Package 'guf'

February 20, 2020

Type Package

Title Generally Useful Functions

Version 1.0.2

Maintainer Ben Johannes <inform@benjohannes.com>

Description Provides simple ``base R"-like functions,
such as %notin% (mirroring %in%), se (standard error, mirroring e.g. sd()), quick rounding of a mixed-class object (mirroring round()), count[ing] (mirroring e.g. sum(), mean(), etc.), and an aggregate function (mirroring aggregate()) that can also do pivoting.

License GPL-3

Encoding UTF-8

LazyData true

RoxygenNote 6.1.1

Date 2020-02-06

Language en-UK

NeedsCompilation no

Author Ben Johannes [aut, cre]

Repository CRAN

Date/Publication 2020-02-20 07:50:02 UTC

R topics documented:

aggro *Aggregates a numerical column*

Description

The function aggro() does roughly the same as the function aggregate(). However, the differences are that (a) aggro() returns the results ordered by all grouping columns; (b) aggro() is polymorphic, that is, two levels of complexity can be specified; and (c) aggro also displays the results per NA if NAs are values in the grouping column(s) or split.by column.

Usage

```
aggro(x, group.by, num.column, FUN, split.by = NULL)
```

Arguments

x	A data.frame with at least one column that can be treated as 'factor', and one numerical column.
group.by	A vector of column names in x. It denotes groups, sub-groups, sub-sub-groups (etc., depending on the number of columns specified) by which counts need to be grouped. See examples.
num.column	A string, denoting the name of the numerical column in x on which to apply function FUN.
FUN	A function that returns a single numerical value when applied to a numerical vector; examples are sum, mean, sd, se, var, etc.
split.by	An optional column name in x, by which to split the counts 'horizontally'. That is, whereas 'group.by' is returned as rows, 'split.by' is returned as columns, whereby every value in 'split.by' will become a column. In that sense, it acts as a pivot specifier. This is one of the main differentiators with aggregate().

Value

A data.frame. The first column(s) is/are the 'group.by' column(s). If 'split.by' is not provided, then the final column name is the column name of the 'num.column'; if 'split'by' is provided, then the return data.frame consists of the columns specified by 'group.by' and the unique values in 'split.by'. See examples.

Examples

```
my_DF = data.frame(var1=factor(c(rep('low', 4),rep('medium', 4),rep('high', 4)),
levels=c('low', 'medium', 'high')), var2=c(1, 2,2, 3,3,3, 4,4,4,4, 3, 2),
var3=rep(c('bbb','aaa','bbb'), 4), stringsAsFactors=FALSE)
aggro(my_DF, c('var3', 'var1'), 'var2', sum)
# Just like with the function count(), the results are grouped by unique combinations of ...
# ...'var3' and 'var1'. Note the following:
# * Column names are given in parenthesis (either single, ', or double, ")
# * Functions are not specified with parenthesis
```

```
# * The output is ordered; in principle according to alphanumerical order, except..
#   ... when a 'group.by' column is an ordered factor, the factor order is followed.
# However, that said, up to this point, the results are the same as with aggregate():
aggregate(var2~var3+var1, my_DF, sum)
# Yet, it is getting more interesting/useful when either the results are split, ...
# ... or when there are NAs envolved; in both cases, aggro() digresses from aggegate();
# see the following:

# With split.by. Also non-factors can be used for 'split.by':
aggro(my_DF, group.by='var1', num.column='var2', FUN=sum, split.by='var3')

# With NAs. For the 'group.by' variable, NAs are treated as 'factor'.
# When there are NAs in the 'split.by' column, then an extra NA column is returned, ...
# ...specifying the counts of the NAs:
my_DF_w_NA = my_DF # same as above, but now...
my_DF_w_NA$var1[1] <- NA
my_DF_w_NA$var2[c(6,10)] <- NA
my_DF_w_NA$var3[10] <- NA
aggro(my_DF_w_NA, c('var1', 'var3'), 'var2', sum)
# Compare with:
aggregate(var2~var1+var3, my_DF_w_NA, sum)

# And indeed, with a split.by:
my_DF_w_NA$var3[8] <- NA
aggro(my_DF_w_NA, group.by='var1', num.column='var2', FUN=sum, split.by='var3')
```

---

count *Count the number of occurences of every unique element in a data object*

Description

The function count() does roughly the same as the function table(). However, the differences are that (a) count() may make the result more directly accessible, because it returns the result in the form of a data.frame; and (b) that it is polymorphic, that is, three levels of complexity can be specified. Perhaps count() may align with a certain psychological expectation that when base R provides functions like mean(some_numerical_vector) and sum(some_numerical_vector), then count(some_numerical_vector) would seem to complement these. However, functions like mean() and sum() only work on numerical vectors, and return one numerical value only (and hence can be used more easily as sub-functions in e.g. apply()), whereas count can deal with various data types and returns a data.frame.

