

Package ‘zmisc’

September 28, 2026

Type Package

Title Vector Look-Ups and Safer Sampling

Version 0.3.0

Description A collection of utility functions that facilitate looking up vector values from a lookup table, annotate values in a table for clearer viewing, and support a safer approach to vector sampling, sequence generation, and aggregation. Also included is a family of argument checks which return their input so that they compose nicely in a pipe.

License MIT + file LICENSE

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

Imports checkmate, rlang, glue

Suggests rmarkdown, labelled, testthat, tibble, dplyr, haven, knitr,
purrr, withr, here, data.table, spelling

VignetteBuilder knitr

Encoding UTF-8

Language en-US

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

Depends R (>= 4.1.0)

NeedsCompilation no

Author Magnus Thor Torfason [aut, cre]

Maintainer Magnus Thor Torfason <m@zulutime.net>

Repository CRAN

Date/Publication 2026-09-27 22:40:02 UTC

Contents

asciify	2
chk_atomic	3
chk_composite	8
chk_other	10
glue_vector	12
lookup	13
notate	15
yencode	16
zample	17
zeq	18
zingle	19
Index	20

asciify	<i>Convert non-ASCII characters to their ASCII equivalents</i>
---------	--

Description

This function replaces non-ASCII characters in a string with their ASCII equivalents. It supports a range of European non-ASCII characters, including Icelandic, Swedish, Norwegian, Danish, Finnish, German, Estonian, Latvian, Lithuanian, Polish, Hungarian, Slovenian, Czech, Slovak, Maltese, Romanian, Albanian, and Croatian.

Usage

```
asciify(x, verify = TRUE)
```

Arguments

- x A character vector to be processed.
- verify A logical value indicating whether to verify that the result is ASCII. Defaults to TRUE. If FALSE, the function will not check that the result is ASCII and it may return non-ASCII characters.

Value

A character vector with non-ASCII characters replaced by their ASCII equivalents.

Examples

```
asciify("Jón Þór Birgisson") # "Jon Thor Birgisson"
asciify("förståndshandikapp") # "forstandshandikapp"
asciify("Viðareiði") # "Vidareidi"
asciify("übermensch") # "uebermensch"
asciify("Jürgen Klopp") # "Juergen Klopp"
asciify("röömsameelsus") # "roomsameelsus"
```

```

asciify("Mężczyzna") # "Mezczyzna"
asciify("Škoda") # "Skoda"

```

chk_atomic

Checks for scalars and atomic vectors

Description

Validity checks for atomic vectors. Each function below takes the same arguments – `na.ok`, `null.ok`, `attr.ok`, and `length` or `range` where they apply.

They are meant to be cheap enough to leave at the top of any function, a passing value with default arguments tends to take under a microsecond on a modern computer; with extra specification arguments it can take up to ten microseconds. Assembling a good message happens on the failing path, which runs once and then stops.

R Type	Scalar	Vector
Any type	chk_scalar(x)	chk_atomic(x)
logical	chk_flag(x)	chk_logical(x)
character	chk_string(x)	chk_character(x)
numeric	chk_number(x)	chk_numeric(x)
integer	chk_inumber(x)	chk_integer(x)
double	chk_dnumber(x)	chk_double(x)
integerish ¹	chk_znumber(x)	chk_integerish(x)
naturalish ²	chk_count(x)	chk_naturalish(x)
factor	³	chk_factor(x)
complex	³	chk_complex(x)
raw	³	chk_raw(x)
Date	chk_day(x)	chk_date(x)
POSIXct	chk_instant(x)	chk_posixct(x)

- ¹ integerish refers to functional integers (numbers that are very close to integer values), regardless of type (integer or double)
- ² naturalish refers to functional integers restricted to the natural numbers (zero and positive numbers)
- ³ No check functions are provided for scalar factor, complex, or raw

Usage

```
chk_flag(x, ..., na.ok = FALSE, null.ok = FALSE, attr.ok = "names")
```

```

chk_logical(
  x,
  ...,
  na.ok = TRUE,

