Package: scrutiny (via r-universe)

August 22, 2024

Title Error Detection in Science

Version 0.4.0

Maintainer Lukas Jung <jung-lukas@gmx.net>

Description Test published summary statistics for consistency (Brown and Heathers, 2017, <doi:10.1177/1948550616673876>; Allard, 2018, <https://aurelienallard.netlify.app/post/anaytic-grimmer-possibility-standard-deviations/>; Heathers and Brown, 2019, <https://osf.io/5vb3u/>). The package also provides infrastructure for implementing new error detection techniques.

License GPL (>= 3)

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.1

Imports cli, corrr, dplyr, ggplot2, ggrepel, glue, lifecycle, magrittr, methods, purrr, rlang (>= 1.0.2), stats, stringr, tibble, tidyr, tidyselect

Collate 'is-numeric-like.R' 'import-reexport.R' 'utils.R' 'mapper-function-helpers.R' 'audit-cols-minimal.R' 'audit.R' 'baseline-consistency-tests.R' 'before-inside-parens.R' 'function-factory-helpers.R' 'round-ceil-floor.R' 'round.R' 'reround.R' 'unround.R' 'sd-binary.R' 'decimal-places.R' 'debit-table.R' 'debit.R' 'grim.R' 'function-map.R' 'grimmer.R' 'grimmer-map.R' 'duplicate-detect.R' 'debit-map.R' 'restore-zeros.R' 'seq-decimal.R' 'manage-extra-cols.R' 'grim-map.R' 'data-doc.R' 'data-frame-predicates.R' 'seq-predicates.R' 'function-map-seq.R' 'debit-map-seq.R' 'disperse.R' 'function-map-total-n.R' 'debit-map-total-n.R' 'debit-plot.R' 'duplicate-count-colpair.R' 'duplicate-count.R' 'grim-granularity.R' 'grim-map-debug.R' 'grim-map-seq.R' 'grim-map-total-n.R' 'grim-plot.R' 'grim-stats.R'

'grimmer-map-seq.R' 'grimmer-map-total-n.R' 'metadata.R'
'method-audit-seq.R' 'method-audit-total-n.R'
'method-debit-map.R' 'method-detect.R'
'method-dup-count-colpair.R' 'method-dup-count.R'
'method-grim-map.R' 'method-grim-sequence.R'
'method-grimmer-map.R' 'method-tally.R' 'reround-to-fraction.R'
'reverse-map-seq.R' 'reverse-map-total-n.R'
'rivets-perfect-mean-sd.R' 'rivets-plot-cols.R'
'rivets-plot-lines.R' 'rivets-t-test.R' 'rivets_new.R'
'rounding-bias.R' 'row-to-colnames.R' 'scrutiny-package.R'
'seq-disperse.R' 'seq-length.R' 'split-by-parens.R'
'subset-superset.R' 'utils-pipe.R' 'utils-tidy-eval.R'
'write-doc-audit.R'

Suggests knitr, pkgload, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

Depends R (>= 3.4.0)

VignetteBuilder knitr

URL `https://lhdjung.github.io/scrutiny/`,
`https://github.com/lhdjung/scrutiny/`

BugReports `https://github.com/lhdjung/scrutiny/issues`

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

RemoteUrl https://github.com/lhdjung/scrutiny

RemoteRef HEAD

RemoteSha a9c4fbe1f571cfc351eae95652a70751b7bf5d13

Contents

`audit` *Summarize scrutiny objects*

Description

`audit()` summarizes the results of scrutiny functions like `grim_map()` that perform tests on data frames.

See below for a record of such functions. Go to the documentation of any of them to learn about its `audit()` method.

Usage

```
audit(data)
```

Arguments

`data`	A data frame that inherits one of the classes named below.

Details

`audit()` is an S3 generic. It looks up the (invisible) scrutiny class of a tibble returned by any function named below. You don't need to deal with the classes directly. Behind the scenes, they mediate between these functions and their associated summary statistics.

Value

A tibble (data frame) with test summary statistics.

Run before `audit()`

Function	**Class**
`grim_map()`	`"scr_grim_map"`
`grimmer_map()`	`"scr_grimmer_map"`
`debit_map()`	`"scr_debit_map"`
`duplicate_count()`	`"scr_dup_count"`
`duplicate_count_colpair()`	`"scr_dup_count_colpair"`
`duplicate_tally()`	`"scr_dup_tally"`
`duplicate_detect()`	`"scr_dup_detect"`
`audit_seq()`	`"scr_audit_seq"`
`audit_total_n()`	`"scr_audit_total_n"`

Examples

```
# For basic GRIM-testing:
pigs1 %>%
  grim_map() %>%
  audit()

# For duplicate detection:
pigs4 %>%
  duplicate_count() %>%
  audit()
```

`audit-special` *Summarize output of sequence mappers and total-n mappers*

Description

`audit_seq()` and `audit_total_n()` summarize the results of functions that end on `_seq` and `_total_n`, respectively.

See below for a record of such functions. Go to the documentation of any of them to learn about the way its output is processed by `audit_seq()` or `audit_total_n()`.

Usage

```
audit_seq(data)

audit_total_n(data)
```

Arguments

`data`	A data frame that inherits one of the classes named below.

Details

All functions named below that end on `_seq` were made by `function_map_seq()`. All that end on `_total_n` were made by `function_map_total_n()`.

Value

A tibble (data frame) with test summary statistics.

Before `audit_seq()`

Function	**Class**
`grim_map_seq()`	`"scr_grim_map_seq"`
`grimmer_map_seq()`	`"scr_grimmer_map_seq"`
`debit_map_seq()`	`"scr_debit_map_seq"`

Before `audit_total_n()`

Function	**Class**
`grim_map_total_n()`	`"scr_grim_map_total_n"`
`grimmer_map_total_n()`	`"scr_grimmer_map_total_n"`
`debit_map_total_n()`	`"scr_debit_map_total_n"`

Examples

```
# For GRIM-testing with dispersed inputs:
out <- pigs1 %>%
  grim_map_seq() %>%
  audit_seq()
out

# Follow up on `audit_seq()` or
# `audit_total_n()` with `audit()`:
audit(out)
```

`audit_cols_minimal` *Compute minimal* `audit()` *summaries*

Description

Call `audit_cols_minimal()` within your `audit()` methods for the output of consistency test mapper functions such as `grim_map()`. It will create a tibble with the three minimal, required columns:

1. `incons_cases` counts the inconsistent cases, i.e., the number of rows in the mapper's output where `"consistency"` is `FALSE`.
2. `all_cases` is the total number of rows in the mapper's output.
3. `incons_rate` is the ratio of `incons_cases` to `all_cases`.

You can still add other columns to this tibble. Either way, make sure to name your method correctly. See examples.

Usage

```
audit_cols_minimal(data, name_test)
```

Arguments

`data`	Data frame returned by a mapper function, such as `grim_map()`.
`name_test`	String (length 1). Short, plain-text name of the consistency test, such as `"GRIM"`. Only needed for a potential alert.

Value

A tibble (data frame) with the columns listed above.