Usage

```
count(x, group.by = NULL, split.by = NULL, row.total = FALSE)
```

Arguments

x	The object to count elements from. Can be a vector, a matrix or a data.frame.
group.by	An optional vector of column names in x. It denotes groups, sub-groups, sub-sub-groups (etc., depending on the number of columns specified) by which counts need to be grouped. See examples.
split.by	An optional column name in x, by which to split the counts 'horizontally'. That is, whereas 'group.by' is returned as rows, 'split.by' is returned as columns, whereby every value in 'split.by' will become a column. In that sense, it acts as a pivot specifier.
row.total	Boolean, specifying whether row totals need to be included as the right-most column. This is only relevant IFF split.by is provided. The column will be called 'row.total'; if that column name already exists in x, then 'row.total' will be trailed by underscores until a unique column name is generated.

Value

A data.frame. If 'x' is a vector, the data.frame has as the first column 'element' which are the elements in x (the vector) that have been counted, and a second column 'count' which represent the counts; if 'x' is a data.frame, then the first column(s) is/are the 'group.by' column(s), and a colum 'count' contains the actual counts of the number of rows for the unique number of rows for 'group.by'. If 'split.by' is also specified, then the return data.frame consists of the columns specified by 'group.by' and the unique values in 'split.by'. See examples.

Examples

```
my_num_vector = c(1,1,1,4,5,5)
mean(my_num_vector)
sum(my_num_vector)
table(my_num_vector)
count(my_num_vector)

my_str_vector = c('R', 'R', 'R', 'S', 'T', 'T')
table(my_str_vector)
count(my_str_vector)

my_DF <- data.frame(var1=rep(c('A','B','C'), 2), var2=c(1,1, 2,2, 3,3),
var3=rep(c('bbb','aaa','bbb'), 2))
count(my_DF, c('var1', 'var2'))
count(my_DF, c('var1', 'var3'))
count(my_DF, c('var2', 'var3'))
count(my_DF, 'var3')
#and compare with:
count(my_DF$var3)

my_DF = data.frame(var1=factor(c(rep('low', 4),rep('medium', 4),rep('high', 4)),
levels=c('low', 'medium', 'high')), var2=c(1, 2,2, 3,3,3, 4,4,4,4, 3, 2),
var3=rep(c('bbb','aaa','bbb'), 4), stringsAsFactors=FALSE)
count(my_DF, c('var3', 'var2'), 'var1')
# The counts are grouped by unique combinations of 'var3' and 'var2', ...
# ...and split out by the unique content of 'var1'.
```

```
# Note that if levels are given (as in this case), then the columns for 'split.by'...
# ...are ordered according to the sequence of the levels; otherwise in alphanumerical order.

# Also non-factors can be used for 'split.by':
count(my_DF, c('var1', 'var3'), 'var2')

# For the 'group.by' variable, NAs are treated as 'factor'.
# When there are NAs in the 'split.by' column, then an extra NA column is returned, ...
# ...specifying the counts of the NAs:
my_DF_w_NA = my_DF # same as above, but now...
my_DF_w_NA$var1[1] <- NA
my_DF_w_NA$var2[c(6,10)] <- NA
my_DF_w_NA$var3[10] <- NA
count(my_DF_w_NA, c('var1', 'var3'), 'var2')

# To show the idea of row totals:
count(my_DF_w_NA, c('var2', 'var3'), 'var1', row.total=TRUE)
```

`left` *Get the specified left most characters of a string*

Description

left() counts from the start of a character string, for a specified number of characters, and returns the resulting substring. If the specified number of characters is more than the total length of the string, then the entire string will be returned.

Usage

```
left(string, nr_of_chars)
```

Arguments

`string`	A character string, or a vector of character strings. This is the (vector of) character strings(s) of which the starting characters are to be returned.
`nr_of_chars`	An integer or a vector of integers, specifying the number of characters from the start of the string to return. If both 'string' and 'nr_or_chars' are a vector, then the vector lengths must match.

Value

A character string, or a vector of character strings if the input is a vector.