```

```
    null.ok = FALSE,  
    attr.ok = "names",  
    length = NULL  
  )
```

```
chk_string(  
  x,  
  ...,  
  na.ok = FALSE,  
  null.ok = FALSE,  
  attr.ok = "names",  
  range = NULL  
)
```

```
chk_character(  
  x,  
  ...,  
  na.ok = TRUE,  
  null.ok = FALSE,  
  attr.ok = "names",  
  length = NULL,  
  range = NULL  
)
```

```
chk_number(  
  x,  
  ...,  
  na.ok = FALSE,  
  null.ok = FALSE,  
  attr.ok = "names",  
  range = NULL  
)
```

```
chk_numeric(  
  x,  
  ...,  
  na.ok = TRUE,  
  null.ok = FALSE,  
  attr.ok = "names",  
  length = NULL,  
  range = NULL  
)
```

```
chk_inumber(  
  x,  
  ...,  
  na.ok = FALSE,  
  null.ok = FALSE,
```

```
    attr.ok = "names",  
    range = NULL  
)
```

```
chk_integer(  
    x,  
    ...,  
    na.ok = TRUE,  
    null.ok = FALSE,  
    attr.ok = "names",  
    length = NULL,  
    range = NULL  
)
```

```
chk_dnumber(  
    x,  
    ...,  
    na.ok = FALSE,  
    null.ok = FALSE,  
    attr.ok = "names",  
    range = NULL  
)
```

```
chk_double(  
    x,  
    ...,  
    na.ok = TRUE,  
    null.ok = FALSE,  
    attr.ok = "names",  
    length = NULL,  
    range = NULL  
)
```

```
chk_znumber(  
    x,  
    ...,  
    na.ok = FALSE,  
    null.ok = FALSE,  
    attr.ok = "names",  
    range = NULL  
)
```

```
chk_integerish(  
    x,  
    ...,  
    na.ok = TRUE,  
    null.ok = FALSE,  
    attr.ok = "names",
```

```
    length = NULL,  
    range = NULL  
)
```

```
chk_count(  
    x,  
    ...,  
    na.ok = FALSE,  
    zero.ok = TRUE,  
    null.ok = FALSE,  
    attr.ok = "names"  
)
```

```
chk_naturalish(  
    x,  
    ...,  
    na.ok = TRUE,  
    zero.ok = TRUE,  
    null.ok = FALSE,  
    attr.ok = "names",  
    length = NULL,  
    range = NULL  
)
```

```
chk_factor(  
    x,  
    ...,  
    na.ok = TRUE,  
    null.ok = FALSE,  
    attr.ok = "names",  
    length = NULL  
)
```

```
chk_complex(  
    x,  
    ...,  
    na.ok = TRUE,  
    null.ok = FALSE,  
    attr.ok = "names",  
    length = NULL  
)
```

```
chk_raw(x, ..., null.ok = FALSE, attr.ok = "names", length = NULL)
```

```
chk_day(  
    x,  
    ...,  
    na.ok = FALSE,
```

```

    null.ok = FALSE,
    attr.ok = "names",
    range = NULL
)

chk_date(
  x,
  ...,
  na.ok = TRUE,
  null.ok = FALSE,
  attr.ok = "names",
  length = NULL,
  range = NULL
)

chk_instant(
  x,
  ...,
  na.ok = FALSE,
  null.ok = FALSE,
  attr.ok = "names",
  range = NULL
)

chk_posixct(
  x,
  ...,
  na.ok = TRUE,
  null.ok = FALSE,
  attr.ok = "names",
  length = NULL,
  range = NULL
)

chk_scalar(x, ..., na.ok = FALSE, null.ok = FALSE, attr.ok = "names")

chk_atomic(x, ..., na.ok = TRUE, attr.ok = "names", length = NULL)

