Package 'memisc'

March 10, 2023

Type Package

Title Management of Survey Data and Presentation of Analysis Results

Version 0.99.31.6

Date 2023-03-07

Author Martin Elff (with contributions from Christopher N. Lawrence, Dave Atkins, Jason W. Morgan, Achim Zeileis, Mael Astruc-Le Souder, Kiril M<c3><bc>ller, and Pieter Schoonees)

Maintainer Martin Elff <memisc@elff.eu>

Description An infrastructure for the management of survey data including value labels, definable missing values, recoding of variables, production of code books, and import of (subsets of) 'SPSS' and 'Stata' files is provided. Further, the package allows to produce tables and data frames of arbitrary descriptive statistics and (almost) publication-ready tables of regression model estimates, which can be exported to 'LaTeX' and HTML.

License GPL-2 | GPL-3

LazyLoad Yes

Depends R (>= 3.3.0), lattice, stats, methods, utils, MASS

Suggests splines, knitr, rmarkdown, sandwich

Enhances AER, car, eha, lme4, ordinal, simex, tibble, haven

Imports grid, data.table, yaml, jsonlite

VignetteBuilder knitr

URL http://memisc.elff.eu,https://github.com/melff/memisc/

BugReports https://github.com/melff/memisc/issues

NeedsCompilation yes

Repository CRAN

Date/Publication 2023-03-10 16:20:05 UTC

R topics documented:

`annotations` *Adding Annotations to Objects*

Description

Annotations, that is, objects of class `"annotation"`, are character vectors with all their elements named. Only one method is defined for this subclass of character vectors, a method for `show`, that shows the annotation in a nicely formatted way. Annotations of an object can be obtained via the function `annotation(x)` and can be set via `annotation(x)<-value`.

Elements of an annotation with names `"description"` and `"wording"` have a special meaning. The first kind can be obtained and set via `description(x)` and `description(x)<-value`, the second kind can be obtained via `wording(x)` and `wording(x)<-value`. `"description"` elements are used in way the "variable labels" are used in SPSS and Stata. `"wording"` elements of annotation objects

are meant to contain the question wording of a questionnaire item represented by an "item" objects. These elements of annotations are treated in a special way in the output of the coodbook function.

Usage

```
annotation(x)
## S4 method for signature 'ANY'
annotation(x)
## S4 method for signature 'item'
annotation(x)
## S4 method for signature 'data.set'
annotation(x)
annotation(x)<-value
## S4 replacement method for signature 'ANY,character'
annotation(x)<-value
## S4 replacement method for signature 'ANY,annotation'
annotation(x)<-value
## S4 replacement method for signature 'item,annotation'
annotation(x)<-value
## S4 replacement method for signature 'vector,annotation'
annotation(x)<-value

description(x)
description(x)<-value

wording(x)
wording(x)<-value

## S4 method for signature 'data.set'
description(x)
## S4 method for signature 'importer'
description(x)
## S4 method for signature 'data.frame'
description(x)
## S4 method for signature 'tbl_df'
description(x)
```

Arguments

x	an object
value	a character or annotation object

Value

annotation(x) returns an object of class "annotation", which is a named character. description(x) and wording(x) each usually return a character string. If description(x) is applied to a data.set or an importer object, however, a character vector is returned, which is named after the variables in the data set or the external file.

Examples

```
vote <- sample(c(1,2,3,8,9,97,99),size=30,replace=TRUE)
labels(vote) <- c(Conservatives         =  1,
                  Labour                =  2,
                  "Liberal Democrats"   =  3,
                  "Don't know"          =  8,
                  "Answer refused"      =  9,
                  "Not applicable"      = 97,
                  "Not asked in survey" = 99
                  )
missing.values(vote) <- c(97,99)
description(vote) <- "Vote intention"
wording(vote) <- "If a general election would take place next tuesday,
                  the candidate of which party would you vote for?"
annotation(vote)
annotation(vote)["Remark"] <- "This is not a real questionnaire item, of course ..."
codebook(vote)
```

---

applyTemplate *Apply a Formatting Template to a Numeric or Character Vector*

Description

applyTemplate is called internally by mtable to format coefficients and summary statistics.

Usage

```
applyTemplate(x,template,float.style=getOption("float.style"),
                      digits=min(3,getOption("digits")),
                      signif.symbols=getOption("signif.symbols"))
```

Arguments

x	a numeric or character vector to be formatted
template	a character vector that defines the template, see details.
float.style	A character string that is passed to formatC by applyTemplate; valid values are "e", "f", "g", "fg", "E", and "G". By default, the float.style setting of options is used. The 'factory fresh' setting is options(float.style="f")
digits	number of significant digits to use if not specified in the template.
signif.symbols	a named vector that specifies how significance levels are symbolically indicated, values of the vector specify significance levels and names specify the symbols. By default, the signif.symbols setting of options is used. The "factory-fresh" setting is options(signif.symbols=c("***"=.001,"**"=.01,"*"=.05)).

Details

Character vectors that are used as templates may be arbitrary. However, certain character sequences may form *template expressions*. A template expression is of the form `($<POS>:<Format spec>)`, where "`($`" indicates the start of a template expression, "`<POS>`" stands for either an index or name that selects an element from `x` and "`<Format spec>`" stands for a *format specifier*. It may contain an letter indicating the style in which the vector element selected by `<POS>` will be formatted by `formatC`, it may contain a number as the number of significant digits, a "`#`" indicating that the number of signifcant digits will be at most that given by `getOption("digits")`, or `*` that means that the value will be formatted as a significance symbol.

Value

`applyTemplate` returns a character vector in which template expressions in `template` are substituted by formatted values from `x`. If `template` is an array then the return value is also an array of the same shape.

Examples

```
applyTemplate(c(a=.0000000000000304,b=3),template=c("($1:g7#)($a:*)"," (($1:f2)) "))
applyTemplate(c(a=.0000000000000304,b=3),template=c("($a:g7#)($a:*)"," (($b:f2)) "))
```

`as.array` *Converting Data Frames into Arrays*

Description

The `as.array` for data frames takes all factors in a data frame and uses them to define the dimensions of the resulting array, and fills the array with the values of the remaining numeric variables.

Currently, the data frame must contain all combinations of factor levels.

Usage

```
## S4 method for signature 'data.frame'
as.array(x,data.name=NULL,...)
```

Arguments

`x`	a data frame
`data.name`	a character string, giving the name attached to the dimension that corresponds to the numerical variables in the data frame (that is, the `name` attached to the corresponding element of the `dimnames` list).
`...`	other arguments, ignored.

Value

An array

Examples

```
BerkeleyAdmissions <- to.data.frame(UCBAdmissions)
BerkeleyAdmissions
as.array(BerkeleyAdmissions,data.name="Admit")
try(as.array(BerkeleyAdmissions[-1,],data.name="Admit"))
```

`as.symbols` *Construction of Lists of Symbols*

Description

`as.symbols` and `syms` are functions potentially useful in connection with `foreach` and `xapply`. `as.symbols` produces a list of symbols from a character vector, while `syms` returns a list of symbols from symbols given as arguments, but it can be used to construct patterns of symbols.

Usage

```
as.symbols(x)
syms(...,paste=FALSE,sep="")
```

Arguments

`x`	a character vector
`...`	character strings or (unquoted) variable names
`paste`	logical value; should the character strings `paste`d into one string?
`sep`	a separator string, passed to `paste`.

Value

A list of language symbols (results of `as.symbol` - not graphical symbols!).

Examples

```
as.symbols(letters[1:8])
syms("a",1:3,paste=TRUE)

sapply(syms("a",1:3,paste=TRUE),typeof)
```

assign_if *Assign a values to a variable for instances where a condition is met*

Description

The %if% operator allows to assign values to a variable only if a condition is met i.e. results in TRUE. It is supposed to be used similar to the replace ... if construct in Stata.

Usage

```
expr %if% condition
# For example
# (variable <- value) %if% (other_variable == 0)
```

Arguments

expr	An expression that assigns a value to variable
condition	A logical vector or a an expression that evaluates to a logical vector

Details

The 'value' that is assigned to the variable in expr should either be a scalar, a vector with as many elements as the condition vector has, or as many elements as the number of elements in the condition vector that are equal (or evaluate to) TRUE.

Examples

```
(test_var <- 1) %if% (1:7 > 3)

test_var

(test_var <- 2) %if% (1:7 <= 3)
test_var

(test_var <- 100*test_var) %if% (1:7%%2==0)
test_var

# This creates a warning about non-matching lengths.
(test_var <- 500:501) %if% (1:7 <= 3)
test_var

(test_var <- 501:503) %if% (1:7 <= 3)
test_var

(test_var <- 401:407) %if% (1:7 <= 3)
test_var
```

attr-operators *Operators for Setting Annotations and Attributes*

Description

The operator %#% can be used to attach a description annotation to an object. %##% can be used to attach a character vector of annotations to an object. %@% returns the attribute with the name given as second argument. With %@% it is also possible to assign attributes.

Usage

```
x %#% descr
x %##% annot
x %@% nm
x %@% nm <- value
```

Arguments

x	an object, usually and item or a vector.
descr	a character string
annot	a named character vector; its contents are added to the "annotation" attribute of x. Existing elements are kept.
nm	a character string, the name of the attribute being set or requested.
value	any kind of object that can be attached as an attribute.

Examples

```
test1 <- 1 %#% "One"
# This is equivalent to:
# test <- 1
# description(test) <- "One"
description(test1)
# Results in "One"

# Not that it makes sense, but ...
test2 <- 2 %##% c(
                    Precedessor = 0,
                    Successor   = 2
                  )
# This is equivalent to:
# test2 <- 2
# annotation(test2) <- c(
#                     Precedessor = 0,
#                     Successor   = 2
#                   )
annotation(test2)

# The following examples are equivalent to
```

```
# attr(test2,"annotation")
test2 %@% annotation

test2 %@% "annotation"

test2 %@% another.attribute <- 42
# This is equivalent to attr(test2,"another.attribute") <- 42

attributes(test2)
```

By *Conditional Evaluation of an Expression*

Description

The function By evaluates an expression within subsets of a data frame, where the subsets are defined by a formula.

Usage

```
By(formula,expr,data=parent.frame())
```

Arguments

formula	an expression or (preferably) a formula containing the names of conditioning variables or factors.
expr	an expression that is evaluated for any unique combination of values of the variables contained in formula.
data	a data frame, an object that can be coerced into a data frame (for example, a table), or an environment, from which values for the variables in formula or expr are taken.

Value

A list of class "by", giving the results for each combination of values of variables in formula.

Examples

```
berkeley <- Aggregate(Table(Admit,Freq)~.,data=UCBAdmissions)
(Bres <- By(~Dept,glm(cbind(Admitted,Rejected)~Gender,family="binomial"),data=berkeley))
# The results all have 'data' components
str(Bres[[1]]$data)

attach(berkeley)
(Bres <- By(~Dept,glm(cbind(Admitted,Rejected)~Gender,family="binomial")))
detach(berkeley)
```

cases *Distinguish between Cases Specified by Logical Conditions*

Description

cases allows to distinguish several cases defined logical conditions. It can be used to code these cases into a vector. The function can be considered as a multi-condition generalization of ifelse.

Usage

```
cases(...,check.xor=c("warn","stop","ignore"),
      .default=NA,.complete=FALSE,
      check.na=c("warn","stop","ignore"),
      na.rm=TRUE)
```

Arguments

...	A sequence of logical expressions or assignment expressions containing logical expressions as "right hand side".
check.xor	character (either "warn", "stop", or "ignore") or logical; if TRUE or equal to "stop" or "warn", cases checks whether the case conditions are mutually exclusive. If this is not satisfied and check.xor equals "warn" (the default), a warning is shown, otherwise an error exception is raised.
.default	a value to be used for unsatisfied conditions.
.complete	logical, if TRUE an additional factor level is created for the unsatisfied conditions.
check.na	character (either "warn", "stop", or "ignore") or logical; if TRUE or equal to "stop" or "warn", cases checks, whether any of the case conditions evaluates to NA. If that case, if check.na is TRUE or equals "stop" an error exception is raised, while if check.na equals "warn" (the default) a warning is shown.
na.rm	a logical value; how to handle NAs (if they do not already lead to an error exception). If FALSE if *any* of the conditions evaluates to NA, the corresponding value of the result vector is NA. If TRUE (the default), the resulting vector or factor is NA only for instances where all conditions result in NA.

Details

There are two distinct ways to use this function. Either the function can be used to construct a factor that represents several logical cases or it can be used to conditionally evaluate an expression in a manner similar to ifelse.

For the first use, the ... arguments have to be a series of logical expressions. cases then returns a factor with as many levels as logical expressions given as ... arguments. The resulting factor will attain its first level if the first condition is TRUE, otherwise it will attain its second level if the second condition is TRUE, etc. The levels will be named after the conditions or, if name tags are attached to the logical expressions, after the tags of the expressions. Not that the logical expressions all need to evaluate to logical vectors of the same length, otherwise an error condition is raised. If

.complete is TRUE then an additional factor level is created for the conditions not satisfied for any of the cases.

For the second use, the ... arguments have to be a series of assignment expression of the type <expression> <- <logical expression> or <logical expression> -> <expression>. For cases in which the first logical expression is TRUE, the result of first expression that appears on the other side of the assignment operator become elements of the vector returned by cases, for cases in which the second logical expression is TRUE, the result of the second expression that appears on the other side of the assignment operator become elements of the vector returned by cases, etc. For cases that do not satisfy any of the given conditions the value of the .default argument is used. Note that the logical expressions also here all need to evaluate to logical vectors of the same length. The expressions on the other side of the assignment operator should also be either vectors of the same length and mode or should scalars of the same mode, otherwise unpredictable results may occur.

Value

If it is called with logical expressions as ... arguments, cases returns a factor, if it is called with assignment expressions the function returns a vector with the same mode as the results of the "assigned" expressions and with the same length as the logical conditions.

Examples

```
# Examples of the first kind of usage of the function
#
df <- data.frame(x = rnorm(n=20), y = rnorm(n=20))
df <- df[do.call(order,df),]
(df <- within(df,{
  x1=cases(x>0,x<=0)
  y1=cases(y>0,y<=0)
  z1=cases(
    "Condition 1"=x<0,
    "Condition 2"=y<0,# only applies if x >= 0
    "Condition 3"=TRUE
    )
  z2=cases(x<0,(x>=0 & y <0), (x>=0 & y >=0))
  }))
xtabs(~x1+y1,data=df)
dd <- with(df,
  try(cases(x<0,
            x>=0,
            x>1,
            check.xor=TRUE)# let's be fussy
            )
  )
dd <- with(df,
  try(cases(x<0,x>=0,x>1))
  )
genTable(range(x)~dd,data=df)

# An example of the second kind of usage of the function:
# A construction of a non-smooth function
#
```

```
fun <- function(x)
  cases(
    x==0      -> 1,
    abs(x)> 1 -> abs(x),
    abs(x)<=1 -> x^2
  )
x <- seq(from=-2,to=2,length=101)
plot(fun(x)~x)

# Demo of the new .default and .complete arguments
x <- seq(from=-2,to=2)
cases(a = x < -1,
      b = x > 1,
      .complete = TRUE)
cases(x < -1,
      x > 1,
      .complete = TRUE)
cases(1 <- x < -1,
      3 <- x > 1,
      .default = 2)

threshhold <- 5
d <- c(1:10, NaN)

d1 <- cases(
  d > threshhold -> 1,
  d <= threshhold -> 2
)

d2 <- cases(
  is.na(d) -> 0,
  d > threshhold -> 1,
  d <= threshhold -> 2
)

# Leads to missing values because some of the conditions result in missing
# even though they could be 'captured'
d3 <- cases(
  is.na(d) -> 0,
  d > threshhold -> 1,
  d <= threshhold -> 2,
  na.rm=FALSE
)

d4 <- cases(
  is.na(d) -> 0,
  d > threshhold +2 -> 1,
  d <= threshhold -> 2,
  na.rm=FALSE
)

cbind(d,d1,d2,d3,d4)
```

```
cases(
  d > threshhold,
  d <= threshhold
)

cases(
  is.na(d),
  d > threshhold,
  d <= threshhold
)

cases(
  d > threshhold,
  d <= threshhold,
  .complete=TRUE
)

cases(
  d > threshhold + 2,
  d <= threshhold,
  .complete=TRUE
)
```

---

coarsen *Coarsen a vector into a factor with a lower number of levels*

Description

coarsen can be used to obtain a factor from a vector, similar to cut, but with less technical and more "aesthetic" labels of the factor levels.

Usage

```
coarsen(x,...)
## S3 method for class 'numeric'
coarsen(x,
        n=5,
        pretty=TRUE,
        quantiles=!pretty,
        breaks=NULL,
        brackets=FALSE,
        sep=if(brackets)";"else if(quantiles) "-" else " - ",
        left="[",
        right="]",
        range=FALSE,
        labels=NULL,
        ...)
```

Arguments

x	a vector, usually a numeric vector
n	number of categories of the resulting factor
pretty	a logical value, whether pretty should be used to compute the breaks.
quantiles	a logical value, whether quantile should be used to compute the breaks.
breaks	a vector of break points or NULL.
brackets	a logical value, whether the labels should include brackets.
sep	a character string, used as a separator between upper and lower boundaries in the labels.
left	a character string, to be used as the left bracket
right	a character string, to be used as the right bracket
range	a logical value, whether the minimum and maximum of x should be included into breaks.
labels	an optional character vector of labels.
...	further arguments, passed on to pretty or quantile if applicable.

Examples

```
x <- rnorm(200)

table(coarsen(x))

table(coarsen(x,quantiles=TRUE))

table(coarsen(x,brackets=TRUE))

table(coarsen(x,breaks=c(-1,0,1)))

table(coarsen(x,breaks=c(-1,0,1),
              range=TRUE,labels=letters[1:4]))
```

---

codebook *Generate a Codebook of a Data Set*

Description

Function codebook collects documentation about an item, or the items in a data set or external data file. It returns an object that, when shown, print this documentation in a nicely formatted way.

Usage

```
codebook(x, weights = NULL, unweighted = TRUE, ...)
## S4 method for signature 'item'
codebook(x, weights = NULL, unweighted = TRUE, ...)
## S4 method for signature 'atomic'
codebook(x, weights = NULL, unweighted = TRUE, ...)
## S4 method for signature 'factor'
codebook(x, weights = NULL, unweighted = TRUE, ...)
## S4 method for signature 'data.set'
codebook(x, weights = NULL, unweighted = TRUE, ...)
## S4 method for signature 'importer'
codebook(x, weights = NULL, unweighted = TRUE, ...)
## S4 method for signature 'data.frame'
codebook(x, weights = NULL, unweighted = TRUE, ...)
## S4 method for signature 'tbl_df'
codebook(x, weights = NULL, unweighted = TRUE, ...)
```

Arguments

x	an item, numeric or character vector, factor, data.set, data.frame or importer object for codebook()
weights	an optional vector of weights.
unweighted	an optional logical vector; if weights are given, it determines of only summaries of weighted data are show or also summaries of unweighted data.
...	other arguments, currently ignored.

Value

An object of class "codebook", for which a show method exists that produces a nicely formatted output.

Examples

```
Data <- data.set(
          vote = sample(c(1,2,3,8,9,97,99),size=300,replace=TRUE),
          region = sample(c(rep(1,3),rep(2,2),3,99),size=300,replace=TRUE),
          income = exp(rnorm(300,sd=.7))*2000
          )

Data <- within(Data,{
  description(vote) <- "Vote intention"
  description(region) <- "Region of residence"
  description(income) <- "Household income"
  wording(vote) <- "If a general election would take place next tuesday,
                    the candidate of which party would you vote for?"
  wording(income) <- "All things taken into account, how much do all
                    household members earn in sum?"
  foreach(x=c(vote,region),{
```

```
    measurement(x) <- "nominal"
    })
  measurement(income) <- "ratio"
  labels(vote) <- c(
                   Conservatives         =  1,
                   Labour                =  2,
                   "Liberal Democrats"   =  3,
                   "Don't know"          =  8,
                   "Answer refused"      =  9,
                   "Not applicable"      = 97,
                   "Not asked in survey" = 99)
  labels(region) <- c(
                   England               =  1,
                   Scotland              =  2,
                   Wales                 =  3,
                   "Not applicable"      = 97,
                   "Not asked in survey" = 99)
  foreach(x=c(vote,region,income),{
    annotation(x)["Remark"] <- "This is not a real survey item, of course ..."
    })
  missing.values(vote) <- c(8,9,97,99)
  missing.values(region) <- c(97,99)
})

description(Data)

codebook(Data)

codebook(Data)$vote
codebook(Data)[2]

codebook(Data[2])

DataFr <- as.data.frame(Data)
DataHv <- as_haven(Data,user_na=TRUE)

codebook(DataFr)
codebook(DataHv)


## Not run:
Write(description(Data),
          file="Data-desc.txt")
Write(codebook(Data),
          file="Data-cdbk.txt")

## End(Not run)
```

---

codeplan *Describe structure of Data Sets and Importers*

Description

The function codeplan() creates a data frame that describes the structure of an item list (a data.set object or an importer object), so that this structure can be stored and and recovered. The resulting data frame has a particular print method that delimits the output to one line per variable.

With setCodeplan an item list structure (as returned by codeplan()) can be applied to a data frame or data set. It is also possible to use an assignment like codeplan(x) <- value to a similar effect.

Usage

```
codeplan(x)
## S4 method for signature 'item.list'
codeplan(x)
## S4 method for signature 'item'
codeplan(x)
setCodeplan(x,value)
## S4 method for signature 'data.frame,codeplan'
setCodeplan(x,value)
## S4 method for signature 'data.frame,NULL'
setCodeplan(x,value)
## S4 method for signature 'data.set,codeplan'
setCodeplan(x,value)
## S4 method for signature 'data.set,NULL'
setCodeplan(x,value)
## S4 method for signature 'item,codeplan'
setCodeplan(x,value)
## S4 method for signature 'item,NULL'
setCodeplan(x,value)
## S4 method for signature 'atomic,codeplan'
setCodeplan(x,value)
## S4 method for signature 'atomic,NULL'
setCodeplan(x,value)
codeplan(x) <- value
read_codeplan(filename,type)
write_codeplan(x,filename,type,pretty)
```

Arguments

x	for codeplan(x) an object that inherits from class "item.list", i.e. can be a "data.set" object or an "importer" object, it can also be an object that inherits from class "item". For write_codeplan an object from class "codeplan".
value	an object as it would be returned by codeplan(x) or NULL.
filename	a character string, the name of the file that is to be read or to be written.
type	a character string (either "yaml" or "json") oder NULL (the default), gives the type of the file into which the codeplan is written or from which it is read. If type is NULL then the file type is inferred from the file name ending (".yaml" or ",yml" for "yaml", ".json" for "json").

`pretty` a logical value, whether the JSON output created by `write_codeplan(...)` should be prettified.

Value

If applicable, `codeplan` returns a list with additional S3 class attribute `"codeplan"`. For arguments for which the relevant information does not exist, the function returns `NULL`.

The list has at least one element or several elements, named after the variable in the "item.list" or "data.set" `x`. Each list element is a list itself with the following elements:

`annotation` a named character vector,

`labels` a named list of labels and labelled values

`value.filter` a list with at least two elements named "class" and "filter", and optionally another element named "range". The "class" element determines the class of the value filter and equals either "missing.values", "valid.values", or "valid.range". An element named "range" may only be needed if "class" is "missing.values", as it is possible (like in SPSS) to have *both* individual missing values and a range of missing values.

`mode` a character string that describes storage mode, such as `"character"`, `"integer"`, or `"numeric"`.

`measurement` a character string with the measurement level, `"nominal"`, `"ordinal"`, `"interval"`, or `"ratio"`.

If `codeplan(x)<-value` or `setCodeplan(x,value)` is used and `value` is `NULL`, all the special information about annotation, labels, value filters, etc. is removed from the resulting object, which then is usually a mere atomic vector or data frame.

