Package 'mosaicCore'

June 24, 2018

Type Package

Title Common Utilities for Other MOSAIC-Family Packages

Version 0.6.0

Date 2018-06-23

Depends R (>= 3.0.0),

Imports stats, dplyr, lazyeval, rlang, tidyr, MASS

Suggests mosaicData, mosaic, ggformula, NHANES, testthat

Author Randall Pruim <rpruim@calvin.edu>, Daniel T. Kaplan <kaplan@macalester.edu>, Nicholas J. Horton <nhorton@amherst.edu>

Maintainer Randall Pruim <rpruim@calvin.edu>

Description Common utilities used in other MOSAIC-family packages are collected here.

License GPL (>= 2)

LazyLoad yes

LazyData yes

URL https://github.com/ProjectMOSAIC/mosaicCore

BugReports https://github.com/ProjectMOSAIC/mosaicCore/issues

RoxygenNote 6.0.1.9000

Encoding UTF-8

NeedsCompilation no

Repository CRAN

Date/Publication 2018-06-24 04:17:57 UTC

R topics documented:

`ash_points` *Compute knot points of an average shifted histogram*

Description

Mainly a utility for the **lattice** and **ggplot2** plotting functions, `ash_points()` returns the points to be plotted.

Usage

```
ash_points(x, binwidth = NULL, adjust = 1)
```

Arguments

x	A numeric vector
binwidth	The width of the histogram bins. If NULL (the default) the binwidth will be chosen so that approximately 10 bins cover the data. adjust can be used to to increase or decrease binwidth.
adjust	A number used to scale binwidth.

Value

A data frame containing x and y coordinates of the resulting ASH plot.

---

coef.function *Extract coefficients from a function*

Description

coef will extract the coefficients attribute from a function. Functions created by applying link{makeFun} to a model produced by lm(), glm(), or nls() store the model coefficients there to enable this extraction.

Usage

```
## S3 method for class 'function'
coef(object, ...)
```

Arguments

object	a function
...	ignored

Examples

```
if (require(mosaicData)) {
  model <- lm( width ~ length, data = KidsFeet)
  f <- makeFun( model )
  coef(f)
}
```

columns *return a vector of row or column indices*

Description

return a vector of row or column indices

Usage

```
columns(x, default = c())

rows(x, default = c())
```

Arguments

x	an object that may or may not have any rows or columns
default	what to return if there are no rows or columns

Value

if x has rows or columns, a vector of indices, else default

Examples

```
dim(iris)
columns(iris)
rows(iris)
columns(NULL)
columns("this doesn't have columns")
```

counts *Compute all proportions or counts*

Description

Compute vector of counts, proportions, or percents for each unique value (and NA if there is missing data) in a vector.

Usage

```
counts(x, ...)

## S3 method for class 'factor'
counts(x, ..., format = c("count", "proportion", "percent"))

## Default S3 method:
counts(x, ..., format = c("count", "proportion", "percent"))

## S3 method for class 'formula'
counts(x, data, ..., format = "count")

props(x, ..., format = "proportion")

percs(x, ..., format = "percent")
```

Arguments

x	A vector or a formula.
...	Arguments passed to methods.
format	One of "count", "proportion", or "percent". May be abbreviated.
data	A data frame.

See Also

mosaic::prop()

mosaic::count()

Examples

```
if (require(mosaicData)) {
  props(HELPrct$substance)
  # numeric version tallies missing values as well
  props(HELPmiss$link)
  # Formula version omits missing data with warning (by default)
  props( ~ link, data = HELPmiss)                      # omit NAs with warning
  props( ~ link, data = HELPmiss, na.action = na.pass)  # no warning; tally NAs
  props( ~ link, data = HELPmiss, na.action = na.omit)  # no warning, omit NAs
  props( ~ substance | sex, data = HELPrct)
  props( ~ substance | sex, data = HELPrct, format = "percent")
  percs( ~ substance | sex, data = HELPrct)
  counts( ~ substance | sex, data = HELPrct)
  df_stats( ~ substance | sex, data = HELPrct, props, counts)
  df_stats( ~ substance | sex, data = HELPmiss, props, na.action = na.pass)
}
```

coverage *Interval statistics*

Description

Calculate coverage intervals and confidence intervals for the sample mean, median, sd, proportion, ... Typically, these will be used within `df_stats()`. For the mean, median, and sd, the variable x must be quantitative. For proportions, the x can be anything; use the `success` argument to specify what value you want the proportion of. Default for `success` is `TRUE` for x logical, or the first level returned by `unique` for categorical or numerical variables.