See Also

For context, see `vignette("consistency-tests-in-depth")`. In case you don't call `audit_cols_minimal()`, you should call `check_audit_special()`.

Examples

```
# For a mapper function called `schlim_map()`
# that applies a test called SCHLIM and returns
# a data frame with the `"scr_schlim_map"` class:
audit.scr_schlim_map <- function(data) {
  audit_cols_minimal(data, name_test = "SCHLIM")
}

# If you like, add other summary columns
# with `dplyr::mutate()` or similar.
```

`check_audit_special` *Alert user if more specific* `audit_*()` *summaries are available*

Description

(Note: Ignore this function if your `audit()` method calls `audit_cols_minimal()`.)

Call `check_audit_special()` within an `audit()` method for a consistency test mapper function, such as `audit.scr_grim_map()`. It checks if the input data frame was the product of a function produced by `function_map_seq()` or `function_map_total_n()`.

If so, the function issues a gentle alert to the user that points to `audit_seq()` or `audit_total_n()`, respectively.

Usage

```
check_audit_special(data, name_test)
```

Arguments

`data`	The `audit()` method's input data frame.
`name_test`	String (length 1). Short, plain-text name of the consistency test, such as `"GRIM"`.

Value

No return value. Might print an alert.

See Also

`vignette("consistency-tests-in-depth")`, for context.

`check_mapper_input_colnames` *Check that a mapper's input has correct column names*

Description

When called within a consistency test mapper function, `check_mapper_input_colnames()` makes sure that the input data frame has correct column names:

- They include all the key columns corresponding to the test applied by the mapper.
- They don't already include `"consistency"`.

If either check fails, the function throws an informative error.

Usage

```
check_mapper_input_colnames(data, reported, name_test)
```

Arguments

`data`	Data frame. Input to the mapper function.
`reported`	String vector of the "key" column names that `data` must have, such as `c("x", "n")` for `grim_map()`.
`name_test`	String (length 1). Short, plain-text name of the consistency test that the mapper function applies, such as `"GRIM"`.

Value

No return value. Might throw an error.

See Also

`vignette("consistency-tests-in-depth")`, for context and the "key columns" terminology.

`data-frame-predicates` *Is an object a consistency test output tibble?*

Description

- `is_map_df()` tests whether an object is the output of a scrutiny-style mapper function for consistency tests, like `grim_map()`. These mapper functions also include those produced by `function_map()`, `function_map_seq()`, and `function_map_total_n()`.
- `is_map_basic_df()` is a variant of `is_map_df()` that tests whether an object is the output of a "basic" mapper function. This includes functions like `grim_map()` and those produced by `function_map()`, but not those produced by `function_map_seq()` or `function_map_total_n()`.
- `is_map_seq_df()` tests whether an object is the output of a function that was produced by `function_map_seq()`.
- `is_map_total_n_df()` tests whether an object is the output of a function that was produced by `function_map_total_n()`.

Usage

```
is_map_df(x)

is_map_basic_df(x)

is_map_seq_df(x)

is_map_total_n_df(x)
```

Arguments

x	Object to be tested.

Details

Sections 3, 6, and 7 of `vignette("consistency-tests-in-depth")` discuss which function factories produce which functions, and which of these new, factory-made functions return which kinds of tibbles.

These tibbles are what the `is_map_*()` functions test for. As an example, `function_map_seq()` produces `grim_map_seq()`, and this new function returns a tibble. `is_map_df()` and `is_map_seq_df()` return `TRUE` for this tibble, but `is_map_basic_df()` and `is_map_total_n_df()` return `FALSE`.

For an overview, see the table at the end of `vignette("consistency-tests-in-depth")`.

Value

Logical (length 1).

Examples

```
# Example test output:
df1 <- grim_map(pigs1)
df2 <- grim_map_seq(pigs1)
df3 <- grim_map_total_n(tibble::tribble(
  ~x1,    ~x2,   ~n,
  "3.43", "5.28", 90,
  "2.97", "4.42", 103
))

# All three tibbles are mapper output:
is_map_df(df1)
is_map_df(df2)
is_map_df(df3)

# However, only `df1` is the output of a
# basic mapper...
is_map_basic_df(df1)
is_map_basic_df(df2)
is_map_basic_df(df3)

# ...only `df2` is the output of a
# sequence mapper...
is_map_seq_df(df1)
is_map_seq_df(df2)
is_map_seq_df(df3)

# ...and only `df3` is the output of a
# total-n mapper:
is_map_total_n_df(df1)
is_map_total_n_df(df2)
is_map_total_n_df(df3)
```

---

debit *The DEBIT (descriptive binary) test*

Description

debit() tests summaries of binary data for consistency: If the mean and the standard deviation of binary data are given, are they consistent with the reported sample size?

The function is vectorized, but it is recommended to use debit_map() for testing multiple cases.

Usage

```
debit(
  x,
  sd,
  n,
  formula = "mean_n",
  rounding = "up_or_down",
  threshold = 5,
  symmetric = FALSE
)
```

Arguments

`x`	String. Mean of a binary distribution.
`sd`	String. Sample standard deviation of a binary distribution.
`n`	Integer. Total sample size.
`formula`	String. Formula used to compute the SD of the binary distribution. Currently, only the default, `"mean_n"`, is supported.
`rounding`	String. Rounding method or methods to be used for reconstructing the SD values to which `sd` will be compared. Default is `"up_or_down"` (from 5). See `vignette("rounding-options")`.
`threshold`	Integer. If `rounding` is set to `"up_from"`, `"down_from"`, or `"up_from_or_down_from"`, set `threshold` to the number from which the reconstructed values should then be rounded up or down. Otherwise irrelevant. Default is `5`.
`symmetric`	Logical. Set `symmetric` to `TRUE` if the rounding of negative numbers with `"up"`, `"down"`, `"up_from"`, or `"down_from"` should mirror that of positive numbers so that their absolute values are always equal. Default is `FALSE`.

Value

Logical. `TRUE` if `x`, `sd`, and `n` are mutually consistent, `FALSE` if not.

References

Heathers, James A. J., and Brown, Nicholas J. L. 2019. DEBIT: A Simple Consistency Test For Binary Data. https://osf.io/5vb3u/.

See Also

`debit_map()` applies `debit()` to any number of cases at once.

Examples

```
# Check single cases of binary
# summary data:
debit(x = "0.36", sd = "0.11", n = 20)
```

debit_map *Apply DEBIT to many cases*

Description

Call `debit_map()` to use DEBIT on multiple combinations of mean, standard deviation, and sample size of binary distributions. Mapping function for `debit()`.

For summary statistics, call `audit()` on the results.

Usage

```
debit_map(
  data,
  x = NULL,
  sd = NULL,
  n = NULL,
  rounding = "up_or_down",
  threshold = 5,
  symmetric = FALSE,
  show_rec = TRUE,
  extra = Inf
)
```

Arguments

`data`	Data frame.
`x`, `sd`, `n`	Optionally, specify these arguments as column names in `data`.
`rounding`, `threshold`, `symmetric`	Arguments passed on to `debit()`, with the same defaults.
`show_rec`	If set to `FALSE`, the resulting tibble only includes the columns `x`, `sd`, `n`, and `consistency`. Default is `TRUE`.
`extra`	Not currently used.