Examples

```
Data1 <- data.set(
          vote = sample(c(1,2,3,8,9,97,99),size=300,replace=TRUE),
          region = sample(c(rep(1,3),rep(2,2),3,99),size=300,replace=TRUE),
          income = exp(rnorm(300,sd=.7))*2000
          )

Data1 <- within(Data1,{
  description(vote) <- "Vote intention"
  description(region) <- "Region of residence"
  description(income) <- "Household income"
  foreach(x=c(vote,region),{
    measurement(x) <- "nominal"
    })
  measurement(income) <- "ratio"
  labels(vote) <- c(
                   Conservatives         =  1,
                   Labour                =  2,
                   "Liberal Democrats"   =  3,
                   "Don't know"          =  8,
                   "Answer refused"      =  9,
                   "Not applicable"      = 97,
```

```
                    "Not asked in survey" = 99)
  labels(region) <- c(
                    England               =  1,
                    Scotland              =  2,
                    Wales                 =  3,
                    "Not applicable"      = 97,
                    "Not asked in survey" = 99)
  foreach(x=c(vote,region,income),{
    annotation(x)["Remark"] <- "This is not a real survey item, of course ..."
    })
  missing.values(vote) <- c(8,9,97,99)
  missing.values(region) <- c(97,99)
})
cpData1 <- codeplan(Data1)

Data2 <- data.frame(
          vote = sample(c(1,2,3,8,9,97,99),size=300,replace=TRUE),
          region = sample(c(rep(1,3),rep(2,2),3,99),size=300,replace=TRUE),
          income = exp(rnorm(300,sd=.7))*2000
          )
codeplan(Data2) <- cpData1
codeplan(Data2)
codebook(Data2)

# Note the difference between 'as.data.frame' and setting
# the codeplan to NULL:
Data2df <- as.data.frame(Data2)
codeplan(Data2) <- NULL
str(Data2)
str(Data2df)
codeplan(Data2) <- NULL # Does not change anything

# Codeplans of survey items can also be inquired and manipulated:
vote <- Data1$vote
str(vote)
cp.vote <- codeplan(vote)
codeplan(vote) <- NULL
str(vote)
codeplan(vote) <- cp.vote
vote

fn.json <- paste0(tempfile(),".json")
write_codeplan(codeplan(Data1),filename=fn.json)
codeplan(Data2) <- read_codeplan(fn.json)
codeplan(Data2)
```

`collect` *Collect Objects*

Description

collect gathers several objects into one, matching the elements or subsets of the objects by names or dimnames.

Usage

```
collect(...,names=NULL,inclusive=TRUE)
## Default S3 method:
collect(...,names=NULL,inclusive=TRUE)
## S3 method for class 'array'
collect(...,names=NULL,inclusive=TRUE)
## S3 method for class 'matrix'
collect(...,names=NULL,inclusive=TRUE)
## S3 method for class 'table'
collect(...,names=NULL,sourcename=".origin",fill=0)
## S3 method for class 'data.frame'
collect(...,names=NULL,inclusive=TRUE,
                                  fussy=FALSE,warn=TRUE,
                                  detailed.warnings=FALSE,use.last=FALSE,
                                  sourcename=".origin")
## S3 method for class 'data.set'
collect(...,names=NULL,inclusive=TRUE,
                                  fussy=FALSE,warn=TRUE,
                                  detailed.warnings=FALSE,use.last=FALSE,
                                  sourcename=".origin")
```

Arguments

...	more atomic vectors, arrays, matrices, tables, data.frames or data.sets
names	optional character vector; in case of the default and array methods, giving dimnames for the new dimension that identifies the collected objects; in case of the data.frame and data.set methods, levels of a factor indentifying the collected objects.
inclusive	logical, defaults to TRUE; should unmatched elements included? See details below.
fussy	logical, defaults to FALSE; should it count as an error, if variables with same names of collected data.frames/data.sets have different attributes?
warn	logical, defaults to TRUE; should an warning be given, if variables with same names of collected data.frames/data.sets have different attributes?
detailed.warnings	logical, whether the attributes of each variable should be printed if they differ, and if warn or fuzzy is TRUE.
use.last	logical, defaults to FALSE. If the function is applied to data frames or similar objects, attributes of variables may differ between data frames (or other objects, respectively). If this argument is TRUE, then the attributes are harmonised based on the variables in the last data frame/object, otherwise the attributes of variables in the first data frame/object are used for harmonisation.
sourcename	name of the factor that identifies the collected data.frames or data.sets

`fill` numeric; with what to fill empty table cells, defaults to zero, assuming the table contains counts

Value

If `x` and all following ... arguments are vectors of the same mode (numeric,character, or logical) the result is a matrix with as many columns as vectors. If argument `inclusive` is TRUE, then the number of rows equals the number of names that appear at least once in each of the vector names and the matrix is filled with `NA` where necessary, otherwise the number of rows equals the number of names that are present in *all* vector names.

If `x` and all ... arguments are matrices or arrays of the same mode (numeric,character, or logical) and n dimension the result will be a $n+1$ dimensional array or table. The extend of the $n+1$th dimension equals the number of matrix, array or table arguments, the extends of the lower dimension depends on the `inclusive` argument: either they equal to the number of dimnames that appear at least once for each given dimension and the array is filled with `NA` where necessary, or they equal to the number of dimnames that appear in all arguments for each given dimension.

If `x` and all ... arguments are data frames or data sets, the result is a data frame or data set. The number of variables of the resulting data frame or data set depends on the `inclusive` argument. If it is true, the number of variables equals the number of variables that appear in each of the arguments at least once and variables are filled with `NA` where necessary, otherwise the number of variables equals the number of variables that are present in all arguments.

Examples

```
x <- c(a=1,b=2)
y <- c(a=10,c=30)

x
y

collect(x,y)
collect(x,y,inclusive=FALSE)

X <- matrix(1,nrow=2,ncol=2,dimnames=list(letters[1:2],LETTERS[1:2]))
Y <- matrix(2,nrow=3,ncol=2,dimnames=list(letters[1:3],LETTERS[1:2]))
Z <- matrix(3,nrow=2,ncol=3,dimnames=list(letters[1:2],LETTERS[1:3]))

X
Y
Z

collect(X,Y,Z)
collect(X,Y,Z,inclusive=FALSE)

X <- matrix(1,nrow=2,ncol=2,dimnames=list(a=letters[1:2],b=LETTERS[1:2]))
Y <- matrix(2,nrow=3,ncol=2,dimnames=list(a=letters[1:3],c=LETTERS[1:2]))
Z <- matrix(3,nrow=2,ncol=3,dimnames=list(a=letters[1:2],c=LETTERS[1:3]))

collect(X,Y,Z)
collect(X,Y,Z,inclusive=FALSE)
```

```
df1 <- data.frame(a=rep(1,5),b=rep(1,5))
df2 <- data.frame(a=rep(2,5),b=rep(2,5),c=rep(2,5))
collect(df1,df2)
collect(df1,df2,inclusive=FALSE)

data(UCBAdmissions)
Male <- as.table(UCBAdmissions[,1,])
Female <- as.table(UCBAdmissions[,2,])
collect(Male,Female,sourcename="Gender")
collect(unclass(Male),unclass(Female))

Male1 <- as.table(UCBAdmissions[,1,-1])
Female2 <- as.table(UCBAdmissions[,2,-2])
Female3 <- as.table(UCBAdmissions[,2,-3])
collect(Male=Male1,Female=Female2,sourcename="Gender")
collect(Male=Male1,Female=Female3,sourcename="Gender")
collect(Male=Male1,Female=Female3,sourcename="Gender",fill=NA)

f1 <- gl(3,5,labels=letters[1:3])
f2 <- gl(3,6,labels=letters[1:3])
collect(f1=table(f1),f2=table(f2))

ds1 <- data.set(x = 1:3)
ds2 <- data.set(x = 4:9,
                y = 1:6)
collect(ds1,ds2)
```

`contr` *Convenience Methods for Setting Contrasts*

Description

This package provides modified versions of `contr.treatment` and `contr.sum`. `contr.sum` gains an optional `base` argument, analog to the one of `contr.treatment`, furthermore, the `base` argument may be the name of a factor level.

`contr` returns a function that calls either `contr.treatment`, `contr.sum`, etc., according to the value given to its first argument.

The `contrasts` method for `"item"` objects returns a contrast matrix or a function to produce a contrast matrix for the factor into which the item would be coerced via `as.factor` or `as.ordered`. This matrix or function can be specified by using `contrasts(x)<-value`

Usage

```
contr(type,...)
contr.treatment(n, base=1,contrasts=TRUE)
contr.sum(n,base=NULL,contrasts=TRUE)
## S4 method for signature 'item'
contrasts(x,contrasts=TRUE,...)
```

```
## S4 replacement method for signature 'item'
contrasts(x,how.many) <- value
# These methods are defined implicitely by making 'contrasts' generic.
## S4 method for signature 'ANY'
contrasts(x,contrasts=TRUE,...)
## S4 replacement method for signature 'ANY'
contrasts(x,how.many) <- value
```

Arguments

type	a character vector, specifying the type of the contrasts. This argument should have a value such that, if e.g. type="something", then there is a function contr.something that produces a contrast matrix.
...	further arguments, passed to contr.treatment, etc.
n	a number of factor levels or a vector of factor levels names, see e.g. contr.treatment.
base	a number of a factor level or the names of a factor level, which specifies the baseline category, see e.g. contr.treatment or NULL.
contrasts	a logical value, see contrasts
how.many	the number of contrasts to generate, see contrasts
x	a factor or an object of class "item"
value	a matrix, a function or the name of a function

Value

contr returns a funtion that calls one of contr.treatment, contr.sum, ... contr.treatment and contr.sum return contrast matrices. contrasts(x) returns the "contrasts" attribute of an object, which may be a function name, a function, a contrast matrix or NULL.

Examples

```
ctr.t <- contr("treatment",base="c")
ctr.t
ctr.s <- contr("sum",base="c")
ctr.h <- contr("helmert")
ctr.t(letters[1:7])
ctr.s(letters[1:7])
ctr.h(letters[1:7])

x <- factor(rep(letters[1:5],3))
contrasts(x)
x <- as.item(x)
contrasts(x)
contrasts(x) <- contr.sum(letters[1:5],base="c")
contrasts(x)
missing.values(x) <- 5
contrasts(x)
contrasts(as.factor(x))

# Obviously setting missing values after specifying
```

```
# contrast matrix breaks the contrasts.
# Using the 'contr' function, however, prevents this:

missing.values(x) <- NULL
contrasts(x) <- contr("sum",base="c")
contrasts(x)
missing.values(x) <- 5
contrasts(x)
contrasts(as.factor(x))
```

`contract` *Contract data into pattern-frequency format*

Description

`contract()` contracts data into pattern-frequency format, similar to a contatenation of `table()` (or `xtabs`) and `as.data.frame()`. Yet it uses much less memory if patterns are sparse, because it does not create rows for patterns that do not occur.

Usage

```
contract(x,...)
## S3 method for class 'data.frame'
contract(x,by=NULL, weights=NULL,name="Freq",
    force.name=FALSE,sort=FALSE,drop.na=TRUE,...)
## S3 method for class 'data.set'
contract(x,by=NULL, weights=NULL,name="Freq",
    force.name=FALSE,sort=FALSE,drop.na=TRUE,...)
```

Arguments

`x`	an object of class `"data.frame"` or `"data.set"`.
`by`	the formula or a vector of variable names (quoted or not quoted). Specifies the patterns (and optionally weights). If `by` is a formula, then the right-hand side specifies the variables the value patterns of which are counted. If the left-hand side of the formula is (the name of) a numeric vector, its values are used as weights (in which case the `weights` argument will be ignored.) If the left-hand side of the formula is (the name of) a factor, counts are computed in separate columns for each of its levels.
`weights`	a numeric vector of weights or `NULL`.
`name`	a character string, the name of the variable that containts the frequency counts of the value patterns.
`force.name`	a logical value, defaults to `FALSE`. If `TRUE` and the left-hand side of `by` formula is a factor, the names of the columns with the counts are combinations of the labels of the factor levels and the argument of `name`; if `FALSE`, the column names are created from the labels of the factor levels only.

sort	a logical value, defaults to FALSE. If TRUE, the resulting data set is sorted by the variables that define the patterns. If FALSE, the row of the resulting data frame or data set are ordered according to the occurrence of the patterns.
drop.na	a logical value, defaults to TRUE. If FALSE, patterns that involve NA are included in the resulting data frame or data set.
...	further arguments, passed to methods or ignored.

Value

If x is a data fame, the value of contract() is also a data frame. If it is a "data.set" object, the result is also a "data.set" object.

Examples

```
iris_ <- sample(iris,size=nrow(iris),replace=TRUE)
w <- rep(1,nrow(iris_))
contract(iris[4:5])
contract(iris[4:5],sort=TRUE)
contract(iris[4:5],weights=w,sort=TRUE)
contract(iris,by=c(Petal.Width,Species),sort=TRUE)
contract(iris,by=~Petal.Width+Species)
contract(iris,by=w~Species)

library(MASS)
contract(housing,
         by=Sat~Infl+Type+Cont,
         weights=Freq)

contract(housing,
         by=Sat~Infl+Type+Cont,
         weights=Freq,
         name="housing",force.name=TRUE
         )
```

---

data.set *Data Set Objects*

Description

"data.set" objects are collections of "item" objects, with similar semantics as data frames. They are distinguished from data frames so that coercion by as.data.fame leads to a data frame that contains only vectors and factors. Nevertheless most methods for data frames are inherited by data sets, except for the method for the within generic function. For the within method for data sets, see the details section.

Thus data preparation using data sets retains all informations about item annotations, labels, missing values etc. While (mostly automatic) conversion of data sets into data frames makes the data amenable for the use of R's statistical functions.

dsView is a function that displays data sets in a similar manner as View displays data frames. (View works with data sets as well, but changes them first into data frames.)

Usage

```
data.set(...,row.names = NULL, check.rows = FALSE, check.names = TRUE,
    stringsAsFactors = FALSE, document = NULL)
as.data.set(x, row.names=NULL, ...)
## S4 method for signature 'list'
as.data.set(x,row.names=NULL,...)
is.data.set(x)
## S3 method for class 'data.set'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
## S4 method for signature 'data.set'
within(data, expr, ...)

dsView(x)

## S4 method for signature 'data.set'
head(x,n=20,...)
## S4 method for signature 'data.set'
tail(x,n=20,...)
```

Arguments

`...`	For the `data.set` function several vectors or items, for `within` further, ignored arguments.
`row.names, check.rows, check.names, stringsAsFactors, optional`	arguments as in `data.frame` or `as.data.frame`, respectively.
`document`	NULL or an optional character vector that contains documenation of the data.
`x`	for `is.data.set(x)`, any object; for `as.data.frame(x,...)` and `dsView(x)` a "data.set" object.
`data`	a data set, that is, an object of class "data.set".
`expr`	an expression, or several expressions enclosed in curly braces.
`n`	integer; the number of rows to be shown by `head` or `tail`

Details

The `as.data.frame` method for data sets is just a copy of the method for list. Consequently, all items in the data set are coerced in accordance to their `measurement` setting, see `as.vector,item-method` and `measurement`.

The `within` method for data sets has the same effect as the `within` method for data frames, apart from two differences: all results of the computations are coerced into items if they have the appropriate length, otherwise, they are automatically dropped.

Currently only one method for the generic function `as.data.set` is defined: a method for "importer" objects.

Value

`data.set` and the `within` method for data sets returns a "data.set" object, `is.data.set` returns a logical value, and `as.data.frame` returns a data frame.

Examples

```
Data <- data.set(
          vote = sample(c(1,2,3,8,9,97,99),size=300,replace=TRUE),
          region = sample(c(rep(1,3),rep(2,2),3,99),size=300,replace=TRUE),
          income = exp(rnorm(300,sd=.7))*2000
          )

Data <- within(Data,{
  description(vote) <- "Vote intention"
  description(region) <- "Region of residence"
  description(income) <- "Household income"
  wording(vote) <- "If a general election would take place next tuesday,
                    the candidate of which party would you vote for?"
  wording(income) <- "All things taken into account, how much do all
                    household members earn in sum?"
  foreach(x=c(vote,region),{
    measurement(x) <- "nominal"
    })
  measurement(income) <- "ratio"
  labels(vote) <- c(
                    Conservatives         =  1,
                    Labour                =  2,
                    "Liberal Democrats"   =  3,
                    "Don't know"          =  8,
                    "Answer refused"      =  9,
                    "Not applicable"      = 97,
                    "Not asked in survey" = 99)
  labels(region) <- c(
                    England               =  1,
                    Scotland              =  2,
                    Wales                 =  3,
                    "Not applicable"      = 97,
                    "Not asked in survey" = 99)
  foreach(x=c(vote,region,income),{
    annotation(x)["Remark"] <- "This is not a real survey item, of course ..."
    })
  missing.values(vote) <- c(8,9,97,99)
  missing.values(region) <- c(97,99)

  # These to variables do not appear in the
  # the resulting data set, since they have the wrong length.
  junk1 <- 1:5
  junk2 <- matrix(5,4,4)

})
# Since data sets may be huge, only a
# part of them are 'show'n
Data

## Not run:

# If we insist on seeing all, we can use 'print' instead
```

```
print(Data)

## End(Not run)

str(Data)
summary(Data)

## Not run:
# If we want to 'View' a data set we can use 'dsView'
dsView(Data)
# Works also, but changes the data set into a data frame first:
View(Data)

## End(Not run)

Data[[1]]
Data[1,]
head(as.data.frame(Data))

EnglandData <- subset(Data,region == "England")
EnglandData

xtabs(~vote+region,data=Data)
xtabs(~vote+region,data=within(Data, vote <- include.missings(vote)))
```

`data.set manipulation` *Manipulation of Data Sets*

Description

Like data frames, `data.set` objects have `subset`, `unique`, `cbind`, `rbind`, `merge` methods defined for them.

The semantics are basically the same as the methods defined for data frames in the `base` package, with the only difference that the return values are `data.set` objects. In fact, the methods described here are front-ends to the corresponding methods for data frames, which are constructed such that the "extra" information attached to variables within `data.set` objects, that is, to `item` objects.

Usage

```
## S3 method for class 'data.set'
subset(x, subset, select, drop = FALSE, ...)

## S4 method for signature 'data.set'
unique(x, incomparables = FALSE, ...)

## S3 method for class 'data.set'
cbind(..., deparse.level = 1)

## S3 method for class 'data.set'
```

```
rbind(..., deparse.level = 1)

## S4 method for signature 'data.set,data.set'
merge(x,y, ...)

## S4 method for signature 'data.set,data.frame'
merge(x,y, ...)

## S4 method for signature 'data.frame,data.set'
merge(x,y, ...)
```

Arguments

x,y	data.set objects. On of the arguments to merge may also be an object coercable into a data frame and the result still is a data.set object.
subset	a logical expression, used to select observations from the data set.
select	a vector with variablen names, which are retained in the data subset.
drop	logical; if TRUE and the result has only one column, the result is an item and not a data set.
...	for subset: a logical vector of the same length as the number of rows of the data.set and, optionally, a vector of variable names (tagged as select); for unique: further arguments, ignored; for cbind, rbind: objects coercable into data frames, with at least one being a data.set object; for merge: further arguments such as arguments tagged with by, by.x, by.y, etc. that specify the variables by which to merge the data sets of data frames x and y.
incomparables	a vector of values that cannot be compared. See unique.
deparse.level	an argument retained for reasons of compatibility of the default methods of cbind and rbind.

Examples

```
ds1 <- data.set(
      a = rep(1:3,5),
      b = rep(1:5,each=3)
  )
ds2 <- data.set(
      a = c(3:1,3,3),
      b = 1:5
  )

ds1 <- within(ds1,{
      description(a) <- "Example variable 'a'"
      description(b) <- "Example variable 'b'"
  })

ds2 <- within(ds2,{
      description(a) <- "Example variable 'a'"
      description(b) <- "Example variable 'b'"
  })
```

```
str(ds3 <- rbind(ds1,ds2))
description(ds3)

ds3 <- within(ds1,{
        c <- a
        d <- b
        description(c) <- "Copy of variable 'a'"
        description(d) <- "Copy of variable 'b'"
        rm(a,b)
    })
str(ds4 <- cbind(ds1,ds3))
description(ds4)

ds5 <- data.set(
        c = 1:3,
        d = c(1,1,2)
        )
ds5 <- within(ds5,{
      description(c) <- "Example variable 'c'"
      description(d) <- "Example variable 'd'"
  })
str(ds6 <- merge(ds1,ds5,by.x="a",by.y="c"))

# Note that the attributes of the left-hand variables
# have priority.
description(ds6)
```

---

deduplicate_labels *Handle duplicated labels*

Description

The function deduplicate_labels can be used with "item" objects, "importer" objects or "data.set" objects to deal with duplicate labels, i.e. labels that are attached to more than one code. There are several ways to de-duplicate labels: by combining values that share their label or by making labels duplicate labels distinct.

Usage

```
deduplicate_labels(x,...)
## S3 method for class 'item'
deduplicate_labels(x,
    method=c("combine codes",
             "prefix values",
             "postfix values"),...)
# Applicable to 'importer' objects and 'data.set' objects
## S3 method for class 'item.list'
deduplicate_labels(x,...)
```

Arguments

x	an item with value labels or that contains items with value labels
method	a character string that determines the method to make value labels unique.
...	other arguments, passed to specific methods of the generic function.

Value

The function deduplicate_labels a copy of x that has unqiue value labels.

Examples

```
x1 <- as.item(rep(1:5,4),
              labels=c(
                  A = 1,
                  A = 2,
                  B = 3,
                  B = 4,
                  C = 5
              ),
              annotation = c(
                  description="Yet another test"
))

x2 <- as.item(rep(1:4,5),
              labels=c(
                  i   = 1,
                  ii  = 2,
                  iii = 3,
                  iii = 4
                  ),
              annotation = c(
                  description="Still another test"
))

x3 <- as.item(rep(1:2,10),
              labels=c(
                  a = 1,
                  b = 2
                  ),
              annotation = c(
                  description="Still another test"
))

codebook(deduplicate_labels(x1))
codebook(deduplicate_labels(x1,method="prefix"))
codebook(deduplicate_labels(x1,method="postfix"))

ds <- data.set(x1,x2,x3)
codebook(deduplicate_labels(ds))
codebook(deduplicate_labels(ds,method="prefix"))
codebook(deduplicate_labels(ds,method="postfix"))
```

Descriptives *Vectors of Univariate Sample Statistics*

Description

`Descriptives(x)` gives a vector of sample statistics for use in codebook.

Usage

```
Descriptives(x,...)
## S4 method for signature 'atomic'
Descriptives(x, weights = NULL, ...)
## S4 method for signature 'item.vector'
Descriptives(x, weights = NULL, ...)
```

Arguments

`x`	an atomic vector or `"item.vector"` object.
`weights`	an optional vector of weights.
`...`	further arguments, to be passed to future methods.

Value

A numeric vector of sample statistics, containing the range, the mean, the standard deviation, the skewness and the (excess) kurtosis.

Examples

```
x <- rnorm(100)
Descriptives(x)
```

dimrename *Change dimnames, rownames, or colnames*

Description

These functions provide an easy way to change the `dimnames`, `rownames` or `colnames` of an array.

Usage

```
dimrename(x, dim = 1, ..., gsub = FALSE, fixed = TRUE, warn = TRUE)
rowrename(x, ..., gsub = FALSE, fixed = TRUE, warn = TRUE)
colrename(x, ..., gsub = FALSE, fixed = TRUE, warn = TRUE)
```

Arguments

x	An array with dimnames
dim	A vector that indicates the dimensions
...	A sequence of named arguments
gsub	a logical value; if TRUE, gsub is used to change the dimnames of the object. That is, instead of substituting whole names, substrings of the dimnames of the object can changed.
fixed	a logical value, passed to gsub. If TRUE, substitutions are by fixed strings and not by regular expressions.
warn	logical; should a warning be issued if the pattern is not found?