```

Arguments

x	Object to check.
...	These dots are for future extensions and must be empty.
na.ok	Are missing values permitted?
null.ok	Is NULL permitted?
attr.ok	Which attributes x may carry beyond those intrinsic to its type: a character vector of permitted attribute names, FALSE for none at all, or TRUE for any.
length	Permitted length. NULL for any length, a scalar for one exact length, or a vector

	whose first and last elements give the minimum and the maximum. Neither may be negative, and NA at an end, or Inf as the maximum, means no bound there.
range	Permitted range of values, under the same first/last rule as length. For the character types it constrains nchar() of the elements instead, and for the date and time types the bounds are themselves Date or POSIXct. NA at an end means no bound there, and so does an infinite end wherever the type keeps that meaning.
zero.ok	Is zero permitted?

Value

The original object if the check passes.

See Also

[chk_composite](#) for lists and list-based objects, and [chk_other](#) for classes, conditions, and various other checks

Examples

```
# A check returns its input (invisibly), so it composes in a pipe
c(2, 4, 6) |> chk_numeric(length = 3) |> sum()

# One parameter set across the whole family
chk_string("abc", range = c(1, 3))
chk_integer(1:5, length = c(1, 10))

# On failure, the error names the argument as the caller wrote it
my_mean <- function(x) {
  chk_numeric(x)
  sum(x) / length(x)
}
tryCatch(my_mean("seven"), error = wrap_error)
```

chk_composite

Checks for lists and composite objects

Description

Checks for composite objects. See [chk_atomic](#) for scalar and vector types, and [chk_other](#) for various other checks.

Function	Passes when
chk_environment(x)	x is an environment
chk_list(x)	x is a list, and carries no class
chk_data_frame(x)	x is a data.frame of sound structure
chk_data_table(x)	x is also a data.table
chk_tibble(x)	x is also a tbl_df

Usage

```
chk_environment(x, ..., null.ok = FALSE, contains = character())  
  
chk_list(x, ..., null.ok = FALSE, attr.ok = "names", length = NULL)  
  
chk_data_frame(x, ..., null.ok = FALSE)  
  
chk_data_table(x, ..., null.ok = FALSE)  
  
chk_tibble(x, ..., null.ok = FALSE)
```

Arguments

x	Object to check.
...	These dots are for future extensions and must be empty.
null.ok	Is NULL permitted?
contains	Character vector of names that must be bound in the environment. Applies to <code>chk_environment()</code> .
attr.ok	Which attributes x may carry beyond those intrinsic to its type: a character vector of permitted attribute names, FALSE for none at all, or TRUE for any. Applies to <code>chk_list()</code> .
length	Permitted length. NULL for any length, a scalar for one exact length, or a vector whose first and last elements give the minimum and the maximum. Neither may be negative, and NA at an end, or Inf as the maximum, means no bound there.

Value

The original object if the check passes.

See Also

[chk_atomic](#) for the scalar and vector types, and [chk_other](#) for classes, conditions, and the [rlang](#) aliases.

Examples

```
chk_data_frame(mtcars)  
chk_list(list(a = 1, b = 2), length = 2)  
  
# chk_list() only accepts bare (unclassed) lists  
tryCatch(chk_list(mtcars), error = wrap_error)
```

chk_other

*Various other check functions***Description**

Various checks not directly related to atomic vectors (see [chk_atomic](#)), or composite objects (see [chk_composite](#)).

Function	Passes when
<code>chk_true(x)</code>	<code>x</code> is TRUE (implement arbitrary checks)
<code>chk_that(x, expr)</code>	<code>expr</code> , with <code>.</code> bound to <code>x</code> , is TRUE
<code>chk_class(x, classes)</code>	<code>x</code> inherits from every class in <code>classes</code>
<code>chk_match(x, values)</code>	<code>x</code> matches one of <code>values</code>
<code>chk_dots_empty()</code>	nothing was passed through <code>...</code>
<code>chk_any(...)</code>	at least one of the checks given passes

`chk_true()` is a catch-all function that can be used to implement arbitrary checks (by checking any expression that should be true).