Value

A tibble with (at least) these columns –

- `x`, `sd`, `n`: the inputs.
- `consistency`: DEBIT consistency of `x`, `sd`, and `n`.

  By default, the tibble also includes the rounding method, boundary values, and information about the boundary values being inclusive or not. The tibble has the `scr_debit_map` class, which is recognized by the `audit()` generic.

Summaries with `audit()`

There is an S3 method for the `audit()` generic, so you can call `audit()` following `debit_map()`. It returns a tibble with these columns —

1. `incons_cases`: the number of DEBIT-inconsistent cases.
2. `all_cases`: the total number of cases.
3. `incons_rate`: the rate of inconsistent cases.
4. `mean_x`: the mean `x` (mean) value.
5. `mean_sd`: the mean `sd` value.
6. `distinct_n`: the number of distinct `n` values.

References

Heathers, James A. J., and Brown, Nicholas J. L. 2019. DEBIT: A Simple Consistency Test For Binary Data. https://osf.io/5vb3u/.

Examples

```
# Call `debit_map()` on binary summary
# data such as these:
pigs3

# The `consistency` column shows
# whether the values to its left
# are DEBIT-consistent:
pigs3 %>%
  debit_map()

# Get test summaries with `audit()`:
pigs3 %>%
  debit_map() %>%
  audit()
```

`debit_map_seq` *Using DEBIT with dispersed inputs*

Description

`debit_map_seq()` applies DEBIT with values surrounding the input values. This provides an easy and powerful way to assess whether small errors in computing or reporting may be responsible for DEBIT inconsistencies in published statistics.

Usage

```
debit_map_seq(
  data,
  x = NULL,
  sd = NULL,
  n = NULL,
  var = Inf,
  dispersion = 1:5,
  out_min = "auto",
  out_max = NULL,
  include_reported = FALSE,
  include_consistent = FALSE,
  ...
)
```

Arguments

data	A data frame that debit_map() could take.
x, sd, n	Optionally, specify column names in data as these arguments.
var	String. Names of the columns that will be dispersed. Default is c("x", "sd", "n").
dispersion	Numeric. Sequence with steps up and down from the var inputs. It will be adjusted to these values' decimal levels. For example, with a reported 8.34, the step size is 0.01. Default is 1:5, for five steps up and down.
out_min, out_max	If specified, output will be restricted so that it's not below out_min or above out_max. Defaults are "auto" for out_min, i.e., a minimum of one decimal unit above zero; and NULL for out_max, i.e., no maximum.
include_reported	Logical. Should the reported values themselves be included in the sequences originating from them? Default is FALSE because this might be redundant and bias the results.
include_consistent	Logical. Should the function also process consistent cases (from among those reported), not just inconsistent ones? Default is FALSE because the focus should be on clarifying inconsistencies.
...	Arguments passed down to debit_map().

Value

A tibble (data frame) with detailed test results.

Summaries with audit_seq()

You can call audit_seq() following debit_map_seq(). It will return a data frame with these columns:

- x, sd, and n are the original inputs, tested for consistency here.
- hits_total is the total number of DEBIT-consistent value sets found within the specified dispersion range.
- hits_x is the number of DEBIT-consistent value sets found by varying x.
- Accordingly with sd and hits_sd as well as n and hits_n.
- (Note that any consistent reported cases will be counted by the hits_* columns if both include_reported and include_consistent are set to TRUE.)
- diff_x reports the absolute difference between x and the next consistent dispersed value (in dispersion steps, not the actual numeric difference). diff_x_up and diff_x_down report the difference to the next higher or lower consistent value, respectively.
- diff_sd, diff_sd_up, and diff_sd_down do the same for sd.
- Likewise with diff_n, diff_n_up, and diff_n_down.

Call audit() following audit_seq() to summarize results even further. It's mostly self-explaining, but na_count and na_rate are the number and rate of times that a difference could not be computed because of a lack of corresponding hits within the dispersion range.

Examples

```
# `debit_map_seq()` can take any input
# that `debit_map()` can take:
pigs3

# Results from testing some few rows:
out <- pigs3 %>%
  dplyr::slice(3:4) %>%
  debit_map_seq()

out

# Case-wise summaries with `audit_seq()`
# can be more important than the raw results:
out %>%
  audit_seq()
```

`debit_map_total_n` *Use DEBIT with hypothetical group sizes*

Description

`debit_map_total_n()` extends DEBIT to cases where only group means and standard deviations (SDs) were reported, not group sizes.

The function is analogous to `grim_map_total_n()` and `grimmer_map_total_n()`, relying on the same infrastructure.

Usage

```
debit_map_total_n(
  data,
  x1 = NULL,
  x2 = NULL,
  sd1 = NULL,
  sd2 = NULL,
  dispersion = 0:5,
  n_min = 1L,
  n_max = NULL,
  constant = NULL,
  constant_index = NULL,
  ...
)
```

Arguments

`data`	Data frame with string columns `x1`, `x2`, `sd1`, and `sd2`, as well as numeric column `n`. The first two are reported group means. `sd1` and `sd2` are reported group SDs.

	n is the reported total sample size. It is not very important whether a value is in x1 or in x2 because, after the first round of tests, the function switches roles between x1 and x2, and reports the outcomes both ways. The same applies to sd1 and sd2. However, do make sure the x* and sd* values are paired accurately, as reported.
x1, x2, sd1, sd2	Optionally, specify these arguments as column names in data.
dispersion	Numeric. Steps up and down from half the n values. Default is 0:5, i.e., half n itself followed by five steps up and down.
n_min	Numeric. Minimal group size. Default is 1.
n_max	Numeric. Maximal group size. Default is NULL, i.e., no maximum.
constant	Optionally, add a length-2 vector or a list of length-2 vectors (such as a data frame with exactly two rows) to accompany the pairs of dispersed values. Default is NULL, i.e., no constant values.
constant_index	Integer (length 1). Index of constant or the first constant column in the output tibble. If NULL (the default), constant will go to the right of n_change.
...	Arguments passed down to debit_map().

Value

A tibble with these columns:

- x and sd, the group-wise reported input statistics, are repeated in row pairs.
- n is dispersed from half the input n, with n_change tracking the differences.
- both_consistent flags scenarios where both reported x and sd values are consistent with the hypothetical n values.
- case corresponds to the row numbers of the input data frame.
- dir is "forth" in the first half of rows and "back" in the second half. "forth" means that x2 and sd2 from the input are paired with the larger dispersed n, whereas "back" means that x1 and sd1 are paired with the larger dispersed n.
- Other columns from debit_map() are preserved.

Summaries with audit_total_n()

You can call audit_total_n() following up on debit_map_total_n() to get a tibble with summary statistics. It will have these columns:

- x1, x2, sd1, sd2, and n are the original inputs.
- hits_total is the number of scenarios in which all of x1, x2, sd1, and sd2 are DEBIT-consistent. It is the sum of hits_forth and hits_back below.
- hits_forth is the number of both-consistent cases that result from pairing x2 and sd2 with the larger dispersed n value.
- hits_back is the same, except x1 and sd1 are paired with the larger dispersed n value.
- scenarios_total is the total number of test scenarios, whether or not both x1 and sd1 as well as x2 and sd2 are DEBIT-consistent.
- hit_rate is the ratio of hits_total to scenarios_total.