Details

dimrename changes the dimnames of x along dimension(s) dim according to the remaining arguments. The argument names are the *old* names, the values are the new names. rowrename is a shorthand for changing the rownames, colrename is a shorthand for changing the colnames of a matrix or matrix-like object.

If gsub is FALSE, argument tags are the *old* dimnames, the values are the new dimnames. If gsub is TRUE, arguments are substrings of the dimnames that are substituted by the argument values.

Value

Object x with changed dimnames.

Examples

```
m <- matrix(1,2,2)
rownames(m) <- letters[1:2]
colnames(m) <- LETTERS[1:2]
m
dimrename(m,1,a="first",b="second")
dimrename(m,1,A="first",B="second")
dimrename(m,2,"A"="first",B="second")

rowrename(m,a="first",b="second")
colrename(m,"A"="first",B="second")

# Since version 0.99.22 - the following also works:

dimrename(m,1,a=first,b=second)
dimrename(m,1,A=first,B=second)
dimrename(m,2,A=first,B=second)
```

duplicated_labels *Check for and report duplicated labels*

Description

The function duplicated_labels can be used with "item" objects, "importer" objects or "data.set" objects to check whether items contain duplicate labels, i.e. labels that are attached to more than one code.

Usage

```
duplicated_labels(x)
## S3 method for class 'item'
duplicated_labels(x)
# Applicable to 'importer' objects and 'data.set' objects
## S3 method for class 'item.list'
duplicated_labels(x)
```

Arguments

x	an item with value labels or that contains items with value labels

Value

The function duplicate.labels returns a list with a class attribute, which allows pretty printing of duplicated value labels

Examples

```
x1 <- as.item(rep(1:5,4),
              labels=c(
                  A = 1,
                  A = 2,
                  B = 3,
                  B = 4,
                  C = 5
              ),
              annotation = c(
                  description="Yet another test"
))

x2 <- as.item(rep(1:4,5),
              labels=c(
                  i   = 1,
                  ii  = 2,
                  iii = 3,
                  iii = 4
                  ),
              annotation = c(
```

```
                  description="Still another test"
))

x3 <- as.item(rep(1:2,10),
              labels=c(
                  a = 1,
                  b = 2
                  ),
              annotation = c(
                  description="Still another test"
))

duplicated_labels(x1)
ds <- data.set(x1,x2,x3)
duplicated_labels(ds)
codebook(ds)

nes1948.por <- unzip(system.file("anes/NES1948.ZIP",package="memisc"),
                     "NES1948.POR",exdir=tempfile())
nes1948 <- spss.portable.file(nes1948.por)
duplicated_labels(nes1948)
```

`foreach` *Loop over Variables in a Data Frame or Environment*

Description

`foreach` evaluates an expression given as untagged argument by substituting in variables. The expression may also contain assignments, which take effect in the caller's environment.

Usage

```
foreach(...,.sorted,.outer=FALSE)
```

Arguments

`...`	tagged and untagged arguments. The tagged arguments define the 'variables' that are looped over, the first untagged argument defines the expression wich is evaluated.
`.sorted`	an optional logical value; relevant only when a range of variable is specified using the column operator ":". Decises whether variable names should be sorted alphabetically before the range of variables are created. If this argument missing, its default value is TRUE, if `foreach()` is called in the global environment, otherwise it is FALSE.
`.outer`	an optional logical value; if TRUE, each combination of the variables is used to evaluate the expression, if FALSE (the default) then the variables all need to have the same length and the corresponding values of the variables are used in the evaluation of the expression.

Examples

```
x <- 1:3
y <- -(1:3)
z <- c("Uri","Schwyz","Unterwalden")
print(x)
print(y)
print(z)
foreach(var=c(x,y,z),          # assigns names
  names(var) <- letters[1:3]   # to the elements of x, y, and z
  )
print(x)
print(y)
print(z)

ds <- data.set(
        a = c(1,2,3,2,3,8,9),
        b = c(2,8,3,2,1,8,9),
        c = c(1,3,2,1,2,8,8)
      )
print(ds)
ds <- within(ds,{
      description(a) <- "First item in questionnaire"
      description(b) <- "Second item in questionnaire"
      description(c) <- "Third item in questionnaire"

      wording(a) <- "What number do you like first?"
      wording(b) <- "What number do you like second?"
      wording(c) <- "What number do you like third?"

      foreach(x=a:c,{ # Lazy data documentation:
        labels(x) <- c(    # a,b,c get value labels in one statement
                         one = 1,
                         two = 2,
                       three = 3,
                "don't know" = 8,
         "refused to answer" = 9)
        missing.values(x) <- c(8,9)
        })
      })

codebook(ds)

# The colon-operator respects the order of the variables
# in the data set, if .sorted=FALSE
with(ds[c(3,1,2)],
     foreach(x=a:c,
             print(description(x))
            ))

# Since .sorted=TRUE, the colon operator creates a range
# of alphabetically sorted variables.
with(ds[c(3,1,2)],
```

```
     foreach(x=a:c,
             print(description(x)),
             .sorted=TRUE
            ))

# The variables in reverse order
with(ds,
     foreach(x=c:a,
             print(description(x))
            ))

# The colon operator can be combined with the
# concatenation function
with(ds,
     foreach(x=c(a:b,c,c,b:a),
             print(description(x))
            ))

# Variables can also be selected by regular expressions.
with(ds,
     foreach(x=rx("[a-b]"),
             print(description(x))
            ))

# A demonstration for '.outer=TRUE'
foreach(l=letters[1:2],
        i=1:3,
        cat(paste0(l,i,"\n")),
        .outer=TRUE)
```

`format_html` *Format Objects in HTML, show the HTML Format or Write it to a File*

Description

`show_html` is for showing objects in a convenient way in HTML format. `write_html` writes them in HTML format into a file. Both functions call the generic `format_html` for the format conversion.

Usage

```
show_html(x, output = NULL, ...)
write_html(x, file, ...)

format_html(x, ...)

## S3 method for class 'data.frame'
format_html(x,
    toprule=2,midrule=1,bottomrule=2,
```

```
    split.dec=TRUE,
    row.names=TRUE,
    digits=getOption("digits"),
    format="f",
    style=df_format_stdstyle,
    margin="2ex auto",
    ...)

## S3 method for class 'matrix'
format_html(x,
    toprule=2,midrule=1,bottomrule=2,
    split.dec=TRUE,
    formatC=FALSE,
    digits=getOption("digits"),
    format="f",
    style=mat_format_stdstyle,
    margin="2ex auto",
    ...)
```

Arguments

x	an object.
output	character string or a function that determines how the HTML formatted object is shown. If output is a function, it is called with the path of a (temporary) file with HTML code, e.g. *RStudio*'s viewer function (which is available in the package rstudioapi. If output equals "stdout", the HTML code is written to the standard output stream (for use e.g. in output produced with knitr), if "file-show", the contents of a file with the HTML code is shown via file.show, and if "browser", the contents of a file with the HTML code is shown by the standard browser (via browseURL). This arguments has different defaults, depending of the type of the session. In non-interactive sessions, the default is "console", in interactive sessions other than RStudio, it is "browser", in interactive sessions with RStudio it is "file-show". These default settings can be overriden by the option "html_viewer" (see options).
file	character string; name or path of the file where to write the HTML code to.
toprule	integer; thickness in pixels of rule at the top of the table.
midrule	integer; thickness in pixels of rules within the table.
bottomrule	integer; thickness in pixels of rule at the bottom of the table.
split.dec	logical; whether numbers should be centered at the decimal point by splitting the table cells.
row.names	logical; whether row names should be shown/exported.
digits	number of digits to be shown after the decimal dot. This is only useful, if the "ftable" object was created from a table created with genTable or the like.

formatC	logical; whether to use formatC instead of format to format cell contents.
format	a format string for formatC
style	string containing the stanard CSS styling of table cells.
margin	character string, determines the margin and thus the position of the HTML table.
...	other arguments, passed on to formatter functions.

Value

format_html character string with code suitable for inclusion into a HTML-file.

---

format_html.codebook *Format Codebooks as HTML*

Description

This is the method of format_html for "codebook" objects as created by the eponymous function (see codebook)

Usage

```
## S3 method for class 'codebook'
format_html(x,
    toprule = 2, midrule = 1,
    padding = 3,
    var_tag = "code",
    varid_prefix = "", title_tag = "p",...)
```

Arguments

x	a "codebook" object
toprule	a non-negative integer; thickness of the line (in pixels) at the top of each codebook entry
midrule	a non-negative integer; thickness of the line (in pixels) that separates the header of an codebook entry from its body
padding	a non-negative integer; left-hand padding in "ex" of the codebook entry contents
var_tag	character string; the HTML tag that contains the name of the variable
varid_prefix	character string; a prefix added to the anchor IDs of the code entry titles (to facilitate the creation of tables of contents etc.)
title_tag	character string; the HTML tag that contains the title of the codebook entry (the variable name and its description)
...	further arguments, ignored.

See Also

See Also as format_html, show_html, write_html.

format_html.ftable *Format "Flattened Tables" as HTML*

Description

This is the method of format_html for "ftable" objects (i.e. flattened contingency tables)

Usage

```
## S3 method for class 'ftable'
format_html(x,
                    show.titles = TRUE,
                    digits = 0,
                    format = "f",
                    toprule = 2, midrule = 1, bottomrule = 2,
                    split.dec = TRUE,
                    style = ftable_format_stdstyle,
                    margin="2ex auto",
                    ...)
## S3 method for class 'ftable_matrix'
format_html(x,
                    show.titles=TRUE,
                    digits=0,
                    format="f",
                    toprule=2,midrule=1,bottomrule=2,
                    split.dec=TRUE,
                    style = ftable_format_stdstyle,
                    margin="2ex auto",
                    varontop,varinfront,
                    ...)
```

Arguments

x	an object of class ftable.
show.titles	logical; should the names of the cross-classified variables be shown?
digits	number of digits to be shown after the decimal dot. This is only useful, if the "ftable" object was created from a table created with genTable or the like.
format	a format string for formatC
toprule	integer; thickness in pixels of rule at the top of the table.
midrule	integer; thickness in pixels of rules within the table.
bottomrule	integer; thickness in pixels of rule at the bottom of the table.
split.dec	logical; whether numbers should be centered at the decimal point by splitting the table cells.
style	string containing the stanard CSS styling of table cells.

margin	character string, determines the margin and thus the position of the HTML table.
varontop	logical; whether names of column variables should appear on top of factor levels
varinfront	logical; whether names of row variables should appear in front of factor levels
...	further arguments, ignored.

See Also

See Also as format_html, show_html, write_html.

---

format_md *Format Codebooks as Markdown*

Description

format_md is for showing objects in a convenient way in Markdown format. Can be included to Rmarkdown file with the cat() function and the results='asis' code block option. The following example should be runned in a Rmd file with different output formats.

Usage

```
## S3 method for class 'codebook'
format_md(x, ...)
## S3 method for class 'codebookEntry'
format_md(x, name = "", add_rules = TRUE, ...)
```

Arguments

x	a "codebook" or "codebookEntry" object
name	a string; the variable name
add_rules	a boolean value; if TRUE adds a horizontal rules before and after the title
...	further arguments, passed to other functions

Value

format_md character string with code suitable for inclusion into a Markdown-file.

Examples

```
library(memisc)

Data1 <- data.set(
  vote = sample(c(1,2,3,8,9,97,99),size=300,replace=TRUE),
  region = sample(c(rep(1,3),rep(2,2),3,99),size=300,replace=TRUE),
  income = exp(rnorm(300,sd=.7))*2000
)

Data1 <- within(Data1,{
```

```
  description(vote) <- "Vote intention"
  description(region) <- "Region of residence"
  description(income) <- "Household income"
  foreach(x=c(vote,region),{
    measurement(x) <- "nominal"
  })
  measurement(income) <- "ratio"
  labels(vote) <- c(
    Conservatives         =  1,
    Labour                =  2,
    "Liberal Democrats"   =  3,
    "Don't know"          =  8,
    "Answer refused"      =  9,
    "Not applicable"      = 97,
    "Not asked in survey" = 99)
  labels(region) <- c(
    England               =  1,
    Scotland              =  2,
    Wales                 =  3,
    "Not applicable"      = 97,
    "Not asked in survey" = 99)
  foreach(x=c(vote,region,income),{
    annotation(x)["Remark"] <- "This is not a real survey item, of course ..."
  })
  missing.values(vote) <- c(8,9,97,99)
  missing.values(region) <- c(97,99)
})

codebook_data <- codebook(Data1)

codebook_md <- format_md(codebook_data, digits = 2)

writeLines(codebook_md)

## Not run:
writeLines(codebook_md,con="codebook-example.md")

## End(Not run)
```

`ftable-matrix` *Combining flattened tables.*

Description

With the method functions described here, flattened (contingency) tables can be combined into more complex objects, of class `"ftable_matrix"`. For objects of these class `format` and `print` methods are provided

Usage

```
## S3 method for class 'ftable'
cbind(..., deparse.level=1)

## S3 method for class 'ftable'
rbind(..., deparse.level=1)

## S3 method for class 'ftable_matrix'
cbind(..., deparse.level=1)

## S3 method for class 'ftable_matrix'
rbind(..., deparse.level=1)

## S3 method for class 'ftable_matrix'
format(x,quote=TRUE,digits=0,format="f",...)

## S3 method for class 'ftable_matrix'
Write(x,
                            file = "",
                            quote = TRUE,
                            append = FALSE,
                            digits = 0,
                            ...)

## S3 method for class 'ftable_matrix'
print(x,quote=FALSE,...)
```

Arguments

`...`	for `cbind` and `rbind` methods, two or more objects of class `"ftable"` or `"ftable_matrix"`; for the other methods: further arguments, ignored.
`deparse.level`	ignored, retained for compatibility reasons only.
`x`	an object used to select a method.
`quote`	logical, indicating whether or not strings should be printed with surrounding quotes.
`digits`	numeric or integer, number of significant digits to be shown.
`format`	a format string as in `formatC`
`file`	character string, containing a file path.
`append`	logical, should the output appended to the file?

Value

`cbind` and `rbind`, when used with `"ftable"` or `"ftable_matrix"` objects, return objects of class `"ftable_matrix"`.

Examples

```
ft1 <- ftable(Sex~Survived,Titanic)
ft2 <- ftable(Age+Class~Survived,Titanic)
ft3 <- ftable(Survived~Class,Titanic)
ft4 <- ftable(Survived~Age,Titanic)
ft5 <- ftable(Survived~Sex,Titanic)

tab10 <- xtabs(Freq~Survived,Titanic)

(c12.10 <- cbind(ft1,ft2,Total=tab10))
(r345.10 <- rbind(ft3,ft4,ft5,Total=tab10))

## Not run:
tf <- tempfile()
Write(c12.10,file=tf)
file.show(tf)

## End(Not run)
```

---

genTable *Generic Tables and Data Frames of Descriptive Statistics*

Description

genTable creates a table of arbitrary summaries conditional on given values of independent variables given by a formula.

Aggregate does the same, but returns a data.frame instead.

fapply is a generic function that dispatches on its data argument. It is called internally by Aggregate and genTable. Methods for this function can be used to adapt Aggregate and genTable to data sources other than data frames.

Usage

```
Aggregate(formula, data=parent.frame(), subset=NULL,
          names=NULL, addFreq=TRUE, drop = TRUE, as.vars=1,
          ...)

genTable(formula, data=parent.frame(), subset=NULL,
         names=NULL, addFreq=TRUE,...)
```

Arguments

formula	a formula. The right hand side includes one or more grouping variables separated by '+'. These may be factors, numeric, or character vectors. The left hand side may be empty, a numerical variable, a factor, or an expression. See details below.

data	an environment or data frame or an object coercable into a data frame.
subset	an optional vector specifying a subset of observations to be used.
names	an optional character vector giving names to the result(s) yielded by the expression on the left hand side of formula. This argument may be redundant if the left hand side results in is a named vector. (See the example below.)
addFreq	a logical value. If TRUE and data is a table or a data frame with a variable named "Freq", a call to table, Table, percent, or nvalid is supplied by an additional argument Freq and a call to table is translated into a call to Table.
drop	a logical value. If TRUE, empty groups (i.e. when there are no observations in the aggregated data frame that contain the defining combination of values or factor levels of the conditioning variables in by) are dropped from the result of Aggregate. Otherwise, result are filled with NA, where appropriate.
as.vars	an integer; relevant only if the left hand side of the formula returns an array or a matrix - which dimension (rows, columns, or layers etc.) will transformed to variables? Defaults to columns in case of matrices and to the highest dimensional extend in case of arrays.
...	further arguments, passed to methods or ignored.

Details

If an expression is given as left hand side of the formula, its value is computed for any combination of values of the values on the right hand side. If the right hand side is a dot, then all variables in data are added to the right hand side of the formula.

If no expression is given as left hand side, then the frequency counts for the respective value combinations of the right hand variables are computed.

If a single factor is on the left hand side, then the left hand side is translated into an appropriate call to table(). Note that also in this case addFreq takes effect.

If a single numeric variable is on the left hand side, frequency counts weighted by this variable are computed. In these cases, genTable is equivalent to xtabs and Aggregate is equivalent to as.data.frame(xtabs(...)).

Value

Aggregate results in a data frame with conditional summaries and unique value combinations of conditioning variables.

genTable returns a table, that is, an array with class "table".

See Also

aggregate.data.frame, xtabs

Examples

```
ex.data <- expand.grid(mu=c(0,100),sigma=c(1,10))[rep(1:4,rep(100,4)),]
ex.data <- within(ex.data,
                  x<-rnorm(
                    n=nrow(ex.data),
```

```
                    mean=mu,
                    sd=sigma
                    )
                  )

Aggregate(~mu+sigma,data=ex.data)
Aggregate(mean(x)~mu+sigma,data=ex.data)
Aggregate(mean(x)~mu+sigma,data=ex.data,name="Average")
Aggregate(c(mean(x),sd(x))~mu+sigma,data=ex.data)
Aggregate(c(Mean=mean(x),StDev=sd(x),N=length(x))~mu+sigma,data=ex.data)
genTable(c(Mean=mean(x),StDev=sd(x),N=length(x))~mu+sigma,data=ex.data)

Aggregate(table(Admit)~.,data=UCBAdmissions)
Aggregate(Table(Admit,Freq)~.,data=UCBAdmissions)
Aggregate(Admit~.,data=UCBAdmissions)
Aggregate(percent(Admit)~.,data=UCBAdmissions)
Aggregate(percent(Admit)~Gender,data=UCBAdmissions)
Aggregate(percent(Admit)~Dept,data=UCBAdmissions)
Aggregate(percent(Gender)~Dept,data=UCBAdmissions)
Aggregate(percent(Admit)~Dept,data=UCBAdmissions,Gender=="Female")
genTable(percent(Admit)~Dept,data=UCBAdmissions,Gender=="Female")
```

`getSummary` *Get Model Summaries for Use with "mtable"*

Description

A generic function and methods to collect coefficients and summary statistics from a model object. It is used in `mtable`

Usage

```
  ## S3 method for class 'lm'
getSummary(obj, alpha=.05,...)
  ## S3 method for class 'glm'
getSummary(obj, alpha=.05,...)
  ## S3 method for class 'merMod'
getSummary(obj, alpha=.05, ...)

# These are contributed by Christopher N. Lawrence
  ## S3 method for class 'clm'
getSummary(obj, alpha=.05,...)
  ## S3 method for class 'polr'
getSummary(obj, alpha=.05,...)
  ## S3 method for class 'simex'
getSummary(obj, alpha=.05,...)

# These are contributed by Jason W. Morgan
```

```
  ## S3 method for class 'aftreg'
getSummary(obj, alpha=.05,...)
  ## S3 method for class 'coxph'
getSummary(obj, alpha=.05,...)
  ## S3 method for class 'phreg'
getSummary(obj, alpha=.05,...)
  ## S3 method for class 'survreg'
getSummary(obj, alpha=.05,...)
  ## S3 method for class 'weibreg'
getSummary(obj, alpha=.05,...)

# These are contributed by Achim Zeileis
  ## S3 method for class 'ivreg'
getSummary(obj, alpha=.05,...)
  ## S3 method for class 'tobit'
getSummary(obj, alpha=.05,...)
  ## S3 method for class 'hurdle'
getSummary(obj, alpha=.05,...)
  ## S3 method for class 'zeroinfl'
getSummary(obj, alpha=.05,...)
  ## S3 method for class 'betareg'
getSummary(obj, alpha=.05,...)
  ## S3 method for class 'multinom'
getSummary(obj, alpha=.05,...)

# A variant that reports exponentiated coefficients.
# The default method calls 'getSummary()' internally and should
# be applicable to all classes for which 'getSummary()' methods exist.
getSummary_expcoef(obj, alpha=.05,...)
  ## Default S3 method:
getSummary_expcoef(obj, alpha=.05,...)
```

Arguments

obj	a model object, e.g. of class lm or glm
alpha	level of the confidence intervals; their coverage should be 1-alpha/2
...	further arguments; ignored.

Details

The generic function getSummary is called by mtable in order to obtain the coefficients and summaries of model objects. In order to adapt mtable to models of classes other than lm or glm one needs to define getSummary methods for these classes and to set a summary template via setSummaryTemplate

Value

Any method of getSummary must return a list with the following components:

`coef` an array with coefficient estimates; the lowest dimension *must* have the following names and meanings:

`est`	the coefficient estimates,
`se`	the estimated standard errors,
`stat`	t- or Wald-z statistics,
`p`	significance levels of the statistics,
`lwr`	lower confidence limits,
`upr`	upper confidence limits.

The higher dimensions of the array correspond to the individual coefficients and, in multi-equation models, to the model equations.

`sumstat` a vector containing the model summary statistics; the components may have arbitrary names.

`Groups` *Operate on grouped data in data frames and data sets*

Description

`Group` creates a grouped variant of an object of class "data.frame" or of class "data.set", for which methods for `with` and `within` are defined, so that these well-known functions can be applied "groupwise".

Usage

```
# Create an object of class "grouped.data" from a
# data frame or a data set.
Groups(data,by,...)
## S3 method for class 'data.frame'
Groups(data,by,...)
## S3 method for class 'data.set'
Groups(data,by,...)
## S3 method for class 'grouped.data'
Groups(data,by,...)

# Recombine grouped data into a data fame or a data set
recombine(x,...)
## S3 method for class 'grouped.data.frame'
recombine(x,...)
## S3 method for class 'grouped.data.set'
recombine(x,...)

# Recombine grouped data and coerce the result appropriately:
## S3 method for class 'grouped.data'
```

```
as.data.frame(x,...)
## S4 method for signature 'grouped.data.frame'
as.data.set(x,row.names=NULL,...)
## S4 method for signature 'grouped.data.set'
as.data.set(x,row.names=NULL,...)

# Methods of the generics "with" and "within" for grouped data
## S3 method for class 'grouped.data'
with(data,expr,...)
## S3 method for class 'grouped.data'
within(data,expr,recombine=FALSE,...)

# This is equivalent to with(Groups(data,by),expr,...)
withGroups(data,by,expr,...)
# This is equivalent to within(Groups(data,by),expr,recombine,...)
withinGroups(data,by,expr,recombine=TRUE,...)
```

Arguments

data	an object of the classes "data.frame", "data.set" if an argument to Groups, withGroups, withinGroups,
by	a formula with the factors the levels of which define the groups.
expr	an expression, or several expressions enclosed in curly braces.
recombine	a logical vector; should the resulting grouped data be recombined?
x	an object of class "grouped.data".
row.names	an optional character vector with row names.
...	other arguments, ignored.

Details

When applied to a data frame Groups returns an object with class attributes "grouped.data.frame", "grouped.data", and "data.frame", when applied do an object with class "data.set", it returns an object with class attributes "grouped.data.set", "grouped.data", and "data.set".

When applied to objects with class attributed "grouped.data", both the functions with() amd within() evaluate expr separately for each group defined by Groups. with() returns an array composed of the results of expr, while within() returns a modified copy of its data argument, which will be a "grouped.data" object ("grouped.data.frame" or "grouped.data.set"), unless the argument recombine=TRUE is set.