`chk_that()` provides an alternative for checking that an expression is true, but separates the value to be checked (`x`) from the expression evaluated on it (`expr`), which makes an arbitrary condition usable on an object passing through a pipe.

`chk_class()` provides a quick way to check the class of an object.

`chk_match()` can be used either as an equivalent to `rlang::arg_match()` or `match.arg()` for checking a function argument against default values in a function, or to check that any (character) variable `x` is an element of a (character) vector of values. When used to select default value in a function, it must be called as `arg <- chk_match(arg)`.

`chk_dots_empty()` verifies that no arguments were passed to the `...` parameters, similarly to `rlang::check_dots_empty()`.

`chk_any(...)` can be used to combine multiple other checks, and will fail only if all constituent checks fail.

Usage

```
chk_true(x, ..., na.ok = FALSE)
```

```
chk_that(x, expr, ..., na.ok = FALSE, bindings = ".")
```

```
chk_class(x, classes, ..., null.ok = FALSE, ordered = FALSE)
```

```
chk_match(x, values = NULL, ..., multiple = FALSE)
```

```
chk_dots_empty()
```

```
chk_any(...)
```

Arguments

<code>x</code>	Object to check.
<code>...</code>	For <code>chk_any()</code> , the checks to try (see examples). For every other function here the dots must be empty.
<code>na.ok</code>	Are missing values permitted?
<code>expr</code>	Expression to evaluate on <code>x</code> , which is bound to <code>.</code> unless <code>bindings</code> says otherwise.
<code>bindings</code>	Character vector with name (or names) to bind <code>x</code> to when evaluating <code>expr</code> .
<code>classes</code>	Character vector of class names <code>x</code> must inherit from.
<code>null.ok</code>	Is <code>NULL</code> permitted?
<code>ordered</code>	Must <code>classes</code> appear in that order at the head of <code>class(x)</code> ?
<code>values</code>	Character vector of values which the object checked by <code>chk_match()</code> must be an element of.
<code>multiple</code>	Logical determining if the return value of <code>chk_match()</code> can contain more than one element.

Details

`chk_any()` evaluates its arguments in turn and returns the value of the first that passes. If none pass, it raises one error reporting every failure. It is how a composite requirement is written, where each individual `chk_*`() function states only one thing:

```
chk_any(chk_string(x), chk_number(x))
```

Only the checks `chk_any()` calls itself are candidates. One reached through a helper function, or from inside a lambda passed to `lapply()`, throws where it stands, and so does everything that is not a failed check: a misspelled function, an argument that does not exist, an object that was never bound.

The arguments are captured as expressions and evaluated in the calling environment, which rules out two ways of reaching `chk_any()` indirectly. `...` cannot be forwarded into it from another function, and an object cannot be piped into it. Both raise an error rather than being accommodated, since the first would evaluate the checks in the wrong scope and the second would return the piped object as a branch that passed. Write the checks at the call site, naming the object in each.

Value

The original object if the check passes. `chk_match()` returns the matched value visibly, `chk_dots_empty()` returns `NULL` invisibly, and `chk_any()` returns the value of the first argument that passes, which is the object that was checked.

See Also

[chk_atomic](#) for the scalar and vector types, and [chk_composite](#) for lists and composite objects.

Examples

```
# Any property that can be written as a condition
chk_true(nrow(mtcars) > 10)
mtcars |> chk_that(nrow(.) > 10) |> ncol()

# Match an argument against the values in its own default
plot_kind <- function(kind = c("scatter", "line", "bar")) chk_match(kind)
plot_kind()
tryCatch(plot_kind("pie"), error = wrap_error)

# chk_any() takes the first check that passes
x <- "a"
chk_any(chk_string(x), chk_number(x))
tryCatch(chk_any(chk_number(x), chk_logical(x)), error = wrap_error)
```

glue_vector

Glue interpolation vectors in pipes

Description

Applies `glue::glue()` to each element of a character vector using a template string, enabling pipe-friendly, element-wise interpolation. Useful when the vector to process is not encapsulated in a `data.frame` or other environment-like object.