Usage

```
coverage(x, level = 0.95, na.rm = TRUE)

ci.mean(x, level = 0.95, na.rm = TRUE)

ci.median(x, level = 0.9, na.rm = TRUE)

ci.sd(x, level = 0.95, na.rm = TRUE)

ci.prop(x, success = NULL, level = 0.95, method = c("Clopper-Pearson",
  "binom.test", "Score", "Wilson", "prop.test", "Wald", "Agresti-Coull",
  "Plus4"))
```

Arguments

`x`	a variable.
`level`	number in 0 to 1 specifying the confidence level for the interval. (Default: 0.95)
`na.rm`	if `TRUE` disregard missing data
`success`	for proportions, this specifies the categorical level for which the calculation of proportion will be done. Defaults: `TRUE` for logicals for which the proportion is to be calculated.
`method`	for `ci.prop()`, the method to use in calculating the confidence interval. See `mosaic::binom.test()` for details.

Details

Methods: `ci.mean()` uses the standard t confidence interval. `ci.median()` uses the normal approximation method. `ci.sd()` uses the chi-squared method. `ci.prop()` uses the binomial method. In the usual situation where the `mosaic` package is available, `ci.prop()` uses `mosaic::binom.test()` internally, which provides several methods for the calculation. See the documentation for `binom.test()` for details about the available methods. Clopper-Pearson is the default method. When used with `df_stats()`, the confidence interval is calculated for each group separately. For "pooled" confidence intervals, see methods such as `lm()` or `glm()`.

Value

a named numerical vector with components `lower` and `upper`, and, in the case of `ci.prop()`, `center`. When used the `df_stats()`, these components are formed into a data frame.

Note

When using these functions with `df_stats()`, omit the `x` argument, which will be supplied automatically by `df_stats()`. See examples.

See Also

`mosaicCore::df_stats()`, `mosaic::binom.test()`, `mosaic::t.test()`

Examples

```
# The central 95% interval
df_stats(hp ~ cyl, data = mtcars, c95 = coverage(0.95))
# The confidence interval on the mean
df_stats(hp ~ cyl, data = mtcars, mean, ci.mean)
# What fraction of cars have 6 cylinders?
df_stats(mtcars, ~ cyl, six_cyl_prop = ci.prop(success = 6, level = 0.90))
# Use without `df_stats()` (rare)
ci.mean(mtcars$hp)
```

---

dfapply *apply-type function for data frames*

Description

An `apply`-type function for data frames.

Usage

```
dfapply(data, FUN, select = TRUE, ...)
```

Arguments

`data`	data frame
`FUN`	a function to apply to (some) variables in the data frame
`select`	a logical, character (naming variables), or numeric vector or a function used to select variables to which `FUN` is applied. If a function, it should take a vector as input and return a single logical. See examples.
`...`	arguments passed along to `FUN`

See Also

`apply()`, `sapply()`, `tapply()`, `lapply()`, `inspect()`

Examples

```
dfapply(iris, mean, select = is.numeric)
dfapply(iris, mosaic::favstats, select = c(TRUE, TRUE, FALSE, FALSE, FALSE))
dfapply(iris, mean, select = c(1,2))
dfapply(iris, mean, select = c("Sepal.Length", "Petal.Length"))
if (require(mosaicData)) {
  dfapply(HELPrct, table, select = is.factor)
  do.call(rbind, dfapply(HELPrct, mean, select = is.numeric))
}
```

---

df_stats *Calculate statistics on a variable*

Description

Creates a data frame of statistics calculated on one variable, possibly for each group formed by combinations of additional variables. The resulting data frame has one column for each of the statistics requested as well as columns for any grouping variables.

Usage

```
df_stats(formula, data, ..., drop = TRUE, fargs = list(), sep = "_",
  format = c("wide", "long"), groups = NULL, long_names = TRUE,
  nice_names = FALSE, na.action = "na.warn")
```

Arguments

formula	A formula indicating which variables are to be used. Semantics are approximately as in `lm()` since `stats::model.frame()` is used to turn the formula into a data frame. But first conditions and `groups` are re-expressed into a form that `stats::model.frame()` can interpret. See details.
data	A data frame or list containing the variables.
...	Functions used to compute the statistics. If this is empty, a default set of summary statistics is used. Functions used must accept a vector of values and return either a (possibly named) single value, a (possibly named) vector of values, or a data frame with one row. Functions can be specified with character strings, names, or expressions that look like function calls with the first argument missing. The latter option provides a convenient way to specify additional arguments. See the examples. Note: If these arguments are named, those names will be used in the data frame returned (see details). Such names may not be among the names of the named arguments of `df_stats()`. If a function is specified using `::`, be sure to include the trailing parens, even if there are no additional arguments required.
drop	A logical indicating whether combinations of the grouping variables that do not occur in `data` should be dropped from the result.
fargs	Arguments passed to the functions in `...`.

sep	A character string to separate components of names. Set to "" if you don't want separation.
format	One of "long" or "wide" indicating the desired shape of the returned data frame.
groups	An expression to be evaluated in data and defining (additional) groups. This isn't necessary, since these can be placed into the formula, but it is provided for similarity to other functions from the **mosaic** package.
long_names	A logical indicating whether the default names should include the name of the variable being summarized as well as the summarizing function name in the default case when names are not derived from the names of the returned object or an argument name.
nice_names	A logical indicating whether make.names() should be used to force names of the returned data frame to by syntactically valid.
na.action	A function (or character string naming a function) that determines how NAs are treated. Options include "na.warn" which removes missing data and emits a warning, "na.pass" which includes all of the data, "na.omit" or "na.exclude" which silently discard missing data, and "na.fail" which fails if there is missing data. See link[stats]{na.pass}() and na.warn() for details. The default is "na.warn" unless no function are specified in ..., in which case "na.pass" is used since the default function reports the number of missing values.