The expression expr may contain references to the variables n_, N_, and i_. n_ is equal to the size of the respective group (the number of rows belonging to it), while N_ is equal to the total number of observations in all groups. The variable i_ equals to the indices of the rows belonging to the respective group of observations.

Examples

```
some.data <- data.frame(x=rnorm(n=100))
some.data <- within(some.data,{
```

```
    f <- factor(rep(1:4,each=25),labels=letters[1:4])
    g <- factor(rep(1:5,each=4,5),labels=LETTERS[1:5])
    y <- x + rep(1:4,each=25) +  0.75*rep(1:5,each=4,5)
})

# For demonstration purposes, we create an
# 'empty' group:
some.data <- subset(some.data,
                      f!="a" | g!="C")

some.grouped.data <- Groups(some.data,
                            ~f+g)

# Computing the means of y for each combination f and g
group.means <- with(some.grouped.data,
                    mean(y))
group.means

# Obtaining a groupwise centered variant of y
some.grouped.data <- within(some.grouped.data,{
    y.cent <- y - mean(y)
},recombine=FALSE)

# The groupwise centered variable should have zero mean
# whithin each group
group.means <- with(some.grouped.data,
                    round(mean(y.cent),15))
group.means

# The following demonstrates the use of n_, N_, and i_
# An external copy of y
y1 <- some.data$y
group.means.n <- with(some.grouped.data,
                      c(mean(y),  # Group means for y
                        n_,       # Group sizes
                        sum(y)/n_,# Group means for y
                        n_/N_,    # Relative group sizes
                        sum(y1)/N_,# NOT the grand mean
                        sum(y1[i_])/n_)) # Group mean for y1
group.means.n

# Names can be attached to the groupwise results
with(some.grouped.data,
     c(Centered=round(mean(y.cent),15),
       Uncentered=mean(y)))

some.data.ungrouped <- recombine(some.grouped.data)
str(some.data.ungrouped)

# It all works with "data.set" objects
some.dataset <- as.data.set(some.data)
some.grouped.dataset <- Groups(some.dataset,~f+g)
```

```
with(some.grouped.dataset,
     c(Mean=mean(y),
       Variance=var(y)))

# The following two expressions are equivalent:
with(Groups(some.data,~f+g),mean(y))
withGroups(some.data,~f+g,mean(y))

# The following two expressions are equivalent:
some.data <- within(Groups(some.data,~f+g),{
    y.cent <- y - mean(y)
    y.cent.1 <- y - sum(y)/n_
})

some.data <- withinGroups(some.data,~f+g,{
    y.cent <- y - mean(y)
    y.cent.1 <- y - sum(y)/n_
})

# Both variants of groupwise centred varaibles should
# have zero groupwise means:
withGroups(some.data,~f+g,{
    c(round(mean(y.cent),15),
      round(mean(y.cent.1),15))
})
```

`html` *Building Blocks for HTML Code*

Description

The functions described here form building blocks for the `format_html` methods functions for `codebook`, `ftable`, `ftable_matrix`, and `mtable` objects, etc.

The most basic of these functions is `html`, which constructs an object that represents a minimal piece of HTML code and is member of the class `"html_elem"`. Unlike a character string containing HTML code, the resulting code element can relatively easily modified using other functions presented here. The actual code is created when the function `as.character` is applied to these objects.

Longer sequences of HTML code can be prepared by concatenating them with `c`, or by `html_group`, or by applying `as.html_group` to a list of `"html_elem"` objects. All these result in objects of class `"html_group"`.

Attributes (such as class, id etc.) of HTML elements can be added to the call to `html`, but can also later recalled or modified with `attribs` or `setAttribs`. An important attribute is the style attribute, which can contain CSS styling. It can be recalled or modified with `style` or `setStyle`. Styling strings can also be created with `hmtl_style` or `as.css`

Usage

```
html(tag, ..., .content = NULL, linebreak = FALSE)
html_group(...)
as.html_group(x)

content(x)
content(x)<-value
setContent(x,value)

attribs(x)
attribs(x)<-value
setAttribs(x,...)
## S3 method for class 'character'
setAttribs(x,...)
## S3 method for class 'html_elem'
setAttribs(x,...)
## S3 method for class 'html_group'
setAttribs(x,...)

css(...)
as.css(x)
style(x)
style(x) <- value
setStyle(x,...)
## S3 method for class 'character'
setStyle(x,...)
## S3 method for class 'html_elem'
setStyle(x,...)
## S3 method for class 'html_group'
setStyle(x,...)
```

Arguments

tag	a character string that determines the opening and closing tags of the HTML element. (The closing tag is relevant only if the element has a content.)
...	optional further arguments, named or not. For `html`: named arguments create the attributes of the HTML element, unnamed arguments define the content of the HTML element, i.e. whatever appears between opening and closing tags (e.g. and). Character strings, `"html_elem"`, or `"html_group"` objects can appear as content of a HTML element. For `setAttribs`: named arguments create the attributes of the HTML element, unnamed arguments are ignored. For `setStyle`: named arguments create the styling of the HTML element, unnamed arguments are ignored. For `html_group`: several objects of class `"html_elem"` or `"html_group"`.

	For `css`: named arguments (character strings!) become components of a styling in CSS format.
`.content`	an optional character string, `"html_elem"`, or `"html_group"` object
`linebreak`	a logical value or vector of length 2, determines whether linebreaks are inserted after the HTML tags.
`x`	an object. For `as.html_group`, this should be a list of objects of class `"html_elem"` or `"html_group"`. For `content`, `setContent`, `attribs`, `setAttribs`, `style`, `setStyle`, this should be an object of class `"html_elem"` or `"html_group"`.
`value`	an object of appropriate class. For `content<-`: a character string, `"html_elem"`, or `"html_group"` object, or a concatenation thereof. For `attribs<-` or `style<-`: a named character vector.

Details

Objects created with `html` are lists with class attribute `"html_elem"` and components

tag a character string

attributes a named character vector

content a character vector, an `"html_elem"` or `"html_group"` object, or a list of such.

linebreak a logical value or vector of length 2.

Objects created with `html_group` or by concatenation of `"html_elem"` or `"html_group"` object are lists of such objects, with class attribute `"html_group"`.

Examples

```
html("img")
html("img",src="test.png")
html("div",class="element",id="first","Sisyphus")
html("div",class="element",id="first",.content="Sisyphus")

div <- html("div",class="element",id="first",linebreak=c(TRUE,TRUE))
content(div) <- "Sisyphus"
div

tag <- html("tag",linebreak=TRUE)
attribs(tag)["class"] <- "something"
attribs(tag)["class"]
tag

style(tag) <- c(color="#342334")
style(tag)
tag

style(tag)["bg"] <- "white"
tag
```

```
setStyle(tag,bg="black")
setStyle(tag,c(bg="black"))

c(div,tag,tag)

c(
  c(div,tag),
  c(div,tag,tag)
)

c(
  c(div,tag),
  div,tag,tag
)

c(
  div,tag,
  c(div,tag,tag)
)

content(div) <- c(tag,tag,tag)
div


css("background-color"="black",
                  color="white")

as.css(c("background-color"="black",
                  color="white"))



Hello <- "Hello World!"
Hello <- html("p",Hello,linebreak=c(TRUE,TRUE))
style(Hello) <- c(color="white",
                  "font-size"="40px",
                  "text-align"="center")

Link <- html("a","More examples here ...",
             href="http://elff.eu/software/memisc",
             title="More examples here ...",
             style=css(color="white"),
             linebreak=c(TRUE,FALSE))
Link <- html("p"," (",Link,")",linebreak=c(TRUE,TRUE))
style(Link) <- c(color="white",
                 "font-size"="15px",
                 "text-align"="center")

Hello <- html("div",c(Hello,Link),linebreak=c(TRUE,TRUE))
style(Hello) <- c("background-color"="#160666",
                  padding="20px")
Hello
```

```
show_html(Hello)
```

Iconv *Convert Annotations, and Value Labels between Encodings*

Description

This function uses the base package function iconv to translate variable descriptions (a.k.a variable labels) and value labels of item, data.set, and importer objects into a specified encoding.

It will be useful in UTF-8 systems when data file come in some ancient encoding like 'Latin-1' as long used by Windows systems.

Usage

```
Iconv(x,from="",to="",...)
## S3 method for class 'annotation'
Iconv(x,from="",to="",...)
## S3 method for class 'data.set'
Iconv(x,from="",to="",...)
## S3 method for class 'importer'
Iconv(x,from="",to="",...)
## S3 method for class 'item'
Iconv(x,from="",to="",...)
## S3 method for class 'value.labels'
Iconv(x,from="",to="",...)
```

Arguments

x	an object of which attributes are to be re-encoded.
from	a character string desribing the original encoding
to	a character string desribing the target encoding
...	further arguments, passed to iconv

Value

Iconv returns a copy of its first argument with re-encoded attributes.

See Also

iconv, iconvlist

Examples

```
## Not run:
# Locate an SPSS 'system' file and get info on variables, their labels etc.
ZA5302 <- spss.system.file("Daten/ZA5302_v6-0-0.sav",to.lower=FALSE)

# Convert labels etc. from 'latin1' to the encoding of the current locale.
ZA5302 <- Iconv(ZA5302,from="latin1")

# Write out the codebook
writeLines(as.character(codebook(ZA5302)),
           con="ZA5302-cdbk.txt")

# Write out the description of the varialbes (their 'variable labels')
writeLines(as.character(description(ZA5302)),
            con="ZA5302-description.txt")

## End(Not run)
```

`importers` *Object Oriented Interface to Foreign Files*

Description

Importer objects are objects that refer to an external data file. Currently only Stata files, SPSS system, portable, and fixed-column files are supported.

Data are actually imported by 'translating' an importer file into a `data.set` using `as.data.set` or `subset`.

The `importer` mechanism is more flexible and extensible than `read.spss` and `read.dta` of package "foreign", as most of the parsing of the file headers is done in R. It is also adapted to efficiently load large data sets. Most importantly, importer objects support the `labels`, `missing.values`, and `descriptions`, provided by this package.

Usage

```
spss.file(file,...)

spss.fixed.file(file,
  columns.file,
  varlab.file=NULL,
  codes.file=NULL,
  missval.file=NULL,
  count.cases=TRUE,
  to.lower=getOption("spss.fixed.to.lower",FALSE),
  iconv=TRUE,
  encoded=getOption("spss.fixed.encoding","cp1252"),
  negative2missing = FALSE)
```

```
spss.portable.file(file,
  varlab.file=NULL,
  codes.file=NULL,
  missval.file=NULL,
  count.cases=TRUE,
  to.lower=getOption("spss.por.to.lower",FALSE),
  iconv=TRUE,
  encoded=getOption("spss.por.encoding","cp1252"),
  negative2missing = FALSE)

spss.system.file(file,
  varlab.file=NULL,
  codes.file=NULL,
  missval.file=NULL,
  count.cases=TRUE,
  to.lower=getOption("spss.sav.to.lower",FALSE),
  iconv=TRUE,
  encoded=getOption("spss.sav.encoding","cp1252"),
  ignore.scale.info = FALSE,
  negative2missing = FALSE)

Stata.file(file,
           iconv=TRUE,
           encoded=if(new_format)
                       getOption("Stata.new.encoding","utf-8")
                   else getOption("Stata.old.encoding","cp1252"),
           negative2missing = FALSE)

## The most important methods for "importer" objects are:
## S3 method for class 'spss.system.importer'
subset(x, subset, select, drop = FALSE, ...)
## S3 method for class 'spss.portable.importer'
subset(x, subset, select, drop = FALSE, ...)
## S3 method for class 'spss.fixed.importer'
subset(x, subset, select, drop = FALSE, ...)
## S3 method for class 'Stata.importer'
subset(x, subset, select, drop = FALSE, ...)
## S3 method for class 'Stata_new.importer'
subset(x, subset, select, drop = FALSE, ...)

## S4 method for signature 'importer'
as.data.set(x,row.names=NULL,optional=NULL,
                    compress.storage.modes=FALSE,...)

## S4 method for signature 'importer'
head(x,n=20,...)
## S4 method for signature 'importer'
tail(x,n=20,...)
```

Arguments

file	character string; the path to the file containing the data
...	Other arguments. spss.file() passes them on to spss.portable.file() of spss.system.file(). Other function ignore further arguments.
columns.file	character string; the path to an SPSS/PSPP syntax file with a DATA LIST FIXED statement
varlab.file	character string; the path to an SPSS/PSPP syntax file with a VARIABLE LABELS statement
codes.file	character string; the path to an SPSS/PSPP syntax file with a VALUE LABELS statement
missval.file	character string; the path to an SPSS/PSPP syntax file with a MISSING VALUES statement
count.cases	logical; should cases in file be counted? This takes effect only if the data file does not already contain information about the number of cases.
to.lower	logical; should variable names changed to lower case?
iconv	logical; should strings (in labels and variables) changed into encoding of the platform?
encoded	a cacharacter string; the way characters are encoded in the improrted file. For the available encoding options see ?iconvlist. Using this argument for spss.system.file this is only a fallback, as the function uses the encoding information present in the file if it is present.
negative2missing	logical; should negative values be marked as missing values? This is the convention of some newer data sets that are available e.g. from the GESIS data archive.
ignore.scale.info	logical; should information about measuremnt scale levels provided in the file be ignored?
x	an object that inherits from class "importer".
subset	a logical vector or an expression containing variables from the external data file that evaluates to logical.
select	a vector of variable names from the external data file. This may also be a named vector, where the names give the names into which the variables from the external data file are renamed.
drop	a logical value, that determines what happens if only one column is selected. If TRUE and only one column is selected, subset returns only a single item object and not a data.set.
row.names	ignored, present only for compatibility.
optional	ignored, present only for compatibility.
compress.storage.modes	logical value; if TRUE floating point values are converted to integers if possible without loss of information.
n	integer; the number of rows to be shown by head or tail

Details

A call to a 'constructor' for an importer object, that is, `spss.fixed.file`, `spss.portable.file`, `spss.sysntax.file`, or `Stata.file`, causes R to read in the header of the data file and/or the syntax files that contain information about the variables, such as the columns that they occupy (in case of `spss.fixed.file`), variable labels, value labels and missing values.

The information in the file header and/or the accompagnying files is then processed to prepare the file for importing. Thus the inner structure of an `importer` object may well vary according to what type of file is to imported and what additional information is given.

The `as.data.set` and `subset` methods for `"importer"` objects internally use the generic functions `seekData`, `readData`, `readSlice`, and `readChunk`, which have methods for the subclasses of `"importer"`. These functions are not callable from outside the package, however.

The `subset` method for `"importer"` objects reads in the data 'chunk-wise' to create the subset of observations if the option `"subset.chunk.size"` is set to a non-`NULL` value, e.g. by `options(subset.chunk.size=1000)`. This may be useful in case of very large data sets from which only a tiny subset of observations is needed for analysis.

Since the functions described here are more or less complete rewrite based on the description of the file structure provided by the documenation for PSPP, they are perhaps not as thorougly tested as the functions in the `foreign` package, apart from the frequent use by the author of this package.

Value

`spss.fixed.file`, `spss.portable.file`, `spss.system.file`, and `Stata.file` return, respectively, objects of class `"spss.fixed.importer"`, `"spss.portable.importer"`, `"spss.system.importer"`, `"Stata.importer"`, or `"Stata_new.importer"`, which, by inheritance, are also objects of class `"importer"`. `"Stata.importer"` is for files in the format of Stata versions up to 12, while `"Stata_new.importer"` is for files in the newer format of Stata versions from 13.

Objects of class `"importer"` have at least the following two slots:

`ptr`	an external pointer
`variables`	a list of objects of class `"item.vector"` which provides a 'prototype' for the `"data.set"` set objects returned by the `as.data.set` and `subset` methods for objects of class `"importer"`

The `as.data.frame` for `importer` objects does the actual data import and returns a data frame. Note that in contrast to `read.spss`, the variable names of the resulting data frame will be lower case, unless the importer function is called with `to.lower=FALSE`. If long variable names are defined (in case of a PSPP/SPSS system file), they take precedence and are *not* coerced to lower case.

See Also

`codebook`, `description`, `read.spss`

Examples

```
# Extract American National Election Study of 1948
nes1948.por <- unzip(system.file("anes/NES1948.ZIP",package="memisc"),
                     "NES1948.POR",exdir=tempfile())
```

```
# Get information about the variables contained.
nes1948 <- spss.portable.file(nes1948.por)

# The data are not yet loaded:
show(nes1948)

# ... but one can see what variables are present:
description(nes1948)

# Now a subset of the data is loaded:
vote.socdem.48 <- subset(nes1948,
              select=c(
                  V480018,
                  V480029,
                  V480030,
                  V480045,
                  V480046,
                  V480047,
                  V480048,
                  V480049,
                  V480050
                  ))

# Let's make the names more descriptive:
vote.socdem.48 <- rename(vote.socdem.48,
                  V480018 = "vote",
                  V480029 = "occupation.hh",
                  V480030 = "unionized.hh",
                  V480045 = "gender",
                  V480046 = "race",
                  V480047 = "age",
                  V480048 = "education",
                  V480049 = "total.income",
                  V480050 = "religious.pref"
        )

# It is also possible to do both
# in one step:
# vote.socdem.48 <- subset(nes1948,
#              select=c(
#                  vote           = V480018,
#                  occupation.hh  = V480029,
#                  unionized.hh   = V480030,
#                  gender         = V480045,
#                  race           = V480046,
#                  age            = V480047,
#                  education      = V480048,
#                  total.income   = V480049,
#                  religious.pref = V480050
#                  ))
```

```
# We examine the data more closely:
codebook(vote.socdem.48)

# ... and conduct some analyses.
#
t(genTable(percent(vote)~occupation.hh,data=vote.socdem.48))

# We consider only the two main candidates.
vote.socdem.48 <- within(vote.socdem.48,{
  truman.dewey <- vote
  valid.values(truman.dewey) <- 1:2
  truman.dewey <- relabel(truman.dewey,
              "VOTED - FOR TRUMAN" = "Truman",
              "VOTED - FOR DEWEY"  = "Dewey")
  })

summary(truman.relig.glm <- glm((truman.dewey=="Truman")~religious.pref,
    data=vote.socdem.48,
    family="binomial",
))
```

`items` *Survey Items*

Description

Objects of class `item` are data vectors with additional information attached to them like "value labels" and "user-defined missing values" known from software packages like SPSS or Stata.

The class `item` is intended to facilitate data management of survey data. Objects in this class should *not* directly used in data analysis. Instead they should changed into "ordinary" vectors or factors before. For this see the documentation for `as.vector,item-method`.

Usage

```
## The constructor for objects of class "item"
## more convenient than new("item",...)
## S4 method for signature 'numeric'
as.item(x,
  labels=NULL,  missing.values=NULL,
  valid.values=NULL,  valid.range=NULL,
  value.filter=NULL,  measurement=NULL,
  annotation=attr(x,"annotation"), ...
  )
## S4 method for signature 'character'
as.item(x,
  labels=NULL,  missing.values=NULL,
  valid.values=NULL,  valid.range=NULL,
```

```
  value.filter=NULL,  measurement=NULL,
  annotation=attr(x,"annotation"), ...
  )

## S4 method for signature 'logical'
as.item(x,...)
# x is first coerced to integer,
# arguments in ... are then passed to the "numeric"
# method.

## S4 method for signature 'factor'
as.item(x,...)
## S4 method for signature 'ordered'
as.item(x,...)
## S4 method for signature 'POSIXct'
as.item(x,...)

## S4 method for signature 'double.item'
as.item(x,
  labels=NULL,  missing.values=NULL,
  valid.values=NULL,  valid.range=NULL,
  value.filter=NULL,  measurement=NULL,
  annotation=attr(x,"annotation"), ...
  )

## S4 method for signature 'integer.item'
as.item(x,
  labels=NULL,  missing.values=NULL,
  valid.values=NULL,  valid.range=NULL,
  value.filter=NULL,  measurement=NULL,
  annotation=attr(x,"annotation"), ...
  )

## S4 method for signature 'character.item'
as.item(x,
  labels=NULL,  missing.values=NULL,
  valid.values=NULL,  valid.range=NULL,
  value.filter=NULL,  measurement=NULL,
  annotation=attr(x,"annotation"), ...
  )

## S4 method for signature 'datetime.item'
as.item(x,
  labels=NULL,  missing.values=NULL,
  valid.values=NULL,  valid.range=NULL,
  value.filter=NULL,  measurement=NULL,
  annotation=attr(x,"annotation"), ...
  )
```

Arguments

x	for as.item methods, any atomic vector; for the unique, summary, str, print, [, and <- methods, a vector with class labelled.
labels	a named vector of the same mode as x.
missing.values	either a vector of the same mode as x, or a list with components "values", vector of the same mode as x (which defines individual missing values) and "range" a matrix with two rows with the same mode as x (which defines a range of missing values), or an object of class "missing.values".
valid.values	either a vector of the same mode as x, defining those values of x that are to be considered as valid, or an object of class "valid.values".
valid.range	either a vector of the same mode as x and length 2, defining a range of valid values of x, or an object of class "valid.range".
value.filter	an object of class "value.filter", that is, of classes "missing.values", "valid.values", or "valid.range".
measurement	level of measurement; one of "nominal", "ordinal", "interval", or "ratio".
annotation	a named character vector, or an object of class "annotation"
...	further arguments, ignored.

See Also

annotation labels value.filter

Examples

```
x <- as.item(rep(1:5,4),
    labels=c(
        "First"      = 1,
        "Second"     = 2,
        "Third"      = 3,
        "Fourth"     = 4,
        "Don't know" = 5
      ),
    missing.values=5,
    annotation = c(
      description="test"
    ))
str(x)
summary(x)
as.numeric(x)

test <- as.item(rep(1:6,2),labels=structure(1:6,
                                    names=letters[1:6]))
test
test == 1
test != 1
test == "a"
```

```
test != "a"
test == c("a","z")
test != c("a","z")
test
test

codebook(test)

Test <- as.item(rep(letters[1:6],2),
                labels=structure(letters[1:6],
                                 names=LETTERS[1:6]))
Test
Test == "a"
Test != "a"
Test == "A"
Test != "A"
Test == c("a","z")
Test != c("a","z")
Test
Test

as.factor(test)
as.factor(Test)
as.numeric(test)
as.character(test)
as.character(Test)

as.data.frame(test)[[1]]
```

`items-to-vectors` *How Survey Items Are Converted into "Ordinary" Data Vectors*

Description

Survey item objects in are numeric or character vectors with some extra information that may helpful for for managing and documenting survey data, but they are not suitable for statistical data analysis. To run regressions etc. one should convert `item` objects into "ordinary" numeric vectors or factors. This means that codes or values declared as "missing" (if present) are translated into the generial missing value `NA`, while value labels (if defined) are translated into factor levels.

