- unclass(HairEyeColor)
dimlabels(a)
a2 <- undimlabel(a) # removes dimlabels from all dimensions
dimlabels(a2)
a3 <- undimlabel(a, dims = c(1,2)) # remove dimlabels from only the first and second dimenions
dimlabels(a3)
a4 <- undimlabel(a, dims = c("Hair","Eye"))
dimlabels(a4)
all.equal(a3, a4)
```

---

undimname *Undimname an Object*

Description

undimname removes dimnames from an object. This function is to allow for removing dimnames from only certain dimensions specified by dims.

Usage

```
undimname(x, dims = seq_along(dim(x)), rm.dim.lab = TRUE)
```

Arguments

x	object with dimnames (usually an array of some kind)
dims	integer vector of dimension positions or character vector of dimlabels specifying the dimensions for which dimnames should be removed. Defaults to all dimensions.
rm.dim.lab	logical vector of length 1 specifying whether the dimlabels from the dims dimensions should be removed and converted to NA.

Value

x without any dimnames for the dimensions specified by dims. If a dimlabel existed for the dims dimensions, they will have been removed if rm.dim.lab = TRUE.

Examples

```
# matrix
m <- array(rep.int(NA, times = 4), dim = c(2,2),
   dimnames = list("lower" = c("a","b"),"UPPER" = c("A","B")))
dimnames(m)
m1 <- undimname(m) # remove dimnames from both dimensions
dimnames(m1)
m2 <- undimname(m, rm.dim.lab = FALSE) # keep dimlabels
dimnames(m2)
m3 <- undimname(m, dims = 1) # remove dimnames from only the first dimenion
dimnames(m3)
m4 <- undimname(m, dims = "lower")
dimnames(m4)
all.equal(m3, m4) # same return object
m5 <- undimname(m, dims = 1, rm.dim.lab = FALSE) # keeps dimlabel
dimnames(m5)
# array
a <- unclass(HairEyeColor)
dimnames(a)
a1 <- undimname(a) # removes dimnames from all dimensions
dimnames(a1)
a2 <- undimname(a, rm.dim.lab = FALSE) # keep dimlabels
dimnames(a2)
a3 <- undimname(a, dims = c(1,2)) # remove dimnames from only the first and second dimenions
dimnames(a3)
a4 <- undimname(a, dims = c("Hair","Eye"))
dimnames(a4)
all.equal(a3, a4)
a5 <- undimname(a, dims = c(1,2), rm.dim.lab = FALSE) # keeps dimlabel
dimnames(a5)
```

---

unstack2 *Unstack one Set of Variables from Long to Wide*

Description

unstack2 converts one set of variables in a data.frame from long to wide format. (If you want to convert multiple sets of variables from long to wide, see reshape.) It is a modified version of unstack that 1) requires a column for the rownames of the data.frame (or equivalently an id column with unique values for each row in the wide format) before it was stacked, 2) can retain additional columns not being unstacked, and 3) can order by rownames original positions rather than their alphanumerical order.

Usage

```
unstack2(
  data,
  rownames.nm = "row_names",
  vrbnames.nm = "vrb_names",
  el.nm = "el",
 keep.nm = pick(x = names(data), val = c(rownames.nm, vrbnames.nm, el.nm), not = TRUE),
  add.missing = TRUE,
  rownamesAsColumn = FALSE
)
```

Arguments

`data` data.frame of data containing stacked variables.

`rownames.nm` character vector of length 1 specifying the colname in `data` for whom its unique values correspond to the rows in the return object.

`vrbnames.nm` character vector of length 1 specifying the colname in `` `data` `` that contains the names of the variables to be unstacked.

`el.nm` character vector of length 1 specifying the colname in `data` containing the elements from the variable to be unstacked.

`keep.nm` optional argument containing a character vector of colnames from `data` specifying the additional columns to be included in the return object. The default is all the other columns in the data.frame besides `rownames.nm`, `vrbnames.nm`, and `el.nm`. If NULL, then no additional columns are retained. The `keep.nm` columns will be the last (aka most right) columns in the return object.

`add.missing` logical vector of length 1 specifying whether missing values should be added when unstacking. This will occur if there are unequal number of rows for each variable in the set. If FALSE, an error will be returned when there are an unequal number of rows and missing values would need to be added to create the returned data.frame.

`rownamesAsColumn` logical vector of length 1 specifying whether the unique values in `rownames.nm` column should be a column in the return object (TRUE) or the rownames of the return object (FALSE).

Details

`unstack2` is also very similar to `reshape::cast.data.frame`. The differences are that it 1) can return the rownames as rownames of the returned data.frame rather than an id column, 2) can retain additional columns not being unstacked, and 3) can order by rownames original positions rather than the variable names being stacked call order.

Value

data.frame with nrow = `length(unique(data[[rownames.nm]]))` from unstacking the elements of `el.nm` alongside one another. New columns are created for each unique value in `vrbnames.nm` as well as columns for any colnames additional specified by `keep.nm`. If `rownamesAsColumn` = TRUE,

then the first column is the unique values in `rownames.nm`; otherwise, they are the rownames of the return object (default).

See Also

`stack2` `unstack` `cast`

Examples