The `glue::glue()` and `glue::glue_data()` functions are also re-exported for convenience

Usage

```
glue_vector(
  ``,
  template = "{.}",
  ...,
  .sep = "",
  .envir = parent.frame(),
  .open = "{",
  .close = "}",
  .na = "NA",
  .null = character(),
  .comment = "#",
  .literal = FALSE,
  .transformer = glue::identity_transformer,
  .trim = TRUE
)
```

Arguments

<code>.</code>	A character vector to be interpolated.
<code>template</code>	A glue template string. Use <code>{.}</code> to refer to the default (unnamed) vector variable, or the names of any other variables accessible in the relevant environment. Variables are recycled using tidyverse recycling rules.
<code>...</code>	Reserved and should not be used.
<code>.sep</code> , <code>.envir</code> , <code>.open</code> , <code>.close</code> , <code>.na</code> , <code>.null</code> , <code>.comment</code> , <code>.literal</code> , <code>.transformer</code> , <code>.trim</code>	Arguments passed on to <code>glue::glue()</code> . Must be passed by name. See <code>glue::glue()</code> for details.

Value

A character vector with interpolated values. The length is determined by tidyverse recycling rules for all referenced variables.

Examples

```
letters |> glue_vector("Letters include {.} and {LETTERS}")
```

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

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

Examples

```
if (getRversion() >= "4") {
  d <- data.frame(
    chr = letters[1:4],
    fct = factor(c("alpha", "bravo", "chrly", "delta")),
    lbl = ll_labelled(c(1, 2, 3, NA),
                      labels = c(one=1, two=2),
                      label = "A labelled vector")
  )
  dn <- notate(d)
  dn
  # View(dn)
}
```

yencode	<i>Yet (another urlencode compatible) encoding scheme</i>
---------	---

Description

Yet (another urlencode compatible) encoding scheme

Usage

```
yencode(string, escape = "%", whitelist = c("._~-", "][!$&'()*+,-:/?@#"))
```

```
yencoder(escape = "%", whitelist = c("._~-", "][!$&'()*+,-:/?@#"))
```

```
ydecode(string, escape = "%")
```

```
ydecoder(escape = "%")
```

Arguments

string	The string to process.
escape	The escape character to use.
whitelist	Any characters that should not be escaped. See details.

Details

Letters and digits are never escaped. Other characters are escaped unless they appear in `whitelist`, which may include multi-byte characters. The escape character is removed from the whitelist, with a warning, if present, and must itself be a single ASCII character.

`yencode()` escapes the UTF-8 representation of `string`, whatever its declared encoding, so the same text always gives the same result. `ydecode()` returns strings marked as UTF-8, and raises an error if an escape sequence is malformed or the decoded bytes are not valid UTF-8.

Value

The processed (encoded or decoded) string.

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 code won't break in unexpected ways when the input vector happens to be of length 1. The sample is taken by subsetting `x`, so the class and attributes of `x` are preserved.

If the goal is indeed to sample from an interval between 1 and `n`, use `sample(n)` or `sample.int(n)` (but make sure to only pass vectors of length one to those functions).

Usage

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

Arguments

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

Value

The resulting sample, of the same class as `x`

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.

Both arguments must be a single integerish value (an integer, or a double that is very close to one), and neither may be NA. Passing a vector of length other than one is an error.

Usage

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

Arguments

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

Value

An integer sequence ranging from from to to, or an empty integer vector if to equals from - 1.