Usage

```
# The following methods can be used to covert items into
# vectors with a given mode or into factors.
## S4 method for signature 'item'
as.vector(x, mode = "any")
## S4 method for signature 'item'
as.numeric(x, ...)
## S4 method for signature 'item'
```

```
as.integer(x, ...)
## S4 method for signature 'item.vector'
as.factor(x)
## S4 method for signature 'item.vector'
as.ordered(x)
## S4 method for signature 'item.vector'
as.character(x, use.labels = TRUE, include.missings = FALSE, ...)
## S4 method for signature 'datetime.item.vector'
as.character()
## S4 method for signature 'Date.item.vector'
as.character()
# The following methods are unlikely to be useful in practice, other than
# that they are called internally by the 'as.data.frame()' method for "data.set"
# objects.
## S3 method for class 'character.item'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
## S3 method for class 'double.item'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
## S3 method for class 'integer.item'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
## S3 method for class 'Date.item'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
## S3 method for class 'datetime.item'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

x	an object in class "item","item.vector", etc., as relevant for the respective conversion method.
mode	the mode of the vector to be returned, usually "numeric", "integer", or "charcater"
use.labels	logical,should value labels be used for creating the character vector?
include.missings	logical; if TRUE, declared missing values are not converted into NA, but into character strings with "*" as the "missingness marker" added at the beginning.
row.names	optional row names, see as.data.frame
optional	a logical value, see as.data.frame
...	other arguments, ignored.

Value

The function as.vector() returns a logical, numeric, or character depending on the mode= argument. If mode="any", the vector has the mode that corresponds to the (internal) mode of the item vector, that is, an item in class "integer.item" will become an integer vector, an item in class "double.item" will become a double-precision numeric vector, an item in class "character.item" will become a character vector; since the internal mode of a "dateitem.item" or a "Date.item" vector is numeric, a numeric vector will be returned.

The functions `as.integer()`, `as.numeric()`, `as.character()`, `as.factor()`, and `as.ordered()` return an integer, numeric, or character vector, or an ordered or unordered factor, respectively.

When `as.data.frame()` is applied to an survey item object, the result is a single-column data frame, where the single column is a numeric vector or character vector or factor depending on the `measurement` attribute of the item. In particular, if the `measurement` attribute equals `"ratio"` or `"interval"` this column will be the result of `as.vector()`, if the `measurement` attribute equals `"ordinal"` this column will be an ordered factor (see `ordered`), and if the `measurement` attribute equals `"nominal"` this column will be an unordered factor (see `factor`).

All these functions have in common that values declared as "missing" by virtue of the `value.filter` attribute will be turned into `NA`.

See Also

`items` `annotation` `labels` `value.filter`

Examples

```
x <- as.item(rep(1:5,4),
    labels=c(
        "First"      = 1,
        "Second"     = 2,
        "Third"      = 3,
        "Fourth"     = 4,
        "Don't know" = 5
      ),
    missing.values=5,
    annotation = c(
      description="test"
    ))
str(x)
summary(x)
as.numeric(x)

test <- as.item(rep(1:6,2),labels=structure(1:6,
                                    names=letters[1:6]))

as.factor(test)
as.numeric(test)
as.character(test)
as.character(test,include.missings=TRUE)

as.data.frame(test)[[1]]
```

`labels` *Value Labels*

Description

Value labels associate character labels to possible values of an encoded survey item. Value labels are represented as objects of class "value.labels".

Value labels of an item can be obtained using `labels(x)` and can be associated to items and to vectors using labels(x) <- value

Value labels also can be updated using the + and - operators.

Usage

```
labels(object,...)
labels(x) <- value
```

Arguments

`object`	any object.
`...`	further arguments for other methods.
`x`	a vector or "item" object.
`value`	an object of class "value.labels" or a vector that can be coerced into an "value.labels" object or NULL

Examples

```
x <- as.item(rep(1:5,4),
    labels=c(
        "First"      = 1,
        "Second"     = 2,
        "Third"      = 3,
        "Fourth"     = 4,
        "Don't know" = 5
      ),
    missing.values=5,
    annotation = c(
      description="test"
    ))
labels(x)
labels(x) <- labels(x) - c("Second"=2)
labels(x)
labels(x) <- labels(x) + c("Second"=2)
labels(x)

puvl <- getOption("print.use.value.labels")
options(print.use.value.labels=FALSE)
x
options(print.use.value.labels=TRUE)
x
options(print.use.value.labels=puvl)
```

List — *Create a list and conveniently supply names to its elements*

Description

`List` creates a list and names its elements after the arguments given, in a manner analogously to `data.frame`

Usage

```
List(...)
```

Arguments

`...`	tagged or untagged arguments from which the list is formed. If the untagged arguments are variables from the englosing environment, their names become the names of the list elements.

Examples

```
num <- 1:3
strng <- c("a","b","A","B")
logi <- rep(FALSE,7)
List(num,strng,logi)
```

Mean — *Convenience wrappers for common statistical functions*

Description

`Mean()`, `Median()`, etc. are mere wrappers of the functions `mean()`, `median()`, etc. with the `na.rm=` optional argument set `TRUE` by default.

Usage

```
Mean(x, na.rm=TRUE, ...)
Median(x, na.rm=TRUE, ...)
Min(x, na.rm=TRUE, ...)
Max(x, na.rm=TRUE, ...)
Weighted.Mean(x, w, ..., na.rm = TRUE)
Var(x, na.rm=TRUE, ...)
StdDev(x, na.rm=TRUE, ...)

Cov(x, y = NULL, use = "pairwise.complete.obs", ...)
Cor(x, y = NULL, use = "pairwise.complete.obs", ...)
Range(..., na.rm = TRUE, finite = FALSE)
```

Arguments

x	a (numeric) vector.
y	a (numeric) vector or NULL.
w	a (numeric) vector of weights.
na.rm	a logical value, see mean.
use	a character string, see cor.
...	other arguments, passed to the wrapped functions.
finite	a logical value, see range.

Means *Means for groups of observations*

Description

The function Means() creates a table of group means, optionally with standard errors, confidence intervals, and numbers of valid observations.

Usage

```
Means(data, ...)
## S3 method for class 'data.frame'
Means(data,
    by, weights=NULL, subset=NULL,
    default=NA,
    se=FALSE, ci=FALSE, ci.level=.95,
    counts=FALSE, ...)
## S3 method for class 'formula'
Means(data, subset, weights, ...)
## S3 method for class 'numeric'
Means(data, ...)
## S3 method for class 'means.table'
as.data.frame(x, row.names=NULL, optional=TRUE, drop=TRUE, ...)
## S3 method for class 'xmeans.table'
as.data.frame(x, row.names=NULL, optional=TRUE, drop=TRUE, ...)
```

Arguments

data	an object usually containing data, or a formula. If data is a numeric vector or an object that can be coerced into a data frame, it is changed into a data frame and the data frame method of Means() is applied to it. If data is a formula, then a data frame is constructed from the variables in the formula and Means is applied to this data frame, while the formula is passed on as a by= argument.

by	a formula, a vector of variable names or a data frame or list of factors. If by is a vector of variable names, they are extracted from data to define the groups for which means are computed, while the variables for which the means are computed are those not named in by. If by is a data frame or a list of factors, these are used to defined the groups for which means are computed, while the variables for which the means are computed are those not in by. If by is a formula, its left-hand side determines the variables of which means are computed, while its right-hand side determines the factors that define the groups.
weights	an optional vector of weights, usually a variable in data.
subset	an optional logical vector to select observations, usually the result of an expression in variables from data.
default	a default value used for empty cells without observations.
se	a logical value, indicates whether standard errors should be computed.
ci	a logical value, indicates whether limits of confidence intervals should be computed.
ci.level	a number, the confidence level of the confidence interval
counts	a logical value, indicates whether numbers of valid observations should be reported.
x	for as.data.frame(), a result of Means().
row.names	an optional character vector. This argmument presently is inconsequential and only included for reasons of compatiblity with the standard methods of as.data.frame.
optional	an optional logical value. This argmument presently is inconsequential and only included for reasons of compatiblity with the standard methods of as.data.frame.
drop	a logical value, determines whether "empty cells" should be dropped from the resulting data frame.
...	other arguments, either ignored or passed on to other methods where applicable.

Value

An array that inherits classes "means.table" and "table". If Means was called with se=TRUE or ci=TRUE then the result additionally inherits class "xmeans.table".

Examples

```
# Preparing example data
USstates <- as.data.frame(state.x77)
USstates <- within(USstates,{
    region <- state.region
    name <- state.name
    abb <- state.abb
    division <- state.division
})
USstates$w <- sample(runif(n=6),size=nrow(USstates),replace=TRUE)
```

```
# Using the data frame method
Means(USstates[c("Murder","division","region")],by=c("division","region"))
Means(USstates[c("Murder","division","region")],by=USstates[c("division","region")])
Means(USstates[c("Murder")],1)
Means(USstates[c("Murder","region")],by=c("region"))

# Using the formula method
# One 'dependent' variable
Means(Murder~1, data=USstates)
Means(Murder~division, data=USstates)
Means(Murder~division, data=USstates,weights=w)
Means(Murder~division+region, data=USstates)
as.data.frame(Means(Murder~division+region, data=USstates))

# Standard errors and counts
Means(Murder~division, data=USstates, se=TRUE, counts=TRUE)
drop(Means(Murder~division, data=USstates, se=TRUE, counts=TRUE))
as.data.frame(Means(Murder~division, data=USstates, se=TRUE, counts=TRUE))

# Confidence intervals
Means(Murder~division, data=USstates, ci=TRUE)
drop(Means(Murder~division, data=USstates, ci=TRUE))
as.data.frame(Means(Murder~division, data=USstates, ci=TRUE))

# More than one dependent variable
Means(Murder+Illiteracy~division, data=USstates)
as.data.frame(Means(Murder+Illiteracy~division, data=USstates))

# Confidence intervals
Means(Murder+Illiteracy~division, data=USstates, ci=TRUE)
as.data.frame(Means(Murder+Illiteracy~division, data=USstates, ci=TRUE))

# Some 'non-standard' but still valid usages:
with(USstates,
     Means(Murder~division+region,subset=region!="Northeast"))

with(USstates,
     Means(Murder,by=list(division,region)))
```

`measurement` *Levels of Measurement of Survey Items*

Description

The measurement level of a `"item"` object, which is one of "nominal", "ordinal", "interval", "ratio", determines what happens to it, if it or the `data.set` containing it is coerced into a `data.frame`. If the level of measurement level is "nominal", the it will be converted into an (unordered) factor, if the level of measurement is "ordinal", the item will be converted into an ordered vector. If the measurement is "interval" or "ratio", the item will be converted into a numerical vector.

Usage

```
## S4 method for signature 'item'
measurement(x)
## S4 replacement method for signature 'item'
measurement(x) <- value
## S4 method for signature 'data.set'
measurement(x)
## S4 replacement method for signature 'data.set'
measurement(x) <- value
is.nominal(x)
is.ordinal(x)
is.interval(x)
is.ratio(x)
as.nominal(x)
as.ordinal(x)
as.interval(x)
as.ratio(x)
set_measurement(x,...)
```

Arguments

x	an object, usually of class "item".
value	for the item method, a character string; either "nominal", "ordinal", "interval", or "ratio"; for the data.set method, a list of character vectors with variable names, where the names of the list corresponds to a measurement level and and the list elements indicates the variables to which the measurement levels are assigned.
...	vectors of variable names, either symbols or character strings, tagged with the intended measurement level.

Value

The item method of measurement(x) returns a character string, the data.set method returns a named character vector, where the name of each element is a variable name and each.

as.nominal, as.ordinal, as.interval, as.ratio return an item with the requested level of measurement setting.

is.nominal, is.ordinal, is.interval, is.ratio return a logical value.

References

Stevens, Stanley S. 1946. "On the theory of scales of measurement." *Science* 103: 677-680.

See Also

data.set, item

Examples

```
vote <- sample(c(1,2,3,8,9),size=30,replace=TRUE)
labels(vote) <- c(Conservatives         =  1,
                  Labour                =  2,
                  "Liberal Democrats"   =  3,
                  "Don't know"          =  8,
                  "Answer refused"      =  9
                  )
missing.values(vote) <- c(8,9)
as.data.frame(vote)[[1]]
measurement(vote) <- "interval"
as.data.frame(vote)[[1]]
vote <- as.nominal(vote)
as.data.frame(vote)[[1]]
group <- sample(c(1,2),size=30,replace=TRUE)
labels(group) <- c(A=1,B=2)
DataS <- data.set(group,vote)
measurement(DataS)
measurement(DataS) <- list(interval=c("group","vote"))
head(as.data.frame(DataS))
DataS <- set_measurement(DataS,
                         nominal=c(group,vote))
head(as.data.frame(DataS))
```

---

measurement_autolevel *Automatically Adapt Measurement Levels*

Description

The generic function `measurement_autolevel` changes the measurement levels of "item" objects to "nominal" or "ordinal", if the proportion of its values that have labels is above a certain threshold.

Usage

```
measurement_autolevel(x, ...)
## S4 method for signature 'ANY'
measurement_autolevel(x, ...) # Returns its argument as is
## S4 method for signature 'item.vector'
measurement_autolevel(x,
                to=getOption("measurement.adapt.default","nominal"),
                threshold=getOption("measurement.adapt.threshold",.75),
                ...)
## S4 method for signature 'data.set'
measurement_autolevel(x,
                to=getOption("measurement.adapt.default","nominal"),
                threshold=getOption("measurement.adapt.threshold",.75),
                except=NULL,
                only=NULL,
                ...)
```

Arguments

x	an object from class "item.vector" or "data.set".
to	a character vector, the target measurement level
threshold	the proportion of values, if reached the target measurement level is set
except	a vector with variable names, either as symbols (without quotation marks) or character strings (with quotation markes), the variables in the data set that are not to be changed by `measurement_autolevel()`.
only	a vector with variable names, either as symbols (without quotation marks) or character strings (with quotation markes), the variables in the data set that are to be changed by `measurement_autolevel()`.
...	other arguments, currently ignored.

Examples

```
exvect <- as.item(rep(1:2,5))
labels(exvect) <- c(a=1,b=2)
codebook(exvect)
codebook(measurement_autolevel(exvect))

avect <- as.item(sample(1:3,16,replace=TRUE))
labels(avect) <- c(a=1,b=2,c=3)
bvect <- as.item(sample(1:4,16,replace=TRUE))
labels(bvect) <- c(A=1,B=2,C=3,D=4)
ds <- data.set(a=avect,b=bvect)
codebook(ds)
codebook(measurement_autolevel(ds))
codebook(measurement_autolevel(ds,except=c(a,b)))
codebook(measurement_autolevel(ds,only=a))
```

Memisc *Introduction to the 'memisc' Package*

Description

This package collects an assortment of tools that are intended to make work with R easier for the author of this package and are submitted to the public in the hope that they will be also be useful to others.

The tools in this package can be grouped into four major categories:

- Data preparation and management
- Data analysis
- Presentation of analysis results
- Programming

Data preparation and management

Survey Items: `memisc` provides facilities to work with what users from other packages like SPSS, SAS, or Stata know as 'variable labels', 'value labels' and 'user-defined missing values'. In the context of this package these aspects of the data are represented by the `"description"`, `"labels"`, and `"missing.values"` attributes of a data vector. These facilities are useful, for example, if you work with survey data that contain coded items like vote intention that may have the following structure:

Question: "If there was a parliamentary election next tuesday, which party would you vote for?"

1	Conservative Party
2	Labour Party
3	Liberal Democrat Party
4	Scottish Nation Party
5	Plaid Cymru
6	Green Party
7	British National Party
8	Other party
96	Not allowed to vote
97	Would not vote
98	Would vote, do not know yet for which party
99	No answer

A statistical package like SPSS allows to attach labels like 'Conservative Party', 'Labour Party', etc. to the codes 1,2,3, etc. and to mark mark the codes 96, 97, 98, 99 as 'missing' and thus to exclude these variables from statistical analyses. `memisc` provides similar facilities. Labels can be attached to codes by calls like `labels(x) <- something` and expendanded by calls like `labels(x) <- labels(x) + something`, codes can be marked as 'missing' by calls like `missing.values(x) <- something` and `missing.values(x) <- missing.values(x) + something`.

`memisc` defines a class called "data.set", which is similar to the class "data.frame". The main difference is that it is especially geared toward containing survey item data. Transformations of and within "data.set" objects retain the information about value labels, missing values etc. Using `as.data.frame` sets the data up for *R*'s statistical functions, but doing this explicitely is seldom necessary. See `data.set`.

More Convenient Import of External Data: Survey data sets are often relative large and contain up to a few thousand variables. For specific analyses one needs however only a relatively small subset of these variables. Although modern computers have enough RAM to load such data sets completely into an R session, this is not very efficient having to drop most of the variables after loading. Also, loading such a large data set completely can be time-consuming, because R has to allocate space for each of the many variables. Loading just the subset of variables really needed for an analysis is more efficient and convenient - it tends to be much quicker. Thus this package provides facilities to load such subsets of variables, without the need to load a complete data set. Further, the loading of data from SPSS files is organized in such a way that all informations about variable labels, value labels, and user-defined missing values are retained. This is made possible by the definition of `importer` objects, for which a `subset` method exists. `importer` objects contain only the information about the variables in the external data set but not the data. The data itself is loaded into memory when the functions `subset` or `as.data.set` are used.

Recoding: `memisc` also contains facilities for recoding survey items. Simple recodings, for example collapsing answer categories, can be done using the function `recode`. More complex recodings, for example the construction of indices from multiple items, and complex case distinctions, can be done using the function `cases`. This function may also be useful for programming, in so far as it is a generalization of `ifelse`.

Code Books: There is a function `codebook` which produces a code book of an external data set or an internal "data.set" object. A codebook contains in a conveniently formatted way concise information about every variable in a data set, such as which value labels and missing values are defined and some univariate statistics.

An extended example of all these facilities is contained in the vignette "anes48", and in `demo(anes48)`

Data Analysis

Tables and Data Frames of Descriptive Statistics: `genTable` is a generalization of `xtabs`: Instead of counts, also descriptive statistics like means or variances can be reported conditional on levels of factors. Also conditional percentages of a factor can be obtained using this function.

In addition an `Aggregate` function is provided, which has the same syntax as `genTable`, but gives a data frame of descriptive statistics instead of a `table` object.

Per-Subset Analysis: `By` is a variant of the standard function `by`: Conditioning factors are specified by a formula and are obtained from the data frame the subsets of which are to be analysed. Therefore there is no need to `attach` the data frame or to use the dollar operator.

Presentation of Results of Statistical Analysis

Publication-Ready Tables of Coefficients:

Journals of the Political and Social Sciences usually require that estimates of regression models are presented in the following form:

```
================================================
                Model 1     Model 2     Model 3
------------------------------------------------
Coefficients
(Intercept)     30.628***    6.360***   28.566***
                (7.409)     (1.252)     (7.355)
pop15           -0.471**                -0.461**
                (0.147)                 (0.145)
pop75           -1.934                  -1.691
                (1.041)                 (1.084)
dpi                          0.001      -0.000
                            (0.001)     (0.001)
ddpi                         0.529*      0.410*
                            (0.210)     (0.196)
------------------------------------------------
Summaries
R-squared         0.262       0.162       0.338
adj. R-squared    0.230       0.126       0.280
N                50          50          50
================================================
```

Such tables of coefficient estimates can be produced by `mtable`. To see some of the possibilities of this function, use `example(mtable)`.

LaTeX Representation of R Objects: Output produced by `mtable` can be transformed into LaTeX tables by an appropriate method of the generic function `toLatex` which is defined in the package `utils`. In addition, `memisc` defines `toLatex` methods for matrices and `ftable` objects. Note that results produced by `genTable` can be coerced into `ftable` objects. Also, a default method for the `toLatex` function is defined which coerces its argument to a matrix and applies the matrix method of `toLatex`.

Programming

Looping over Variables: Sometimes users want to contruct loops that run over variables rather than values. For example, if one wants to set the missing values of a battery of items. For this purpose, the package contains the function `foreach`. To set 8 and 9 as missing values for the items `knowledge1`, `knowledge2`, `knowledge3`, one can use

```
foreach(x=c(knowledge1,knowledge2,knowledge3),
    missing.values(x) <- 8:9)
```

Changing Names of Objects and Labels of Factors: `R` already makes it possible to change the names of an object. Substituting the `names` or `dimnames` can be done with some programming tricks. This package defines the function `rename`, `dimrename`, `colrename`, and `rowrename` that implement these tricks in a convenient way, so that programmers (like the author of this package) need not reinvent the weel in every instance of changing names of an object.

Dimension-Preserving Versions of `lapply` and `sapply`: If a function that is involved in a call to `sapply` returns a result an array or a matrix, the dimensional information gets lost. Also, if a list object to which `lapply` or `sapply` are applied have a dimension attribute, the result looses this information. The functions `Lapply` and `Sapply` defined in this package preserve such dimensional information.

Combining Vectors and Arrays by Names: The generic function `collect` collects several objects of the same mode into one object, using their names, `rownames`, `colnames` and/or `dimnames`. There are methods for atomic vectors, arrays (including matrices), and data frames. For example

```
a <- c(a=1,b=2)
b <- c(a=10,c=30)
collect(a,b)
```

leads to

```
   x  y
a  1 10
b  2 NA
c NA 30
```

Reordering of Matrices and Arrays: The `memisc` package includes a `reorder` method for arrays and matrices. For example, the matrix method by default reorders the rows of a matrix according the results of a function.

memisc-deprecated *Deprecated Functions in Package* **memisc**

Description

These functions are provided for compatibility with older versions of **memisc** only, and may be defunct as soon as the next release.

Usage

```
fapply(formula,data,...) # calls UseMethod("fapply",data)
## Default S3 method:
fapply(formula, data, subset=NULL,
      names=NULL, addFreq=TRUE,...)
```

Arguments

formula	a formula. The right hand side includes one or more grouping variables separated by '+'. These may be factors, numeric, or character vectors. The left hand side may be empty, a numerical variable, a factor, or an expression. See details below.
data	an environment or data frame or an object coercable into a data frame.
subset	an optional vector specifying a subset of observations to be used.
names	an optional character vector giving names to the result(s) yielded by the expression on the left hand side of formula. This argument may be redundant if the left hand side results in is a named vector. (See the example below.)
addFreq	a logical value. If TRUE and data is a table or a data frame with a variable named "Freq", a call to table, Table, percent, or nvalid is supplied by an additional argument Freq and a call to table is translated into a call to Table.
...	further arguments, passed to methods or ignored.

mtable *Comparative Table of Model Estimates*

Description

mtable produces a table of estimates for several models.

Usage

```
mtable(...,coef.style=getOption("coef.style"),
    summary.stats=TRUE,
    signif.symbols=getOption("signif.symbols"),
    factor.style=getOption("factor.style"),
    show.baselevel=getOption("show.baselevel"),
    baselevel.sep=getOption("baselevel.sep"),
    getSummary=eval.parent(quote(getSummary)),
    float.style=getOption("float.style"),
    digits=min(3,getOption("digits")),
    sdigits=digits,
    show.eqnames=getOption("mtable.show.eqnames",NA),
    gs.options=NULL,
    controls=NULL,
    collapse.controls=FALSE,
    control.var.indicator=getOption("control.var.indicator",c("Yes","No"))
  )
## S3 method for class 'memisc_mtable'
relabel(x, ..., gsub = FALSE, fixed = !gsub, warn = FALSE)

## S3 method for class 'memisc_mtable'
format(x,target=c("print","LaTeX","HTML","delim"),
    ...
    )

## S3 method for class 'memisc_mtable'
print(x,
    center.at=getOption("OutDec"),
    topsep="=",bottomsep="=",sectionsep="-",...)

write.mtable(object,file="",
             format=c("delim","LaTeX","HTML"),...)