```
# ordered by rownames
stacked <- stack2(data = mtcars, select.nm = c("disp","hp","drat","wt","qsec"),
   keep.nm = c("vs","am"), order.by.rownames = TRUE)
x <- unstack2(stacked)
# ordered by vrbnames
stacked <- stack2(data = mtcars, select.nm = c("disp","hp","drat","wt","qsec"),
   keep.nm = c("vs","am"), order.by.rownames = FALSE)
y <- unstack2(stacked)
identical(x, y)

# rownames as a column
z <- unstack2(data = stacked, rownamesAsColumn = TRUE)

# compare to utils:::unstack.data.frame and reshape::cast
stacked <- stack2(data = mtcars, select.nm = c("disp","hp","drat","wt","qsec"),
   keep.nm = c("vs","am"))
x <- unstack(x = stacked, form = el ~ vrb_names) # automatically sorts the colnames alphabetically
y <- reshape::cast(data = stacked, formula = row_names ~ vrb_names,
   value = "el") # automatically sorts the rownames alphabetically
z <- unstack2(stacked) # is able to keep additional variables
head(x); head(y); head(z)

# unequal number of rows for each unique value in `data`[[`vrbnames.nm`]]
# this can occur if you are using unstack2 without having called stack2 right before
row_keep <- sample(1:nrow(stacked), size = nrow(stacked) / 2)
stacked_rm <- stacked[row_keep, ]
unstack2(data = stacked_rm, rownames.nm = "row_names", vrbnames.nm = "vrb_names", el.nm = "el")
## Not run:  # error when `add.missing` = FALSE
   unstack2(data = stacked_rm, rownames.nm = "row_names", vrbnames.nm = "vrb_names",
      el.nm = "el", add.missing = FALSE)

## End(Not run)
```

`v2d` *(Atomic) Vector to Data-Frame*

Description

`v2m` converts an (atomic) vector to a single row or single column data.frame. The benefit of `v2m` over `as.data.frame.vector` is that the dimension along which the vector is binded can be either

rows or columns, whereas in `as.data.frame.vector` it can only be binded along a column, and that `v2m` will keep the names of `v` in the dimnames of the returned data.frame.

Usage

```
v2d(v, along = 2, rtn.dim.nm = NULL, stringsAsFactors = FALSE, check = TRUE)
```

Arguments

`v`	(atomic) vector.
`along`	numeric vector of length 1 that is equal to either 1 or 2 specifying which dimension to bind `v` along. 1 means that `v` is binded along rows (i.e., dimension 1) into a one row data.frame. 2 means that `v` is binded along columns (i.e., dimension 2) into a one column data.frame.
`rtn.dim.nm`	character vector of length 1 specifying what dimname to use for the dimension of length 1 in the returned data.frame. If `along = 1`, then `rtn.dim.nm` will be the single rowname. If `along = 2`, then `rtn.dim.nm` will be the single colname. If NULL, then the dimension of length 1 will be created by default with `data.frame` internally, which will have the rowname be "1" and the colname "V1".
`stringsAsFactors`	logical vector of length 1 specifying if `v` should be converted to a factor in the case that typeof is character.
`check`	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `v` is an atomic vector. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

data.frame with typeof = `typeof(v)`. If `along = 1`, then the dimensions = `c(1L, length(v))` and dimnames = `list(rtn.dim.nm, names(v))`. If `along = 2`, then the dimensions = `c(length(v), 1L)` and dimnames = `list(names(v), rtn.dim.nm)`.

Examples

```
x <- setNames(mtcars[, "mpg"], nm = row.names(mtcars))
v2d(x)
v2d(v = x, along = 1)
v2d(v = x, rtn.dim.nm = "mpg")
```

v2fct *Character Vector to (Unordered) Factor*

Description

v2fct converts a character vector to a (unordered) factor. It goes beyond as.factor by allowing you to specify how you want the levels ordered and whether you want NA treated as a level.

Usage

