Package 'expss'

October 19, 2016

Type Package

Title Some Useful Functions from Spreadsheets and 'SPSS' Statistics

Version 0.5.5

Date 2016-10-19

Author Gregory Demin

Maintainer Gregory Demin <gdemin@gmail.com>

URL https://github.com/gdemin/expss

BugReports https://github.com/gdemin/expss/issues

Depends R (>= 3.3)

Imports foreign, utils, stats

Suggests knitr

Description Package implements several popular functions from Excel ('COUNTIF', 'VLOOKUP', etc.) and 'SPSS' Statistics ('RECODE', 'COUNT', etc.). Also there are functions for basic tables with value labels/variable labels support. Package aimed to help people to move data processing from Excel/'SPSS' to R.

VignetteBuilder knitr

LazyData yes

License GPL (>= 2)

RoxygenNote 5.0.1

NeedsCompilation no

Repository CRAN

Date/Publication 2016-10-19 00:39:24

R topics documented:

`add_rows` *Add rows to data.frame/matrix/table*

Description

`add_rows` is similar to rbind but it handles non-matching column names. `%add_rows%` is an infix version of `add_rows`. There is also special method for the results of `cro_*/fre`. `.add_rows` is version for addding rows to default dataset. See default_dataset.

Usage

```
add_rows(...)

## S3 method for class 'data.frame'
add_rows(..., nomatch_columns = c("add", "drop", "stop"))

x %add_rows% y

.add_rows(..., nomatch_columns = c("add", "drop", "stop"))
```

Arguments

...	data.frame/matrix/table for binding
nomatch_columns	action if there are non-matching columns between data.frames. Possible values are "add", "drop", "stop". "add" will combine all columns, "drop" will leave only common columns, "stop" will raise an error.
x	data.frame/matrix/table for binding
y	data.frame/matrix/table for binding

Value

See rbind, cro, fre

Examples

```
a = data.frame(x = 1:5, y = 6:10)
b = data.frame(y = 6:10, z = 11:15)

add_rows(a, b) # x, y, z
a %add_rows% b # the same result

add_rows(a, b, nomatch_columns = "drop")  # y

# simple tables
data(mtcars)

mtcars = modify(mtcars, {
            var_lab(mpg) = "Miles/(US) gallon"
            var_lab(vs) = "vs"
            val_lab(vs) = c("V-engine" = 0, "Straight engine" = 1)
            var_lab(am) = "am"
            val_lab(am) = c("automatic transmission" = 1, "manual transmission" = 0)
            var_lab(gear) = "gear"
            var_lab(carb) = "carb"
})

tab_mean = with(mtcars, cro_mean(mpg, am))
tab_percent = with(mtcars, cro_cpct(vs, am))
```

```
tab_mean %add_rows% tab_percent
```

`as.labelled` *Recode vector into integer vector with value labels*

Description

Recode vector into integer vector with value labels

Usage

```
as.labelled(x, label = NULL)
```

Arguments

`x`	numeric vector/character vector/factor
`label`	optional variable label

Value

integer vector with labels

Examples

```
character_vector = c("one", "two",  "two", "three")
as.labelled(character_vector, label = "Numbers")
```

`category` *Convert dichotomy matrix/data.frame to matrix/data.frame with category encoding*

Description

Convert dichotomy matrix/data.frame to matrix/data.frame with category encoding

Usage

```
category(x, prefix = NULL, use_var_lab = TRUE, counted_value = 1,
  compress = TRUE)

category_df(x, prefix = NULL, use_var_lab = TRUE, counted_value = 1,
  compress = TRUE)
```

Arguments

x	Dichotomy matrix (usually with 0,1 coding).
prefix	If not is NULL then column names will be added in the form prefix+column number.
use_var_lab	logical If TRUE then we will try to use variable labels as value labels instead of column names.
counted_value	Vector. Values that should be considered as indicator of category presence. By default it equals to 1.
compress	Logical. Should we drop columns with all NA?

Value

Matrix or data.frame with numeric values that correspond to column numbers of counted values. Column names of x or variable labels are added as value labels.

See Also

dichotomy for reverse conversion.

Examples

```
set.seed(123)

# Let's imagine it's matrix of consumed products
dichotomy_matrix = matrix(sample(0:1,40,replace = TRUE,prob=c(.6,.4)),nrow=10)
colnames(dichotomy_matrix) = c("Milk","Sugar","Tea","Coffee")
category(dichotomy_matrix,compress=FALSE) # uncompressed version
category_matrix=category(dichotomy_matrix)

 # should be TRUE
identical(val_lab(category_matrix),c(Milk = 1L,Sugar = 2L,Tea = 3L,Coffee = 4L))
all(dichotomy(category_matrix,use_na = FALSE)==dichotomy_matrix)

# with prefix
category(dichotomy_matrix, prefix = "products_")

# data.frame with variable labels
dichotomy_dataframe = as.data.frame(dichotomy_matrix)
colnames(dichotomy_dataframe) = paste0("product_", 1:4)
var_lab(dichotomy_dataframe[[1]]) = "Milk"
var_lab(dichotomy_dataframe[[2]]) = "Sugar"
var_lab(dichotomy_dataframe[[3]]) = "Tea"
var_lab(dichotomy_dataframe[[4]]) = "Coffee"

category_df(dichotomy_dataframe, prefix = "products_")
```

compute *Experimental functions for operations with default dataset*

Description

Workflow for these functions is rather simple. You should set up default data.frame with default_dataset and then operate with it without any reference to your data.frame. There are two kinds of operations. The first kind modify default dataset, the second kind will be evaluated in the context of the default dataset but doesn't modify it. It is not recommended to use one of these functions in the scope of another of these functions. By now their performance is not so high, especially `.do_if`/`.modify_if` can be very slow.

Usage

```
.modify(expr)

.modify_if(cond, expr)

.do_if(cond, expr)

.compute(expr)

.with(expr, ...)

.val_lab(...)

.var_lab(...)

.set_var_lab(x, ...)

.set_val_lab(x, ...)

.add_val_lab(x, ...)

.if_val(x, ...)

.recode(x, ...)

.fre(...)

.cro(...)

.cro_cpct(...)

.cro_rpct(...)

.cro_tpct(...)
```

```
.cro_mean(...)

.cro_sum(...)

.cro_median(...)

.cro_fun(...)

.cro_fun_df(...)

.set(varnames, value = NA)
```

Arguments

`expr`	set of expressions in curly brackets which will be evaluated in the context of default dataset
`cond`	logical vector/expression
`...`	further arguments
`x`	vector/data.frame - variable names in the scope of default dataset
`varnames`	character vector. Names of variables which should be created in the default dataset. Expressions inside backticks in `varnames` will be expanded as with subst.
`value`	value/vector/matrix/data.frame. Value for newly created/existing variables.

Details

Functions which modify default dataset:

- `.modify` Add and modify variables inside default data.frame. See modify.
- `.compute` Shortcut for `.modify`. Name is inspired by SPSS COMPUTE operator. See modify.
- `.modify_if` Add and modify variables inside subset of default data.frame. See modify_if.
- `.do_if` Shortcut for `.modify_if`. Name is inspired by SPSS DO IF operator. See modify_if.
- `.where` Leave subset of default data.frame which meet condition. See where, subset.
- `.set_var_lab` Set variable label in the default data.frame. See set_var_lab.
- `.set_val_lab` Set value labels for variable in the default data.frame. See set_val_lab.
- `.add_val_lab` Add value labels for variable in the default data.frame. See add_val_lab.
- `.if_val` Change, rearrange or consolidate the values of an existing variable inside default data.frame. See if_val.
- `.recode` Shortcut for `.if_val`. Name is inspired by SPSS RECODE. See if_val.
- `.set` Set variables values in the default dataset with given names filled with `value`. It is possible to set multiple variables at once. Expressions inside backticks in `varnames` will be expanded as with subst. `set` (without dot) is also available inside `.compute`, `.modify`, `.modify_if`, `.do_if`, modify, modify_if.

Other functions:

- `.var_lab` Return variable label from default dataset. See var_lab.
- `.val_lab` Return value labels from default dataset. See val_lab.
- `.fre` Simple frequencies of variable in the default data.frame. See fre.
- `.cro`/`.cro_cpct`/`.cro_rpct`/`.cro_tpct` Simple crosstabulations of variable in the default data.frame. See cro.
- `.cro_mean`/`.cro_sum`/`.cro_median`/`.cro_fun`/`.cro_fun_df` Simple crosstabulations of variable in the default data.frame. See cro_fun.
- `.with` Evaluate arbitrary expression in the context of data.frame. See with.

Examples

```
data(mtcars)

default_dataset(mtcars) # set mtcars as default dataset

# calculate new variables
.compute({
    mpg_by_am = ave(mpg, am, FUN = mean)
    hi_low_mpg = ifs(mpg<mean(mpg) ~ 0, default = 1)
})

# set labels
.set_var_lab(mpg, "Miles/(US) gallon")
.set_var_lab(cyl, "Number of cylinders")
.set_var_lab(disp, "Displacement (cu.in.)")
.set_var_lab(hp, "Gross horsepower")
.set_var_lab(mpg_by_am, "Average mpg for transimission type")
.set_var_lab(hi_low_mpg, "Miles per gallon")
.set_val_lab(hi_low_mpg, ml_left("
                                  0 Low
                                  1 High
                                  "))

.set_var_lab(vs, "Engine")
.set_val_lab(vs, ml_left("
                         0 V-engine
                         1 Straight engine
                         "))

.set_var_lab(am, "Transmission")
.set_val_lab(am, ml_left("
                         0 automatic
                         1 manual
                         "))

# calculate frequencies
.fre(hi_low_mpg)
.cro(cyl, hi_low_mpg)
.cro_mean(mpg, am)
```

```
.cro_mean(data.frame(mpg, disp, hp), vs)

# disable default dataset
default_dataset(NULL)

# Example of .recode

data(iris)

default_dataset(iris) # set iris as default dataset

.recode(Sepal.Length, lo %thru% median(Sepal.Length) ~ "small", other ~ "large")

.fre(Sepal.Length)

# example of .do_if

.do_if(Species == "setosa",{
     Petal.Length = NA
     Petal.Width = NA
})

.cro_mean(data.frame(Petal.Length, Petal.Width), Species)

# disable default dataset
default_dataset(NULL)
```

`count_if` *Count/sum/average/other functions on values that meet a criterion*

Description

These functions calculate count/sum/average/etc on values that meet a criterion that you specify. `apply_if_*` apply custom functions. There are different flavors of these functions: `*_if` work on entire dataset/matrix/vector, `*_row_if` works on each row and `*_col_if` works on each column.