## S3 method for class 'memisc_mtable'
toLatex(object,...)
```

Arguments

`...`	as argument to `mtable`: several model objects, e.g. of class `lm`; as argument to `print.memisc_mtable`, `toLatex.memisc_mtable`, `write.memisc_mtable`: further arguments passed to `format.memisc_mtable`; as argument to `format.memisc_mtable`: further arguments passed to `format.default`; as argument to `relabel.memisc_mtable`: further arguments passed to `dimrename`.
`coef.style`	a character string which specifies the style of coefficient values, whether standard errors, Wald/t-statistics, or significance levels are reported, etc. See `coef.style`.
`summary.stats`	if `FALSE`, no summary statistics are repored. If `TRUE` then for each object in `...` either all summary statistics are reported, or those specified by the option `"summary.stats.<cls>"`, where `<cls>` is the class of the respective object.

This argument may also contain a character vector with the names of the summary statistics to report, or a list of character vectors with names of summary statistics for each object passed as argument in ....

`signif.symbols` a named numeric vector to specify the "significance levels" and corresponding symbols. The numeric elements define the significance levels, the attached names define the associated symbols.

`factor.style` a character string that specifies the style in which factor contrasts are labled. See `factor.style`.

`show.baselevel` logical; determines whether base levels of factors are indicated for dummy coefficients

`baselevel.sep` character that is used to separate the base level from the level that a dummy variable represents

`getSummary` a function that computes model-related statistics that appear in the table. See `getSummary`.

`float.style` default format for floating point numbers if no format is specified by `coef.style`.

`digits` number of significant digits if not specified by the template returned from `getCoefTemplate` `getSummaryTemplate`

`sdigits` integer; number of digits after decimal dot for summary statistics.

`show.eqnames` logical; if `TRUE`, left-hand sides of equations are (always) shown in the table header; if `FALSE`, left-hand sides of equations are not shown; if `NA`, left-hand sides of equations are shown only if left-hand sides differ among models or one of the models has multiple equations.

`gs.options` an optional list of arguments passed on to `getSummary`

`controls` an optional formula or character vector that designates "control variables" for which no coefficients are reported, but only whether they are present in the model.

`collapse.controls`
a logical values; should the report about inclusion of control variables collapsed to a single value? If yes, models should either contain none or all of the control variables.

`control.var.indicator`
a character vector with to elements; the first element being used to indicate the presence of a control variable or all control variables (if `collapse.controls=TRUE`), the second element being used otherwise. By default these elements are `"Yes"` and `"No"`.

`x, object` an object of class `mtable`

`gsub, warn, fixed`
logical values, see `relabel`

`target` a character string which indicates the target format. Currenlty the targets "print" (see `mtable_format_print`), "LaTeX" (see `mtable_format_latex`), "HTML" (see `mtable_format_html`), and "delim" (see `mtable_format_delim`) are supported.

`center.at` a character string on which resulting values are centered. Typically equal to ".". This is the default when `forLaTeX==TRUE`. If `NULL`, reported values are not centered.

topsep	a character string that is recycled to a top rule.
bottomsep	a character string that is recycled to a bottom rule.
sectionsep	a character string that is recycled to seperate coefficients from summary statistics.
file	name of the file where to write to; defaults to console output.
format	character string that specifies the desired format.

Details

mtable constructs a table of estimates for regression-type models. format.memisc_mtable formats suitable for use with output or conversion functions such as print.memisc_mtable, toLatex.memisc_mtable, or write.memisc_mtable.

Value

A call to mtable results in an object of class "mtable" with the following components:

coefficients	a list that contains the model coefficients,
summaries	a matrix that contains the model summaries,
calls	a list of calls that created the model estimates being summarised.

Examples

```
#### Basic workflow

lm0 <- lm(sr ~ pop15 + pop75,              data = LifeCycleSavings)
lm1 <- lm(sr ~                 dpi + ddpi, data = LifeCycleSavings)
lm2 <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings)

options(summary.stats.lm=c("R-squared","N"))
mtable("Model 1"=lm0,"Model 2"=lm1,"Model 3"=lm2)

options(summary.stats.lm=c("sigma","R-squared","N"))
mtable("Model 1"=lm0,"Model 2"=lm1,"Model 3"=lm2)

options(summary.stats.lm=NULL)

mtable123 <- mtable("Model 1"=lm0,"Model 2"=lm1,"Model 3"=lm2,
    summary.stats=c("sigma","R-squared","F","p","N"))

(mtable123 <- relabel(mtable123,
  "(Intercept)" = "Constant",
          pop15 = "Percentage of population under 15",
          pop75 = "Percentage of population over 75",
            dpi = "Real per-capita disposable income",
           ddpi = "Growth rate of real per-capita disp. income"
  ))

# This produces output in tab-delimited format:
write.mtable(mtable123)
```

```
## Not run:
# This produces output in tab-delimited format:
file123 <- "mtable123.txt"
write.mtable(mtable123,file=file123)
file.show(file123)
# The contents of this file can be pasted into Word
# and converted into a Word table.

## End(Not run)

## Not run: texfile123 <- "mtable123.tex"
write.mtable(mtable123,format="LaTeX",file=texfile123)
file.show(texfile123)
## End(Not run)

#### Examples with UC Berkeley data

berkeley <- Aggregate(Table(Admit,Freq)~.,data=UCBAdmissions)

berk0 <- glm(cbind(Admitted,Rejected)~1,data=berkeley,family="binomial")
berk1 <- glm(cbind(Admitted,Rejected)~Gender,data=berkeley,family="binomial")
berk2 <- glm(cbind(Admitted,Rejected)~Gender+Dept,data=berkeley,family="binomial")

mtable(berk0,summary.stats=c("Deviance","N"))
mtable(berk1,summary.stats=c("Deviance","N"))

mtable(berk0,berk1,berk2,summary.stats=c("Deviance","N"))

mtable(berk0,berk1,berk2,
          coef.style="horizontal",
          summary.stats=c("Deviance","AIC","N"))
mtable(berk0,berk1,berk2,
          coef.style="stat",
          summary.stats=c("Deviance","AIC","N"))
mtable(berk0,berk1,berk2,
          coef.style="ci",
          summary.stats=c("Deviance","AIC","N"))
mtable(berk0,berk1,berk2,
          coef.style="ci.se",
          summary.stats=c("Deviance","AIC","N"))
mtable(berk0,berk1,berk2,
          coef.style="ci.se.horizontal",
          summary.stats=c("Deviance","AIC","N"))
mtable(berk0,berk1,berk2,
          coef.style="ci.p.horizontal",
          summary.stats=c("Deviance","AIC","N"))
mtable(berk0,berk1,berk2,
          coef.style="ci.horizontal",
          summary.stats=c("Deviance","AIC","N"))
mtable(berk0,berk1,berk2,
          coef.style="all",
          summary.stats=c("Deviance","AIC","N"))
```

```
mtable(berk0,berk1,berk2,
          coef.style="all.nostar",
          summary.stats=c("Deviance","AIC","N"))

mtable(by(berkeley,berkeley$Dept,
  function(x)glm(cbind(Admitted,Rejected)~Gender,
        data=x,family="binomial")),
      summary.stats=c("Likelihood-ratio","N"))


mtable(By(~Gender,
  glm(cbind(Admitted,Rejected)~Dept,
        family="binomial"),
        data=berkeley),
      summary.stats=c("Likelihood-ratio","N"))

berkfull <- glm(cbind(Admitted,Rejected)~Dept/Gender - 1,
                     data=berkeley,family="binomial")
relabel(mtable(berkfull),Dept="Department",gsub=TRUE)

#### Array-like semantics

mtable123 <- mtable("Model 1"=lm0,"Model 2"=lm1,"Model 3"=lm2,
    summary.stats=c("sigma","R-squared","F","p","N"))

dim(mtable123)
dimnames(mtable123)
mtable123[c("dpi","ddpi"),
          c("Model 2","Model 3")]

#### Concatention
mt01 <- mtable(lm0,lm1,summary.stats=c("R-squared","N"))
mt12 <- mtable(lm1,lm2,summary.stats=c("R-squared","F","N"))
c(mt01,mt12) # not that this makes sense, but ...
c("Group 1"=mt01,
  "Group 2"=mt12)
```

`mtable_format_delim` *Format for 'mtable' Objects for Writing into File*

Description

`mtable_mtable_print` formats 'mtable' in a way suitable for output into a file with `write.table`

Usage

```
mtable_format_delim(x,
          colsep="\t",
          rowsep="\n",
          interaction.sep = " x ",
```

```
            ...
            )
```

Arguments

x	an object of class mtable
colsep	a character string which seperates the columns in the output.
rowsep	a character string which seperates the rows in the output.
interaction.sep	a character string that separates factors that are involved in an interaction effect
...	further arguments, ignored.

Value

A character string.

`mtable_format_html` *HTML Formatting for 'mtable' Results*

Description

These functions formats 'mtable' objects into HTML format.

Usage

```
mtable_format_html(x,
                   interaction.sep = NULL,
                   toprule=2,midrule=1,bottomrule=2,
                   split.dec=TRUE,
                   style=mtable_format_stdstyle,
                   margin="2ex auto",
                   sig.notes.style=c(width="inherit"),
                   ...
  )
## S3 method for class 'memisc_mtable'
format_html(x,
                   interaction.sep = NULL,
                   toprule=2,midrule=1,bottomrule=2,
                   split.dec=TRUE,
                   style=mtable_format_stdstyle,
                   margin="2ex auto",
                   sig.notes.style=c(width="inherit"),
                   ...
  )
```

Arguments

x	an object of class mtable
toprule	integer; thickness in pixels of rule at the top of the table.
midrule	integer; thickness in pixels of rules within the table.
bottomrule	integer; thickness in pixels of rule at the bottom of the table.
interaction.sep	a character string that separates factors that are involved in an interaction effect or NULL. If NULL then a reasonable default is used (either a unicode character or an ampersand encoded HTML entity).
split.dec	logical; whether numbers should be centered at the decimal point by splitting the table cells.
style	string containing default the CSS styling.
margin	character string, determines the margin and thus the position of the HTML table.
sig.notes.style	a character vector with named elements, allows extra styling of the p-values notes at the bottom of the table.
...	further arguments, ignored.

Value

A character string with code suitable for inclusion into a HTML-file.

Examples

```
lm0 <- lm(sr ~ pop15 + pop75,              data = LifeCycleSavings)
lm1 <- lm(sr ~                 dpi + ddpi, data = LifeCycleSavings)
lm2 <- lm(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings)

mtable123 <- mtable("Model 1"=lm0,"Model 2"=lm1,"Model 3"=lm2,
                    summary.stats=c("sigma","R-squared","F","p","N"))

(mtable123 <- relabel(mtable123,
                      "(Intercept)" = "Constant",
                      pop15 = "Percentage of population under 15",
                      pop75 = "Percentage of population over 75",
                      dpi = "Real per-capita disposable income",
                      ddpi = "Growth rate of real per-capita disp. income"
))

# Use HTML entity '&minus;' for minus sign
options(html.use.ampersand=TRUE)
show_html(mtable123)
show_html(mtable123[1:2],
          sig.notes.style=c(width="30ex"))

# Use unicode for minus sign (default)
options(html.use.ampersand=FALSE)
show_html(mtable123)
```

`mtable_format_latex` *Format 'mtable' Results for LaTeX*

Description

This function formats objects created by `mtable` for inclusion into LaTeX files.

Usage

```
mtable_format_latex(x,
          useDcolumn=getOption("useDcolumn",TRUE),
          colspec=if(useDcolumn)
                    paste("D{.}{",LaTeXdec,"}{",ddigits,"}",sep="")
                  else "l",
          LaTeXdec=".",
          ddigits=min(3,getOption("digits")),
          useBooktabs=getOption("useBooktabs",TRUE),
          toprule=if(useBooktabs) "\\toprule" else "\\hline\\hline",
          midrule=if(useBooktabs) "\\midrule" else "\\hline",
          cmidrule=if(useBooktabs) "\\cmidrule" else "\\cline",
          bottomrule=if(useBooktabs) "\\bottomrule" else "\\hline\\hline",
          interaction.sep = " $\\times$ ",
          sdigits=min(1,ddigits),
          compact=FALSE,
          sumry.multicol=FALSE,
          escape.tex=getOption("toLatex.escape.tex",FALSE),
          signif.notes.type=getOption("toLatex.signif.notes.type","include"),
      signif.notes.spec=getOption("toLatex.signif.notes.spec","p{.5\\linewidth}"),
          ...
)
```

Arguments

`x`	an object of class `mtable`
`useDcolumn`	should the `dcolumn` LaTeX package be used? If true, you will have to include `\usepackage{dcolumn}` into the preamble of your LaTeX document.
`colspec`	LaTeX table column format specifyer(s).
`LaTeXdec`	the decimal point in the final LaTeX output.
`ddigits`	alignment specification or digits after the decimal point.
`useBooktabs`	should the `booktabs` LaTeX package be used? If true, you will have to include `\usepackage{booktabs}` into the preamble of your LaTeX document.
`toprule`	appearance of the top border of the LaTeX `tabular` environment.
`midrule`	how are coefficients and summary statistics separated in the LaTeX `tabular` environment.
`cmidrule`	appearance of rules under section headings.

bottomrule	appearance of the bottom border of the LaTeX tabular environment.
interaction.sep	a character string that separates factors that are involved in an interaction effect
sdigits	integer; number of digits after decimal dot for summary statistics.
compact	logical; should the table be compact, without extra columns between multi-equation models?
sumry.multicol	logical, should summaries enclosed into \multicol commands?
escape.tex	logical, should symbols $, _, and ^ be escaped with backslashes?
signif.notes.type	character string; should be either "include","append", "drop", or "tnotes". If "append", (very simple) LaTeX code is appended that contains notes that relate significance symbols to p-values. If "include", the LaTeX table will include a (multi-column) cell with these notes. If "drop", notes will not be added. If "tnotes", the exported LaTeX table is wrapped in a threeparttable environment and the p-value notes are wrapped in a tablenotes environment. This requires the LaTeX package threeparttable in order to work.
signif.notes.spec	character string; specifies format of cells that include notes about p-values; relevant only if signif.notes.type="include"
...	further arguments, ignored.

Value

A character string with code suitable for inclusion into a LaTeX-file.

`mtable_format_print` *Print Format for 'mtable' Objects*

Description

`mtable_format_print` formats 'mtable' in a way suitable for screen output with 'print'.

Usage

```
mtable_format_print(x,
  topsep="=",
  bottomsep="=",
  sectionsep="-",
  interaction.sep = " x ",
  center.at=getOption("OutDec"),
  align.integers=c("dot","right","left"),
  padding = "  ",
  ...
  )
```

Arguments

x	an object of class mtable
topsep	a character string that is recycled to a top rule.
bottomsep	a character string that is recycled to a bottom rule.
sectionsep	a character string that is recycled to seperate coefficients from summary statistics.
interaction.sep	a character string that separates factors that are involved in an interaction effect
center.at	a character string on which resulting values are centered. Typically equal to ".". This is the default when forLaTeX==TRUE. If NULL, reported values are not centered.
align.integers	how to align integer values.
padding	a character string, usually whitespace, used to insert left- and right-padding of table contents.
...	further arguments, ignored.

Value

A character string.

---

negative match *Negative Match*

Description

%nin% is a convenience operator: x %nin% table is equivalent to !(x %in% table).

Usage

```
x %nin% table
```

Arguments

x	the values to be matched
table	a values to be match against

Value

A logical vector

Examples

```
x <- sample(1:6,12,replace=TRUE)
x %in% 1:3
x %nin% 1:3
```

percent *Table of Percentages with Percentage Base*

Description

percent returns a table of percentages along with the percentage base. It will be useful in conjunction with Aggregate or genTable.

Usage

```
  percent(x,...)
  ## Default S3 method:
percent(x,weights=NULL,total=!(se || ci),
      se=FALSE,ci=FALSE,ci.level=.95,
      total.name="N",perc.label="Percentage",...)
  ## S3 method for class 'logical'
percent(x,weights=NULL,total=!(se || ci),
      se=FALSE,ci=FALSE,ci.level=.95,
      total.name="N",perc.label="Percentage",...)
```

Arguments

x	a numeric vector or factor.
weights	a optional numeric vector of weights of the same length as x.
total	logical; should the total sum of counts from which the percentages are computed be included into the output?
se	logical; should standard errors of the percentages be included?
ci	logical; should confidence intervals of the percentages be included?
ci.level	numeric; nominal coverage of confidence intervals
total.name	character; name given for the total sum of counts
perc.label	character; label given for the percentages if the table has more than one dimensions, e.g. if se or ci is TRUE.
...	for percent.mresp: one or several 1-0 vectors or matrices otherwise, further arguments, currently ignored.

Value

A table of percentages.

Examples

```
x <- rnorm(100)
y <- rnorm(100)
z <- rnorm(100)
```

```
f <- sample(1:3,100,replace=TRUE)
f <- factor(f,labels=c("a","b","c"))


percent(x>0)
percent(f)

genTable(
  cbind(percent(x>0),
        percent(y>0),
        percent(z>0)) ~ f
  )

gt <- genTable(
  cbind("x > 0" = percent(x>0,ci=TRUE),
        "y > 0" = percent(y>0,ci=TRUE),
        "z > 0" = percent(z>0,ci=TRUE)) ~ f
  )

ftable(gt,row.vars=3:2,col.vars=1)

ex.data <- expand.grid(mean=c(0,25,50),sd=c(1,10,100))[rep(1:9,rep(250,9)),]
ex.data <- within(ex.data,x <- rnorm(n=nrow(ex.data),mean=ex.data$mean,sd=ex.data$sd))
ex.data <- within(ex.data,x.grp <- cases( x < 0,
                                          x >= 0 & x < 50,
                                          x >= 50 & x < 100,
                                          x >= 100
                                        ))
genTable(percent(x.grp)~mean+sd,data=ex.data)

Aggregate(percent(Admit,weight=Freq)~Gender+Dept,data=UCBAdmissions)
```

`percentages` *Easy Creation of Tables of Percentages*

Description

The generic function `percentages` and its methods create one- or multidimensional tables of percentages. As such, the function `percentages` can be viewed as a convenience interface to `prop.table`. However, it also allows to obtain standard errors and confidence intervals.

Usage

```
percentages(obj, ...)
## S3 method for class 'table'
percentages(obj,
      by=NULL, which=NULL, se=FALSE, ci=FALSE, ci.level=.95, ...)
## S3 method for class 'formula'
percentages(obj,
```

```
      data=parent.frame(), weights=NULL, ...)
## Default S3 method:
percentages(obj,
      weights=NULL, ...)
## S3 method for class 'data.frame'
percentages(obj,
      weights=NULL, ...)
## S3 method for class 'list'
percentages(obj,
      weights=NULL, ...)
## S3 method for class 'percentage.table'
as.data.frame(x, ...)
## S3 method for class 'xpercentage.table'
as.data.frame(x, ...)
```

Arguments

obj	an object; a contingency table or a formula. If it is a formula, its left-hand side determines the factor or combination of factors for which percentages are computed while its right-hand side determines the factor or combination of factors that define the groups within which percentages are computed.
by	a character vector with the names of the factor variables that define the groups within which percentages are computed. Percentages sum to 100 within combination of levels of these factors.
which	a character vector with the names of the factor variables for which percentages are computed.
se	a logical value; determines whether standard errors are computed.
ci	a logical value; determines whether confidence intervals are computed. Note that the confidence intervals are for infinite (or very large) populations.
ci.level	a numerical value, the required confidence level of the confidence intervals.
data	a contingency table (an object that inherits from "table") or a data frame or an object coercable into a data frame.
weights	an optional vector of weights. Should be NULL or a numeric vector.
...	Further arguments passed on to the "table" method of percentages or ignored in case of a call to as.data.frame.
x	an object coerced into a data frame.

Value

An array that inherits classes "percentage.table" and "table". If percentages was called with se=TRUE or ci=TRUE then the result additionally inherits class "xpercentage.table".

Examples

```
percentages(UCBAdmissions)

# Three equivalent ways to create the same table of conditional
```

```
# percentages
percentages(Admit~Gender+Dept,data=UCBAdmissions)
percentages(UCBAdmissions,by=c("Gender","Dept"))
percentages(UCBAdmissions,which="Admit")
# Percentage table as data frame
as.data.frame(percentages(Admit~Gender+Dept,data=UCBAdmissions))

# Standard errors and confidence intervals
percentages(Admit~Dept,data=UCBAdmissions,se=TRUE)
percentages(Admit~Dept,data=UCBAdmissions,ci=TRUE)
(p<- percentages(Admit~Dept,data=UCBAdmissions,ci=TRUE,se=TRUE))

# An extended table of percentages as data frame
as.data.frame(p)

# A table of percentages of a factor
percentages(iris$Species)

UCBA <- as.data.frame(UCBAdmissions)
percentages(UCBA$Admit,weights=UCBA$Freq)

percentages(UCBA,weights=UCBA$Freq)
```

---

query *Query an Object for Information*

Description

The function query can be used to search an object for a keyword.

The data.set and importer methods perform such a search through the annotations and value labels of the items in the data set.

Usage

```
query(x,pattern,...)
## S4 method for signature 'data.set'
query(x,pattern,...)
## S4 method for signature 'importer'
query(x,pattern,...)

## S4 method for signature 'item'
query(x,pattern,...)
# (Called by the methods above.)
```

Arguments

x	an object

pattern — a character string that gives the pattern to be searched for

... — optional arguments such as

fuzzy logical, TRUE by default; use fuzzy search via agrep or regexp search via grep

extended logical, defaults to FALSE; passed to grep

perl logical, defaults to TRUE; passed to grep

fixed logical, defaults to TRUE; passed to grep

ignore.case logical, defaults to TRUE; passed to grep or agrep

insertions numerical value, defaults to 0.999999999; passed to agrep

deletions numerical value, defaults to 0; passed to agrep

substitutions numerical value, defaults to 0; passed to agrep

Value

If both the annotation and the value labels of an item match the pattern the query method for 'item' objects returns a list containing the annotation and the value labels, otherwise if only the annotation or the value labels match the pattern, either the annotation or the value labels are returned, otherwise if neither matches the pattern, query returns NULL.

The methods of query for 'data.set' and 'importer' objects return a list of all non-NULL query results of all items contained by these objects, or NULL.

Examples

```
nes1948.por <- unzip(system.file("anes/NES1948.ZIP",package="memisc"),
                     "NES1948.POR",exdir=tempfile())
nes1948 <- spss.portable.file(nes1948.por)
query(nes1948,"TRUMAN")
```

---

recode *Recode Items, Factors and Numeric Vectors*

Description

recode substitutes old values of a factor or a numeric vector by new ones, just like the recoding facilities in some commercial statistical packages.

Usage

```
recode(x,...,
       copy=getOption("recode_copy",identical(otherwise,"copy")),
       otherwise=NA)
## S4 method for signature 'vector'
recode(x,...,
    copy=getOption("recode_copy",identical(otherwise,"copy")),
    otherwise=NA)
## S4 method for signature 'factor'
```

```
recode(x,...,
    copy=getOption("recode_copy",identical(otherwise,"copy")),
    otherwise=NA)
## S4 method for signature 'item'
recode(x,...,
    copy=getOption("recode_copy",identical(otherwise,"copy")),
    otherwise=NA)
```

Arguments

x	An object
...	One or more assignment expressions, each of the form new.value <- old.values. new.value should be a scalar numeric value or character string. If one of the new.values is a character string, the return value of recode will be a factor and each new.value will be coerced to a character string that labels a level of the factor. Each old.value in an assignment expression may be a (numeric or character) vector. If x is numeric such an assignment expression may have the form new.value <- range(lower,upper) In that case, values between lower and upper are exchanged by new.value. If one of the arguments to range is min, it is substituted by the minimum of x. If one of the arguments to range is max, it is substituted by the maximum of x. In case of the method for labelled vectors, the *tags* of arguments of the form tag = new.value <- old.values will define the labels of the new codes. If the old.values of different assignment expressions overlap, an error will be raised because the recoding is ambigous.
copy	logical; should those values of x not given an explicit new code copied into the resulting vector?
otherwise	a character string or some other value that the result may obtain. If equal to NA or "NA", original codes not given an explicit new code are recoded into NA. If equal to "copy", original codes not given an explicit new code are copied.

Details

recode relies on the lazy evaluation mechanism of *R*: Arguments are not evaluated until required by the function they are given to. recode does not cause arguments that appear in ... to be evaluated. Instead, recode parses the ... arguments. Therefore, although expressions like 1 <- 1:4 would cause an error action, if evaluated at any place elsewhere in *R*, they will not cause an error action, if given to recode as an argument. However, a call of the form recode(x,1=1:4), would be a syntax error.

If John Fox' package "car" is installed, recode will also be callable with the syntax of the recode function of that package.

Value

A numerical vector, factor or an item object.

See Also

recode of package "car".

Examples

```
x <- as.item(sample(1:6,20,replace=TRUE),
        labels=c( a=1,
                  b=2,
                  c=3,
                  d=4,
                  e=5,
                  f=6))
print(x)


codebook(
    recode(x,
           a = 1 <- 1:2,
           b = 2 <- 4:6))


codebook(
    recode(x,
           a = 1 <- 1:2,
           b = 2 <- 4:6,
           copy = TRUE))

# Note the handling of labels if the recoding rules are bijective
codebook(
    recode(x,
           1 <- 2,
           2 <- 1,
           copy=TRUE))

codebook(
    recode(x,
           a = 1 <- 2,
           b = 2 <- 1,
           copy=TRUE))