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)

# Each bound must be a single whole number, so this errors as well
tryCatch(zeq(c(11,12),15), error=wrap_error)
```

zingle

Return the single (unique) value found in a vector

Description

`zingle()` returns the only value present in a vector. If the vector contains more than one distinct value, it throws an error. This is a guard for aggregations where all values within a group should be identical, but where you want that assumption checked rather than assumed. Only values are compared. Names are ignored and the result is unnamed, in line with other aggregation functions.

Usage

```
zingle(  
  x,  
  ...,  
  empty.ok = FALSE,  
  na.ok.partial = FALSE,  
  na.ok.all = FALSE,  
  nan.ok.all = FALSE  
)
```

Arguments

<code>x</code>	Vector of elements that should all be identical
<code>...</code>	Unused, reserved to force later arguments to be named
<code>empty.ok</code>	Is an empty vector ok?
<code>na.ok.partial</code>	Is a mix of NA and one distinct non-missing value ok?
<code>na.ok.all</code>	Is a vector of only NA values ok?
<code>nan.ok.all</code>	Is NaN ok as the returned value?

Value

The single element in the vector, unnamed. For `na.ok.partial` the non-missing value is returned.

Examples

```
zingle(c("Alpha", "Alpha", "Alpha"))  
zingle(c("Alpha", NA, "Alpha"), na.ok.partial = TRUE)  
zingle(c(NA, NA), na.ok.all = TRUE)  
zingle(c(NaN, NaN), nan.ok.all = TRUE)
```

Index

`asciify`, [2](#)

`chk_any` (`chk_other`), [10](#)
`chk_atomic`, [3](#), [8–11](#)
`chk_character` (`chk_atomic`), [3](#)
`chk_class` (`chk_other`), [10](#)
`chk_complex` (`chk_atomic`), [3](#)
`chk_composite`, [8](#), [8](#), [10](#), [11](#)
`chk_count` (`chk_atomic`), [3](#)
`chk_data_frame` (`chk_composite`), [8](#)
`chk_data_table` (`chk_composite`), [8](#)
`chk_date` (`chk_atomic`), [3](#)
`chk_day` (`chk_atomic`), [3](#)
`chk_dnumber` (`chk_atomic`), [3](#)
`chk_dots_empty` (`chk_other`), [10](#)
`chk_double` (`chk_atomic`), [3](#)
`chk_environment` (`chk_composite`), [8](#)
`chk_factor` (`chk_atomic`), [3](#)
`chk_flag` (`chk_atomic`), [3](#)
`chk_instant` (`chk_atomic`), [3](#)
`chk_integer` (`chk_atomic`), [3](#)
`chk_integerish` (`chk_atomic`), [3](#)
`chk_inumber` (`chk_atomic`), [3](#)
`chk_list` (`chk_composite`), [8](#)
`chk_logical` (`chk_atomic`), [3](#)
`chk_match` (`chk_other`), [10](#)
`chk_naturalish` (`chk_atomic`), [3](#)
`chk_number` (`chk_atomic`), [3](#)
`chk_numeric` (`chk_atomic`), [3](#)
`chk_other`, [8](#), [9](#), [10](#)
`chk_posixct` (`chk_atomic`), [3](#)
`chk_raw` (`chk_atomic`), [3](#)
`chk_scalar` (`chk_atomic`), [3](#)
`chk_string` (`chk_atomic`), [3](#)
`chk_that` (`chk_other`), [10](#)
`chk_tibble` (`chk_composite`), [8](#)
`chk_true` (`chk_other`), [10](#)
`chk_znumber` (`chk_atomic`), [3](#)

`glue::glue()`, [12](#), [13](#)

`glue::glue_data()`, [12](#)
`glue_vector`, [12](#)

`lookup`, [13](#)
`lookup()`, [13](#), [14](#)
`lookuper` (`lookup`), [13](#)
`lookuper()`, [13](#), [14](#)

`match.arg()`, [10](#)

`notate`, [15](#)

`rlang`, [9](#)
`rlang::arg_match()`, [10](#)
`rlang::check_dots_empty()`, [10](#)

`sample()`, [17](#)

`ydecode` (`yencode`), [16](#)
`ydecoder` (`yencode`), [16](#)
`yencode`, [16](#)
`yencoder` (`yencode`), [16](#)

`zample`, [17](#)
`zample()`, [17](#)
`zeq`, [18](#)
`zeq()`, [18](#)
`zingle`, [19](#)
`zingle()`, [19](#)