Usage

```
count_if(criterion = NULL, ...)

count_row_if(criterion = NULL, ...)

count_col_if(criterion = NULL, ...)

criterion %in_row% x

criterion %in_col% x

sum_if(criterion = NULL, ..., data = NULL)
```

```
sum_row_if(criterion = NULL, ..., data = NULL)

sum_col_if(criterion = NULL, ..., data = NULL)

mean_if(criterion = NULL, ..., data = NULL)

mean_row_if(criterion = NULL, ..., data = NULL)

mean_col_if(criterion = NULL, ..., data = NULL)

sd_if(criterion = NULL, ..., data = NULL)

sd_row_if(criterion = NULL, ..., data = NULL)

sd_col_if(criterion = NULL, ..., data = NULL)

median_if(criterion = NULL, ..., data = NULL)

median_row_if(criterion = NULL, ..., data = NULL)

median_col_if(criterion = NULL, ..., data = NULL)

max_if(criterion = NULL, ..., data = NULL)

max_row_if(criterion = NULL, ..., data = NULL)

max_col_if(criterion = NULL, ..., data = NULL)

min_if(criterion = NULL, ..., data = NULL)

min_row_if(criterion = NULL, ..., data = NULL)

min_col_if(criterion = NULL, ..., data = NULL)

apply_row_if(fun, criterion = NULL, ..., data = NULL)

apply_col_if(fun, criterion = NULL, ..., data = NULL)
```

Arguments

criterion	Vector with counted values, logical vector/matrix or function. See details and examples.
...	Data on which criterion will be applied. Vector, matrix, data.frame, list. Shorter arguments will be recycled.
x	Counted values or criterion for counting. Vector, matrix, data.frame, list, function. Shorter columns in list will be recycled.

data	Data on which function will be applied. Doesn't applicable to count_*_if functions. If omitted then function will be applied on the ... argument.
fun	Custom function that will be applied based on criterion.

Details

Possible type for criterion argument:

- vector/single value All values in ... which equal to elements of vector in criteria will be used as function argument.
- function Values for which function gives TRUE will be used as function argument. There are some special functions for convenience (e. g. gt(5) is equivalent ">5" in spreadsheet) - see criteria.
- logical vector/matrix/data.frame Values for which element of criterion equals to TRUE will be used as function argument. Logical vector will be recycled across all columns of ...data. If criteria is logical matrix/data.frame then column from this matrix/data.frame will be used for corresponding column/element of ...data. Note that this kind of criterion doesn't use ... so ... can be used instead of data argument.

If criterion is missing (or is NULL) then non-NA's values will be used for function.

count* and %in*% never returns NA's. Other functions remove NA's before calculations (as na.rm = TRUE in base R functions).

Function criterion should return logical vector of same size and shape as its argument. This function will be applied to each column of supplied data and TRUE results will be used. There is asymmetrical behavior in *_row_if and *_col_if for function criterion: in both cases function criterion will be applied columnwise.

Value

*_if return single value (vector of length 1). *_row_if returns vector for each row of supplied arguments. *_col_if returns vector for each column of supplied arguments. %in_row%/%in_col% return logical vector - indicator of presence of criterion in each row/column.

Examples

```
set.seed(123)
dfs = as.data.frame(
       matrix(sample(c(1:10,NA),30,replace = TRUE),10)
)

result  = modify(dfs, {
             # count 8
             exact = count_row_if(8, V1, V2, V3)
             # count values greater than 8
             greater = count_row_if(gt(8), V1, V2, V3)
             # count integer values between 5 and 8, e. g. 5, 6, 7, 8
             integer_range = count_row_if(5:8, V1, V2, V3)
             # count values between 5 and 8
             range = count_row_if(5 %thru% 8, V1, V2, V3)
             # count NA
```

```
            na = count_row_if(is.na, V1, V2, V3)
            # count not-NA
            not_na = count_row_if(, V1, V2, V3)
            # are there any 5 in each row?
            has_five = 5 %in_row% cbind(V1, V2, V3)
        })
result

mean_row_if(6, dfs$V1, data = dfs)
median_row_if(gt(2), dfs$V1, dfs$V2, dfs$V3)
sd_row_if(5 %thru% 8, dfs$V1, dfs$V2, dfs$V3)

if_na(dfs) = 5 # replace NA

# custom apply
apply_col_if(prod, gt(2), dfs$V1, data = dfs) # product of all elements by columns
apply_row_if(prod, gt(2), dfs$V1, data = dfs) # product of all elements by rows

# Examples borrowed from Microsoft Excel help for COUNTIF
df1 = data.frame(
    a=c("apples",   "oranges",     "peaches",     "apples"),
    b = c(32, 54, 75, 86)
)

count_if("apples",df1$a) # 2

count_if("apples",df1) # 2

with(df1,count_if("apples",a,b)) # 2

count_if(gt(55),df1$b) # greater than 55 = 2

count_if(neq(75),df1$b) # not equal 75 = 3

count_if(gte(32),df1$b) # greater than or equal 32 = 4

count_if(gt(32) & lt(86),df1$b) # 2

# count only integer values between 33 and 85
count_if(33:85,df1$b) # 2

# values with letters
count_if(regex("^[A-z]+$"),df1) # 4

# values that started on 'a'
count_if(regex("^a"),df1) # 2

# count_row_if
count_row_if(regex("^a"),df1) # c(1,0,0,1)

'apples' %in_row% df1  # c(TRUE,FALSE,FALSE,TRUE)

# Some of Microsoft Excel examples for SUMIF/AVERAGEIF/etc
```

```
dfs = read.csv(
    text = "
    property_value,commission,data
    100000,7000,250000
    200000,14000,
    300000,21000,
    400000,28000,"
)

# Sum of commision for property value greater than 160000
with(dfs, sum_if(gt(160000), property_value, data = commission)) # 63000

# Sum of property value greater than 160000
with(dfs, sum_if(gt(160000), property_value)) # 900000

# Sum of commision for property value equals to 300000
with(dfs, sum_if(300000, property_value, data = commission)) # 21000

# Sum of commision for property value greater than first value of data
with(dfs, sum_if(gt(data[1]), property_value, data = commission)) # 49000


dfs = data.frame(
    category = c("Vegetables", "Vegetables", "Fruits", "", "Vegetables", "Fruits"),
    food = c("Tomatoes", "Celery", "Oranges", "Butter", "Carrots", "Apples"),
    sales = c(2300, 5500, 800, 400, 4200, 1200),
    stringsAsFactors = FALSE
)

# Sum of sales for Fruits
with(dfs, sum_if("Fruits", category, data = sales)) # 2000

# Sum of sales for Vegetables
with(dfs, sum_if("Vegetables", category, data = sales)) # 12000

# Sum of sales for food which is ending on 'es'
with(dfs, sum_if(perl("es$"), food, data = sales)) # 4300

# Sum of sales for empty category
with(dfs, sum_if("", category, data = sales))  # 400


dfs = read.csv(
    text = "
    property_value,commission,data
    100000,7000,250000
    200000,14000,
    300000,21000,
    400000,28000,"
)

# Commision average for comission less than 23000
with(dfs, mean_if(lt(23000), commission)) # 14000
```

```
# Property value average for property value less than 95000
with(dfs, mean_if(lt(95000), property_value)) #  NaN

# Commision average for property value greater than 250000
with(dfs, mean_if(gt(250000), property_value, data = commission)) # 24500


dfs = data.frame(
    region = c("East", "West", "North", "South (New Office)",  "MidWest"),
    profits = c(45678, 23789, -4789, 0, 9678),
    stringsAsFactors = FALSE
)


# Mean profits for 'west' regions
with(dfs, mean_if(fixed("West"), region, data = profits)) # 16733.5


# Mean profits for regions wich doesn't contain New Office
with(dfs, mean_if(!fixed("(New Office)"), region, data = profits))  # 18589


dfs = read.csv(
    text = '
    grade,weight
    89,1
    93,2
    96,2
    85,3
    91,1
    88,1'
    ,stringsAsFactors = FALSE
)

# Minimum gade for weight equals to 1
with(dfs, min_if(1, weight, data = grade)) # 88


# Maximum gade for weight equals to 1
with(dfs, max_if(1, weight, data = grade)) #91


# Example with offset
dfs = read.csv(
    text = '
    weight,grade
    10,b
    11,a
    100,a
    111,b
    1,a
```

```
    1,a'
    ,stringsAsFactors = FALSE
)

with(dfs, min_if("a", grade[2:5], data = weight[1:4])) # 10
```

`criteria` *Criteria functions*

Description

These functions returns criteria functions which could be used in different situation - see keep, except, if_val, na_if, %i%, %d%, count_if, match_row etc. For example, `gt(5)` returns function which tests whether its argument greater than five. `fixed("apple")` return function which tests whether its argument contains "apple". Logical operations (|, &, !, xor) defined for these functions. List of functions:

- `gt` greater than
- `gte`/`ge` greater than or equal
- `eq` equal
- `neq`/`ne` not equal
- `lt` less than
- `lte`/`le` less than or equal
- `thru` checks whether value is inside interval. `thru(0,1)` is equivalent of `x>=0 & x<=1` or `gte(0) & lte(1)`
- `%thru%` infix version of `thru`, e. g. `0 %thru% 1`
- `regex` use POSIX 1003.2 extended regular expressions. For details see grepl
- `perl` perl-compatible regular expressions. For details see grepl
- `fixed` pattern is a string to be matched as is. For details see grepl
- `to` returns function which gives TRUE for all elements of vector before the first occurrence of `x` and for `x`.
- `from` returns function which gives TRUE for all elements of vector after the first occurrence of `x` and for `x`. `from` and `to` are intended for usage with keep and except.
- `not_na` return TRUE for all non-NA elements of vector.
- `other` return TRUE for all elements of vector. It is intended for usage with `if_val`, `keep`, `except`

Usage

```
eq(x)

neq(x)

ne(x)

lt(x)

gt(x)

lte(x)

le(x)

gte(x)

ge(x)

perl(pattern, ignore.case = FALSE, useBytes = FALSE)

regex(pattern, ignore.case = FALSE, useBytes = FALSE)

fixed(pattern, ignore.case = FALSE, useBytes = FALSE)

thru(lower, upper)

lower %thru% upper

from(x)

to(x)

not_na(x)

other(x)
```

Arguments

x	vector
pattern	character string containing a regular expression (or character string for `fixed`) to be matched in the given character vector. Coerced by as.character to a character string if possible.
ignore.case	logical see grepl
useBytes	logical see grepl
lower	vector/single value - lower bound of interval
upper	vector/single value - upper bound of interval

Value

function of class 'criterion' which tests its argument against condition and return logical value

See Also

keep, except, count_if, match_row, if_val, na_if, %i%, %d%

Examples

```
# operations on vector
1:6 %d% gt(4) # 1:4

1:6 %d% (1 | gt(4)) # 2:4

letters %i% (fixed("a") | fixed("z")) # a, z

letters %i% from("w")  # w, x, y, z

letters %i% to("c")  # a, b, c

letters %i% (from("b") & to("e"))  # b, d, e

c(1, 2, NA, 3) %i% other # c(1, 2, 3)

# examples with count_if
df1 = data.frame(
    a=c("apples",   "oranges",     "peaches",     "apples"),
    b = c(32, 54, 75, 86)
)

count_if(gt(55), df1$b) # greater than 55 = 2

count_if(neq(75), df1$b) # not equal 75 = 3

count_if(gte(32), df1$b) # greater than or equal 32 = 4

count_if(gt(32) & lt(86), df1$b) # greater than 32 and less than 86 = 2

# via different kinds of 'thru'
count_if(thru(35, 80), df1$b) # greater than or equals to 35 and less than or equals to 80 = 2
# infix version
count_if(35 %thru% 80, df1$b) # greater than or equals to 35 and less than or equals to 80 = 2

# values that started on 'a'
count_if(regex("^a"),df1) # 2

# count_row_if
count_row_if(regex("^a"),df1) # c(1,0,0,1)

# examples with 'keep' and 'except'

data(iris)
```

```
iris %keep% to("Petal.Width") # column 'Species' will be removed

# 'Sepal.Length', 'Sepal.Width' will be left
iris %except% from("Petal.Length")

# if_val examples
# From SPSS: RECODE QVAR(1 THRU 5=1)(6 THRU 10=2)(11 THRU HI=3)(ELSE=0).
set.seed(123)
qvar = sample((-5):20, 50, replace = TRUE)
if_val(qvar, 1 %thru% 5 ~ 1, 6 %thru% 10 ~ 2, 11 %thru% hi ~ 3, other ~ 0)
# the same result
if_val(qvar, 1 %thru% 5 ~ 1, 6 %thru% 10 ~ 2, gte(11) ~ 3, other ~ 0)
```

`default_dataset` *Get or set reference to default dataset. Experimental feature.*

Description

Use data.frame or data.frame name to set it as default. Use NULL as an argument to disable default dataset. If argument is missing then function will return reference to default dataset. Use ref to modify it. Also see .compute for usage patterns.