Details

Use a one-sided formula to compute summary statistics for the left hand side expression over the entire data. Use a two-sided formula to compute summary statistics for the left hand expression for each combination of levels of the expressions occurring on the right hand side. This is most useful when the left hand side is quantitative and each expression on the right hand side has relatively few unique values. A function like mosaic::ntiles() is often useful to create a few groups of roughly equal size determined by ranges of a quantitative variable. See the examples.

Note that unlike dplyr::summarise(), df_stats() ignores any grouping defined in data if data is a grouped tibble.

Value

A data frame. Names of columns in the resulting data frame consist of three parts separated by sep. The first part is the argument name, if it exists, else the function. The second part is the name of the variable being summarised if long_names == TRUE and the first part is the function name, else "" The third part is the names of the object returned by the summarizing function, if they exist, else a sequence of consecutive integers or "" if there is only one component returned by the summarizing function. See the examples.

Cautions Regarding Formulas

The use of | to define groups is tricky because (a) stats::model.frame() doesn't handle this sort of thing and (b) | is also used for logical or. The current algorithm for handling this will turn the first occurrence of | into an attempt to condition, so logical or cannot be used before conditioning in the formula. If you have need of logical or, we suggest creating a new variable that contains the results of evaluating the expression.

Similarly, addition (+) is used to separate grouping variables, not for arithmetic.

Examples

```
df_stats( ~ hp, data = mtcars)
# There are several ways to specify functions
df_stats( ~ hp, data = mtcars, mean, trimmed_mean = mean(trim = 0.1), "median",
  range, Q = quantile(c(0.25, 0.75)))
# When using ::, be sure to include parents, even if there are no additional arguments.
df_stats( ~ hp, data = mtcars, mean = base::mean(), trimmed_mean = base::mean(trim = 0.1))

# force names to by syntactically valid
df_stats( ~ hp, data = mtcars, Q = quantile(c(0.25, 0.75)), nice_names = TRUE)
# shorter names
df_stats( ~ hp, data = mtcars, mean, trimmed_mean = mean(trim = 0.1), "median", range,
  long_names = FALSE)
# wide vs long format
df_stats( hp ~ cyl, data = mtcars, mean, median, range)
df_stats( hp ~ cyl, data = mtcars, mean, median, range, format = "long")
# More than one grouping variable -- 3 ways.
df_stats( hp ~ cyl + gear, data = mtcars, mean, median, range)
df_stats( hp ~ cyl | gear, data = mtcars, mean, median, range)
df_stats( hp ~ cyl, groups = gear, data = mtcars, mean, median, range)

# because the result is a data frame, df_stats() is also useful for creating plots
if(require(ggformula)) {
  gf_violin(hp ~ cyl, data = mtcars, group = ~ cyl) %>%
  gf_point(mean_hp ~ cyl, data = df_stats(hp ~ cyl, data = mtcars, mean),
    color = ~ "mean") %>%
  gf_point(median_hp ~ cyl, data = df_stats(hp ~ cyl, data = mtcars, median),
    color = ~"median") %>%
  gf_labs(color = "")
}

# magrittr style piping is also supported
if (require(ggformula)) {
  mtcars %>%
    df_stats(hp ~ cyl, mean, median, range)
  mtcars %>%
    df_stats(hp ~ cyl + gear, mean, median, range) %>%
    gf_point(mean_hp ~ cyl, color = ~ factor(gear)) %>%
    gf_line(mean_hp ~ cyl, color = ~ factor(gear))
}

# can be used with a categorical response, too
if (require(mosaic)) {
  df_stats(sex ~ substance, data = HELPrct, table, prop_female = prop)
}
if (require(mosaic)) {
  df_stats(sex ~ substance, data = HELPrct, table, props)
}
```

ediff *Lagged Differences with equal length*

Description

Often when creating lagged differences, it is awkward that the differences vector is shorter than the original. `ediff` pads with `pad.value` to make its output the same length as the input.

Usage

```
ediff(x, lag = 1, differences = 1, pad = c("head", "tail", "symmetric"),
  pad.value = NA, frontPad, ...)
```

Arguments

`x`	a numeric vector or a matrix containing the values to be differenced
`lag`	an integer indicating which lag to use
`differences`	an integer indicating the order of the difference
`pad`	one of `"head"`, `"tail"`, or `"symmetric"`. indicating where the `pad.value` padding should be added to the result.
`pad.value`	the value to be used for padding.
`frontPad`	logical indicating whether padding is on the front (head) or back (tail) end. This exists for backward compatibility. New code should use `pad` instead.
`...`	further arguments to be passed to or from methods

See Also

`diff()` since `ediff` is a thin wrapper around `diff()`.

