Package: zmisc (via r-universe)

November 5, 2024

Type Package

Title Vector Look-Ups and Safer Sampling

Version 0.2.3.9002

Author Magnus Thor Torfason

Maintainer Magnus Thor Torfason <m@zulutime.net>

Description A collection of utility functions that facilitate looking up vector values from a lookup table, annotate values in at table for clearer viewing, and support a safer approach to vector sampling, sequence generation, and aggregation.

License MIT + file LICENSE

URL https://github.com/torfason/zmisc/, https://torfason.github.io/zmisc/

Suggests desc, dplyr, haven, knitr, labelled, purrr, rmarkdown, roxygen2, rprojroot, stringr, testthat, tibble

VignetteBuilder knitr

Encoding UTF-8

Language en-US

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Repository https://torfason.r-universe.dev

RemoteUrl https://github.com/torfason/zmisc

RemoteRef HEAD

RemoteSha 8b90b6485b21f0879558e41622215d53015aaeca

Contents

`lookup` *Lookup values from a lookup table*

Description

The `lookup()` function implements lookup of values (such as variable names) from a lookup table which maps keys onto values (such as variable labels or descriptions).

The lookup table can be in the form of a two-column `data.frame`, in the form of a named `vector`, or in the form of a `list`. If the table is in the form of a `data.frame`, the key column should be named either `key` or `name`, and the value column should be named `value` (for the value). If the lookup table is in the form of a named `vector` or `list`, the names are used as the key, and the returned value is taken from the values in the vector or list.

The underlying lookup is done using `base::match()`, and all atomic data types except `factor` are supported. Factors are omitted due to the ambiguity in what should be looked up (the values or the levels). It is important that `x`, `.default` and the columns of `lookup_table` are all of the same type (specifically of the same `base::mode()`). If the lookup table is specified as a `vector` or `list`, only the `character` variables are supported, because `name(lookup_table)` is always of mode `character`.

Original values are returned if they are not found in the lookup table. Alternatively, a `.default` can be specified for values that are not found. Note that it is possible to specify `NA` as one of the keys to look up NA values (only when using a `data.frame` as lookup table).

Any names or attributes of x are preserved.

The `lookuper()` function returns *a function* equivalent to the `lookup()` function, except that instead of taking a lookup table as an argument, the lookup table is embedded in the function itself.

This can be very useful, in particular when using the lookup function as an argument to other functions that expect a function which maps `character->character` (or other data types), but do not offer a good way to pass additional arguments to that function.

Usage

```
lookup(x, lookup_table, ..., .default = x)

lookuper(lookup_table, ..., .default = NULL)
```

Arguments

`x`	A vector whose elements are to be looked up.
`lookup_table`	The lookup table to use.
`...`	Reserved for future use.
`.default`	If a value is not found in the lookup table, the value will be taken from `.default`. This must be a vector of the same mode as x, and either of length 1 or the same length as `x`. Useful values include `x` (the default setting), `NA`, or `""` (an empty string). Specifying `.default = NULL` implies that `x` will be used for missing values.

Value

The `lookup()` function returns a vector based on `x`, with values replaced with the lookup values from `lookup_table`. Any values not found in the lookup table are taken from `.default`.

The `lookuper()` function returns *a function* that takes vectors as its argument `x`, and returns either the corresponding values from the underlying lookup table, or the original values from `x` for those elements that are not found in the lookup table (or looks them up from the `default`).

Examples

```
fruit_lookup_vector <- c(a = "Apple", b = "Banana", c = "Cherry")
lookup(letters[1:5], fruit_lookup_vector)
lookup(letters[1:5], fruit_lookup_vector, .default = NA)

mtcars_lookup_data_frame <- data.frame(
  name = c("mpg", "hp", "wt"),
  value = c("Miles/(US) gallon", "Gross horsepower", "Weight (1000 lbs)"))
lookup(names(mtcars), mtcars_lookup_data_frame)

# A more complex example, with numeric and NA values
numeric_lookup_table <- data.frame(
  key = c(1:5, NA), value = c(sqrt(1:5), 99999))
lookup(c(0:6, NA), numeric_lookup_table)

lookup_fruits <- lookuper(list(a = "Apple", b = "Banana", c = "Cherry"))
lookup_fruits(letters[1:5])
lookup_fruits_nomatch_na <-
  lookuper(list(a = "Apple", b = "Banana", c = "Cherry"), .default = NA)
lookup_fruits_nomatch_na(letters[1:5])
```

`notate` *Embed factor levels and value labels in values.*

Description

This function adds level/label information as an annotation to either factors or `labelled` variables. This function is called `notate()` rather than `annotate()` to avoid conflict with `ggplot2::annotate()`. It is a generic that can operate either on individual vectors or on a `data.frame`.

When printing `labelled` variables from a `tibble` in a console, both the numeric value and the text label are shown, but no variable labels. When using the `View()` function, only variable labels are shown but no value labels. For factors, there is no way to view the integer levels and values at the same time.