Call audit() following audit_total_n() to summarize results even further.

References

Bauer, P. J., & Francis, G. (2021). Expression of Concern: Is It Light or Dark? Recalling Moral Behavior Changes Perception of Brightness. *Psychological Science*, 32(12), 2042–2043. https://journals.sagepub.com/doi/10.11

Heathers, J. A. J., & Brown, N. J. L. (2019). DEBIT: A Simple Consistency Test For Binary Data. https://osf.io/5vb3u/.

See Also

`function_map_total_n()`, which created the present function using `debit_map()`.

Examples

```
# Run `debit_map_total_n()` on data like these:
df <- tibble::tribble(
  ~x1,  ~x2,  ~sd1,  ~sd2,  ~n,
  "0.30", "0.28", "0.17", "0.10", 70,
  "0.41", "0.39", "0.09", "0.15", 65
)
df

debit_map_total_n(df)
```

`debit_plot` *Visualize DEBIT results*

Description

Plot a distribution of binary data and their mutual DEBIT consistency. Call this function only on a data frame that resulted from a call to `debit_map()`.

Various parameters of the individual geoms can be controlled via arguments.

Usage

```
debit_plot(
  data,
  show_outer_boxes = TRUE,
  show_labels = TRUE,
  show_full_scale = TRUE,
  show_theme_other = TRUE,
  color_cons = "royalblue1",
  color_incons = "red",
  line_alpha = 1,
  line_color = "black",
  line_linetype = 1,
  line_width = 0.5,
  line_size = 0.5,
  rect_alpha = 1,
  tile_alpha = 0.15,
  tile_height_offset = 0.025,
  tile_width_offset = 0.025,
  tile_height_min = 0.0375,
  tile_width_min = 0.0385,
  label_alpha = 0.5,
  label_linetype = 3,
  label_size = 3.5,
  label_linesize = 0.75,
  label_force = 175,
  label_force_pull = 0.75,
  label_padding = 0.5
)
```

Arguments

`data`	Data frame. Result of a call to `debit_map()`.
`show_outer_boxes`	Logical. Should outer tiles surround the actual data points, making it easier to spot them and to assess their overlap? Default is `TRUE`.
`show_labels`	Logical. Should the data points have labels (of the form "mean; SD")? Default is `TRUE`.
`show_full_scale`	Logical. Should the plot be fixed to full scale, showing the entire consistency line independently of the data? Default is `TRUE`.
`show_theme_other`	Logical. Should the theme be modified in a way fitting the plot structure? Default is `TRUE`.
`color_cons`, `color_incons`	Strings. Colors of the geoms representing consistent and inconsistent values, respectively.
`line_alpha`, `line_color`, `line_linetype`, `line_width`, `line_size`	Parameters of the curved DEBIT line.
`rect_alpha`	Parameter of the DEBIT rectangles. (Due to the nature of the data mapping, there can be no leeway regarding the shape or size of this particular geom.)
`tile_alpha`, `tile_height_offset`, `tile_width_offset`, `tile_height_min`, `tile_width_min`	Parameters of the outer tiles surrounding the DEBIT rectangles. Offset refers to the distance from the rectangles within.
`label_alpha`, `label_linetype`, `label_size`, `label_linesize`, `label_force`, `label_force_pull`, `label_padding`	Parameters of the labels showing mean and SD values. Passed on to `ggrepel::geom_text_repel()`; see there for more information.

Details

The labels are created via `ggrepel::geom_text_repel()`, so the algorithm is designed to minimize overlap with the tiles and other labels. Yet, they don't take the DEBIT line into account, and their locations are ultimately random. You might therefore have to resize the plot or run the function a few times until the labels are localized in a satisfactory way.

An alternative to the present function would be an S3 method for `ggplot2::autoplot()`. However, a standalone function such as this allows for customizing geom parameters and might perhaps provide better accessibility overall.

Value

A ggplot object.

References

Heathers, James A. J., and Brown, Nicholas J. L. 2019. DEBIT: A Simple Consistency Test For Binary Data. https://osf.io/5vb3u/.

Examples

```
# Run `debit_plot()` on the output
# of `debit_map()`:
pigs3 %>%
  debit_map() %>%
  debit_plot()
```

`decimal_places` *Count decimal places*

Description

`decimal_places()` counts the decimal places in a numeric vector, or in a string vector that can be coerced to numeric.

`decimal_places_scalar()` is much faster but only takes a single input. It is useful as a helper within other single-case functions.

Usage

```
decimal_places(x, sep = "\\.")

decimal_places_scalar(x, sep = "\\.")
```

Arguments

`x`	Numeric (or string that can be coerced to numeric). Object with decimal places to count.
`sep`	Substring that separates the mantissa from the integer part. Default is `"\\."`, which renders a decimal point.

Details

Decimal places in numeric values can't be counted accurately if the number has 15 or more characters in total, including the integer part and the decimal point. A possible solutions is to enter the number as a string to count all digits. (Converting to string is not sufficient – those numbers need to be *entered* in quotes.)

The functions ignore any whitespace at the end of a string, so they won't mistake spaces for decimal places.

Value

Integer. Number of decimal places in `x`.

Trailing zeros

If trailing zeros matter, don't convert numeric values to strings: In numeric values, any trailing zeros have already been dropped, and any information about them was lost (e.g., `3.70` returns `3.7`). Enter those values as strings instead, such as `"3.70"` instead of `3.70`. However, you can restore lost trailing zeros with `restore_zeros()` if the original number of decimal places is known.

If you need to enter many such values as strings, consider using `tibble::tribble()` and drawing quotation marks around all values in a `tribble()` column at once via RStudio's multiple cursors.

See Also

`decimal_places_df()`, which applies `decimal_places()` to all numeric-like columns in a data frame.

Examples

```
# `decimal_places()` works on both numeric values
# and strings...
decimal_places(x = 2.851)
decimal_places(x = "2.851")

# ... but trailing zeros are only counted within
# strings:
decimal_places(x = c(7.3900, "7.3900"))

# This doesn't apply to non-trailing zeros; these
# behave just like any other digit would:
decimal_places(x = c(4.08, "4.08"))

# Whitespace at the end of a string is not counted:
decimal_places(x = "6.0     ")

# `decimal_places_scalar()` is much faster,
# but only works with a single number or string:
decimal_places_scalar(x = 8.13)
decimal_places_scalar(x = "5.024")
```

`decimal_places_df` *Count decimal places in a data frame*

Description

For every value in a column, `decimal_places_df()` counts its decimal places. By default, it operates on all columns that are coercible to numeric.

Usage

```
decimal_places_df(
  data,
  cols = everything(),
  check_numeric_like = TRUE,
  sep = "\\."
)
```

Arguments

`data`	Data frame.
`cols`	Select columns from `data` using tidyselect. Default is `everything()`, but restricted by `check_numeric_like`.
`check_numeric_like`	Logical. If `TRUE` (the default), the function only operates on numeric columns and other columns coercible to numeric, as determined by `is_numeric_like()`.
`sep`	Substring that separates the mantissa from the integer part. Default is `"\\."`, which renders a decimal point.