Examples

```
ediff(1:10)
ediff(1:10, pad.value = 0)
ediff(1:10, 2)
ediff(1:10, 2, 2)
x <- cumsum(cumsum(1:10))
ediff(x, lag = 2)
ediff(x, differences = 2)
ediff(x, differences = 2, pad = "symmetric")
ediff(.leap.seconds)
if (require(mosaic)) {
  Men <- subset(SwimRecords, sex == "M")
  Men <- mutate(Men, change = ediff(time), interval = ediff(year))
  head(Men)
}
```

evalFormula *Evaluate a formula*

Description

Evaluate a formula

Usage

```
evalFormula(formula, data = parent.frame(), subset, ops = c("+", "&"))
```

Arguments

formula	a formula (`y ~ x | z`) to evaluate
data	a data frame or environment in which evaluation occurs
subset	an optional vector describing a subset of the observations to be used. Currently only implemented when data is a data frame.
ops	a vector of operator symbols allowable to separate variables in rhs

Value

a list containing data frames corresponding to the left, right, and condition slots of `formula`

Examples

```
if (require(mosaicData)) {
data(CPS85)
cps <- CPS85[1:6,]
cps
evalFormula(wage ~ sex & married & age | sector & race, data=cps)
}
```

evalSubFormula *Evaluate a part of a formula*

Description

Evaluate a part of a formula

Usage

```
evalSubFormula(x, data = NULL, ops = c("+", "&"), env = parent.frame())
```

Arguments

x	an object appearing as a subformula (typically a name or a call)
data	a data frame or environment in which things are evaluated
ops	a vector of operators that are not evaluated as operators but instead used to further split x
env	an environment in which to search for objects not in data.

Value

a data frame containing the terms of the evaluated subformula

Examples

```
if (require(mosaicData)) {
data(CPS85)
cps <- CPS85[1:6,]
cps
evalSubFormula( rhs( ~ married & sector), data=cps )
}
```

---

fit_distr_fun *Fit a distribution to data and return a function*

Description

Given the name of a family of 1-dimensional distributions, this function chooses a particular member of the family that fits the data and returns a function in the selected p, d, q, or r format. When analytical solutions do not exist, `MASS::fitdistr()` is used to estimate the parameters by numerical maximum likelihood.

Usage

```
fit_distr_fun(data, formula, dist, start = NULL, ...)
```

Arguments

data	A data frame.
formula	A formula. A distribution will be fit to the data defined by the right side and evaluated in data.
dist	A string naming the function desired. Tyically this will be "d", "p", "q", or "r" followed by the (abbrevation for) a family of distributions such as "pnorm", "rgamma". Fitting is done use `MASS::fitdistr()`; see the help there for a list of distributions that are available.
start	Starting values for the numerical maximum likelihood method (passed to `MASS::fitdistr`).
...	Additional arguments to MASS::fitdistr()

Value

A function of one variable that acts like, say, `pnorm()`, `dnorm()`, `qnorm()`, or `rnorm()`, but with the default values of the parameters set to their maximum likelihood estimates.

Examples

```
fit_distr_fun( ~ cesd, data = mosaicData::HELPrct, dist = "dnorm")
fit_distr_fun( ~ cesd, data = mosaicData::HELPrct, dist = "pnorm")
fit_distr_fun( ~ cesd, data = mosaicData::HELPrct, dist = "qpois")
```

---

formularise *Convert lazy objects into formulas*

Description

Convert lazy objects into a formula

Usage

```
formularise(lazy_formula, envir = parent.frame())
```

Arguments

`lazy_formula`	an object of class `lazy`
`envir`	an environment that will be come the environment of the returned formula

Details

The expression of the lazy object is evaluated in its environment. If the result is not a formula, then the formula is created with an empty left hand side and the expression on the right hand side.

Value

a formula

Examples

```
formularise(lazyeval::lazy(foo))
formularise(lazyeval::lazy(y ~ x))
bar <- a ~ b
formularise(lazyeval::lazy(bar))
```

infer_transformation *Infer a Back-Transformation*

Description

For a handful of transformations on y, infer the reverse transformation. If the transformation is not recognized, return the identity function. This is primarily intended to be used for setting a default value in other functions.

Usage

```
infer_transformation(formula, warn = TRUE)
```

Arguments

formula	A formula as used by, for example, lm().
warn	A logical.

Value

A function.

inspect *Inspect objects*

Description

Print a short summary of the contents of an object. Most useful as a way to get a quick overview of the variables in data frame.