Usage

```
default_dataset(x)
```

Arguments

x	data.frame or data.frame name which we want to make default for some operations.

Value

formula reference to default dataset or NULL

See Also

ref

Examples

```
data(iris)
default_iris = iris
default_dataset(default_iris) # set default dataset

.compute({
```

```
    new_col = 1
    Sepal.Length = Sepal.Length*2
})

# for comparison

iris$new_col = 1
iris$Sepal.Length = iris$Sepal.Length*2
identical(iris, default_iris) # should be TRUE

default_dataset(NULL) # disable default dataset
```

`dichotomy` *Convert variable (possibly multiple choice question) to matrix of dummy variables.*

Description

This function converts variable/multiple response variable(matrix/data.frame) with category encoding into matrix with dichotomy encoding (0/1) suited for most statistical analysis, e. g. clustering, factor analysis, linear regression and so on.

- `dichotomy` returns matrix with 0, 1 and possibly NA.
- `dichotomy1` drops last column in dichotomy matrix. It is useful in many cases because any column of such matrix usually is linear combinations of other columns.
- `dummy` is another shortcut for `dichotomy`.
- `*_df` are the same functions as `dichotomy` etc. but return data.frame instead of matrix.

Usage

```
dichotomy(x, prefix = NULL, keep_unused = FALSE, use_na = TRUE,
  keep_values = NULL, keep_labels = NULL, drop_values = NULL,
  drop_labels = NULL)

dichotomy1(x, prefix = NULL, keep_unused = FALSE, use_na = TRUE,
  keep_values = NULL, keep_labels = NULL, drop_values = NULL,
  drop_labels = NULL)

dichotomy1_df(x, prefix = NULL, keep_unused = FALSE, use_na = TRUE,
  keep_values = NULL, keep_labels = NULL, drop_values = NULL,
  drop_labels = NULL)

dichotomy_df(x, prefix = NULL, keep_unused = FALSE, use_na = TRUE,
  keep_values = NULL, keep_labels = NULL, drop_values = NULL,
  drop_labels = NULL)
```

```
dummy(x, prefix = NULL, keep_unused = FALSE, use_na = TRUE,
  keep_values = NULL, keep_labels = NULL, drop_values = NULL,
  drop_labels = NULL)

dummy_df(x, prefix = NULL, keep_unused = FALSE, use_na = TRUE,
  keep_values = NULL, keep_labels = NULL, drop_values = NULL,
  drop_labels = NULL)
```

Arguments

x	vector/factor/matrix/data.frame.
prefix	character. If it is not NULL it instead of labels will be used prefix+values.
keep_unused	Logical. Should we create columns for unused value labels/factor levels.
use_na	Logical. Should we use NA for rows with all NA or use 0's instead.
keep_values	Numeric/character. Values that should be kept. By default all values will be kept.
keep_labels	Numeric/character. Labels/levels that should be kept. By default all labels/levels will be kept.
drop_values	Numeric/character. Values that should be dropped. By default all values will be kept. Ignored if keep_values/keep_labels are provided.
drop_labels	Numeric/character. Labels/levels that should be dropped. By default all labels/levels will be kept. Ignored if keep_values/keep_labels are provided.

Value

matrix or data.frame with 0,1 which column names are value labels or values with prefix. If label doesn't exist for particular value then this value will be used as column name.

See Also

category for reverse conversion.

Examples

```
# toy example
# brands - multiple response question
# Which brands do you use during last three months?
set.seed(123)
brands = t(replicate(20,sample(c(1:5,NA),4,replace = FALSE)))
# score - evaluation of tested product
score = sample(-1:1,20,replace = TRUE)
var_lab(brands) = "Used brands"
val_lab(brands) = make_labels("
                              1 Brand A
                              2 Brand B
                              3 Brand C
                              4 Brand D
                              5 Brand E
```

```
                               ")

var_lab(score) = "Evaluation of tested brand"
val_lab(score) = make_labels("
                             -1 Dislike it
                              0 So-so
                              1 Like it
                             ")

# percentage of used brands
colMeans(dichotomy(brands))

# percentage of brands within each score
cro_mean(dichotomy(brands), score)
# the same as
cro_cpct(brands, score)

# percentage of brands within each score - same numbers
aggregate(dichotomy(brands) ~ f(score), FUN = mean)

# or, same result in another form
by(dichotomy(brands), f(score), FUN = colMeans)

# customer segmentation by used brands
kmeans(dichotomy(brands),3)

# model of influence of used brands on evaluation of tested product
summary(lm(score ~ dichotomy(brands)))

# prefixed data.frame
dichotomy_df(brands, prefix = "brand_")
```

`dtfrm` *Make data.frame without conversion to factors and without fixing names*

Description

`dtfrm` and `as.dtfrm` are shortcuts to `data.frame` and `as.data.frame` with stringsAsFactors = FALSE, check.names = FALSE. `lst` creates list with names. `.dtfrm`, `.lst` the same as above but work in the scope of default dataset.

Usage

```
dtfrm(...)

as.dtfrm(x, ...)

lst(...)
```

```
.dtfrm(...)

.lst(...)
```

Arguments

...	objects, possibly named
x	object to be coerced

Value

data.frame/list

See Also

default_dataset, data.frame, as.data.frame, list

Examples

```
# see the difference
df1 = data.frame(a = letters[1:3], "This is my long name" = 1:3)
df2 = dtfrm(a = letters[1:3], "This is my long name" = 1:3)

str(df1)
str(df2)

# lst
a = 1:3
b = 3:1

list1 = list(a, b)
list2 = lst(a, b)

str(list1)
str(list2)

data(iris)
default_dataset(iris)

.dtfrm(Sepal.Width,  Sepal.Length)
.lst(Sepal.Width,  Sepal.Length)
```

expss *expss: Some Useful Functions from Spreadsheets and SPSS Statistics*

Description

'expss' package implements some popular functions from spreadsheets and SPSS Statistics software. Implementations are not complete copies of their originals. I try to make them consistent with other R functions. See examples in vignette and in help.

Excel

- IF ifelse
- AVERAGE mean_row
- SUM sum_row
- MIN min_row
- MAX max_row
- VLOOKUP vlookup
- COUNTIF count_if
- AVERAGEIF mean_row_if
- SUMIF sum_row_if
- MINIF min_row_if
- MAXIF max_row_if
- IFS ifs
- IFNA if_na
- MATCH match_row
- INDEX index_row

SPSS

- COMPUTE modify
- DO IF modify_if
- RECODE if_val
- COUNT count_row_if
- VARIABLE LABELS var_lab
- VALUE LABELS val_lab
- ANY any_in_row
- FREQUENCIES fre
- CROSSTABS cro

f *Convert labelled variable to factor*

Description

f converts labelled variable to factor. Factor levels are constructed as values labels. If label doesn't exist for particular value then this value remain as is - so there is no information lost. This levels look like as "Variable_label|Value label". If variable doesn't have labels then usual factor will be applied.

Usage

```
f(x, ...)
```

Arguments

x	a vector of data with labels.
...	optional arguments for factor

Value

an object of class factor. For details see base factor documentation.

See Also

values2labels, names2labels, val_lab, var_lab. Materials for base functions: factor, as.factor, ordered, as.ordered

Examples

```
data(mtcars)

var_lab(mtcars$am) = "Transmission"
val_lab(mtcars$am) = c(automatic = 0, manual=1)

## Not run:
plot(f(mtcars$am))

## End(Not run)

table(f(mtcars$am))

summary(lm(mpg ~ am, data = mtcars)) # no labels
summary(lm(mpg ~ f(am), data = mtcars)) # with labels
summary(lm(mpg ~ f(unvr(am)), data = mtcars)) # without variable label
```

fre *Simple frequencies and crosstabs with support of labels, weights and multiple response variables.*

Description

- fre returns data.frame with six columns: labels or values, counts, valid percent (excluding NA), percent (with NA), percent of responses(for single-column x it equals to valid percent) and cumulative percent of responses.
- cro returns data.frame with counts (possibly weighted) with column and row totals.
- cro_pct, cro_cpct, cro_rpct return data.frame with table/column/row percent with column and row totals. There are always weighted counts instead of margin with 100%. Empty labels/factor levels are removed from results of these functions. Base for multiple response (x is data.frame) percent is number of valid cases (not sum of responses) so sum of percent may be greater than 100. Case is considered as valid if it has at least one non-NA value.
- cro_mean, cro_sum, cro_median return data.frame with mean/sum/median. Empty labels/factor levels are removed from results of these functions. NA's are always omitted.
- cro_fun, cro_fun_df return data.frame with custom summary statistics defined by 'fun' argument. Empty labels/factor levels in predictor are removed from results of these functions. NA's treatment depends on your 'fun' behavior. To use weight you should have 'weight' argument in 'fun' and some logic for its proccessing inside.cro_fun applies 'fun' on each column in 'x' separately, cro_fun_df gives to 'fun' x as a whole data.frame. So cro_fun(iris[, -5], iris$Species, fun = mean) gives the same result as cro_fun_df(iris[, -5], iris$Spec colMeans). For cro_fun_df names of 'x' will converted to labels if they are available before 'fun' is applied. You should take care to return from 'fun' rectangular object with appropriate row/column names - they will be used in final result as labels.