Value

Data frame. The values of the selected columns are replaced by the numbers of their decimal places.

See Also

Wrapped functions: `decimal_places()`, `dplyr::across()`.

Examples

```
# Coerce all columns to string:
iris <- iris %>%
  tibble::as_tibble() %>%
  dplyr::mutate(across(everything(), as.character))

# The function will operate on all
# numeric-like columns but not on `"Species"`:
iris %>%
  decimal_places_df()
```

```
# Operate on some select columns only
# (from among the numeric-like columns):
iris %>%
  decimal_places_df(cols = starts_with("Sepal"))
```

disperse *Vary hypothetical group sizes*

Description

Some published studies only report a total sample size but no group sizes. However, group sizes are crucial for consistency tests such as GRIM. Call `disperse()` to generate possible group sizes that all add up to the total sample size, if that total is even.

`disperse2()` is a variant for odd totals. It takes two consecutive numbers and generates decreasing values from the lower as well as increasing values from the upper. In this way, all combinations still add up to the total.

`disperse_total()` directly takes the total sample size, checks if it's even or odd, splits it up accordingly, and applies `disperse()` or `disperse2()`, respectively.

These functions are primarily intended as helpers. They form the backbone of `grim_map_total_n()` and all other functions created with `function_map_total_n()`.

Usage

```
disperse(
  n,
  dispersion = 0:5,
  n_min = 1L,
  n_max = NULL,
  constant = NULL,
  constant_index = NULL
)

disperse2(
  n,
  dispersion = 0:5,
  n_min = 1L,
  n_max = NULL,
  constant = NULL,
  constant_index = NULL
)

disperse_total(
  n,
  dispersion = 0:5,
  n_min = 1L,
  n_max = NULL,
  constant = NULL,
  constant_index = NULL
)
```

Arguments

`n`	Numeric: • In `disperse()`, single number from which to go up and down. This should be half of an even total sample size. • In `disperse2()`, the two consecutive numbers closest to half of an odd total sample size (e.g., `c(25, 26)` for a total of 51). • In `disperse_total()`, the total sample size.
`dispersion`	Numeric. Vector that determines the steps up and down from `n` (or, in `disperse_total()`, from half `n`). Default is `0:5`.
`n_min`	Numeric. Minimal group size. Default is `1L`.
`n_max`	Numeric. Maximal group size. Default is `NULL`, i.e., no maximum.
`constant`	Optionally, add a length-2 vector or a list of length-2 vectors (such as a data frame with exactly two rows) to accompany the pairs of dispersed values. Default is `NULL`, i.e., no constant values.
`constant_index`	Integer (length 1). Index of `constant` or the first `constant` column in the output tibble. If `NULL` (the default), `constant` will go to the right of `n_change`.

Details

If any group size is less than `n_min` or greater than `n_max`, it is removed. The complementary size of the other group is also removed.

`constant` values are pairwise repeated. That is why `constant` must be a length-2 atomic vector or a list of such vectors. If `constant` is a data frame or some other named list, the resulting columns will have the same names as the list-element names. If the list is not named, the new column names will be `"constant1"`, `"constant2"`, etc; or just `"constant"`, for a single pair.

Value

A tibble (data frame) with these columns:

- `n` includes the dispersed `n` values. Every pair of consecutive rows has `n` values that each add up to the total.
- `n_change` records how the input `n` was transformed to the output `n`. In `disperse2()`, the `n_change` strings label the lower of the input `n` values `n1` and the higher one `n2`.

References

Bauer, P. J., & Francis, G. (2021). Expression of Concern: Is It Light or Dark? Recalling Moral Behavior Changes Perception of Brightness. *Psychological Science*, 32(12), 2042–2043. https://journals.sagepub.com/doi/10.11

See Also

function_map_total_n(), grim_map_total_n(), and seq_distance_df().

Examples

```
# For a total sample size of 40,
# set `n` to `20`:
disperse(n = 20)

# Specify `dispersion` to control
# the steps up and down from `n`:
disperse(n = 20, dispersion = c(3, 6, 10))

# In `disperse2()`, specify `n` as two
# consecutive numbers -- i.e., group sizes:
disperse2(n = c(25, 26))

# Use the total sample size directly
# with `disperse_total()`. An even total
# internally triggers `disperse()`...
disperse_total(n = 40)

# ...whereas an odd total triggers `disperse2()`:
disperse_total(n = 51)

# You may add values that repeat along with the
# dispersed ones but remain constant themselves.
# Such values can be stored in a length-2 vector
# for a single column...
disperse_total(37, constant = c("5.24", "3.80"))

# ... or a list of length-2 vectors for multiple
# columns. This includes data frames with 2 rows:
df_constant <- tibble::tibble(
  name = c("Paul", "Mathilda"), age = 27:28,
  registered = c(TRUE, FALSE)
)
disperse_total(37, constant = df_constant)
```

---

duplicate_count *Count duplicate values*

Description

duplicate_count() returns a frequency table. When searching a data frame, it includes values from all columns for each frequency count.

This function is a blunt tool designed for initial data checking. It is not too informative if many values have few characters each.

For summary statistics, call audit() on the results.

Usage

```
duplicate_count(
  x,
  ignore = NULL,
  locations_type = c("character", "list"),
  numeric_only = deprecated()
)
```

Arguments

`x`	Vector or data frame.
`ignore`	Optionally, a vector of values that should not be counted.
`locations_type`	String. One of `"character"` or `"list"`. With `"list"`, each `locations` value is a vector of column names, which is better for further programming. By default (`"character"`), the column names are pasted into a string, which is more readable.
`numeric_only`	[Deprecated] No longer used: All values are coerced to character.

Details

Don't use `numeric_only`. It no longer has any effect and will be removed in the future. The only reason for this argument was the risk of errors introduced by coercing values to numeric. This is no longer an issue because all values are now coerced to character, which is more appropriate for checking reported statistics.

Value

If `x` is a data frame or another named vector, a tibble with four columns. If `x` isn't named, only the first two columns appear:

- `value`: All the values from `x`.
- `frequency`: Absolute frequency of each value in `x`, in descending order.
- `locations`: Names of all columns from `x` in which `value` appears.
- `locations_n`: Number of columns named in `locations`.

The tibble has the `scr_dup_count` class, which is recognized by the `audit()` generic.

Summaries with `audit()`

There is an S3 method for the `audit()` generic, so you can call `audit()` following `duplicate_count()`. It returns a tibble with summary statistics for the two numeric columns, `frequency` and `locations_n` (or, if `x` isn't named, only for `frequency`).

See Also

- `duplicate_count_colpair()` to check each combination of columns for duplicates.
- `duplicate_tally()` to show instances of a value next to each instance.
- `janitor::get_dupes()` to search for duplicate rows.