Usage

```
inspect(object, ...)

## S3 method for class 'list'
inspect(object, max.level = 2, ...)

## S3 method for class 'character'
inspect(object, ...)

## S3 method for class 'logical'
inspect(object, ...)

## S3 method for class 'numeric'
inspect(object, ...)
```

```
## S3 method for class 'factor'
inspect(object, ...)

## S3 method for class 'Date'
inspect(object, ...)

## S3 method for class 'POSIXt'
inspect(object, ...)

## S3 method for class 'data.frame'
inspect(object, select = TRUE,
  digits = getOption("digits", 3), ...)

## S3 method for class 'inspected_data_frame'
print(x, digits = NULL, ...)
```

Arguments

object	a data frame or a vector
...	additional arguments passed along to specific methods
max.level	an integer giving the depth to which lists should be expanded
select	a logical, character (naming variables), or numeric vector or a function used to select variables to which FUN is applied. If a function, it should take a vector as input and return a single logical. See examples here and at link{dfapply}.
digits	and integer giving the number of digits to display
x	an object

Examples

```
if (require(mosaicData)) {
  inspect(Births78)
  inspect(Births78, is.numeric)
}
```

---

joinFrames *Join data frames*

Description

Join data frames

Usage

```
joinFrames(...)

joinTwoFrames(left, right)
```

Arguments

`...`	data frames to be joined
`left, right`	data frames

Value

a data frame containing columns from each of data frames being joined.

`logical2factor` *Convert logical vector into factor*

Description

Turn logicals into factors with levels ordered with `TRUE` before `FALSE`. Other inputs are returned unaltered.

Usage

```
logical2factor(x, ...)

## Default S3 method:
logical2factor(x, ...)

## S3 method for class 'data.frame'
logical2factor(x, ...)
```

Arguments

`x`	a vector or data frame
`...`	additional arguments (currently ignored)

Value

If `x` is a vector either `x` or the result of converting `x` into a factor with levels `TRUE` and `FALSE` (in that order); if `x` is a data frame, a data frame with all logicals converted to factors in this manner.

logit *Logit and inverse logit functions*

Description

Logit and inverse logit functions

Usage

```
logit(x)

ilogit(x)
```

Arguments

x	a numeric vector

Value

For `logit` the value is

$$log(x/(1-x))$$

For `ilogit` the value is

$$exp(x)/(1+exp(x))$$

Examples

```
p <- seq(.1, .9, by=.10)
l <- logit(p); l
ilogit(l)
ilogit(l) == p
```

makeFun *Create a function from a formula*

Description

Provides an easy mechanism for creating simple "mathematical" functions via a formula interface.

Usage

```
makeFun(object, ...)

## S3 method for class 'function'
makeFun(object, ..., strict.declaration = TRUE,
  use.environment = TRUE, suppress.warnings = FALSE)

## S3 method for class 'formula'
makeFun(object, ..., strict.declaration = TRUE,
  use.environment = TRUE, suppress.warnings = TRUE)

## S3 method for class 'lm'
makeFun(object, ..., transformation = NULL)

## S3 method for class 'glm'
makeFun(object, ..., type = c("response", "link"),
  transformation = NULL)

## S3 method for class 'nls'
makeFun(object, ..., transformation = NULL)
```

Arguments

object	an object from which to create a function. This should generally be specified without naming.
...	additional arguments in the form var = val that set default values for the inputs to the function.
strict.declaration	if TRUE (the default), an error is thrown if default values are given for variables not appearing in the object formula.
use.environment	if TRUE, then variables implicitly defined in the object formula can take default values from the environment at the time makeFun is called. A warning message alerts the user to this situation, unless suppress.warnings is TRUE.
suppress.warnings	A logical indicating whether warnings should be suppressed.
transformation	a function used to transform the response. This can be useful to invert a transformation used on the response when creating the model. If NULL, an attempt will be made to infer the transformation from the model formula. A few simple transformations (log, log2, sqrt) are recognized. For other transformations, transformation should be provided explicitly.
type	one of 'response' (default) or 'link' specifying scale to be used for value of function returned.

Details

The definition of the function is given by the left side of a formula. The right side lists at least one of the inputs to the function. The inputs to the function are all variables appearing on either the left

or right sides of the formula. Those appearing in the right side will occur in the order specified. Those not appearing in the right side will appear in an unspecified order.

When creating a function from a model created with `lm`, `glm`, or `nls`, the function produced is a wrapper around the corresponding version of `predict`. This means that having variables in the model with names that match arguments of `predict` will lead to potentially ambiguous situations and should be avoided.