# A recoded version of x is returned
# containing the values 1, 2, 3, which are
# labelled as "A", "B", "C".
recode(x,
  A = 1 <- range(min,2),
  B = 2 <- 3:4,
  C = 3 <- range(5,max), # this last comma is ignored
  )

# This causes an error action: the sets
# of original values overlap.
```

```
try(recode(x,
  A = 1 <- range(min,2),
  B = 2 <- 2:4,
  C = 3 <- range(5,max)
  ))

recode(x,
  A = 1 <- range(min,2),
  B = 2 <- 3:4,
  C = 3 <- range(5,6),
  D = 4 <- 7
  )

# This results in an all-missing vector:
recode(x,
  D = 4 <- 7,
  E = 5 <- 8
  )

f <- as.factor(x)
x <- as.integer(x)

recode(x,
  1 <- range(min,2),
  2 <- 3:4,
  3 <- range(5,max)
  )

# This causes another error action:
# the third argument is an invalid
# expression for a recoding.
try(recode(x,
  1 <- range(min,2),
  3:4,
  3 <- range(5,max)
  ))

# The new values are character strings,
# therefore a factor is returned.
recode(x,
  "a" <- range(min,2),
  "b" <- 3:4,
  "c" <- range(5,6)
  )

recode(x,
  1 <- 1:3,
  2 <- 4:6
  )

recode(x,
  4 <- 7,
  5 <- 8,
```

```
  otherwise = "copy"
  )

recode(f,
  "A" <- c("a","b"),
  "B" <- c("c","d"),
  otherwise="copy"
  )

recode(f,
  "A" <- c("a","b"),
  "B" <- c("c","d"),
  otherwise="C"
  )

recode(f,
  "A" <- c("a","b"),
  "B" <- c("c","d")
  )

DS <- data.set(x=as.item(sample(1:6,20,replace=TRUE),
        labels=c( a=1,
                  b=2,
                  c=3,
                  d=4,
                  e=5,
                  f=6)))
print(DS)

DS <- within(DS,{
    xf <- recode(x,
                 "a" <- range(min,2),
                 "b" <- 3:4,
                 "c" <- range(5,6)
                 )
    xn <- x@.Data
    xc <- recode(xn,
                 "a" <- range(min,2),
                 "b" <- 3:4,
                 "c" <- range(5,6)
                 )
    xc <- as.character(x)
    xcc <- recode(xc,
                  1 <- letters[1:2],
                  2 <- letters[3:4],
                  3 <- letters[5:6]
                  )
})

DS

DS <- within(DS,{
    xf <- recode(x,
```

```
                    "a" <- range(min,2),
                    "b" <- 3:4,
                    "c" <- range(5,6)
                    )
        x1 <- recode(x,
                    1 <- range(1,2),
                    2 <- range(3,4),
                    copy=TRUE
                    )
        xf1 <- recode(x,
                    "A" <- range(1,2),
                    "B" <- range(3,4),
                    copy=TRUE
                    )
    })
    DS
    codebook(DS)

    DF <- data.frame(x=rep(1:6,4,replace=TRUE))
    DF <- within(DF,{
        xf <- recode(x,
                    "a" <- range(min,2),
                    "b" <- 3:4,
                    "c" <- range(5,6)
                    )
        x1 <- recode(x,
                    1 <- range(1,2),
                    2 <- range(3,4),
                    copy=TRUE
                    )
        xf1 <- recode(x,
                    "A" <- range(1,2),
                    "B" <- range(3,4),
                    copy=TRUE
                    )
        xf2 <- recode(x,
                    "B" <- range(3,4),
                    "A" <- range(1,2),
                    copy=TRUE
                    )
    })
    DF
    codebook(DF)
```

`relabel` *Change labels of factors or labelled objects*

Description

Function `relabel` changes the labels of a factor or any object that has a `names`, `labels`, `value.labels`, or `variable.labels` attribute. Function `relabel4` is an (internal) generic which is called by `relabel` to handle S4 objects.

Usage

```
## Default S3 method:
relabel(x, ..., gsub = FALSE, fixed = TRUE, warn = TRUE)
## S3 method for class 'factor'
relabel(x, ..., gsub = FALSE, fixed = TRUE, warn = TRUE)


## S4 method for signature 'item'
relabel4(x, ...)
# This is an internal method, see details.
# Use relabel(x, \dots) for 'item' objects
```

Arguments

`x`	An object with a `names`, `labels`, `value.labels`, or `variable.labels` attribute
`...`	A sequence of named arguments, all of type character
`gsub`	a logical value; if TRUE, `gsub` is used to change the labels of the object. That is, instead of substituting whole labels, substrings of the labels of the object can changed.
`fixed`	a logical value, passed to `gsub`. If TRUE, substitutions are by fixed strings and not by regular expressions.
`warn`	a logical value; if TRUE, a warning is issues if a a change of labels was unsuccessful.

Details

This function changes the names or labels of `x` according to the remaining arguments. If `gsub` is FALSE, argument tags are the *old* labels, the values are the new labels. If `gsub` is TRUE, arguments are substrings of the labels that are substituted by the argument values.

Function `relabel` is S3 generic. If its first argument is an S4 object, it calls the (internal) `relabel4` generic function.

Value

The object `x` with new labels defined by the ... arguments.

Examples

```
f <- as.factor(rep(letters[1:4],5))
levels(f)
F <- relabel(f,
  a="A",
```

```
  b="B",
  c="C",
  d="D"
  )
levels(F)

f <- as.item(f)
labels(f)
F <- relabel(f,
  a="A",
  b="B",
  c="C",
  d="D"
  )
labels(F)

# Since version 0.99.22 - the following also works:

f <- as.factor(rep(letters[1:4],5))
levels(f)
F <- relabel(f,
  a=A,
  b=B,
  c=C,
  d=D
  )
levels(F)

f <- as.item(f)
labels(f)
F <- relabel(f,
  a=A,
  b=B,
  c=C,
  d=D
  )
labels(F)
```

---

rename *Change Names of a Named Object*

Description

rename changes the names of a named object.

Usage

```
rename(x, ..., gsub = FALSE, fixed = TRUE, warn = TRUE)
```

Arguments

x	Any named object
...	A sequence of named arguments, all of type character
gsub	a logical value; if TRUE, gsub is used to change the row and column labels of the resulting table. That is, instead of substituting whole names, substrings of the names of the object can changed.
fixed	a logical value, passed to gsub. If TRUE, substitutions are by fixed strings and not by regular expressions.
warn	a logical value; should a warning be issued if those names to change are not found?

Details

This function changes the names of x according to the remaining arguments. If gsub is FALSE, argument tags are the *old* names, the values are the new names. If gsub is TRUE, arguments are substrings of the names that are substituted by the argument values.

Value

The object x with new names defined by the ... arguments.

Examples

```
x <- c(a=1, b=2)
rename(x,a="A",b="B")
# Since version 0.99.22 - the following also works:
rename(x,a=A,b=B)

str(rename(iris,
                Sepal.Length="Sepal_Length",
                Sepal.Width ="Sepal_Width",
                Petal.Length="Petal_Length",
                Petal.Width ="Petal_Width"
                ))
str(rename(iris,
                .="_"
                ,gsub=TRUE))

# Since version 0.99.22 - the following also works:
str(rename(iris,
                Sepal.Length=Sepal_Length,
                Sepal.Width =Sepal_Width,
                Petal.Length=Petal_Length,
                Petal.Width =Petal_Width
                ))
```

`reorder.array` *Reorder an Array or Matrix*

Description

`reorder.array` reorders an array along a specified dimension according given names, indices or results of a function applied.

Usage

```
## S3 method for class 'array'
reorder(x,dim=1,names=NULL,indices=NULL,FUN=mean,...)
## S3 method for class 'matrix'
reorder(x,dim=1,names=NULL,indices=NULL,FUN=mean,...)
```

Arguments

`x`	An array
`dim`	An integer specifying the dimension along which `x` should be ordered.
`names`	A character vector
`indices`	A numeric vector
`FUN`	A function that can be used in `apply(x,dim,FUN)`
`...`	further arguments, ignored.

Details

Typical usages are

```
reorder(x,dim,names)
reorder(x,dim,indices)
reorder(x,dim,FUN)
```

The result of `rename(x,dim,names)` is `x` reordered such that `dimnames(x)[[dim]]` is equal to the concatenation of those elements of `names` that are in `dimnames(x)[[dim]]` and the remaining elements of `dimnames(x)[[dim]]`.

The result of `rename(x,dim,indices)` is `x` reordered along `dim` according to `indices`.

The result of `rename(x,dim,FUN)` is `x` reordered along `dim` according to `order(apply(x,dim,FUN))`.

Value

The reordered object `x`.

See Also

The default method of `reorder` in package `stats`.

Examples

```
(M <- matrix(rnorm(n=25),5,5,dimnames=list(LETTERS[1:5],letters[1:5])))
reorder(M,dim=1,names=c("E","A"))
reorder(M,dim=2,indices=3:1)
reorder(M,dim=1)
reorder(M,dim=2)
```

Reshape *Reshape data frames or data sets*

Description

Reshape is a convenience wrapper around reshape with a somewhat simpler syntax.

Usage

```
Reshape(data,...,id,within_id,drop,direction)
```

Arguments

data	a data frame or data set to be reshaped.
...	Further arguments that specify the variables in long and in wide format as well as the time variable. The name tags of the arguments given here specify variable names in long format, the arguments themselves specify the variables in wide format (or observations in long vormat) and the variable of the "time" variable. The time variable is usually the last of these arguments. An "automatic" time variable can be specified if only a single argument in ... is given.
id	a variable name or a concatenation of variable names (either as character strings or as unquoted symbols), that identify individual units. Defaults to "id" or the id variable specified in the "reshapeLong" attribute of the data argument. Needed only if the data are reshaped from long to wide format.
within_id	an optional variable name (either as character string or as unquoted symbol), that identifies individual observations on units. Relevant only if the data are reshaped from long to wide format.
drop	a variable name or a concatenation of variable names (either as character strings or as unquoted symbols), thast specifies the variables to be dropped before reshaping.
direction	a character string, should be either equal "long" or "wide".

Examples

```
example.data.wide <- data.frame(
    v  = c(35,42),
    x1 = c(1.1,2.1),
    x2 = c(1.2,2.2),
    x3 = c(1.3,2.3),
```

```
    x4 = c(1.4,2.4),
    y1 = c(2.5,3.5),
    y2 = c(2.7,3.7),
    y3 = c(2.9,3.9))
example.data.wide

# The following two calls are equivalent:
example.data.long <- Reshape(data=example.data.wide,
                             x=c(x1,x2,x3,x4),
                             # N.B. it is possible to
                             # specify 'empty' i.e. missing
                             # measurements
                             y=c(y1,y2,y3,),
                             t=1:4,
                             direction="long")

example.data.long <- Reshape(data=example.data.wide,
                             list(
                                 x=c(x1,x2,x3,x4),
                                 # N.B. it is possible to
                                 # specify 'empty' i.e. missing
                                 # measurements
                                 y=c(y1,y2,y3,)
                             ),
                             t=1:4,
                             direction="long")

example.data.long

# Since the data frame contains an "reshapeLong" attribute
# an id variable is already specified and part of the data
# frame.
example.data.wide <- Reshape(data=example.data.long,
                             x=c(x1,x2,x3,x4),
                             y=c(y1,y2,y3,),
                             t=1:4,
                             direction="wide")
example.data.wide

# Here we examine the case where no "reshapeLong" attribute
# is present:
example.data.wide <- Reshape(data=example.data.long,
                             x=c(x1,x2,x3,x4),
                             y=c(y1,y2,y3,),
                             t=1:4,
                             id=v,
                             direction="wide")
example.data.wide


# Here, an "automatic" time variable is created. This works
# only if there is a single argument other than the data=
# and direction= arguments
```

```
example.data.long <- Reshape(data=example.data.wide,
                             list(
                                 x=c(x1,x2,x3,x4),
                                 y=c(y1,y2,y3,)
                             ),
                             direction="long")

example.data.long

example.data.wide <- Reshape(data=example.data.long,
                             list(
                                 x=c(x1,x2,x3,x4),
                                 y=c(y1,y2,y3,)
                             ),
                             direction="wide")
example.data.wide
```

`retain` *Retain Objects in an Environment*

Description

`retain` removes all objects from the environment except those mentioned as argument.

Usage

```
retain(..., list = character(0), envir = parent.frame(),force=FALSE)
```

Arguments

`...`	names of objects to be retained, as names (unquoted) or character strings(quoted).
`list`	a character vector naming the objects to be retained.
`envir`	the environment from which the objects are removed that are not to be retained.
`force`	logical value. As a measure of caution, this function removes objects only from local environments, unless `force` equals TRUE. In that case, `retain` can also be used to clear the global environment, the user's workspace.

Examples

```
local({
  foreach(x=c(a,b,c,d,e,f,g,h),x<-1)
  cat("Objects before call to 'retain':\n")
  print(ls())
  retain(a)
  cat("Objects after call to 'retain':\n")
  print(ls())
})
x <- 1
y <- 2
retain(x)
```

`reversed` *Reverse the codes of a survey item or the levels of a factor*

Description

The function `reversed()` returns a copy of its argument with codes or levels in reverse order.

Usage

```
reversed(x)
## S4 method for signature 'item.vector'
reversed(x)
## S4 method for signature 'factor'
reversed(x)
```

Arguments

x	An object – an "item" object or a "data.set" object

Value

If the argument of the function `reversed()` than either the unique valid values or the labelled valid values recoded into the reverse order.

If th argument is a factor than the function returns the factor with levels in reverse order.

Examples

```
ds <- data.set(
    x = as.item(sample(c(1:3,9),100,replace=TRUE),
                labels=c("One"=1,
                         "Two"=2,
                         "Three"=3,
                         "Missing"=9)))
df <- as.data.frame(ds)
ds <- within(ds,{
    xr <- reversed(x)
})
codebook(ds)
df <- within(df,{
    xr <- reversed(x)
})
codebook(df)
```

sample-methods *Take a Sample from a Data Frame-like Object*

Description

The methods below are convenience short-cuts to take samples from data frames and data sets. They result in a data frame or data set, respectively, the rows of which are a sample of the complete data frame/data set.

Usage

```
## S4 method for signature 'data.frame'
sample(x, size, replace = FALSE, prob = NULL)
## S4 method for signature 'data.set'
sample(x, size, replace = FALSE, prob = NULL)
## S4 method for signature 'importer'
sample(x, size, replace = FALSE, prob = NULL)
```

Arguments

x	a data frame or data set.
size	an (optional) numerical value, the sample size, defaults to the total number of rows of x.
replace	a logical value, determines whether sampling takes place with or without replacement.
prob	a vector of sampling probabities or NULL.

Value

A data frame or data set.

Examples

```
for(.i in 1:4)
  print(sample(iris,5))
```

Sapply *A Dimension Preserving Variant of "sapply" and "lapply"*

Description

Sapply is equivalent to sapply, except that it preserves the dimension and dimension names of the argument X. It also preserves the dimension of results of the function FUN. It is intended for application to results e.g. of a call to by. Lapply is an analog to lapply insofar as it does not try to simplify the resulting list of results of FUN.

Usage

```
Sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
Lapply(X, FUN, ...)
```

Arguments

X	a vector or list appropriate to a call to sapply.
FUN	a function.
...	optional arguments to FUN.
simplify	a logical value; should the result be simplified to a vector or matrix if possible?
USE.NAMES	logical; if TRUE and if X is character, use X as names for the result unless it had names already.

Value

If FUN returns a scalar, then the result has the same dimension as X, otherwise the dimension of the result is enhanced relative to X.

Examples

```
berkeley <- Aggregate(Table(Admit,Freq)~.,data=UCBAdmissions)
berktest1 <- By(~Dept+Gender,
                glm(cbind(Admitted,Rejected)~1,family="binomial"),
                data=berkeley)
berktest2 <- By(~Dept,
                glm(cbind(Admitted,Rejected)~Gender,family="binomial"),
                data=berkeley)

sapply(berktest1,coef)
Sapply(berktest1,coef)

sapply(berktest1,function(x)drop(coef(summary(x))))
Sapply(berktest1,function(x)drop(coef(summary(x))))

sapply(berktest2,coef)
Sapply(berktest2,coef)
sapply(berktest2,function(x)coef(summary(x)))
Sapply(berktest2,function(x)coef(summary(x)))
```

---

sort-methods *Convenience Methods to Sort Data Frames and Data Sets*

Description

The methods below return a sorted version of the data frame or data set, given as first argument.

Usage

```
## S3 method for class 'data.frame'
sort(x,decreasing=FALSE,by=NULL,na.last=NA,...)
## S3 method for class 'data.set'
sort(x,decreasing=FALSE,by=NULL,na.last=NA,...)
```

Arguments

x	a data frame or data set.
decreasing	a logical value, should sorting be in increasing or decreasing order?
by	a character name of variable names, by which to sort; a formula giving the variables, by which to sort; NULL, in which case, the data frame / data set is sorted by all of its variables.
na.last	for controlling the treatment of 'NA's. If 'TRUE', missing values in the data are put last; if 'FALSE', they are put first; if 'NA', they are removed
...	other arguments, currently ignored.

Value

A sorted copy of x.

Examples

```
DF <- data.frame(
        a = sample(1:2,size=20,replace=TRUE),
        b = sample(1:4,size=20,replace=TRUE))
sort(DF)
sort(DF,by=~a+b)
sort(DF,by=~b+a)
sort(DF,by=c("b","a"))
sort(DF,by=c("a","b"))
```

---

styles *Formatting Styles for Coefficients, Factor Contrasts, and Summary Statistics*

Description

Methods for setting and getting templates for formatting model coefficients and summaries for use in mtable.

Usage

```
setCoefTemplate(...)
getCoefTemplate(style)
getSummaryTemplate(x)
setSummaryTemplate(...)
summaryTemplate(x)
```

Arguments

`...`	sevaral tagged arguments; in case of `setCoefTemplate` the tags specify the `coef.styles`, in case of `setSummaryTemplate` they specify model classes. The associated values are templates.
`style`	a character string with the name of a coefficient style, if left empty, all coefficient templates are returned.
`x`	a model or a name of a model class, for example `"lm"` or `"glm"`; if left empty, all summary templates are returned.

Details

The style in which model coefficients are formatted by `mtable` is by default selected from the `coef.style` setting of `options`, "factory-fresh" setting being `options(coef.style="default")`.

The appearance of factor levels in an `mtable` can be influenced by the `factor.style` setting of `options`. The "factory-fresh" setting is `options(factor.style="($f): ($l)")`, where `($f)` stands for the factor name and `($l)` stands for the factor level. In case of treatment contrasts, the baseline level will also appear in an `mtable` separated from the current factor level by the `baselevel.sep` setting of `options`. The "factory-fresh" setting is `options(baselevel.sep="-")`,

Users may specify additional coefficient styles by a call to `setCoefTemplate`.

In order to adapt the display of summary statistics of other model classes, users need to set a template for model summaries via a call to `setSummaryTemplate` or to define a method of the generic function `summaryTemplate`.

`Substitute` *Substitutions in Language Objects*

Description

`Substitute` differs from `substitute` in so far as its first argument can be a variable that contains an object of mode "language". In that case, substitutions take place inside this object.

Usage

```
Substitute(lang,with)
```

Arguments

`lang`	any object, unevaluated expression, or unevaluated language construct, such as a sequence of calls inside braces
`with`	a named list, environment, data frame or data set.

Details

The function body is just `do.call("substitute",list(lang,with))`.

Value

An object of storage mode "language" or "symbol".

Examples

```
lang <- quote(sin(x)+z)
substitute(lang,list(x=1,z=2))
Substitute(lang,list(x=1,z=2))
```

Table *One-Dimensional Table of Frequences and/or Percentages*

Description

`Table` is a generic function that produces a table of counts or weighted counts and/or the corresponding percentages of an atomic vector, factor or `"item.vector"` object. This function is intended for use with `Aggregate` or `genTable`. The `"item.vector"` method is the workhorse of `codebook`.

Usage

```
## S4 method for signature 'atomic'
Table(x,weights=NULL,counts=TRUE,percentage=FALSE,...)
## S4 method for signature 'factor'
Table(x,weights=NULL,counts=TRUE,percentage=FALSE,...)
## S4 method for signature 'item.vector'
Table(x,weights=NULL,counts=TRUE,percentage=(style=="codebook"),
              style=c("table","codebook","nolabels"),
              include.missings=(style=="codebook"),
              missing.marker=if(style=="codebook") "M" else "*",...)
```

Arguments

`x`	an atomic vector, factor or `"item.vector"` object
`counts`	logical value, should the table contain counts?
`percentage`	logical value, should the table contain percentages? Either the `counts` or the `percentage` arguments or both should be `TRUE`.
`style`	character string, the style of the names or rownames of the table.
`weights`	a numeric vector of weights of the same length as `x`.
`include.missings`	a logical value; should missing values included into the table?
`missing.marker`	a character string, used to mark missing values in the table (row)names.
`...`	other, currently ignored arguments.

Value

The atomic vector and factor methods return either a vector of counts or vector of percentages or a matrix of counts and percentages. The same applies to the `"item.vector"` vector method unless `include.missing=TRUE` and `percentage=TRUE`, in which case total percentages and percentages of valid values are given.

Examples

```
with(as.data.frame(UCBAdmissions),Table(Admit,Freq))
Aggregate(Table(Admit,Freq)~.,data=UCBAdmissions)

A <- sample(c(1:5,9),size=100,replace=TRUE)
labels(A) <- c(a=1,b=2,c=3,d=4,e=5,dk=9)
missing.values(A) <- 9
Table(A,percentage=TRUE)
```

---

tibbles *Interface to Packages 'tibble' and 'haven'*

Description

A `as_tibble` method (`as_table.data.set`) allows to transform `"data.set"` objects into objects of class `"tbl_df"` as defined by the package "tibble".

`as.item` methods for objects of classes `"haven_labelled"` and `"have_labelled_spss"` allow to transform a "tibble" imported using `read_dta`, `read_spss`, etc. from the package "haven" into an object of class `"data.set"`.

`as_haven` can be used to transform `"data.set"` objects into objects of class `"tbl_df"` with that additional information that objects imported using the "haven" package usually have, i.e. variable labels and value labels (as the `"label"` and `"labels"` attributes of the columns).

Usage

```
as_tibble.data.set(x,...)
## S4 method for signature 'haven_labelled'
as.item(x,...)
## S4 method for signature 'haven_labelled_spss'
as.item(x,...)
as_haven(x,...)
## S4 method for signature 'data.set'
as_haven(x,user_na=FALSE,...)
## S4 method for signature 'item.vector'
as_haven(x,user_na=FALSE,...)
## S4 method for signature 'tbl_df'
as.data.set(x,row.names=NULL,...)
```

Arguments

x	for as_tibble.data.set and as_haven, an object of class "data.set"; for as.item, an object of class "haven_labelled" or "haven_labelled_spss"; an object of class "tbl_df" for as.data.set.
user_na	logical; if TRUE then the resulting vectors have an "na_values" and/or "na_range" attribute.
row.names	NULL or an optional character vector of row names.
...	further arguments, passed through to other the the as_tibble method for lists, or ignored.

Value

as_tibble.data.set and the "data.set"-method of as_haven return a "tibble". The "item.vector"-method (which is for internal use only) returns a vector with S3 class either "haven_labelled" or "haven_labelled_spss".

---

to.data.frame *Convert an Array into a Data Frame*

Description

to.data.frame converts an array into a data frame, in such a way that a chosen dimensional extent forms variables in the data frame. The elements of the array must be either atomic, data frames with matching variables, or coercable into such data frames.

Usage

```
to.data.frame(X,as.vars=1,name="Freq")
```

Arguments

X	an array.
as.vars	a numeric value or a character string. If it is a numeric value then it indicates the dimensional extend which defines the variables. If it is a character string then it is matched against the names of the dimenstional extents. This is applicable e.g. if X is a contingency table and the dimensional extents are named after the cross-classified factors. Takes effect only if X is an atomic array. If as.vars equals zero, a new variable is created that contains the values of the array, that is, to.data.frame acts on the array X like as.data.frame(as.table(X))
name	a character string; the name of the variable created if X is an atomic array and as.vars equals zero.

Value

A data frame.

Examples

```
berkeley <- Aggregate(Table(Admit,Freq)~.,data=UCBAdmissions)
berktest1 <- By(~Dept+Gender,
                glm(cbind(Admitted,Rejected)~1,family="binomial"),
                data=berkeley)
berktest2 <- By(~Dept,
                glm(cbind(Admitted,Rejected)~Gender,family="binomial"),
                data=berkeley)
Stest1 <- Lapply(berktest2,function(x)predict(x,,se.fit=TRUE)[c("fit","se.fit")])
Stest2 <- Sapply(berktest2,function(x)coef(summary(x)))
Stest2.1 <- Lapply(berktest1,function(x)predict(x,,se.fit=TRUE)[c("fit","se.fit")])
to.data.frame(Stest1)
to.data.frame(Stest2,as.vars=2)
to.data.frame(Stest2.1)
# Recasting a contingency table
to.data.frame(UCBAdmissions,as.vars="Admit")
```

---

toLatex *Additional Methods for LaTeX Representations for R objects*

Description

Methods for the generic function `toLatex` of package "utils" are provided for generating LaTeX representations of matrices and flat contingency tables (see `ftable`). Also a default method is defined that coerces its first argument into a matrix and applies the matrix method.

Usage

```
## Default S3 method:
toLatex(object,...)