Examples

```
# Count duplicate values...
iris %>%
  duplicate_count()

# ...and compute summaries:
iris %>%
  duplicate_count() %>%
  audit()

# Any values can be ignored:
iris %>%
  duplicate_count(ignore = c("setosa", "versicolor", "virginica"))
```

---

duplicate_count_colpair

Count duplicate values by column

Description

`duplicate_count_colpair()` takes a data frame and checks each combination of columns for duplicates. Results are presented in a tibble, ordered by the number of duplicates.

Usage

```
duplicate_count_colpair(
  data,
  ignore = NULL,
  show_rates = TRUE,
  na.rm = deprecated()
)
```

Arguments

`data`	Data frame.
`ignore`	Optionally, a vector of values that should not be checked for duplicates.
`show_rates`	Logical. If `TRUE` (the default), adds columns `rate_x` and `rate_y`. See value section. Set `show_rates` to `FALSE` for higher performance.
`na.rm`	[Deprecated] Missing values are never counted in any case.

Value

A tibble (data frame) with these columns –

- `x` and `y`: Each line contains a unique combination of `data`'s columns, stored in the `x` and `y` output columns.
- `count`: Number of "duplicates", i.e., values that are present in both `x` and `y`.

- total_x, total_y, rate_x, and rate_y (added by default): total_x is the number of non-missing values in the column named under x. Also, rate_x is the proportion of x values that are duplicated in y, i.e., count / total_x. Likewise with total_y and rate_y. The two rate_* columns will be equal unless NA values are present.

Summaries with audit()

There is an S3 method for audit(), so you can call audit() following duplicate_count_colpair(). It returns a tibble with summary statistics.

See Also

- duplicate_count() for a frequency table.
- duplicate_tally() to show instances of a value next to each instance.
- janitor::get_dupes() to search for duplicate rows.
- corrr::colpair_map(), a versatile tool for pairwise column analysis which the present function wraps.

Examples

```
# Basic usage:
mtcars %>%
  duplicate_count_colpair()

# Summaries with `audit()`:
mtcars %>%
  duplicate_count_colpair() %>%
  audit()
```

---

duplicate_detect *Detect duplicate values*

Description

[Superseded]

duplicate_detect() is superseded because it's less informative than duplicate_tally() and duplicate_count(). Use these functions instead.

For every value in a vector or data frame, duplicate_detect() tests whether there is at least one identical value. Test results are presented next to every value.

This function is a blunt tool designed for initial data checking. Don't put too much weight on its results.

For summary statistics, call audit() on the results.

Usage

```
duplicate_detect(x, ignore = NULL, colname_end = "dup", numeric_only)
```

Arguments

`x`	Vector or data frame.
`ignore`	Optionally, a vector of values that should not be checked. In the test result columns, they will be marked `NA`.
`colname_end`	String. Name ending of the logical test result columns. Default is `"dup"`.
`numeric_only`	[Deprecated] No longer used: All values are coerced to character.

Details

This function is not very informative with many input values that only have a few characters each. Many of them may have duplicates just by chance. For example, in R's built-in `iris` data set, 99% of values have duplicates.

In general, the fewer values and the more characters per value, the more significant the results.

Value

A tibble (data frame). It has all the columns from `x`, and to each of these columns' right, the corresponding test result column.

The tibble has the `scr_dup_detect` class, which is recognized by the `audit()` generic.

Summaries with `audit()`

There is an S3 method for the `audit()` generic, so you can call `audit()` following `duplicate_detect()`. It returns a tibble with these columns —

- `term`: The original data frame's variables.
- `dup_count`: Number of "duplicated" values of that `term` variable: those which have at least one duplicate anywhere in the data frame.
- `total`: Number of all non-`NA` values of that `term` variable.
- `dup_rate`: Rate of "duplicated" values of that `term` variable.

The final row, `.total`, summarizes across all other rows: It adds up the `dup_count` and `total_count` columns, and calculates the mean of the `dup_rate` column.

See Also

- `duplicate_tally()` to count instances of a value instead of just stating whether it is duplicated.
- `duplicate_count()` for a frequency table.
- `duplicate_count_colpair()` to check each combination of columns for duplicates.
- `janitor::get_dupes()` to search for duplicate rows.

Examples

```
# Find duplicate values in a data frame...
duplicate_detect(x = pigs4)

# ...or in a single vector:
duplicate_detect(x = pigs4$snout)

# Summary statistics with `audit()`:
pigs4 %>%
  duplicate_detect() %>%
  audit()

# Any values can be ignored:
pigs4 %>%
  duplicate_detect(ignore = c(8.131, 7.574))
```

`duplicate_tally` *Count duplicates at each observation*

Description

For every value in a vector or data frame, `duplicate_tally()` counts how often it appears in total. Tallies are presented next to each value.

For summary statistics, call `audit()` on the results.

Usage

```
duplicate_tally(x, ignore = NULL, colname_end = "n")
```

Arguments

`x`	Vector or data frame.
`ignore`	Optionally, a vector of values that should not be checked. In the test result columns, they will be marked `NA`.
`colname_end`	String. Name ending of the logical test result columns. Default is `"n"`.

Details

This function is not very informative with many input values that only have a few characters each. Many of them may have duplicates just by chance. For example, in R's built-in `iris` data set, 99% of values have duplicates.

In general, the fewer values and the more characters per value, the more significant the results.

Value

A tibble (data frame). It has all the columns from `x`, and to each of these columns' right, the corresponding tally column.

The tibble has the `scr_dup_detect` class, which is recognized by the `audit()` generic.

Summaries with audit()

There is an S3 method for the audit() generic, so you can call audit() following duplicate_tally(). It returns a tibble with summary statistics.

See Also

- duplicate_count() for a frequency table.
- duplicate_count_colpair() to check each combination of columns for duplicates.
- janitor::get_dupes() to search for duplicate rows.

Examples

```
# Tally duplicate values in a data frame...
duplicate_tally(x = pigs4)

# ...or in a single vector:
duplicate_tally(x = pigs4$snout)

# Summary statistics with `audit()`:
pigs4 %>%
  duplicate_tally() %>%
  audit()

# Any values can be ignored:
pigs4 %>%
  duplicate_tally(ignore = c(8.131, 7.574))
```

---

fractional-rounding   *Generalized rounding to the nearest fraction of a specified denominator*

Description

Two functions that round numbers to specific fractions, not just to the next higher decimal level. They are inspired by janitor::round_to_fraction() but feature all the options of reround():

- reround_to_fraction() closely follows janitor::round_to_fraction() by first rounding to fractions of a whole number, then optionally rounding the result to a specific number of digits in the usual way.
- reround_to_fraction_level() rounds to the nearest fraction of a number at the specific decimal level (i.e., number of digits), without subsequent rounding. This is closer to conventional rounding functions.