Value

a function

Examples

```
f <- makeFun( sin(x^2 * b) ~ x & y & a); f
g <- makeFun( sin(x^2 * b) ~ x & y & a, a = 2 ); g
h <- makeFun( a * sin(x^2 * b) ~ b & y, a = 2, y = 3); h
if (require(mosaicData)) {
  model <- lm( log(length) ~ log(width), data = KidsFeet)
  f <- makeFun(model, transformation = exp)
  f(8.4)
  head(KidsFeet, 1)
}

if (require(mosaicData)) {
  model <- lm(wage ~ poly(exper, degree = 2), data = CPS85)
  fit <- makeFun(model)
  if (require(ggformula)) {
    gf_point(wage ~ exper, data = CPS85) %>%
    gf_fun(fit(exper) ~ exper, color = "red")
  }
}
if (require(mosaicData)) {
model <- glm(wage ~ poly(exper, degree = 2), data = CPS85, family = gaussian)
fit <- makeFun(model)
  if (require(ggformula)) {
    gf_jitter(wage ~ exper, data = CPS85) %>%
    gf_fun(fit(exper) ~ exper, color = "red")
    gf_jitter(wage ~ exper, data = CPS85) %>%
    gf_function(fun = fit, color = "blue")
  }
}
if (require(mosaicData)) {
model <- nls( wage ~ A + B * exper + C * exper^2, data = CPS85, start = list(A = 1, B = 1, C = 1) )
fit <- makeFun(model)
  if (require(ggformula)) {
    gf_point(wage ~ exper, data = CPS85) %>%
    gf_fun(fit(exper) ~ exper, color = "red")
  }
}
```

make_df *Convert to a data frame*

Description

A generic and several methods for converting objects into data frames.

Usage

```
make_df(object, ...)

## S3 method for class 'list'
make_df(object, ...)

## S3 method for class 'matrix'
make_df(object, ...)

## S3 method for class 'numeric'
make_df(object, ...)

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

Arguments

`object`	An object to be converted into a data frame.
`...`	Additional arguments used by methods.

Details

These methods are primarily for internal use inside `df_stats()`, but are exported in case they have other uses. The conversion works as follows. Data frames are left as is. Matrices are converted column-by-column and the columns assembled with `as.data.frame()`; this allows matrices that are lists to be converted into data frames where columns can have differing types. The names are then set to the column names of `object`, even if that results in `NULL`. A numeric vector is converted into a data frame with 1 column. If `object` is a list, each element is converted using `vector2df()` and the resulting columns are joined with `bind_rows()`.

modelVars *extract predictor variables from a model*

Description

extract predictor variables from a model

Usage

```
modelVars(model)
```

Arguments

model	a model, typically of class lm or glm

Value

a vector of variable names

Examples

```
if (require(mosaicData)) {
  model <- lm( wage ~ poly(exper, degree = 2), data = CPS85 )
  modelVars(model)
}
```

---

mosaic_formula *Convert formulas into standard shapes*

Description

These functions convert formulas into standard shapes, including by incorporating a groups argument.

Usage

```
mosaic_formula(formula, groups = NULL, envir = parent.frame(),
  max.slots = 3, groups.first = FALSE)

mosaic_formula_q(formula, groups = NULL, max.slots = 3,
  groups.first = FALSE, ...)
```

Arguments

formula	a formula
groups	a name used for grouping
envir	the environment in which the resulting formula may be evaluated. May also be NULL, a list, a data frame, or a pairlist.
max.slots	an integer specifying the maximum number of slots for the resulting formula. An error results from trying to create a formula that is too complex.
groups.first	a logical indicating whether groups should be inserted ahead of the condition (else after).
...	additional arguments (currently ignored)

Details

mosaic_formula_q uses nonstandard evaluation of groups that may be necessary for use within other functions. mosaic_formula is a wrapper around mosaic_formula_q and quotes groups before passing it along.

Examples

```
mosaic_formula( ~ x | z )
mosaic_formula( ~ x, groups=g )
mosaic_formula( y ~ x, groups=g )
# this is probably not what you want for interactive use.
mosaic_formula_q( y ~ x, groups=g )
# but it is for programming
foo <- function(x, groups=NULL) {
    mosaic_formula_q(x, groups=groups, envir=parent.frame())
}
foo( y ~ x , groups = g)
```

---

na.warn *Exclude Missing Data with Warning*

Description

Similar to stats::na.exclude() this function excludes missing data. When missing data are excluded, a warning message indicating the number of excluded rows is emited as a caution for the user.

Usage

```
na.warn(object, ...)
```

Arguments

object	an R object, typically a data frame
...	further arguments special methods could require.

---

named *List extraction*

Description

These functions create subsets of lists based on their names

Usage

```
named(l)

unnamed(l)

named_among(l, n)
```

Arguments

l	A list.
n	A vector of character strings (potential names).

Value

A sublist of `l` determined by `names(l)`.

`nice_names` *Nice names*

Description

Convert a character vector into a similar character vector that would work better as names in a data frame by avoiding certain awkward characters

Usage

```
nice_names(x, unique = TRUE)
```

Arguments

x	a character vector
unique	a logical indicating whether returned values should be uniquified.

Value

a character vector

Examples

```
nice_names( c("bad name", "name (crazy)", "a:b", "two-way") )
```

n_missing *Counting missing/non-missing elements*

Description

Counting missing/non-missing elements

Usage

```
n_missing(..., type = c("any", "all"))

n_not_missing(..., type = c("any", "all"))

n_total(..., type = c("any", "all"))
```

Arguments

...	vectors of equal length to be checked in parallel for missing values.
type	one of "any" (default) or "all".

Value

a vector of counts of missing or non-missing values.

Examples