Examples

```
left("hello world", 2) # returns 'he'
left(c("hello", "world"), 2) # returns vector c('he', 'wo')
left(c("hello", "world"), c(1,3)) # returns vector c('h', 'wor')
left("hello world", 80) # returns 'hello world'
```

right *Get the specified right most characters of a string*

Description

right() counts from the right most position of a character string, for a specified number of characters, and returns the resulting substring. If the specified number of characters is more than the total length of the string, then the entire string will be returned.

Usage

```
right(string, nr_of_chars)
```

Arguments

string	A character string, or a vector of character strings. This is the (vector of) character strings(s) of which the right most characters are to be returned.
nr_of_chars	An integer or a vector of integers, specifying the right most number of characters to return. If both 'string' and 'nr_or_chars' are a vector, then the vector lengths must match.

Value

A character string, or a vector of character strings if the input is a vector.

Examples

```
right("hello world", 3) # returns 'rld'
right(c("hello", "world"), 3) # returns vector c('llo', 'rld')
right(c("hello", "world"), c(1,3)) # returns vector c('o', 'rld')
right("hello world", 80) # returns 'hello world'
```

---

round_something *The function round_something() does numerical rounding with a specified number of decimals. The 'something' will typically be a data.frame with mixed column types, of which the numerical columns must be rounded. This function makes that easier: all non-numerical columns are left in place, the numerical columns are rounded. Columns that are character but can be interpreted as numerical are first converted to numerical, and then rounded.*

Description

The function round_something() does numerical rounding with a specified number of decimals. The 'something' will typically be a data.frame with mixed column types, of which the numerical columns must be rounded. This function makes that easier: all non-numerical columns are left in place, the numerical columns are rounded. Columns that are character but can be interpreted as numerical are first converted to numerical, and then rounded.

Usage

```
round_something(something, decimals = 0)
```

Arguments

something	The object of which the numbers need to be rounded. Can be a numerical or character vector, a matrix, a data.frame or a list containing these elements.
decimals	An integer specifying how many decimal points the result must contain.

Value

The same structure as used for the input of 'something', but then with rounded numbers (where applicable).

Examples

```
my_DF = data.frame(A=c(1.111,2.22,3.3,4), B=c("A", "Bb", "Ccc", "Dddd"), row.names=c(1,2,3,4))
round_something(my_DF)
round_something(my_DF, 2)
my_list = list(A=c(1.111,2.22,3.3,4), B=c("A", "Bb", "Ccc", "Dddd"))
round_something(my_list)
round_something(my_list, 2)
```

---

se *Standard Error*

Description

The Standard Error ('se') of a numerical vector x.

Usage

```
se(x, na.rm = TRUE)
```

Arguments

x	A vector of numerical values, be it integers and/or real numbers.
na.rm	A Boolean, indicating whether NAs need to be removed from x before calculating the Standard Error; defaults to TRUE

Value

One real number, being the Standard Error.

Examples

```
my_vector = c(1, 3, 4, 7.5, 8)
var(my_vector); sd(my_vector); se(my_vector)
my_vector = c(NA, 3, 4, 7.5, 8)
var(my_vector); sd(my_vector); se(my_vector)
# note that se() by default leaves out NAs. But compare:
var(my_vector); sd(my_vector); se(my_vector, na.rm=FALSE)
```

`%notin%` *The inverse of %in%*

Description

`%notin%` tells you which values of vector x are not in vector y.

Usage

```
x %notin% y
```

Arguments

x	A vector
y	A vector

Details

Although this function can be called as `` `%notin%`(x,y) ``, the intended use is like the function `%in%`: similar to `x %in% y`, this function's supposed use is `x %notin% y`. Vector types (more formally: modes) don't need to match (see last example).

Value

A vector of of booleans with the length of vector x. `TRUE` indicates that the particular element of x is not in y, `FALSE` indicates that the element of x is in y.

Examples

```
c(1,2,3) %notin% c(0,2,4) # returns TRUE FALSE TRUE
# compare to:
c(1,2,3) %in% c(0,2,4) # returns FALSE TRUE FALSE

c(1,2) %notin% c(0,2,4) # returns TRUE FALSE
# versus (vectors changed place, hence, result adapts to length of x):
c(0,2,4) %notin% c(1,2) # returns TRUE FALSE TRUE

c('Hello', 'world') %notin% unlist(strsplit('The world is not enough', ' ')) # TRUE FALSE
c('Hello', 'world') %notin% c(1,2,3,4,5) # returns TRUE TRUE. Vector types don't need to match.
```

Index