Usage

```
reround_to_fraction(
  x = NULL,
  denominator = 1,
  digits = Inf,
  rounding = "up_or_down",
  threshold = 5,
  symmetric = FALSE
)

reround_to_fraction_level(
  x = NULL,
  denominator = 1,
  digits = 0L,
  rounding = "up_or_down",
  threshold = 5,
  symmetric = FALSE
)
```

Arguments

`x`	Numeric. Vector of numbers to be rounded.
`denominator`	Numeric (>= 1) . `x` will be rounded to the nearest fraction of `denominator`. Default is `1`.
`digits`	Numeric (whole numbers). • In `reround_to_fraction()`: If `digits` is specified, the values resulting from fractional rounding will subsequently be rounded to that many decimal places. If set to `"auto"`, it internally becomes `ceiling(log10(denominator)) + 1`, as in `janitor::round_to_fraction()`. Default is `Inf`, in which case there is no subsequent rounding. • In `reround_to_fraction_level()`: This function will round to a fraction of the number at the decimal level specified by `digits`. Default is `0`.
`rounding`, `threshold`, `symmetric`	More arguments passed down to `reround()`.

Value

Numeric vector of the same length as `x` unless `rounding` is either of `"up_or_down"`, `"up_from_or_down_from"`, and `"ceiling_or_floor"`. In these cases, it will always have length 2.

See Also

`reround()`, which the functions wrap, and `janitor::round_to_fraction()`, part of which they copy.

Examples

```
#`reround_to_fraction()` rounds `0.4`
# to `0` if `denominator` is `1`, which
# is the usual integer rounding...
reround_to_fraction(0.4, denominator = 1, rounding = "even")

# ...but if `denominator` is `2`, it rounds to the nearest
# fraction of 2, which is `0.5`:
reround_to_fraction(0.4, denominator = 2, rounding = "even")

# Likewise with fractions of 3:
reround_to_fraction(0.25, denominator = 3, rounding = "even")

# The default for `rounding` is to round
# both up and down, as in `reround()`:
reround_to_fraction(0.4, denominator = 2)

# These two rounding procedures differ
# at the tie points:
reround_to_fraction(0.25, denominator = 2)

# `reround_to_fraction_level()`, in contrast,
# uses `digits` to determine some decimal level,
# and then rounds to the closest fraction at
# that level:
reround_to_fraction_level(0.12345, denominator = 2, digits = 0)
reround_to_fraction_level(0.12345, denominator = 2, digits = 1)
reround_to_fraction_level(0.12345, denominator = 2, digits = 2)
```

`function_map` *Create new* `*_map()` *functions*

Description

`function_map()` creates new basic mapper functions for consistency tests, such as `grim_map()` or `debit_map()`.

For context, see *Creating basic mappers with* `function_map()`.

Usage

```
function_map(
  .fun,
  .reported,
  .name_test,
  .name_key_result = "consistency",
  .name_class = NULL,
  .args_disabled = NULL,
  .col_names = NULL,
  .col_control = NULL,
  .col_filler = NULL,
  ...
)
```

Arguments

`.fun`	Single-case consistency testing function that will be applied to each row in a data frame. The function must return a single logical value, i.e., `TRUE`, `FALSE`, or `NA`.
`.reported`	String. Names of the columns to be tested.
`.name_test`	String (length 1). Plain-text name of the consistency test, such as `"GRIM"`.
`.name_key_result`	(Experimental) Optionally, a single string that will be the name of the key result column in the output. Default is `"consistency"`.
`.name_class`	String. Optionally, one or more classes to be added to the output data frame. Default is `NULL`, i.e., no extra class (but see *Details*).
`.args_disabled`	Optionally, a string vector with names of arguments of the `*_scalar()` function that don't work with the factory-made function. If the user tries to specify these arguments, an informative error will be thrown.
`.col_names`	(Experimental) Optionally, a string vector with the names of additional columns that are derived from the `*_scalar()` function. Requires `.col_control` and `.col_filler` specifications.
`.col_control`	(Experimental) Optionally, a single string with the name of the `*_scalar()` function's logical argument that controls if the columns named in `.col_names` will be displayed.
`.col_filler`	(Experimental) Optionally, a vector specifying the values of `.col_names` columns in rows where the `*_scalar()` function only returned the `consistency` value.
`...`	These dots must be empty.

Details

The output tibble returned by the factory-made function will inherit one or two classes independently of the `.name_class` argument:

- It will inherit a class named `"scr_{tolower(.name_test)}_map"`; for example, the class is `"scr_grim_map"` if `.name_test` is `"GRIM"`.
- If a `rounding` argument is specified via `...`, or else if `.fun` has a `rounding` argument with a default, the output tibble will inherit a class named `"scr_rounding_{rounding}"`; for example, `"scr_rounding_up_or_down"`.

Value

A factory-made function with these arguments:

- `data`: Data frame with all the columns named in `.reported`. It must have columns named after the key arguments in `.fun`. Other columns are permitted.

- Arguments named after the `.reported` values. They can be specified as the names of `data` columns so that the function will rename that column using the `.reported` name.
- `reported`, `fun`, `name_class`: Same as when calling `function_map()` but spelled without dots. You can override these defaults when calling the factory-made function.
- `...`: Arguments passed down to `.fun`. This does not include the column-identifying arguments derived from `.reported`.

Value returned by the factory-made function

A tibble that includes `"consistency"`: a logical column showing whether the values to its left are mutually consistent (`TRUE`) or not (`FALSE`).

Examples

```
# Basic test implementation for "SCHLIM",
# a mock test with no real significance:
schlim_scalar <- function(y, n) {
  (y / 3) > n
}

# Let the function factory produce
# a mapper function for SCHLIM:
schlim_map <- function_map(
  .fun = schlim_scalar,
  .reported = c("y", "n"),
  .name_test = "SCHLIM"
)

# Example data:
df1 <- tibble::tibble(y = 16:25, n = 3:12)

# Call the "factory-made" function:
schlim_map(df1)
```

`function_map_seq` *Create new* `*_map_seq()` *functions*

Description

`function_map_seq()` is the engine that powers functions such as `grim_map_seq()`. It creates new, "factory-made" functions that apply consistency tests such as GRIM or GRIMMER to sequences of specified variables. The sequences are centered around the reported values of those variables.

By default, only inconsistent values are dispersed from and tested. This provides an easy and powerful way to assess whether small errors in computing or reporting may be responsible for inconsistencies in published statistics.

For background and more examples, see the sequence mapper section of *Consistency tests in depth*.

Usage

```
function_map_seq(
  .fun,
  .var = Inf,
  .reported,
  .name_test,
  .name_key_result = "consistency",
  .name_class = NULL,
  .args_disabled = NULL,
  .dispersion = 1:5,
  .out_min = "auto",
  .out_max = NULL,
  .include_reported = FALSE,
  .include_consistent = FALSE,
  ...
)
```

Arguments

`.fun`	Function such as `grim_map()`, or one made by `function_map()`: It will be used to test columns in a data frame for consistency. Test results are logical and need to be contained in a column called `"consistency"` that is added to the input data frame. This modified data frame is then returned by `.fun`.
`.var`	String. Variables that will be dispersed by the manufactured function. Defaults to `.reported`.
`.reported`	String. All variables the manufactured function can disperse in principle.
`.name_test`	String (length 1). The name of the consistency test, such as `"GRIM"`, to be optionally shown in a message when using the manufactured function.
`.name_key_result`	(Experimental) Optionally, a single string that will be the name of the key result column in the output. Default is `"consistency"`.
`.name_class`	String. If specified, the tibbles returned by the manufactured function will inherit this string as an S3 class. Default is `NULL`, i.e., no extra class.
`.args_disabled`	String. Optionally, names of the basic `*_map()` function's arguments. These arguments will throw an error if specified when calling the factory-made function.
`.dispersion`	Numeric. Sequence with steps up and down from the reported values. It will be adjusted to these values' decimal level. For example, with a reported `8.34`, the step size is `0.01`. Default is `1:5`, for five steps up and down.
`.out_min, .out_max`	If specified when calling a factory-made function, output will be restricted so that it's not below `.out_min` or above `.out_max`. Defaults are `"auto"` for `.out_min`, i.e., a minimum of one decimal unit above zero; and `NULL` for `.out_max`, i.e., no maximum.
`.include_reported`	Logical. Should the reported values themselves be included in the sequences originating from them? Default is `FALSE` because this might be redundant and bias the results.