```
if (require(NHANES) && require(mosaic)) {
  tally( ~ is.na(Height) + is.na(Weight), data = NHANES, margins = TRUE)
  NHANES %>%
    summarise(
      mean.wt = mean(Weight, na.rm = TRUE),
      missing.Wt = n_missing(Weight),
      missing.WtAndHt = n_missing(Weight, Height, type = "all"),
      missing.WtOrHt = n_missing(Weight, Height, type = "any")
      )
    }
```

parse.formula *Parse Formulas*

Description

Utilities for extracting portions of formulas.

Usage

```
parse.formula(formula, ...)

rhs(x, ...)

lhs(x, ...)

condition(x, ...)

operator(x, ...)

## S3 method for class 'formula'
rhs(x, ...)

## S3 method for class 'formula'
lhs(x, ...)

## S3 method for class 'formula'
condition(x, ...)

## S3 method for class 'formula'
operator(x, ...)

## S3 method for class 'parsedFormula'
rhs(x, ...)

## S3 method for class 'parsedFormula'
lhs(x, ...)

## S3 method for class 'parsedFormula'
operator(x, ...)

## S3 method for class 'parsedFormula'
condition(x, ...)
```

Arguments

formula,	a formula
...	additional arguments, current ignored
x,	an object (currently a formula or parsedFormula)

Details

currently this is primarily concerned with extracting the operator, left hand side, right hand side (minus any condition) and the condition. Improvements/extensions may come in the future.

Value

an object of class `parsedFormula` from which information is easy to extract

---

`print.msummary.lm` *Modified summaries*

Description

`msummary` provides modified summary objects that typically produce output that is either identical to or somewhat terser than their `summary()` analogs. The contents of the object itself are unchanged (except for an augmented class) so that other downstream functions should work as before.

Usage

```
## S3 method for class 'msummary.lm'
print(x, digits = max(3L, getOption("digits") - 3L),
  symbolic.cor = x$symbolic.cor,
  signif.stars = getOption("show.signif.stars"), ...)

## S3 method for class 'msummary.glm'
print(x, digits = max(3L, getOption("digits") - 3L),
  symbolic.cor = x$symbolic.cor,
  signif.stars = getOption("show.signif.stars"), ...)

msummary(object, ...)

## Default S3 method:
msummary(object, ...)

## S3 method for class 'lm'
msummary(object, ...)

## S3 method for class 'glm'
msummary(object, ...)
```

Arguments

`x`	an object to summarize
`digits`	desired number of digits to display
`symbolic.cor`	see `summary()`
`signif.stars`	a logical indicating whether to display stars to indicate significance
`...`	additional arguments
`object`	an object to summarise

Examples

```
msummary(lm(Sepal.Length ~ Species, data = iris))
```

---

prop *Compute proportions, percents, or counts for a single level*

Description

Compute proportions, percents, or counts for a single level

Usage

```
prop(x, data = parent.frame(), useNA = "no", ..., success = NULL,
  level = NULL, long.names = TRUE, sep = ".", format = c("proportion",
  "percent", "count"), quiet = TRUE, pval.adjust = FALSE)

prop1(..., pval.adjust = TRUE)

count(x, ...)

perc(x, data = parent.frame(), ..., format = "percent")
```

Arguments

x	an R object, usually a formula
data	a data frame in which x is to be evaluated
useNA	an indication of how NA's should be handled. By default, they are ignored.
...	arguments passed through to tally()
success	the level for which counts, proportions or percents are calculated
level	Deprecated. Use sucess.
long.names	a logical indicating whether long names should be when there is a conditioning variable
sep	a character used to separate portions of long names
format	one of proportion, percent, or count, possibly abbreviated
quiet	a logical indicating whether messages regarding the success level should be supressed.
pval.adjust	a logical indicating whether the "p-value" adjustment should be applied. This adjustment adds 1 to the numerator and denominator counts.

Details

prop1 is intended for the computation of p-values from randomization distributions and differs from prop only in its default value of pval.adjust.

Note

For 0-1 data, success is set to 1 by default since that a standard coding scheme for success and failure.

Examples

```
if (require(mosaicData)) {
  prop( ~sex, data=HELPrct)
  prop( ~sex, data=HELPrct, success = "male")
  count( ~sex | substance, data=HELPrct)
  prop( ~sex | substance, data=HELPrct)
  perc( ~sex | substance, data=HELPrct)
}
```

---

reop_formula *Insert Inhibition of Interpretation/Conversion into formulas*

Description

model.frame() assumes that certain operations (e.g. /, *, ^) have special meanings. These can be inhibited using I(). This function inserts I() into a formula when encountering a specified operator or parens.

Usage

```
reop_formula(x, ops = c("/", "*", "^"))
```

Arguments

x	a formula (or a call of length 2 or 3, for recursive processing of formulas). Other objects are returned unchanged.
ops	a vector of character represention of operators to be inhibited.

Value

a formula with I() inserted where required to inhibit interpretation/conversion.

Examples

```
reop_formula(y ~ x * y)
reop_formula(y ~ (x * y))
reop_formula(y ~ x ^ y)
reop_formula(y ~ x * y ^ z)
```

`tally` *Tabulate categorical data*

Description

Tabulate categorical data

Usage

```
tally(x, ...)