```
v2fct(
  v,
  order.lvl = "position",
  decreasing = FALSE,
  na.lvl = FALSE,
  check = TRUE
)
```

Arguments

v	character vector. If it is not a character vector (e.g., factor, numeric vector), then it is coerced to a character vector within v2fct.
order.lvl	character vector of length 1 specifying how you want to order the levels of the factor. The options are "alphanum", which sorts the levels alphanumerically (with NA last); "position", which sorts the levels by the position the level first appears; "frequency", which sorts the levels by their frequency. If any frequencies are tied, then the ties are sorted alphanumerically (with NA last).
decreasing	logical vector of length 1 specifying whether the ordering of the levels should be decreasing (TRUE) rather than increasing (FALSE).
na.lvl	logical vector of length 1 specifying if NA should be considered a level.
check	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether v is an atomic vector. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Details

When order.lvl = "alpanum" the levels are sorted alphabetically if letters or a combination of letters and numbers/numerals are in present in v. If only numbers/numerals are present in v, then levels are sorted numerically.

Value

factor of length = length(x) and names = names(x).

Examples

```
# no missing values
state_region <- as.character(state.region)
v2fct(state_region, order.lvl = "position") # in position order
v2fct(v = state_region, order.lvl = "frequency",
   decreasing = TRUE) # most frequent to least frequent
v2fct(v = state_region, order.lvl = "alphanum") # in alphanumerical order
v2fct(v = state_region, na.lvl = TRUE) # na.lvl is inert because no NAs in `v`
# with missing values
state_region <- c(NA_character_, as.character(state.region), NA_character_)
v2fct(v = state_region, order.lvl = "position", na.lvl = TRUE)
v2fct(v = state_region, order.lvl = "frequency", decreasing = TRUE, na.lvl = TRUE)
v2fct(v = state_region, order.lvl = "alphanum", na.lvl = TRUE)
identical(x = v2fct(v = state_region, order.lvl = "alphanum"),
   y = as.factor(state_region)) # equal to as.factor()
# numeric vectors
v2fct(v = round(faithful$"eruptions"), order.lvl = "position")
v2fct(v = round(faithful$"eruptions"), order.lvl = "frequency", decreasing = TRUE)
v2fct(v = round(faithful$"eruptions"), order.lvl = "alphanum")
# cnumeric vectors
cnum <- c("100","99","10","9","1","0","100","99","10","9","1","0")
factor(cnum) # not in numerical order
v2fct(v = cnum, order.lvl = "alphanum") # yes in numerical order
# ties on frequency
v2fct(v = rev(npk$"block"), order.lvl = "alphanum") # ties sorted alphanumerically
v2fct(v = rev(npk$"block"), order.lvl = "position") # no possibility of ties
```

`v2frm` *Character Vector to Formula*

Description

`v2frm` converts a character vector to a formula. The formula has the simple structure of y ~ x1 + x2 + x3 + ... + xn. This function is a simple wrapper for `reformulate`.

Usage

```
v2frm(v, y = 1L, intercept = TRUE)
```

Arguments

`v`	character vector of term(s) and/or response to be included on either side of the returned formula. If it is not a character vector (e.g., factor, numeric vector), then it is coerced to a character vector within `v2frm`. Note, if the length of `v` is 1, then `y.which` must be NULL because at least one term on the right hand side is required, otherwise an error is returned.
`y`	character vector of length 1 specifying the value of the element within `v`, or integer of length 1 specifying the position of the element within `v`, that is the response to be placed on the left hand side of the returned formula. If NULL, then no elements of `v` are treated as response(s) and the left hand side is empty.

`intercept` logical vector of length 1 specifying whether the intercept should be included in the returned formula. The default is TRUE and no change is made to the returned formula. If FALSE, then a -1 is added to the end of the right hand side.

Value

formula with element `v[y]` on the left hand side and `v[-y]` elements on the right hand side (rhs) separated by plus signs (+) with a -1 if `intercept` = FALSE.

Examples

```
v2frm(v = names(attitude))
v2frm(v = names(attitude), y = 7L)
v2frm(v = names(attitude), y = NULL)
v2frm(v = "rating", y = NULL)
try_expr(v2frm(v = "rating")) # error is returned
```

`v2lv` *(Atomic) Vector to List of (Atomic) Vectors*

Description

`v2lv` converts a (atomic) vector to a list of atomic vectors. The default is conversion to a list vector where each element of the list has only one element. The `n.break` argument allows for the input vector to be broken up into larger sections with each section being a list element in the return object.

Usage

```
v2lv(v, use.names = TRUE, n.break = 1L, warn.break = TRUE, check = TRUE)
```

Arguments

`v` (atomic) vector.

`use.names` logical vector of length 1 specifying whether the names from `v` should be retained in the return object.

`n.break` integer vector of length 1 specifying how `v` should be broken up. Every `n.break` elements while seq_along `v`, a new element of the list is created and subsequent elements of `v` are stored there. If `n.break` is not a multiple of `length(v)`, then NAs are appended to the end of `v` to ensure that each list element has (atomic) vectors of the same length. Note, the default is 1L resulting in a list vector.

`warn.break` logical vector of length one specifying whether a warning should be printed if `length(v) / n.break` is not a whole number, which would then result in NAs being appended to the end of the vector before converting to a list.

`check` logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `v` is an atomic vector. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Details

Future versions of this function plan to allow for use similar to the `utils::relist` function to allow reconstruction after flattening a matrix-like object to a single vector.

Value

list of (atomic) vectors that are the elements of `v` broken up according to `n.break`. The list only has names if `v` has names and `n.break` = 1L.

Examples