Usage

```
fre(x, weight = NULL)

cro(x, predictor, weight = NULL)

cro_cpct(x, predictor, weight = NULL)

cro_rpct(x, predictor, weight = NULL)

cro_tpct(x, predictor, weight = NULL)

cro_mean(x, predictor, weight = NULL)

cro_sum(x, predictor, weight = NULL)

cro_median(x, predictor)
```

```
cro_fun(x, predictor, fun, ..., weight = NULL)

cro_fun_df(x, predictor, fun, ..., weight = NULL)
```

Arguments

x	vector/data.frame. data.frames are considered as multiple response variables.
weight	numeric vector. Optional case weights. NA's and negative weights treated as zero weights.
predictor	vector. By now multiple-response predictor is not supported.
fun	custom summary function. It should always return scalar/vector/matrix of the same size.
...	further arguments for fun

Value

object of class 'simple_table'/'summary_table'. Basically it's a data.frame but class is needed for custom print method.

Examples

```
data(mtcars)
mtcars = modify(mtcars,{
    var_lab(vs) = "Engine"
    val_lab(vs) = c("V-engine" = 0,
                    "Straight engine" = 1)
    var_lab(am) = "Transmission"
    val_lab(am) = c(automatic = 0,
                    manual=1)
})

fre(mtcars$vs)
with(mtcars, cro(am, vs))
with(mtcars, cro_cpct(am, vs))

# multiple-choise variable
# brands - multiple response question
# Which brands do you use during last three months?
set.seed(123)
brands = data.frame(t(replicate(20,sample(c(1:5,NA),4,replace = FALSE))))
# score - evaluation of tested product
score = sample(-1:1,20,replace = TRUE)
var_lab(brands) = "Used brands"
val_lab(brands) = make_labels("
                              1 Brand A
                              2 Brand B
                              3 Brand C
                              4 Brand D
                              5 Brand E
                              ")
```

```
var_lab(score) = "Evaluation of tested brand"
val_lab(score) = make_labels("
                             -1 Dislike it
                             0 So-so
                             1 Like it
                             ")

fre(brands)
cro(brands, score)
cro_cpct(brands, score)

# 'cro_mean'

data(iris)
cro_mean(iris[, -5], iris$Species)

# 'cro_fun'

data(mtcars)
mtcars = modify(mtcars,{
    var_lab(vs) = "Engine"
    val_lab(vs) = c("V-engine" = 0,
                    "Straight engine" = 1)
    var_lab(hp) = "Gross horsepower"
    var_lab(mpg) = "Miles/(US) gallon"
})

# Label for 'disp' forgotten intentionally
with(mtcars, cro_fun(data.frame(hp, mpg, disp), vs, summary))

# or, the same with transposed summary
with(mtcars, cro_fun(data.frame(hp, mpg, disp), vs, function(x) t(summary(x))))

# very artificial example
a = c(1,1,1, 1, 1)
b = c(0, 1, 2, 2, NA)
weight = c(0, 0, 1, 1, 1)
cro_fun(b, a, weight = weight,
     fun = function(x, weight, na.rm){
                 weighted.mean(x, w = weight, na.rm = na.rm)
             },
     na.rm = TRUE)


# comparison 'cro_fun' and 'cro_fun_df'

data(iris)
cro_fun(iris[, -5], iris$Species, fun = mean)
# same result
cro_fun_df(iris[, -5], iris$Species, fun = colMeans)

# usage for 'cro_fun_df' which is not possible for 'cro_fun'
```

```
# calculate correlations of variables with Sepal.Length inside each group
cro_fun_df(iris[,-5], iris$Species, fun = function(x) cor(x)[,1])

# or, pairwise correlations inside groups
cro_fun_df(iris[,-5], iris$Species, fun = cor)
```

`if_na` *Replace NA values in vector/data.frame/matrix/list with supplied value*

Description

Function replaces NA values in vector/data.frame/matrix/list with supplied value. If x is vector then `if_na(x) = 99` is equivalent to `x[is.na(x)] = 99`. In more complex cases when x is data.frame/matrix/list this function tries to replace NA recursively. If replacement value is vector/data.frame/matrix/list then `if_na` uses for replacement values from appropriate places. For example if both `x` and `value` are vectors then `if_na(x) = value` is equivalent to `x[is.na(x)] = value[is.na(x)]`. Single column/row value recycled to conform to x. See examples.

Usage

```
if_na(x, value)

if_na(x) <- value

x %if_na% value
```

Arguments

`x`	vector/matrix/data.frame/list
`value`	vector/matrix/data.frame/list

Value

x with replaced NA

See Also

For reverse operation see na_if.

Examples

```
# simple case
a = c(NA, 2, 3, 4, NA)
if_na(a, 99)

# the same result
a %if_na% 99
```

```
# the same result
if_na(a) = 99
a # c(99, 2, 3, 4, 99)


# replacement with values from other variable
a = c(NA, 2, 3, 4, NA)
if_na(a) = 1:5
a # 1:5

# replacement with group means

# make data.frame
set.seed(123)
group = sample(1:3, 30, replace = TRUE)
param = runif(30)
param[sample(30, 10)] = NA # place 10 NA's
df = data.frame(group, param)

# replace NA's with group means
df = modify(df, {
        if_na(param) = ave(param, group, FUN = mean_col)
    })

df

# replacement with column means

# make data.frame
set.seed(123)
x1 = runif(30)
x2 = runif(30)
x3 = runif(30)
x1[sample(30, 10)] = NA # place 10 NA's
x2[sample(30, 10)] = NA # place 10 NA's
x3[sample(30, 10)] = NA # place 10 NA's

df = data.frame(x1, x2, x3)

# replace NA's with column means
if_na(df) = t(mean_col(df))

df
```

`if_val` *Change, rearrange or consolidate the values of an existing/new variable. Inspired by RECODE command from SPSS.*

Description

`if_val` change, rearrange or consolidate the values of an existing variable based on conditions. Design of this function inspired by RECODE from SPSS. Sequence of recodings provided in the form of formulas. For example, 1:2 ~ 1 means that all 1's and 2's will be replaced with 1. Each value will recoded only once. In the assignment form `if_val(...) = ...` of this function values which doesn't meet any condition remain unchanged. In case of the usual form `... = if_val(...)` values which doesn't meet any condition will be replaced with NA. As a condition one can use just values or more sophisticated logical values and functions. There are several special functions for usage as criteria - for details see criteria. Simple common usage looks like: `if_val(x, 1:2 ~ -1, 3 ~ 0, 1:2 ~ 1, 99 ~ NA)`. For more information, see details and examples. The `ifs` function checks whether one or more conditions are met and returns a value that corresponds to the first TRUE condition. `ifs` can take the place of multiple nested `ifelse` statements and is much easier to read with multiple conditions. `ifs` works in the same manner as `if_val` - e. g. with formulas or with from/to notation. But conditions should be only logical and it doesn't operate on multicolumn objects.

Usage

```
if_val(x, ..., from = NULL, to = NULL)

if_val(x, from = NULL) <- value

ifs(..., from = NULL, to = NULL, default = NA)

lo

hi

copy(x)
```

Arguments

`x`	vector/matrix/data.frame/list
`...`	sequence of formulas which describe recodings. They are used when `from`/`to` arguments are not provided.
`from`	list of conditions for values which should be recoded (in the same format as LHS of formulas).
`to`	list of values into which old values should be recoded (in the same format as RHS of formulas).
`value`	list with formulas which describe recodings in assignment form of function/`to` list if `from`/`to` notation is used.
`default`	single value or vector. Default value - NA. This value will be used for values of result with all conditions FALSE/NA.

Format

An object of class `numeric` of length 1.

Details

Input conditions - possible values for left hand side (LHS) of formula or element of `from` list:

- vector/single value All values in `x` which equal to elements of vector in LHS will be replaced with RHS.
- function Values for which function gives TRUE will be replaced with RHS. There are some special functions for convenience - see criteria. One of special functions is `other`. It means all other unrecoded values (ELSE in SPSS RECODE). All other unrecoded values will be changed to RHS of formula or appropriate element of `to`.
- logical vector/matrix/data.frame Values for which LHS equals to TRUE will be recoded. Logical vector will be recycled across all columns of `x`. If LHS is matrix/data.frame then column from this matrix/data.frame will be used for corresponding column/element of `x`.

Output values - possible values for right hand side (RHS) of formula or element of `to` list:

- value replace elements of `x`. This value will be recycled across rows and columns of `x`.
- vector values of this vector will be replace values in corresponding position in rows of `x`. Vector will be recycled across columns of `x`.
- list/matrix/data.frame Element of list/column of matrix/data.frame will be used as a replacement value for corresponding column/element of `x`.
- function This function will be applied to values of `x` which satisfy recoding condition.There is special auxiliary function `copy` which just returns its argument. So in the `if_val` it just copies old value (COPY in SPSS RECODE). See examples. `copy` is useful in the usual form of `if_val` and doesn't do anything in the case of the assignment form `if_val() = ...` because this form don't modify values which are not satisfying any of the conditions.

`lo` and `hi` are shortcuts for `-Inf` and `Inf`. They can be useful in expressions with `%thru%`, e. g. `1 %thru% hi`.