## S3 method for class 'tbl'
tally(x, wt, sort = FALSE, ..., envir = parent.frame())

## S3 method for class 'data.frame'
tally(x, wt, sort = FALSE, ..., envir = parent.frame())

## S3 method for class 'formula'
tally(x, data = parent.frame(2), format = c("count",
  "proportion", "percent", "data.frame", "sparse", "default"),
  margins = FALSE, quiet = TRUE, subset, groups = NULL, useNA = "ifany",
  groups.first = FALSE, ...)
```

Arguments

`x`	an object
`...`	additional arguments passed to `table()`
`wt`	for weighted tallying, see `dplyr::tally()` in **dplyr**
`sort`	a logical, see `dplyr::tally()` in **dplyr**
`envir`	an environment in which to evaluate
`data`	a data frame or environment in which evaluation occurs. Note that the default is `data=parent.frame()`. This makes it convenient to use this function interactively by treating the working envionment as if it were a data frame. But this may not be appropriate for programming uses. When programming, it is best to use an explicit `data` argument – ideally supplying a data frame that contains the variables mentioned
`format`	a character string describing the desired format of the results. One of `'default'`, `'count'`, `'proportion'`, `'percent'`, `'data.frame'`, `'sparse'`, or `'default'`. In case of `'default'`, counts are used unless there is a condition, in which case proportions are used instead. Note that prior to version 0.9.3, `'default'` was the default, now it is `'count'`. `'data.frame'` converts the table to a data frame with one row per cell; `'sparse'` additionally removes any rows with 0 counts.
`margins`	a logical indicating whether marginal distributions should be displayed.
`quiet`	a logical indicating whether messages about order in which marginal distributions are calculated should be suppressed. See `stats::addmargins()`.

subset	an expression evaluating to a logical vector used to select a subset of data
groups	used to specify a condition as an alternative to using a formula with a condition.
useNA	as in table(), but the default here is "ifany".
groups.first	a logical indicating whether groups should be inserted ahead of the condition (else after).

Details

The **dplyr** package also exports a dplyr::tally() function. If x inherits from class "tbl" or "data frame", then **dplyr**'s dplyr::tally() is called. This makes it easier to have the two packages coexist.

Otherwise, tally() is designed as an alternative to table() and xtabs(). The primary use case it to describe a (possibly multi-dimensional) table using a formula. For a table of counts, each component of the formula becomes one of the dimensions of the cross table. For tables of proportions or percents, conditional proportions and percents are computed, conditioned on each level of all "secondary" (i.e., conditioning) variables, defined as everything other than the left hand side, if there is a left hand side to the formula; and everything except the right hand side if the left hand side of the formula is empty. Note that groups is folded into the formula prior to this determination and becomes part of the conditioning.

When marginal totals are added, they are added for all of the conditioning dimensions, and proportions should sum to 1 for each level of the conditioning variables. This can be useful to make it clear which conditional proportions are being computed.

See the examples for some typical use cases.

Value

A object of class "table", unless passing through to **dplyr** or converted to a data frame because format is "data.frame" or "sparse".

Note

The current implementation when format = "sparse" first creates the full data frame and then removes the unneeded rows. So the savings is in terms of space, not time.

Examples

```
if (require(mosaicData)) {
tally( ~ substance, data = HELPrct)
tally( ~ substance + sex , data = HELPrct)
tally( sex ~ substance, data = HELPrct)   # equivalent to tally( ~ sex | substance, ... )
tally( ~ substance | sex , data = HELPrct)
tally( ~ substance | sex , data = HELPrct, format = 'count', margins = TRUE)
tally( ~ substance + sex , data = HELPrct, format = 'percent', margins = TRUE)
tally( ~ substance | sex , data = HELPrct, format = 'percent', margins = TRUE)
# force NAs to show up
tally( ~ sex, data = HELPrct, useNA = "always")
# show NAs if any are there
tally( ~ link, data = HELPrct)
# ignore the NAs
```

```
tally( ~ link, data = HELPrct, useNA = "no")
}
```

---

vector2df *Convert a vector to a data frame*

Description

Convert a vector into a 1-row data frame using the names of the vector as column names for the data frame.

Usage

```
vector2df(x, nice_names = FALSE)
```

Arguments

x	A vector.
nice_names	A logical indicating whether names should be nicified.

Value

A data frame.

Examples

```
vector2df(c(1, b = 2, `(Intercept)` = 3))
vector2df(c(1, b = 2, `(Intercept)` = 3), nice_names = TRUE)
```

Index