```
vec <- setNames(object = mtcars[[1]], nm = row.names(mtcars))
v2lv(vec)
v2lv(vec, use.names = FALSE)
vec <- unlist(mtcars)
v2lv(vec, n.break = 32) # n.break > 1L and multiple of length(v)
v2lv(vec, n.break = 30) # n.break > 1L and NOT multiple of length(v)
```

---

v2m *(Atomic) Vector to Matrix*

Description

`v2m` converts an (atomic) vector to a single row or single column matrix. The matrix will be the same typeof as the atomic vector. The benefit of `v2m` over `as.matrix.default` is that the dimension along which the vector is binded can be either rows or columns, whereas in `as.matrix.default` it can only be binded along a column.

Usage

```
v2m(v, along = 2, rtn.dim.nm = NULL, check = TRUE)
```

Arguments

`v`	(atomic) vector.
`along`	numeric vector of length 1 that is equal to either 1 or 2 specifying which dimension to bind `v` along. 1 means that `v` is binded along rows (i.e., dimension 1) into a one row matrix. 2 means that `v` is binded along columns (i.e., dimension 2) into a one column matrix.
`rtn.dim.nm`	character vector of length 1 specifying what dimname to use for the dimension of length 1 in the returned matrix. If `along` = 1, then `rtn.dim.nm` will be the single rowname. If `along` = 2, then `rtn.dim.nm` will be the single colname. If NULL, then the dimension of length 1 has no dimname.
`check`	logical vector of length 1 specifying whether to check the structure of the input arguments. For example, check whether `v` is an atomic vector. This argument is available to allow flexibility in whether the user values informative error messages (TRUE) vs. computational efficiency (FALSE).

Value

matrix with typeof = `typeof(v)`. If `along` = 1, then the dimensions = `c(1L, length(v))` and dimnames = `list(rtn.dim.nm, names(v))`. If `along` = 2, then the dimensions = `c(length(v), 1L)` and dimnames = `list(names(v), rtn.dim.nm)`.

Examples

```
mtcars2 <- as.matrix(mtcars, rownames.force = TRUE) # to make sure dimnames stay in the example
v2m(mtcars2[, "mpg"])
identical(x = v2m(mtcars2[, "mpg"]),
   y = as.matrix(mtcars2[, "mpg"])) # default = as.matrix.default()
v2m(mtcars2[, "mpg"], along = 1)
identical(x = v2m(mtcars2[, "mpg"], along = 1),
   y = t(as.matrix(mtcars2[, "mpg"]))) # = t(as.matrix.default())
v2m(v = mtcars2[, "mpg"], rtn.dim.nm = "mpg")
```

Index

a2d, 4
a2la, 6
a2ld, 7
a2lm, 7
a2v, 8
abind<-, 9
all_diff, 11
all_same, 12
append<-, 13

cast, *87*
cat0, 14
cbind<-, 15
cbind_fill, 17, *19*
cbind_fill_matrix, *17*, 18
codes, 20

d2a, 21
d2d, 23
d2ld, 25
d2lv, 26
d2m, 27
d2v, 28
dimlabels, 30
dimlabels<-, 31

e2l, 32

fct2v, 33

grab, 34

inbtw, 35
is.avector, 36
is.cnumeric, 37
is.colnames, 38
is.Date, 38
is.dummy, 39
is.empty, 40
is.names, 40
is.POSIXct, 41
is.POSIXlt, 42
is.row.names, 42
is.rownames, 43
is.whole, 44

Join, 44
join, *45*
join_all, *45*

la2a, 47
laynames, 48
ld2a, 49
ld2d, 50
ld2v, 52
lm2a, 54
lm2d, 55
lm2v, 57
lv2d, 59
lv2m, 61
lv2v, 63

m2d, 65
m2lv, 66
m2v, 67
melt.data.frame, *78*
merge, *45*

ndim, 68
nlay, 69
not.colnames, 70
not.names, 70
not.row.names, 71
not.rownames, 71

order.custom, 72

pick, 73

rbind.fill, *17*
rbind.fill.matrix, *19*
rbind<-, 75

reshape, *85*

sn, 77
stack, *78*
stack2, 77, *87*
str2str (str2str-package), 3
str2str-package, 3

t_list, 81
try_expr, 79
try_fun, 80

undim, 83
undimlabel, 83
undimname, 84
unstack, *87*
unstack2, *78*, 85

v2d, 87
v2fct, 89
v2frm, 90
v2lv, 91
v2m, 92