Value

object of same form as `x` with recoded values

Examples

```
# `ifs` examples
a = 1:5
b = 5:1
ifs(b>3 ~ 1)                       # c(1, 1, NA, NA, NA)
ifs(b>3 ~ 1, default = 3)          # c(1, 1, 3, 3, 3)
ifs(b>3 ~ 1, a>4 ~ 7, default = 3) # c(1, 1, 3, 3, 7)
ifs(b>3 ~ a, default = 42)         # c(1, 2, 42, 42, 42)
# some examples from SPSS manual
# RECODE V1 TO V3 (0=1) (1=0) (2, 3=-1) (9=9) (ELSE=SYSMIS)
set.seed(123)
v1  = sample(c(0:3, 9, 10), 20, replace = TRUE)
if_val(v1) = c(0 ~ 1, 1 ~ 0, 2:3 ~ -1, 9 ~ 9, other ~ NA)
v1
```

```
# RECODE QVAR(1 THRU 5=1)(6 THRU 10=2)(11 THRU HI=3)(ELSE=0).
set.seed(123)
qvar = sample((-5):20, 50, replace = TRUE)
if_val(qvar, 1 %thru% 5 ~ 1, 6 %thru% 10 ~ 2, 11 %thru% hi ~ 3, other ~ 0)
# the same result
if_val(qvar, 1 %thru% 5 ~ 1, 6 %thru% 10 ~ 2, gte(11) ~ 3, other ~ 0)

# RECODE STRNGVAR ('A', 'B', 'C'='A')('D', 'E', 'F'='B')(ELSE=' ').
strngvar = LETTERS
if_val(strngvar, c('A', 'B', 'C') ~ 'A', c('D', 'E', 'F') ~ 'B', other ~ ' ')

# RECODE AGE (MISSING=9) (18 THRU HI=1) (0 THRU 18=0) INTO VOTER.
set.seed(123)
age = sample(c(sample(5:30, 40, replace = TRUE), rep(9, 10)))
voter = if_val(age, NA ~ 9, 18 %thru% hi ~ 1, 0 %thru% 18 ~ 0)
voter

# example with function in RHS
set.seed(123)
a = rnorm(20)
# if a<(-0.5) we change it to absolute value of a (abs function)
if_val(a, lt(-0.5) ~ abs, other ~ copy)

# the same example with logical criteria
if_val(a, a<(-.5) ~ abs, other ~ copy)

# replace with specific value for each column
# we replace values greater than 0.75 with column max and values less than 0.25 with column min
# and NA with column means
# make data.frame
set.seed(123)
x1 = runif(30)
x2 = runif(30)
x3 = runif(30)
x1[sample(30, 10)] = NA # place 10 NA's
x2[sample(30, 10)] = NA # place 10 NA's
x3[sample(30, 10)] = NA # place 10 NA's
dfs = data.frame(x1, x2, x3)

#replacement. Note the necessary transpose operation
if_val(dfs,
        lt(0.25) ~ t(min_col(dfs)),
        gt(0.75) ~ t(max_col(dfs)),
        NA ~ t(mean_col(dfs)),
        other ~ copy
      )

# replace NA with row means
# some rows which contain all NaN remain unchanged because mean_row for them also is NaN
if_val(dfs, NA ~ mean_row(dfs), other ~ copy)

# some of the above examples with from/to notation
```

```
set.seed(123)
v1  = sample(c(0:3,9,10), 20, replace = TRUE)
# RECODE V1 TO V3 (0=1) (1=0) (2,3=-1) (9=9) (ELSE=SYSMIS)
fr = list(0, 1, 2:3, 9, other)
to = list(1, 0, -1, 9, NA)
if_val(v1, from = fr) = to
v1

# RECODE QVAR(1 THRU 5=1)(6 THRU 10=2)(11 THRU HI=3)(ELSE=0).
fr = list(1 %thru% 5, 6 %thru% 10, gte(11), other)
to = list(1, 2, 3, 0)
if_val(qvar, from = fr, to = to)

# RECODE STRNGVAR ('A','B','C'='A')('D','E','F'='B')(ELSE=' ').
fr = list(c('A','B','C'), c('D','E','F') , other)
to = list("A", "B", " ")
if_val(strngvar, from = fr, to = to)

# RECODE AGE (MISSING=9) (18 THRU HI=1) (0 THRU 18=0) INTO VOTER.
fr = list(NA, 18 %thru% hi, 0 %thru% 18)
to = list(9, 1, 0)
voter = if_val(age, from = fr, to = to)
voter
```

`info` *Provides variables description for dataset*

Description

`info` returns data.frame with variables description and some summary statistics. Resulting data.frame mainly intended to keep in front of eyes in RStudio viewer or to be saved as csv to view in the spreadsheet software as reference about working dataset.

Usage

```
info(x, stats = TRUE, frequencies = TRUE, max_levels = 10)
```

Arguments

`x`	vector/factor/list/data.frame.
`stats`	Logical. Should we calculate summary for each variable?
`frequencies`	Logical. Should we calculate frequencies for each variable? This calculation can take significant amount of time for large datasets.
`max_levels`	Numeric. Maximum levels for using in frequency calculations. Levels above this value will convert to 'Other values'.

Value

data.frame with following columns: Name, Class, Length, NotNA, NA, Distincts, Label, ValueLabels, Min., 1st Qu., Median, Mean, 3rd Qu., Max., Frequency.

Examples

```
data(mtcars)
var_lab(mtcars$am) = "Transmission"
val_lab(mtcars$am) = c("Automatic"=0, "Manual"=1)
info(mtcars,max_levels = 5)
```

---

keep *Keep or drop elements by name/criteria in data.frame/matrix/list/vector*

Description

`keep` selects variables/elements from data.frame by their names or by criteria (see criteria). `except` drops variables/elements from data.frame by their names or by criteria. There is no non-standard evaluation in these functions by design so use quotes for names of your variables or use qc. `%keep%/%except%` are infix versions of these functions. `.keep/.except` are versions which works with default_dataset.

Usage

```
keep(data, ...)

except(data, ...)

data %keep% variables

data %except% variables

.keep(...)

.except(...)
```

Arguments

`data`	data.frame/matrix/list/vector
`...`	column names/element names of type character or criteria/logical functions
`variables`	column names/element names of type character or criteria/logical functions

Value

object of the same type as `data`

Examples

```
keep(iris, "Sepal.Length", "Sepal.Width")
keep(iris, qc(Sepal.Length, Sepal.Width)) # same result with non-standard eval
except(iris, "Species")

keep(iris, "Species", other) # move 'Species' to the first position
keep(iris, to("Petal.Width")) # keep all columns except 'Species'

except(iris, perl("^Petal")) # remove columns which names starts with 'Petal'

keep(airquality, (from("Ozone") & to("Wind"))) # keep columns from 'Ozone' to 'Wind'

# the same examples with infix operators

iris %keep% c("Sepal.Length", "Sepal.Width")
iris %keep% qc(Sepal.Length, Sepal.Width) # same result with non-standard eval
iris %except% "Species"

iris %keep% c("Species", other) # move 'Species' to the first position
iris %keep% to("Petal.Width")   # keep all columns except 'Species'

iris %except% perl("^Petal")    # remove columns which names starts with 'Petal'

airquality %keep% (from("Ozone") & to("Wind")) # keep columns from 'Ozone' to 'Wind'
```

`match_row` *Match finds value in rows or columns/index returns value by index from rows or columns*

Description

`match` finds value in rows or columns. `index` returns value by index from row or column. One can use functions as criteria for `match`. In this case position of first value on which function equals to TRUE will be returned. For convenience there are special predefined functions - see criteria. If value doesn't found NA will be returned.

Usage

```
match_row(criterion, ...)

match_col(criterion, ...)

index_row(index, ...)

index_col(index, ...)
```

Arguments

criterion	Vector of values to be matched, or function.
...	data. Vectors, matrixes, data.frames, lists. Shorter arguments will be recycled.
index	vector of positions in rows/columns from which values should be returned.

Value

vector with length equals to number of rows for *_row and equals to number of columns for *_col.

Examples

```
# toy data
v1 = 1:3
v2 = 2:4
v3 = 7:5

# postions of 1,3,5 in rows
match_row(c(1, 3, 5), v1, v2, v3) # 1:3
# postions of 1,3,5 in columnss
match_col(1, v1, v2, v3) # c(v1 = 1, v2 = NA, v3 = NA)

# postion of first value greater than 2
ix = match_row(gt(2), v1, v2, v3)
ix # c(3,2,1)
# return values by result of previous
index_row(ix, v1, v2, v3) # c(7,3,3)

# the same actions with data.frame
dfs = data.frame(v1, v2, v3)

# postions of 1,3,5 in rows
match_row(c(1, 3, 5), dfs) # 1:3
# postions of 1,3,5 in columnss
match_col(1, dfs) # c(v1 = 1, v2 = NA, v3 = NA)

# postion of first value greater than 2
ix = match_row(gt(2), dfs)
ix # c(3,2,1)
# return values by result of previous
index_row(ix, dfs) # c(7,3,3)
```

---

merge.simple_table *Merge two tables/data.frames*

Description

%merge% is infix shortcut for base merge with all.x = TRUE and all.y = FALSE (left join). There is also special method for combining results of cro_* and fre. For them all = TRUE (full join). It allows make complex tables from simple ones. See examples. Strange result is possible if one or two arguments have duplicates in first column (column with labels).

Usage

```
x %merge% y
```

Arguments

x	data.frame or results of fre/cro_*
y	data.frame or results of fre/cro_*

Value

data.frame

See Also

fre, cro, merge

Examples

```
data(mtcars)
# apply labels
mtcars = modify(mtcars,{
                var_lab(mpg) = "Miles/(US) gallon"
                var_lab(cyl) = "Number of cylinders"
                var_lab(disp) = "Displacement (cu.in.)"
                var_lab(hp) = "Gross horsepower"
                var_lab(drat) = "Rear axle ratio"
                var_lab(wt) = "Weight (lb/1000)"
                var_lab(qsec) = "1/4 mile time"
                var_lab(vs) = "V/S"
                val_lab(vs) = c("V-engine" = 0, "Straight engine" = 1)
                var_lab(am) = "Transmission (0 = automatic, 1 = manual)"
                val_lab(am) = c(automatic = 0, manual = 1)
                var_lab(gear) = "Number of forward gears"
                var_lab(carb) = "Number of carburetors"
})

# table by 'am'
tab1 = with(mtcars, cro_cpct(gear, am))
# table with percents
tab2 = with(mtcars, cro_cpct(gear, vs))

# combine tables
# %n_d% remove first total
tab1 %n_d% "#Total" %merge% tab2
```

`modify` *Modify data.frame/conditionally modify data.frame*

Description

`modify` evaluates expression `expr` in the context of data.frame `data`. It works similar to `within` in base R but try to return new variables in order of their appearance in the expression. `modify_if` modifies only rows for which `cond` has TRUE. Other rows remain unchanged. Newly created variables also will have values only in rows for which `cond` has TRUE. There will be NA's in other rows. This function tries to mimic SPSS "DO IF(). ... END IF." statement. There is a special constant `.n` which equals to number of cases in `data` for usage in expression inside `modify`. Inside `modify_if .n` gives number of rows which will be affected by expressions. Inside these functions you can use `set` function which creates variables with given name/set values to existing variables - .set. It is possible with `set` to assign values to multiple variables at once.

Usage

```
modify(data, expr)

data %modify% expr

modify_if(data, cond, expr)
```

Arguments

`data`	data.frame
`expr`	expression(s) that should be evaluated in the context of data.frame `data`
`cond`	logical vector or expression. Expression will be evaluated in the context of the data.

Value

Both functions returns modified data.frame

Examples

```
dfs = data.frame(
    test = 1:5,
    aa = rep(10, 5),
    b_ = rep(20, 5),
    b_1 = rep(11, 5),
    b_2 = rep(12, 5),
    b_3 = rep(13, 5),
    b_4 = rep(14, 5),
    b_5 = rep(15, 5)
)
```

```
# calculate sum of b* variables
modify(dfs, {
    b_total = sum_row(b_, b_1 %to% b_5)
    var_lab(b_total) = "Sum of b"
    random_numbers = runif(.n) # .n usage
})

# 'set' function
# new variables filled with NA
modify(dfs, {
    set('new_b`1:5`')
})

# 'set' function
# set values to existing/new variables
# expression in backticks will be expanded - see ?subst
modify(dfs, {
    set('new_b`1:5`', b_1 %to% b_5)
})


# conditional modification
modify_if(dfs, test %in% 2:4, {
    aa = aa + 1
    a_b = aa + b_
    b_total = sum_row(b_, b_1 %to% b_5)
    random_numbers = runif(.n) # .n usage
})
```

`names2labels` *Replace data.frame/list names with corresponding variables labels.*

Description

`names2labels` replaces data.frame/list names with corresponding variables labels. If there are no labels for some variables their names remain unchanged. `n2l` is just shortcut for `names2labels`.