In order to allow the viewing of both variable and value labels at the same time, this function converts both `factor` and `labelled` variables to `character`, including both numeric levels (`labelled` values) and character values (`labelled` labels) in the output.

Usage

```
notate(x)
```

Arguments

x	The object (either vector or `date.frame` of vectors), that one desires to annotate and/or view.

Value

The processed `data.frame`, suitable for viewing, in particular through the `View()` function.

---

zample *Sample from a vector in a safe way*

Description

The `zample()` function duplicates the functionality of `sample()`, with the exception that it does not attempt the (sometimes dangerous) user-friendliness of switching the interpretation of the first element to a number if the length of the vector is 1. `zample()` *always* treats its first argument as a vector containing elements that should be sampled, so your code won't break in unexpected ways when the input vector happens to be of length 1.

Usage

```
zample(x, size = length(x), replace = FALSE, prob = NULL)
```

Arguments

x	The vector to sample from
size	The number of elements to sample from x (defaults to `length(x)`)
replace	Should elements be replaced after sampling (defaults to `false`)
prob	A vector of probability weights (defaults to equal probabilities)

Details

If what you really want is to sample from an interval between 1 and n, you can use `sample(n)` or `sample.int(n)` (but make sure to only pass vectors of length one to those functions).

Value

The resulting sample

Examples

```
# For vectors of length 2 or more, zample() and sample() are identical
set.seed(42); zample(7:11)
set.seed(42); sample(7:11)

# For vectors of length 1, zample() will still sample from the vector,
# whereas sample() will "magically" switch to interpreting the input
# as a number n, and sampling from the vector 1:n.
set.seed(42); zample(7)
set.seed(42); sample(7)

# The other arguments work in the same way as for sample()
set.seed(42); zample(7:11, size=13, replace=TRUE, prob=(5:1)^3)
set.seed(42); sample(7:11, size=13, replace=TRUE, prob=(5:1)^3)

# Of course, sampling more than the available elements without
# setting replace=TRUE will result in an error
set.seed(42); tryCatch(zample(7, size=2), error=wrap_error)
```

`zeq` *Generate sequence in a safe way*

Description

The `zeq()` function creates an increasing integer sequence, but differs from the standard one in that it will not silently generate a decreasing sequence when the second argument is smaller than the first. If the second argument is one smaller than the first it will generate an empty sequence, if the difference is greater, the function will throw an error.

Usage

```
zeq(from, to)
```

Arguments

`from`	The lower bound of the sequence
`to`	The higher bound of the sequence

Value

A sequence ranging from `from` to `to`

Examples

```
# For increasing sequences, zeq() and seq() are identical
zeq(11,15)
zeq(11,11)

# If second argument equals first-1, an empty sequence is returned
zeq(11,10)

# If second argument is less than first-1, the function throws an error
tryCatch(zeq(11,9), error=wrap_error)
```

---

zingle *Return the single (unique) value found in a vector*

Description

The zingle() function returns the first element in a vector, but only if all the other elements are identical to the first one (the vector only has a zingle value). If the elements are not all identical, it throws an error. The vector must contain at least one non-NA value, or the function errors out as well. This is especially useful in aggregations, when all values in a given group should be identical, but you want to make sure.

Usage

```
zingle(x, na.rm = FALSE)
```

Arguments

x	Vector of elements that should all be identical
na.rm	Should NA elements be removed prior to comparison

Details

Optionally takes a na.rm parameter, similarly to sum, mean and other aggregate functions. If TRUE, NA values will be removed prior to comparing the elements, so the function will accept input values that contain a combination of the single value and any NA values (but at least one non-NA value is required).

Only values are tested for equality. Any names are simply ignored, and the result is an unnamed value. This is in line with how other aggregation functions handle names.

Value

The zingle element in the vector

Examples

```
# If all elements are identical, all is good.
# The value of the element is returned.
zingle(c("Alpha", "Alpha", "Alpha"))

# If any elements differ, an error is thrown
tryCatch(zingle(c("Alpha", "Beta", "Alpha")), error=wrap_error)

if (require("dplyr", quietly=TRUE, warn.conflicts=FALSE)) {
  d <- tibble::tribble(
    ~id, ~name, ~fouls,
    1, "James", 3,
    2, "Jack",  2,
    1, "James", 4
  )

  # If the data is of the correct format, all is good
  d %>%
    dplyr::group_by(id) %>%
    dplyr::summarise(name=zingle(name), total_fouls=sum(fouls))
 }

if (require("dplyr", quietly=TRUE, warn.conflicts=FALSE)) {
  # If a name does not match its ID, we should get an error
  d[1,"name"] <- "Jammes"
  tryCatch({
    d %>%
      dplyr::group_by(id) %>%
      dplyr::summarise(name=zingle(name), total_fouls=sum(fouls))
  }, error=wrap_error)
}
```

Index