## S3 method for class 'matrix'
toLatex(object,
    show.titles=TRUE,
    show.vars=FALSE,
    show.xvar=show.vars,
    show.yvar=show.vars,
    digits=if(is.table(object)) 0 else getOption("digits"),
    format="f",
    useDcolumn=getOption("useDcolumn",TRUE),
    colspec=if(useDcolumn)
                paste("D{.}{",LaTeXdec,"}{",ddigits,"}",sep="")
            else "r",
    LaTeXdec=".",
    ddigits=digits,
    useBooktabs=getOption("useBooktabs",TRUE),
    toprule=if(useBooktabs) "\\toprule" else "\\hline\\hline",
    midrule=if(useBooktabs) "\\midrule" else "\\hline",
```

```
    cmidrule=if(useBooktabs) "\\cmidrule" else "\\cline",
    bottomrule=if(useBooktabs) "\\bottomrule" else "\\hline\\hline",
    toLatex.escape.tex=getOption("toLatex.escape.tex",FALSE),
    ...)


## S3 method for class 'data.frame'
toLatex(object,
    digits=getOption("digits"),
    format="f",
    useDcolumn=getOption("useDcolumn",TRUE),
    numeric.colspec=if(useDcolumn)
                      paste("D{.}{",LaTeXdec,"}{",ddigits,"}",sep="")
                    else "r",
    factor.colspec="l",
    LaTeXdec=".",
    ddigits=digits,
    useBooktabs=getOption("useBooktabs",TRUE),
    toprule=if(useBooktabs) "\\toprule" else "\\hline\\hline",
    midrule=if(useBooktabs) "\\midrule" else "\\hline",
    cmidrule=if(useBooktabs) "\\cmidrule" else "\\cline",
    bottomrule=if(useBooktabs) "\\bottomrule" else "\\hline\\hline",
    row.names=is.character(attr(object,"row.names")),
    NAas="",
    toLatex.escape.tex=getOption("toLatex.escape.tex",FALSE),
    ...)

## S3 method for class 'ftable'
toLatex(object,
    show.titles=TRUE,
    digits=if(is.integer(object)) 0 else getOption("digits"),
    format=if(is.integer(object)) "d" else "f",
    useDcolumn=getOption("useDcolumn",TRUE),
    colspec=if(useDcolumn)
                paste("D{.}{",LaTeXdec,"}{",ddigits,"}",sep="")
             else "r",
    LaTeXdec=".",
    ddigits=digits,
    useBooktabs=getOption("useBooktabs",TRUE),
    toprule=if(useBooktabs) "\\toprule" else "\\hline\\hline",
    midrule=if(useBooktabs) "\\midrule" else "\\hline\n",
    cmidrule=if(useBooktabs) "\\cmidrule" else "\\cline",
    bottomrule=if(useBooktabs) "\\bottomrule" else "\\hline\\hline",
    extrarowsep = NULL,
    toLatex.escape.tex=getOption("toLatex.escape.tex",FALSE),
    fold.leaders=FALSE,
    ...)
```

```
## S3 method for class 'ftable_matrix'
toLatex(object,
    show.titles=TRUE,
    digits=getOption("digits"),
    format="f",
    useDcolumn=getOption("useDcolumn",TRUE),
    colspec=if(useDcolumn)
                paste("D{.}{",LaTeXdec,"}{",ddigits,"}",sep="")
            else "r",
    LaTeXdec=".",
    ddigits=digits,
    useBooktabs=getOption("useBooktabs",TRUE),
    toprule=if(useBooktabs) "\\toprule" else "\\hline\\hline",
    midrule=if(useBooktabs) "\\midrule" else "\\hline",
    cmidrule=if(useBooktabs) "\\cmidrule" else "\\cline",
    bottomrule=if(useBooktabs) "\\bottomrule" else "\\hline\\hline",
    compact=FALSE,
    varontop,varinfront,
    groupsep="3pt",
    grouprule=midrule,
    toLatex.escape.tex=getOption("toLatex.escape.tex",FALSE),
    ...)
```

Arguments

object	an ftable, a matrix or an object coercable into a matrix.
show.titles	logical, should variable names (in case of the ftable and table methods) or row and column names (in case of the matrix method) be appear in the LaTeX code?
show.vars,show.xvar,show.yvar	logical, should the names of the dimnames of object be shown in the margins of the LaTeX tabular? Such names usually represent the row and/or column variables of a two-dimensional table.
digits	number of significant digits.
format	character containing a format specifier, see format.
useDcolumn	logical, should the facilities of the dcolumn LaTeX package be used? Note that, if TRUE, you will need to include \usepackage{dcolumn} in the preamble of your LaTeX document.
colspec	character, LaTeX table column format specifyer(s).
numeric.colspec	character, LaTeX table column format specifyer(s) for numeric vectors in the data frame.
factor.colspec	character, LaTeX table column format specifyer(s) for factors in the data frame.
LaTeXdec	character, the decimal point in the final LaTeX output.
ddigits	integer, digits after the decimal point.

useBooktabs	logical, should the facilities of the booktabs LaTeX package be used? Note that, if TRUE, you will need to include \usepackage{booktabs} in the preamble of your LaTeX document.
toprule	character string, TeX code that determines the appearance of the top border of the LaTeX tabular environment.
midrule	character string, TeX code that determines how coefficients and summary statistics are separated in the LaTeX tabular environment.
cmidrule	character string, TeX code that determines the appearance of rules under section headings.
bottomrule	character string, TeX code that determines the appearance of the bottom border of the LaTeX tabular environment.
extrarowsep	character string, extra code to be inserted between the column titles and the table body produced by toLatex.
compact	logical, if TRUE, extra column space between sub-tables is suppressed. Defaults to FALSE
varontop	logical, whether names of column variables should appear on top of factor levels
varinfront	logical, whether names of row variables should appear in front of factor levels
groupsep	character string, containing a TeX length; extra vertical space inserted between sub-tables, unless compact is TRUE.
grouprule	character string, TeX code that determines how sub-table headings are embellished.
row.names	logical, whether row names should be included in exported LaTeX code.
NAas	character string, how missing values should be represented.
toLatex.escape.tex	logical, should symbols "$", "_", and "^" be escaped with backslashes?
fold.leaders	logical, if TRUE, factor levels of row variables are not distributed into different columns, but 'folded' into a single column.
...	further argument, currently ignored.

Examples

```
toLatex(diag(5))

toLatex(ftable(UCBAdmissions))

toLatex(rbind(
  ftable(margin.table(UCBAdmissions,c(2,1))),
  ftable(margin.table(UCBAdmissions,c(3,1)))
))
```

`trim_labels` *Trim Codes from the Labels of an Item*

Description

Occasionally, labels of codes in a survey data sets (e.g. from the 2016 American National Election Study) include a character representation of the codes being labelled. While there may be technical reasons for this, it is often inconvenient (e.g. if one wants to reorder the labelled codes). The function `trim_labels` trims the code representations (if they are present.)

Usage

```
trim_labels(x,...)
## S4 method for signature 'item.vector'
trim_labels(x,...)
## S4 method for signature 'data.set'
trim_labels(x,...)
```

Arguments

`x`	An object – an "item" object or a "data.set" object
`...`	Further arguments, currently ignored

Details

The "data.set" method applies the "item.vector" method to all the labelled items in the data set.

The "item.vector" returns a copy of its argument with modified labels, where a label such as "1. First alternative" is changed into "First alternative".

Examples

```
x <- as.item(sample(1:3,10,replace=TRUE),
             labels=c("1. One"=1,
                      "2. Two"=2,
                      "2. Three"=3))
y <- as.item(sample(1:2,10,replace=TRUE),
             labels=c("1. First category"=1,
                      "2. Second category"=2))
ds <- data.set(x,y)
x <- trim_labels(x)
codebook(x)
ds <- trim_labels(ds)
codebook(ds)
```

Utility classes *Named Lists, Lists of Items, and Atomic Vectors*

Description

The classes "named.list" and "item.list" are merely some 'helper classes' for the construction of the classes "data.set" and "importer".

Class "named.list" extends the basic class "list" by an additional slot "names". Its `initialize` method assures that the names of the list are unique.

Class "item.list" extends the class "named.list", but does not add any slots. From "named.list" it differs only by the `initialize` method, which calls that for "named.list" and makes sure that all elements of the list belong to class "item".

Classes "atomic" and "double" are merely used for method selection.

Examples

```
new("named.list",a=1,b=2)

# This should generate an error, since the names
# are not unique.
try(new("named.list",a=1,a=2))

# Another error, one name is missing.
try(new("named.list",a=1,2))

# Also an error, the resulting list would be unnamed.
try(new("named.list",1,2))


new("item.list",a=1,b=2)

# Also an error: "item.list"s are "named.lists",
# and here the names would be non-unique.
try(new("item.list",a=1,a=2))
```

value.filter *Value Filters*

Description

Value filters, that is objects that inherit from class "value.filter", are a mechanism to distinguish between valid codes of a survey item and codes that are considered to be missing, such as the codes for answers like "don't know" or "answer refused".

Value filters are optional slot values of "item" objects. They determine which codes of "item" objects are replaced by `NA` when they are coerced into a vector or a factor.

There are three (sub)classes of value filters: "missing.values", which specify individual missing values and/or a range of missing values; "valid.values", which specify individual valid values (that is, all other values of the item are considered as missing); "valid.range", which specify a range of valid values (that is, all values outside the range are considered as missing). Value filters of class "missing.values" correspond to missing-values declarations in SPSS files, imported by `spss.fixed.file`, `spss.portable.file`, or `spss.system.file`.

Value filters also can be updated using the + and - operators.

Usage

```
value.filter(x)

missing.values(x)
missing.values(x)<-value

valid.values(x)
valid.values(x)<-value

valid.range(x)
valid.range(x)<-value

is.valid(x)
nvalid(x)
is.missing(x)
include.missings(x,mark="*")
```

Arguments

`x,value`	objects of the appropriate class.
`mark`	a character string, used to pasted to value labels of `x` (if present).

Value

`value.filter(x)`, `missing.values(x)`, `valid.values(x)`, and `valid.range(x)`, return the value filter associated with `x`, an object of class "value.filter", that is, of class "missing.values", "valid.values", or "valid.range", respectively.

`is.missing(x)` returns a logical vector indicating for each element of `x` whether it is a missing value or not. `is.valid(x)` returns a logical vector indicating for each element of `x` whether it is a valid value or not. `nvalid(x)` returns the number of elements of `x` that are valid.

For convenience, `is.missing(x)` and `is.valid(x)` also work for atomic vectors and factors, where they are equivalent to `is.na(x)` and `!is.na(x)`. For atomic vectors and factors, `nvalid(x)` returns the number of elements of `x` for which `!is.na(x)` is TRUE.

`include.missings(x,...)` returns a copy of `x` that has all values declared as valid.

Examples

```
x <- rep(c(1:4,8,9),2,length=60)
labels(x) <- c(
```

```
    a=1,
    b=2,
    c=3,
    d=4,
    dk=8,
    refused=9
    )
missing.values(x) <- 9
missing.values(x)
missing.values(x) <- missing.values(x) + 8
missing.values(x)
missing.values(x) <- NULL
missing.values(x)
missing.values(x) <- list(range=c(8,Inf))
missing.values(x)
valid.values(x)
print(x)
is.missing(x)
is.valid(x)
as.factor(x)
as.factor(include.missings(x))
as.integer(x)
as.integer(include.missings(x))
```

`view` *A Generic Viewing Function*

Description

The function `view` provides generic interface to the non-generic function `View`.

In contrast to the implementation of `View` provided by either basic *R* or *RStudio*, this function can be extended to handle new kinds of objects by defining `viewPrep` methods for them. Further, `view` can be adapted to other GUIs by specifying the `"vfunc"` option or the `vfunc=` optional argument.

Internally, `view` usues the generic function `viewPrep` to prepare data so it can be passed on to the (non-generic) function `View` or (optionally) a different graphical user interface function that can be used to display matrix- or data frame-like objects.

The `vfunc` argument determines how the result of `viewPrep` is displayed. Its default is the function `View`, but an alternative is `view_html` which creates and displays an HTML grid.

Usage

```
view(x,
     title=deparse(substitute(x)),
     vfunc=getOption("vfunc","View"),
     ...)

# The internal generic, not intended to be used by the end-user.
viewPrep(x,title,...)
```

```
## S3 method for class 'data.set'
viewPrep(x,title,...)
## S3 method for class 'data.frame'
viewPrep(x,title,...)
## S3 method for class 'descriptions'
viewPrep(x,title,...)
## S3 method for class 'codeplan'
viewPrep(x,title,compact=FALSE,...)
## S3 method for class 'importer'
viewPrep(x,title,compact=TRUE,...)
```

Arguments

x	an object, e.g. a data frame, data.set, or importer.
title	an optional character string; shown as the title of the display.
vfunc	a character string; a name of a GUI function to call with the results of viewPrep()
compact	a logical value; should the codeplan be shown in a compact form - one line per variable - or in a more expanive form - one line per labelled value?
...	further arguments; view() passes them on to viewPrep.

Examples

```
## Not run:
    example(data.set)
    view(Data)
    view(description(Data))
    view(codeplan(Data))
    # Note that this file is *not* included in the package
    # and has to be obtained from GESIS in order to run the
    # following
    ZA7500sav <- spss.file("ZA7500_v2-0-0.sav")
    view(ZA7500sav)

## End(Not run)
```

---

view_html *HTML Output for 'view.*

Description

An alternative to 'View' for use with 'view'.

Usage

```
view_html(x,title=deparse(substitute(x)),output,...)
```

Arguments

x	the result of viewPrep, a matrix of character strings.
title	an optional character string; shown as the title of the display.
output	a function or the name of a function. It determines how where the HTML code is directed to. If the working environment is RStudio, the default value is "file.show". In other interactive environments it is "browser". In non-interactive sessions it is "stdout". If output equals "browser" the generated HTML code is shown using browseURL. If output equals "stdout" the HTML code is written to the console output window. If output equals "file.show", the function file.show is used. If view_html is called within a *Jupyter* session, the HTML code created is envelopped in a pair of tags and included into the Jupyter output.
...	other arguments; ignored.

Examples

```
## Not run:
    example(data.set)
    view(Data,vfunc=view_html)

## End(Not run)
```

---

wild.codes *Table of frequencies for unlabelled codes*

Description

The function wild.codes creates a table of frequencies of those codes of an item that do not have labelled attached to them. This way, it helps to identify coding errors.

Usage

```
wild.codes(x)
## S4 method for signature 'item'
wild.codes(x)
```

Arguments

x	an object of class "item"

Value

A table of frequencies (i.e. an array of class "table")

`within-operators` *Operators to abbreviate use of "with" and "within"*

Description

The operators `%$%` and `%$$%` provide abbrevitions for calls to `with()` and `within()` respectively. The function `Within()` is a variant of `with()` were the resulting data frame contains any newly created variables in the order in which they are created (and not in the reverse order).

Usage

```
data %$% expr
data %$$% expr
Within(data,expr,...)
## S3 method for class 'data.frame'
Within(data,expr,...)
```

Arguments

`data`	a data frame or similar object, see `with` and `within`
`expr`	a single or compound expression (i.e. several expressions enclosed in curly braces), see `with` and `within`
`...`	Further arguments, currently ignored

See Also

`with` and `within` in package "base".

Examples

```
df <- data.frame(a = 1:7,
                 b = 7:1)

df

df <- within(df,{
  ab <- a + b
  a2b2 <- a^2 + b^2
})
df

df <- data.frame(a = 1:7,
                 b = 7:1)
df <- Within(df,{
  ab <- a + b
  a2b2 <- a^2 + b^2
})
df
```

```
df <- data.frame(a = 1:7,
                 b = 7:1)
df

ds <- as.data.set(df)
ds

df %$$% {
  ab <- a + b
  a2b2 <- a^2 + b^2
}
df

ds %$$% {
  ab <- a + b
  a2b2 <- a^2 + b^2
}
ds

df %$% c(a.ssq = sum(a^2),
         b.ssq = sum(b^2))
```

---

withSE *Add Alternative Variance Estimates to Models Estimates*

Description

A simple object-orientation infrastructure to add alternative standard errors, e.g. sandwich estimates or New-West standard errors to fitted regression-type models, such as fitted by `lm()` or `glm()`.

Usage

```
withSE(object, vcov, ...)

withVCov(object, vcov, ...)

## S3 method for class 'lm'
withVCov(object, vcov, ...)

## S3 method for class 'withVCov'
summary(object, ...)
## S3 method for class 'withVCov.lm'
summary(object, ...)
```

Arguments

object	a fitted model object
vcov	a function that returns a variance matrix estimate, a given matrix that is such an estimate, or a character string that identifies a function that returns a variance matrix estimate (e.g. "HAC" for vcovHAC).
...	further arguments, passed to vcov() or, respectively, to the parent method of summary()

Details

Using withVCov() an alternative variance-covariance matrix is attributed to a fitted model object. Such a matrix may be produced by any of the variance estimators provided by the "sandwich" package or any package that extends it.

withVCov() has no consequences on how a fitted model itself is printed or represented, but it does have consequences what standard errors are reported, when the function summary() or the function mtable() is applied.

withSE() is a convenience front-end to withVCov(). It can be called in the same way as withVCov, but also allows to specify the type of variance estimate by a character string that identifies the function that gives the covariance matrix (e.g. "OPG" for vcovOPG).

Value

withVCov returns a slightly modified model object: It adds an attribute named ".VCov" that contains the alternate covaraince matrix and modifies the class attribute. If e.g. the original model object has class "lm" then the model object modified by withVCov has the class attribute c("withVCov.lm", "withVCov", "lm").

Examples

```
## Generate poisson regression relationship
x <- sin(1:100)
y <- rpois(100, exp(1 + x))
## compute usual covariance matrix of coefficient estimates
fm <- glm(y ~ x, family = poisson)

library(sandwich)
fmo <- withVCov(fm,vcovOPG)
vcov(fm)
vcov(fmo)

summary(fm)
summary(fmo)

mtable(Default=fm,
       OPG=withSE(fm,"OPG"),
       summary.stats=c("Deviance","N")
       )

vo <- vcovOPG(fm)
```

```
mtable(Default=fm,
       OPG=withSE(fm,vo),
       summary.stats=c("Deviance","N")
       )
```

`Write` *Write Codebooks and Variable Descriptions into a Text File*

Description

This is a convenience function to facilitate the creation of data set documents in text files.

Usage

```
Write(x,...)
## S3 method for class 'codebook'
Write(x,file=stdout(),...)
## S3 method for class 'descriptions'
Write(x,file=stdout(),...)
```

Arguments

`x`	a "codebook" or "descriptions" object.
`file`	a connection, see connections.
`...`	further arguments, ignored or passed on to particular methods.

`xapply` *Apply a function to ranges of variables*

Description

`xapply` evaluates an expression given as second argument by substituting in variables. The results are collected in a list or array in a similar way as done by `Sapply` or `lapply`.

Usage

```
xapply(...,.sorted,simplify=TRUE,USE.NAMES=TRUE,.outer=FALSE)
```

Arguments

`...`	tagged and untagged arguments. The tagged arguments define the 'variables' that are looped over, the first untagged argument defines the expression wich is evaluated.
`.sorted`	an optional logical value; relevant only when a range of variable is specified using the column operator ":". Decises whether variable names should be sorted alphabetically before the range of variables are created. If this argument missing, its default value is TRUE, if `xapply()` is called in the global environment, otherwise it is FALSE.
`simplify`	a logical value; should the result be simplifies in `Sapply`?
`USE.NAMES`	a logical value or a positive integer. If an integer, determines which variable is used to name the highest dimension of the result (its columns, in case is it a matrix). If TRUE, the first variable is used.
`.outer`	an optional logical value; if TRUE, each combination of the variables is used to evaluate the expression, if FALSE (the default) then the variables all need to have the same length and the corresponding values of the variables are used in the evaluation of the expression.

Examples

```
x <- 1:3
y <- -(1:3)
z <- c("Uri","Schwyz","Unterwalden")
print(x)
print(y)
print(z)
foreach(var=c(x,y,z),          # assigns names
  names(var) <- letters[1:3]   # to the elements of x, y, and z
  )
print(x)
print(y)
print(z)

ds <- data.set(
        a = c(1,2,3,2,3,8,9),
        b = c(2,8,3,2,1,8,9),
        c = c(1,3,2,1,2,8,8)
      )
print(ds)
ds <- within(ds,{
      description(a) <- "First item in questionnaire"
      description(b) <- "Second item in questionnaire"
      description(c) <- "Third item in questionnaire"

      wording(a) <- "What number do you like first?"
      wording(b) <- "What number do you like second?"
      wording(c) <- "What number do you like third?"

      foreach(x=a:c,{ # Lazy data documentation:
```

```
        labels(x) <- c(    # a,b,c get value labels in one statement
                        one = 1,
                        two = 2,
                      three = 3,
              "don't know" = 8,
         "refused to answer" = 9)
        missing.values(x) <- c(8,9)
        })
      })

codebook(ds)

# The colon-operator respects the order of the variables
# in the data set, if .sorted=FALSE
with(ds[c(3,1,2)],
     xapply(x=a:c,
            description(x)
            ))

# Since .sorted=TRUE, the colon operator creates a range
# of alphabetically sorted variables.
with(ds[c(3,1,2)],
     xapply(x=a:c,
            description(x),
            .sorted=TRUE
            ))

# The variables in reverse order
with(ds,
     xapply(x=c:a,
             description(x)
            ))

# The colon operator can be combined with the
# concatenation function
with(ds,
     xapply(x=c(a:b,c,c,b:a),
             description(x)
            ))

# Variables can also be selected by regular expressions.
with(ds,
     xapply(x=rx("[a-b]"),
             description(x)
            ))

# Demonstrating the effects of the 'USE.NAMES' argument.
with(ds,
     xapply(x=a:c,mean(x)))

with(ds,
     xapply(x=a:c,mean(x),
     USE.NAMES=FALSE))
```

```
t(with(ds,
      xapply(i=1:3,
             x=a:c,
             c(Index=i,
               Mean=mean(x)),
      USE.NAMES=2)))

# Result with 'simplify=FALSE'
with(ds,
     xapply(x=a:c,mean(x),
     simplify=FALSE))

# It is also possible to loop over functions:
xapply(fun=c(exp,log),
       fun(1))

# Two demonstrations for '.outer=TRUE'
with(ds,
      xapply(x=a:c,
             y=a:c,
             cov(x,y),
             .outer=TRUE))

with(ds,
      xapply(x=a:c,
             y=a:c,
             fun=c(cov,cor),
             fun(x,y),
             .outer=TRUE))
```

Index