Usage

```
names2labels(x, exclude = NULL, keep_names = FALSE)

n2l(x, exclude = NULL, keep_names = FALSE)
```

Arguments

`x`	data.frame/list.
`exclude`	logical/integer/character columns which names should be left unchanged. Only applicable to list/data.frame.

keep_names — logical. If TRUE original column names will be appended to labels in round brackets. Only applicable to list/data.frame.

Value

Object of the same type as x but with variable labels instead of names.

See Also

values2labels, f, val_lab, var_lab

Examples

```
data(mtcars)
mtcars = modify(mtcars,{
                var_lab(mpg) = "Miles/(US) gallon"
                var_lab(cyl) = "Number of cylinders"
                var_lab(disp) = "Displacement (cu.in.)"
                var_lab(hp) = "Gross horsepower"
                var_lab(drat) = "Rear axle ratio"
                var_lab(wt) = "Weight (lb/1000)"
                var_lab(qsec) = "1/4 mile time"
                var_lab(vs) = "V/S"
                var_lab(am) = "Transmission (0 = automatic, 1 = manual)"
                var_lab(gear) = "Number of forward gears"
                var_lab(carb) = "Number of carburetors"
})

# without original names
# note: we exclude dependent variable 'mpg' from conversion to use its short name in formula
summary(lm(mpg ~ ., data = names2labels(mtcars, exclude = "mpg")))
# with names
summary(lm(mpg ~ ., data = names2labels(mtcars, exclude = "mpg", keep_names = TRUE)))
```

---

na_if *Replace certain values with NA*

Description

There are following options for value:

- vector Vector of values which should be replaced with NA in x.
- logical vector/matrix/data.frame NA's will be set in places where value is TRUE. value will be recycled if needed.
- function NA's will be set in places where value(x) is TRUE. Function will be applied columnwise. Additionally, there are special functions for common cases of comparison. For example na_if(my_var, gt(98)) will replace all values greater 98 in my_var with NA. For detailed description of special functions see criteria

Usage

```
na_if(x, value)

na_if(x) <- value

x %na_if% value
```

Arguments

x	vector/matrix/data.frame/list
value	vector/matrix/data.frame/function

Value

x with NA's instead of value

See Also

For reverse operation see if_na, if_val for more general recodings.

Examples

```
a = c(1:5, 99)

# 99 to NA
na_if(a, 99)    # c(1:5, NA)

a %na_if% 99    # same result

# values which greater than 5 to NA
na_if(a, gt(5)) # c(1:5, NA)

set.seed(123)
dfs = data.frame(
      a = c("bad value", "bad value", "good value", "good value", "good value"),
      b = runif(5)
)

# rows with 'bad value' will be filled with NA
# logical argument and recycling by columns
na_if(dfs, dfs$a=="bad value")

a = rnorm(50)
# values greater than 1 or less than -1 will be set to NA
# special functions usage
na_if(a, lt(-1) | gt(1))

# values inside [-1, 1] to NA
na_if(a, -1 %thru% 1)
```

`product_test` *Data from product test of chocolate confectionary*

Description

It is rather artificial dataset with data from product test of two samples of chocolate sweets. 150 respondents tested two kinds of sweets (codenames: VSX123 and SDF546). Sample was divided into two groups (cells) of 75 respondents in each group. In cell 1 product VSX123 was presented first and then SDF546. In cell 2 sweets were presented in reversed order. Questions about respondent impressions about first product are in the block A (and about second tested product in the block B). At the end of the questionnaire there is a question about preferences between sweets.

Usage

```
product_test
```

Format

A data frame with 150 rows and 18 variables:

id Respondent Id.

cell First tested product (cell number).

s2a Age.

a1_1 What did you like in these sweets? Multiple response. First tested product.

a1_2 (continue) What did you like in these sweets? Multiple response. First tested product.

a1_3 (continue) What did you like in these sweets? Multiple response. First tested product.

a1_4 (continue) What did you like in these sweets? Multiple response. First tested product.

a1_5 (continue) What did you like in these sweets? Multiple response. First tested product.

a1_6 (continue) What did you like in these sweets? Multiple response. First tested product.

a22 Overall liking. First tested product.

b1_1 What did you like in these sweets? Multiple response. Second tested product.

b1_2 (continue) What did you like in these sweets? Multiple response. Second tested product.

b1_3 (continue) What did you like in these sweets? Multiple response. Second tested product.

b1_4 (continue) What did you like in these sweets? Multiple response. Second tested product.

b1_5 (continue) What did you like in these sweets? Multiple response. Second tested product.

b1_6 (continue) What did you like in these sweets? Multiple response. Second tested product.

b22 Overall liking. Second tested product.

c1 Preferences.

prop *Compute proportions from numeric vector/matrix/data.frame*

Description

prop returns proportion to sum of entire x. prop_col returns proportion to sum of each column of x. prop_row returns proportion to sum of each row of x. Non-numeric columns in the data.frame are ignored. NA's are also ignored.

Usage

```
prop(x)

prop_col(x)

prop_row(x)
```

Arguments

x	numeric vector/matrix/data.frame

Value

the same structure as x but with proportions of original values from sum of original values.

Examples

```
a = c(25, 25, NA)
prop(a)

# data.frame with non-numeric columns
fac = factor(c("a", "b", "c"))
char = c("a", "b", "c")
dat = as.POSIXct("2016-09-27")
a = dtfrm(fac, a = c(25, 25, NA), b = c(100, NA, 50), char, dat)

prop(a)
prop_row(a)
prop_col(a)

# the same as result as with prop.table
tbl = table(state.division, state.region)

prop(tbl)
prop_row(tbl)
prop_col(tbl)
```

qc *Create vector of characters from unquoted strings (variable names)*

Description

In many cases one need to address variables in list/data.frame in such manner: `dfs[ , c("var1", "var2", "var3")]`. `qc` ("quoted c") is a shortcut for the such cases to reduce keystrokes. With `qc` you can write: `dfs[ , qc(var1, var2, var3)]`. `subst` is simple string interpolation function. It searches in its arguments expressions in backticks (`), evaluate it and substitute it with result of evaluation. See examples.

Usage

```
qc(...)

subst(...)
```

Arguments

`...`	characters in `subst`/unquoted names of variables in `qc`

Value

Vector of characters

Examples

```
## qc
qc(a, b, c)
identical(qc(a, b, c), c("a", "b", "c"))

mtcars[, qc(am, mpg, gear)]

## subst
i = 1:5
subst("q`i`")

i = 1:3
j = 1:3
subst("q1_`i`_`j`")

data(iris)
subst("'iris' has `nrow(iris)` rows.")
```

read_spss *Read an SPSS Data File*

Description

read_spss reads data from a file stored in SPSS *.sav format. It returns data.frame and never converts string variables to factors. Also it prepares SPSS values/variables labels for working with val_lab/var_lab functions. User-missings values are ignored. read_spss is simple wrapper around read.spss function from package foreign.

Usage

```
read_spss(file, reencode = TRUE)

read_spss_to_list(file, reencode = TRUE)
```

Arguments

file	Character string: the name of the file or URL to read.
reencode	logical: should character strings be re-encoded to the current locale. The default is TRUE. NA means to do so in a UTF-8 locale, only. Alternatively, a character string specifying an encoding to assume for the file.

Value

read_spss returns data.frame.

read_spss_to_list returns list of variables from SPSS files.

See Also

read.spss in package foreign, val_lab, var_lab

Examples

```
## Not run:

w = read_spss("project_123.sav") # to data.frame
list_w = read_spss_to_list("project_123.sav") # to list


## End(Not run)
```

`ref` *Auxiliary functions to pass arguments to function by reference*

Description

These two functions aimed to simplify build functions with side-effects (e. g. for modifying variables in place). Of cause it is not the R way of doing things but sometimes it can save several keystrokes.

Usage

```
ref(x)

ref(x) <- value
```

Arguments

`x`	Reference to variable, it is formula, ~var_name.
`value`	Value that should be assigned to modified variable.

Details

To create reference to variable one can use formula: `b = ~a`. `b` is reference to `a`. So `ref(b)` returns value of `a` and `ref(b) = new_val` will modify `a`. If argument `x` of these functions is not formula then these functions have no effect e. g. `ref(a)` is identical to `a` and after `ref(a) = value` `a` is identical to `value`. It is not possible to use function as argument `x` in assignment form. For example, `ref(some_function(x)) = some_value` will rise error. Use `y = some_function(x); ref(y) = some_value` instead.

Value

`ref` returns value of referenced variable. `ref<-` modifies referenced variable.

Examples

```
# Simple example
a = 1:3
b = ~a  # b is reference to a
identical(ref(b),a) # TRUE

ref(b)[2] = 4 # here we modify a
identical(a,c(1,4,3)) # TRUE

# usage inside function

# top 10 rows
head10 = function(x){
 ds = head(ref(x),10)
```

```
 ref(x) = ds
 invisible(ds) # for usage without references
}

data(iris)
ref_to_iris = ~iris
head10(ref_to_iris) # side-effect
nrow(iris) # 10

# argument is not formula - no side-effect
data(mtcars)
mtcars10 = head10(mtcars)

nrow(mtcars10) # 10
nrow(mtcars) # 32
```

`sort_asc` *Sort data.frames/matrices/vectors*

Description

`sort_asc` sorts in ascending order and `sort_desc` sorts in descending order. There is no non-standard evaluation in these functions by design so use quotes for names of your variables or use qc. `%sort_asc%/%sort_desc%` are infix versions of these functions. `.sort_asc/.sort_desc` are versions for working with default_dataset.

Usage

```
sort_asc(data, ..., na.last = FALSE)

data %sort_asc% variables

.sort_asc(..., na.last = FALSE)

sort_asc(data) <- value

sort_desc(data, ..., na.last = TRUE)

.sort_desc(..., na.last = TRUE)

data %sort_desc% variables

sort_desc(data) <- value
```

Arguments

`data`	data.frame/matrix/vector

...	character/numeric. Column names/numbers for data.frame/matrix by which object will be sorted. Ignored for vectors.
na.last	for controlling the treatment of NAs. If TRUE, missing values in the data are put last; if FALSE, they are put first; if NA, they are removed.
variables	character/numeric. Column names/numbers for data.frame/matrix by which object will be sorted for infix functions. Ignored for vectors.
value	character/numeric. Column names/numbers for data.frame/matrix by which object will be sorted. The same as ... but for assignment versions of functions.

Value

sorted data

Examples

```
data(mtcars)
sort_asc(mtcars, "mpg")
sort_asc(mtcars, "cyl", "mpg") # by two column

# same results with column nums
sort_asc(mtcars, 1)
sort_asc(mtcars, 2:1) # by two column
sort_asc(mtcars, 2, 1) # by two column

# 'qc'  usage
sort_asc(mtcars, qc(cyl, mpg))

# infix version
mtcars %sort_asc% "mpg"
mtcars %sort_asc% c("cyl", "mpg")
mtcars %sort_asc% qc(cyl, mpg)
```

---

sum_row *Compute sum/mean/sd/median/max/min/custom function on rows/columns*

Description

This are convenience functions for usage inside modify, modify_if, with, within and dplyr mutate functions. sum/mean/sd/median/max/min always omits NA. any_in_* checks existence of any TRUE in each row/column. It is equivalent of any applied to each row/column. all_in_* is equivalent of all applied to each row/column. They don't remove NA.

Usage

```
sum_row(...)

sum_col(...)

mean_row(...)

mean_col(...)

sd_row(...)

sd_col(...)

median_row(...)

median_col(...)

max_row(...)

max_col(...)

min_row(...)

min_col(...)

apply_row(fun, ...)

apply_col(fun, ...)

any_in_row(...)

any_in_col(...)

all_in_row(...)

all_in_col(...)
```

Arguments

...	data. Vectors, matrixes, data.frames, list. Shorter arguments will be recycled.
fun	custom function that will be applied to ...

Value

All functions except apply_* return numeric vector of length equals the number of argument columns/rows. Value of apply_* depends on supplied fun function.

See Also

modify, modify_if, %to%, count_if, sum_if, mean_if, median_if, sd_if, min_if, max_if

Examples

```
## Inside example
iris = modify(iris,{
  new_median = median_row(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width)
  new_mean = mean_row(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width)
  })

dfs = data.frame(
    test = 1:5,
    aa = rep(10, 5),
    b_ = rep(20, 5),
    b_1 = rep(11, 5),
    b_2 = rep(12, 5),
    b_4 = rep(14, 5),
    b_5 = rep(15, 5)
)

# calculate sum of b* variables
modify(dfs, {
    b_total = sum_row(b_, b_1 %to% b_5)
})

# conditional modification
modify_if(dfs, test %in% 2:4, {
    b_total = sum_row(b_, b_1 %to% b_5)
})

# Examples from rowSums/colSums manual.
## Compute row and column sums for a matrix:
x = cbind(x1 = 3, x2 = c(4:1, 2:5))
sum_row(x); sum_col(x)
dimnames(x)[[1]] <- letters[1:8]
sum_row(x); sum_col(x); mean_row(x); mean_col(x)
```

---

unlab *Drop variable label and value labels*

Description

unlab returns variable x without variable labels and value labels

Usage

```
unlab(x)
```

Arguments

x	Variable(s). Vector/data.frame/list.

Value

unlab returns original variable x without variable label, value labels and class.

See Also

unvr unvl

Examples

```
raw_var = rep(1:2,5)
var_with_lab = set_var_lab(raw_var,"Income")
val_lab(var_with_lab) = c("Low"=1,"High"=2)
identical(raw_var,unlab(var_with_lab)) # should be TRUE
```

---

values2labels *Replace vector/matrix/data.frame/list values with corresponding value labels.*

Description

values2labels replaces vector/matrix/data.frame/list values with corresponding value labels. If there are no labels for some values they are converted to characters in most cases. If there are no labels at all for variable it remains unchanged. v2l is just shortcut to values2labels.

Usage

```
values2labels(x)

v2l(x)
```

Arguments

x	vector/matrix/data.frame/list

Value

Object of the same form as x but with value labels instead of values.

See Also

f, names2labels, val_lab, var_lab

Examples

```
data(mtcars)
mtcars = modify(mtcars,{
                var_lab(mpg) = NULL
                val_lab(am) = c(" automatic" = 0, " manual" =  1)
})

summary(lm(mpg ~ ., data = values2labels(mtcars[,c("mpg","am")])))
```

`val_lab` *Set or get value labels*

Description

These functions set/get/drop value labels. Duplicated values are not allowed. If argument `x` is data.frame or list then labels applied to all elements of data.frame/list. To drop value labels, use `val_lab(var) <-NULL` or `unvl(var)`. `make_labels` converts text from the form that usually used in questionnaires to named vector. See examples. For utilizing labels in base R see f, names2labels, values2labels, unlab, dichotomy. For variable labels see var_lab.

- `val_lab` returns value labels or NULL if labels doesn't exist.
- `val_lab<-` set value labels.
- `set_val_lab` returns variable with value labels.
- `add_val_lab<-` add value labels to already existing value labels.
- `unvl` drops value labels.
- `make_labels` makes named vector from text for usage as value labels.
- `ml_left`, `ml_right` and `ml_autonum` are shortcuts for `make_labels` with `code_postion` 'left', 'right' and 'autonum' accordingly.

Usage

```
val_lab(x)

val_lab(x) <- value

set_val_lab(x, value, add = FALSE)

add_val_lab(x, value)

add_val_lab(x) <- value

unvl(x)

make_labels(text, code_position = c("left", "right", "autonum"))
```

```
ml_left(text)

ml_right(text)

ml_autonum(text)
```

Arguments

x	Variable(s). Vector/data.frame/list.
value	Named vector. Names of vector are labels for the appropriate values of variable x.
add	Logical. Should we add value labels to old labels or replace it? Deafult is FALSE - we completely replace old values. If TRUE new value labels will be combined with old value labels.
text	text that should be converted to named vector
code_position	Possible values "left", "right" - position of numeric code in text. "autonum" - makes codes by autonumbering lines of text.

Details

Value labels are stored in attribute "labels" (attr(x,"labels")). We set variable class to "labelled" for preserving labels from dropping during some operations (such as c and `[`). There are special methods of subsetting and concatenation for this class.

Value

val_lab return value labels (named vector). If labels doesn't exist it return NULL . val_lab<- and set_val_lab return variable (vector x) of class "labelled" with attribute "labels" which contains value labels. make_labels return named vector for usage as value labels.

Examples

```
# toy example
set.seed(123)
# score - evaluation of tested product

score = sample(-1:1,20,replace = TRUE)
var_lab(score) = "Evaluation of tested brand"
val_lab(score) = c("Dislike it" = -1,
                   "So-so" = 0,
                   "Like it" = 1
                   )

# frequency of product scores
fre(score)

# brands - multiple response question
# Which brands do you use during last three months?

brands = t(replicate(20,sample(c(1:5,NA),4,replace = FALSE)))
```

```
var_lab(brands) = "Used brands"
val_lab(brands) = make_labels("
                              1 Brand A
                              2 Brand B
                              3 Brand C
                              4 Brand D
                              5 Brand E
                              ")


# percentage of used brands
fre(brands)

# percentage of brands within each score
cro(brands, score)


aggregate(dichotomy(brands) ~ f(score), FUN = mean)

# customer segmentation by used brands
kmeans(dichotomy(brands),3)

# model of influence of used brands on evaluation of tested product
summary(lm(score ~ dichotomy(brands)))

## make labels from text copied from questionnaire

age = c(1, 2, 1, 2)

val_lab(age) = make_labels("
 1. 18 - 26
 2. 27 - 35
")

f(age)

# or, if in original codes is on the right side

products = 1:8

val_lab(products) = ml_right("
 Chocolate bars    1
 Chocolate sweets (bulk) 2
 Slab chocolate(packed) 3
 Slab chocolate (bulk) 4
 Boxed chocolate sweets 5
 Marshmallow/pastilles in chocolate coating 6
 Marmalade in chocolate coating 7
 Other 8
")

f(products)
```

vars_range *Get range of variables/variables by pattern/by name*

Description

- `%to%` returns all variables with names in range from pattern_num1 to pattern_num2 (similar to SPSS 'to'). Result doesn't depend from order of variables in data.frame. 'num1' and 'num2' should be numbers. Results are always arranged in ascending order and include all variables with such pattern even if these variables located in different parts of dataframe. `vars_range` has the same functionality but intended for programming.
- `vars_pattern` returns all variables by pattern (regular expression). Functions with word 'list' in name return lists of variables instead of dataframes.
- `vars` returns all variables by their names. Expressions in backticks inside characters will be expanded as with subst.

Functions with word 'list' in name return lists of variables instead of dataframes.

Usage

```
vars_range(start, end)

vars_range_list(start, end)

e1 %to% e2

e1 %to_list% e2

vars_pattern_list(pattern)

vars_pattern(pattern)

vars_list(...)

vars(...)
```

Arguments

`start`	character Name of start variable (e. g. a_1)
`end`	character Name of start variable (e. g. a_5)
`e1`	unquoted name of start variable (e. g. a_1)
`e2`	unquoted name of start variable (e. g. a_5)
`pattern`	character pattern of variable(-s) name
`...`	characters names of variables.

Value

data.frame/list with variables

Examples

```
# In global environement
aa = rep(10, 5)
b = rep(20, 5)
a1 = rep(1, 5)
a2 = rep(2, 5)
a3 = rep(3, 5)
a4 = rep(4, 5)
a5 = rep(5, 5)

# identical results
vars_range("a1", "a5")
a1 %to% a5
vars("a`1:5`")
vars_pattern("^a[0-9]$")

# sum each row
sum_row(a1 %to% a5)

# In data.frame
dfs = data.frame(
    aa = rep(10, 5),
    b_ = rep(20, 5),
    b_1 = rep(11, 5),
    b_2 = rep(12, 5),
    b_3 = rep(13, 5),
    b_4 = rep(14, 5),
    b_5 = rep(15, 5)
)

# all variables that starts with 'b'
with(dfs, vars_pattern("^b"))

# calculate sum of b_* variables
modify(dfs,{
    b_total = sum_row(b_1 %to% b_5)
    b_total2 = sum_row(vars("b_`1:5`"))
})
```

---

var_lab *Set or get variable label*

Description

These functions set/get/drop variable labels. For utilizing labels in base R see f, names2labels, values2labels, unlab. For value labels see val_lab.

- var_lab returns variable label or NULL if label doesn't exist.
- var_lab<- set variable label.
- set_var_lab returns variable with label.
- unvr drops variable label.

Usage

```
var_lab(x)

var_lab(x) <- value

set_var_lab(x, value)

unvr(x)
```

Arguments

x	Variable. In the most cases it is numeric vector.
value	A character scalar - label for the variable x.

Details

Variable label is stored in attribute "label" (attr(x,"label")). For preserving from dropping this attribute during some operations (such as c) variable class is set to "labelled". There are special methods of subsetting and concatenation for this class. To drop variable label use var_lab(var) <- NULL or unvr(var).

Value

var_lab return variable label. If label doesn't exist it return NULL . var_lab<- and set_var_lab return variable (vector x) of class "labelled" with attribute "label" which equals submitted value.

Examples

```
data(mtcars)
mtcars = modify(mtcars,{
                var_lab(mpg) = "Miles/(US) gallon"
                var_lab(cyl) = "Number of cylinders"
                var_lab(disp) = "Displacement (cu.in.)"
                var_lab(hp) = "Gross horsepower"
                var_lab(drat) = "Rear axle ratio"
                var_lab(wt) = "Weight (lb/1000)"
                var_lab(qsec) = "1/4 mile time"
                var_lab(vs) = "V/S"
                var_lab(am) = "Transmission (0 = automatic, 1 = manual)"
                var_lab(gear) = "Number of forward gears"
                var_lab(carb) = "Number of carburetors"
})

# note: we exclude dependent variable 'mpg' from conversion to use its short name in formula
```

```
summary(lm(mpg ~ ., data = n2l(mtcars, exclude = "mpg")))

data(mtcars)

var_lab(mtcars$am) = "Transmission"
val_lab(mtcars$am) = c(automatic = 0, manual=1)

## Not run:
plot(f(mtcars$am))

## End(Not run)

fre(mtcars$am)
```

`vectors` *Infix operations on vectors - append, diff, intersection, union, replication*

Description

- `%a%` a(ppends) second argument to first argument.
- `%u%` u(nites) first and second arguments. Remove elements from second argument that exist in first argument.
- `%d%` d(iffs) second argument from first argument. Second argument could be a function which returns logical value. In this case elements of first argument which give TRUE will be removed.
- `%i%` i(ntersects) first argument and second argument. Second argument could be a function which returns logical value. In this case elements of first argument which give FALSE will be removed.
- `%e%` e(xclusive OR). Returns elements that contained only in one of arguments.
- `%r%` r(epeats) first argument second argument times.
- `%n_d%`n(ames) d(iff) - diffs second argument from names of first argument. Second argument could be a function which returns logical value. In this case elements of first argument which names give TRUE will be removed.
- `%n_i%`n(ames) i(ntersect) - intersects names of first argument with second argument. Second argument could be a function which returns logical value. In this case elements of first argument which names give FALSE will be removed.

All these functions except `%n_d%`, `%n_i%` preserve names of vectors and don't remove duplicates. For `%d%`, `%i%`, `%n_d%`, `%n_i%` one can use criteria functions. See criteria for details.

Usage

```
e1 %a% e2

e1 %u% e2
```

```
e1 %d% e2

e1 %i% e2

e1 %e% e2

e1 %r% e2

e1 %n_d% e2

e1 %n_i% e2
```

Arguments

e1	vector (possibly data.frame/matrix/list for %n_d%, %n_i%)
e2	vector (or function for %d%, %i%)

Value

vector (possibly data.frame/matrix/list for %n_d%, %n_i%)

Examples

```
1:4 %a% 5:6   # 1:6

1:4 %a% 4:5   # 1,2,3,4,4,5

1:4 %u% 4:5   # 1,2,3,4,5

1:6 %d% 5:6   # 1:4

# function as criterion
1:6 %d% gt(4) # 1:4

1:4 %i% 4:5   # 4

# function as criterion
letters %i% perl("[a-d]") # a,b,c,d

# function as criterion
letters %i% (fixed("a") | fixed("z")) # a, z

1:4 %e% 4:5   # 1, 2, 3, 5

1:2 %r% 2     # 1, 2, 1, 2

# %n_i%, %n_d%

# remove column Species
```

```
iris %n_d% "Species"

# leave only columns which names start with "Sepal"
iris %n_i% perl("^Sepal")

# leave column "Species" and columns which names start with "Sepal"
iris %n_i% (perl("^Sepal")|"Species")
```

---

vlookup *Look up values in dictionary.*

Description

This function is inspired by VLOOKUP spreadsheet function. Looks for a lookup_value in the lookup_column of the dict, and then returns values in the same rows from result_column.

Usage

```
vlookup(lookup_value, dict, result_column, lookup_column = 1)

vlookup_df(lookup_value, dict, result_column = NULL, lookup_column = 1)
```

Arguments

lookup_value	Vector of looked up values
dict	Dictionary. Should be vector/matrix or data.frame
result_column	Resulting columns of dict. Should be numeric or character vector. There are special values: 'row.names', 'rownames', 'names'. If result_column equals to one of these special values and dict is matrix/data.frame then row names of dict will be returned. If dict is vector then names of vector will be returned. For vlookup_df default result_column is NULL and result will be entire rows. For vlookup result_column = NULL gives error.
lookup_column	Column of dict in which lookup value will be searched. By default it is the first column of the dict. There are special values: 'row.names', 'rownames', 'names'. If lookup_column equals to one of these special values and dict is matrix/data.frame then values will be searched in the row names of dict. If dict is vector then values will be searched in names of the dict.

Value

vlookup always return vector, vlookup_df always returns data.frame.

Examples

```
# with data.frame
dict = data.frame(num=1:26, small=letters, cap=LETTERS, stringsAsFactors = FALSE)
rownames(dict) = paste0('rows', 1:26)
identical(vlookup_df(1:3, dict), dict[1:3,]) # should be TRUE
vlookup(c(45,1:3,58), dict, result_column='cap')
vlookup_df(c('z','d','f'), dict, lookup_column = 'small')
vlookup_df(c('rows7', 'rows2', 'rows5'), dict, lookup_column = 'row.names')

# with vector
dict=1:26
names(dict) = letters

vlookup(c(2,4,6), dict, result_column='row.names')

# The same results
vlookup(c(2,4,6), dict, result_column='rownames')
vlookup(c(2,4,6), dict, result_column='names')

# Just for fun. Examples borrowed from Microsoft Excel.
# It is not the R way of doing things.

# Example 2

ex2 = utils::read.table(header = TRUE, text = "
    Item_ID Item Cost Markup
    ST-340 Stroller 145.67  0.30
    BI-567 Bib 3.56  0.40
    DI-328 Diapers  21.45  0.35
    WI-989 Wipes  5.12  0.40
    AS-469 Aspirator 2.56  0.45
", stringsAsFactors = FALSE)

# Calculates the retail price of diapers by adding the markup percentage to the cost.
vlookup("DI-328", ex2, 3) * (1 + vlookup("DI-328", ex2, 4)) # 28.9575

# Calculates the sale price of wipes by subtracting a specified discount from
# the retail price.
(vlookup("WI-989", ex2, "Cost") * (1 + vlookup("WI-989", ex2, "Markup"))) * (1 - 0.2)  # 5.7344

A2 = ex2[1, "Item_ID"]
A3 = ex2[2, "Item_ID"]

# If the cost of an item is greater than or equal to $20.00, displays the string
# "Markup is nn%"; otherwise, displays the string "Cost is under $20.00".
ifelse(vlookup(A2, ex2, "Cost") >= 20,
       paste0("Markup is " , 100 * vlookup(A2, ex2, "Markup"),"%"),
       "Cost is under $20.00") # Markup is 30%


# If the cost of an item is greater than or equal to $20.00, displays the string
# Markup is nn%"; otherwise, displays the string "Cost is $n.nn".
```

```
ifelse(vlookup(A3, ex2, "Cost") >= 20,
       paste0("Markup is: " , 100 * vlookup(A3, ex2, "Markup") , "%"),
       paste0("Cost is $", vlookup(A3, ex2, "Cost"))) #Cost is $3.56


# Example 3

ex3 = utils::read.table(header = TRUE, text = "
    ID  Last_name  First_name  Title Birth_date
    1 Davis Sara 'Sales Rep.'  12/8/1968
    2 Fontana Olivier 'V.P. of Sales' 2/19/1952
    3 Leal Karina 'Sales Rep.' 8/30/1963
    4 Patten Michael 'Sales Rep.' 9/19/1958
    5 Burke Brian 'Sales Mgr.' 3/4/1955
    6 Sousa Luis 'Sales Rep.'  7/2/1963
", stringsAsFactors = FALSE)

# If there is an employee with an ID of 5, displays the employee's last name;
# otherwise, displays the message "Employee not found".
if_na(vlookup(5, ex3, "Last_name"), "Employee not found") # Burke

# Many employees
if_na(vlookup(1:10, ex3, "Last_name"), "Employee not found")

# For the employee with an ID of 4, concatenates the values of three cells into
# a complete sentence.
paste0(vlookup(4, ex3, "First_name"), " ",
       vlookup(4, ex3, "Last_name"), " is a ",
       vlookup(4, ex3, "Title")) # Michael Patten is a Sales Rep.
```

`where` *Subsetting Data Frames*

Description

cond will be evaluated in the context of the data frame, so columns can be referred to (by name) as variables in the expression (see the examples). `.where` is version for working with default dataset. See default_dataset. `%where%` is infix function with the same functional. See examples. There is a special constant `.n` which equals to number of cases in `data` for usage in cond expression.

Usage

```
where(data, cond)

data %where% cond

.where(cond)
```

Arguments

data	data.frame to be subsetted
cond	logical or numeric expression indicating elements or rows to keep: missing values (NA) are taken as false.

Value

data.frame which contains just selected rows.

Examples

```
# leave only 'setosa'
where(iris, Species == "setosa")
# leave only first five rows
where(iris, 1:5)

# infix version
# note that '%where%' have higher precendence than '=='
# so we need to put condition inside brackets
iris %where% (Species == "setosa")

iris %where% 1:5

# example of .n usage. Very artificial examples
set.seed(42)
train = iris %where% sample(.n, 100)
str(train)

set.seed(42)
test = iris %where% -sample(.n, 100)
str(test)
```

---

write_labels *Write data with labels to file in R code or in SPSS syntax.*

Description

write_labelled_* functions write data in the CSV format and file with R code/SPSS syntax for labelling data. SPSS syntax also contains code for reading data in SPSS. write_labelled_* doesn't save rownames of data.frame. write_labels_* functions write R code/SPSS syntax for labelling data. It allows to extract labels from *.sav files that come without accompanying syntax. read_labelled_csv reads data file in CSV format and apply labels from accompanying file with R code.

Usage

```
write_labels(x, filename, fileEncoding = "")

read_labelled_csv(filename, fileEncoding = "", ...)

write_labelled_csv(x, filename, fileEncoding = "", ...)

write_labelled_spss(x, filename, fileEncoding = "", ...)

write_labels_spss(x, filename)
```

Arguments

x	data.frame to be written/data.frame whose labels to be written
filename	the name of the file which the data are to be read from/write to.
fileEncoding	character string: if non-empty declares the encoding to be used on a file (not a connection) so the character data can be re-encoded as they are written. See file.
...	additional arguments for read.table/write.table

Value

Functions for writing invisibly return NULL. Functions for reading return labelled data.frame.

Examples

```
## Not run:
data(mtcars)
mtcars = modify(mtcars,{
                var_lab(mpg) = "Miles/(US) gallon"
                var_lab(cyl) = "Number of cylinders"
                var_lab(disp) = "Displacement (cu.in.)"
                var_lab(hp) = "Gross horsepower"
                var_lab(drat) = "Rear axle ratio"
                var_lab(wt) = "Weight (lb/1000)"
                var_lab(qsec) = "1/4 mile time"
                var_lab(vs) = "Engine"
                val_lab(vs) = c("V-engine" = 0,
                                "Straight engine" = 1)
                var_lab(am) = "Transmission"
                val_lab(am) = c(automatic = 0,
                                manual=1)
                var_lab(gear) = "Number of forward gears"
                var_lab(carb) = "Number of carburetors"
})

# to R code
# rownames are not preserved
write_labelled_csv(mtcars, "mtcars.csv")
new_mtcars = read_labelled_csv("mtcars.csv")
```

```
# to SPSS syntax
write_labelled_spss(mtcars, "mtcars.csv")


## End(Not run)
```

Index