`.include_consistent`
: Logical. Should the function also process consistent cases (from among those reported), not just inconsistent ones? Default is `FALSE` because the focus should be on clarifying inconsistencies.

`...`
: These dots must be empty.

Details

All arguments of `function_map_seq()` set the defaults for the arguments in the manufactured function. They can still be specified differently when calling the latter.

If functions created this way are exported from other packages, they should be written as if they were created with purrr adverbs; see explanations there, and examples in the export section of *Consistency tests in depth*.

This function is a so-called function factory: It produces other functions, such as `grim_map_seq()`. More specifically, it is a function operator because it also takes functions as inputs, such as `grim_map()`. See Wickham (2019, ch. 10-11).

Value

A function such as those below. ("Testable statistics" are variables that can be selected via `var`, and are then varied. All variables except for those in parentheses are selected by default.)

Manufactured function	**Testable statistics**	**Test vignette**
`grim_map_seq()`	`"x", "n", ("items")`	`vignette("grim")`
`grimmer_map_seq()`	`"x", "sd", "n", ("items")`	`vignette("grimmer")`
`debit_map_seq()`	`"x", "sd", "n"`	`vignette("debit")`

The factory-made function will also have dots, `...`, to pass arguments down to `.fun`, i.e., the basic mapper function such as `grim_map()`.

Conventions

The name of a function returned by `function_map_seq()` should mechanically follow from that of the input function. For example, `grim_map_seq()` derives from `grim_map()`. This pattern fits best if the input function itself is named after the test it performs on a data frame, followed by `_map`: `grim_map()` applies GRIM, `grimmer_map()` applies GRIMMER, etc.

Much the same is true for the classes of data frames returned by the manufactured function via the `.name_class` argument of `function_map_seq()`. It should be the function's own name preceded by the name of the package that contains it, or by an acronym of that package's name. Therefore, some existing classes are `scr_grim_map_seq` and `scr_grimmer_map_seq`.

References

Wickham, H. (2019). *Advanced R* (Second Edition). CRC Press/Taylor and Francis Group. https://adv-r.hadley.nz/index.html

Examples

```
# Function definition of `grim_map_seq()`:
grim_map_seq <- function_map_seq(
  .fun = grim_map,
  .reported = c("x", "n"),
  .name_test = "GRIM",
)
```

`function_map_total_n` *Create new* `*_map_total_n()` *functions*

Description

`function_map_total_n()` is the engine that powers functions such as `grim_map_total_n()`. It creates new, "factory-made" functions for consistency tests such as GRIM or GRIMMER. The new functions take reported summary statistics (e.g., means) and apply those tests in cases where only a total sample size is known, not group sizes.

This works by making `disperse_total()` create multiple pairs of hypothetical group sizes, all of which add up to the reported total. There need to be exactly two groups.

For background and more examples, see the total-n mapper section of *Consistency tests in depth*.

Usage

```
function_map_total_n(
  .fun,
  .reported,
  .name_test,
  .name_key_result = "consistency",
  .name_class = NULL,
  .dispersion = 0:5,
  .n_min = 1L,
  .n_max = NULL,
  .constant = NULL,
  .constant_index = NULL,
  ...
)
```

Arguments

`.fun`	Function such as `grim_map()`, or one made by `function_map()`: It will be used to test columns in a data frame for consistency. Test results are logical and need to be contained in a column called `consistency` that is added to the input data frame. This modified data frame is then returned by `.fun`.
`.reported`	String. Names of the columns containing group-specific statistics that were reported alongside the total sample size(s). They will be tested for consistency with the hypothetical group sizes. Examples are `"x"` for GRIM and `c("x",`

"sd") for DEBIT. In the data frame with reported group statistics that the manufactured function takes as an input, each will need to fan out like "x1", "x2", "sd1", and "sd2".

`.name_test`	String (length 1). The name of the consistency test, such as "GRIM", to be optionally shown in a message when using the manufactured function.
`.name_key_result`	(Experimental) Optionally, a single string that will be the name of the key result column in the output. Default is "consistency".
`.name_class`	String. If specified, the tibbles returned by the manufactured function will inherit this string as an S3 class. Default is `NULL`, i.e., no extra class.
`.dispersion, .n_min, .n_max, .constant, .constant_index`	Arguments passed down to `disperse_total()`, using defaults from there.
`...`	These dots must be empty.

Details

If functions created by `function_map_total_n()` are exported from other packages, they should be written as if they were created with purrr adverbs; see explanations there, and examples in the export section of *Consistency tests in depth*.

This function is a so-called function factory: It produces other functions, such as `grim_map_total_n()`. More specifically, it is a function operator because it also takes functions as inputs, such as `grim_map()`. See Wickham (2019), ch. 10-11.

Value

A function such as these:

Manufactured function	**Reported statistics**	**Test vignette**
`grim_map_total_n()`	"x"	`vignette("grim")`
`grimmer_map_total_n()`	"x", "sd"	`vignette("grimmer")`
`debit_map_total_n()`	"x", "sd"	`vignette("debit")`

The factory-made function will also have dots, `...`, to pass arguments down to `.fun`, i.e., the basic mapper function.

Conventions

The name of a function returned by `function_map_total_n()` should mechanically follow from that of the input function. For example, `grim_map_total_n()` derives from `grim_map()`. This pattern fits best if the input function itself is named after the test it performs on a data frame, followed by _map: `grim_map()` applies GRIM, `grimmer_map()` applies GRIMMER, etc.

Much the same is true for the classes of data frames returned by the manufactured function via the `.name_class` argument of `function_map_total_n()`. It should be the function's own name preceded by the name of the package that contains it, or by an acronym of that package's name. Therefore, some existing classes are `scr_grim_map_total_n` and `scr_grimmer_map_total_n`.

References

Bauer, P. J., & Francis, G. (2021). Expression of Concern: Is It Light or Dark? Recalling Moral Behavior Changes Perception of Brightness. *Psychological Science*, 32(12), 2042–2043. https://journals.sagepub.com/doi/10.11

Wickham, H. (2019). *Advanced R* (Second Edition). CRC Press/Taylor and Francis Group. https://adv-r.hadley.nz/index.html

See Also

function_map_seq()

Examples

```
# Function definition of `grim_map_total_n()`:
grim_map_total_n <- function_map_total_n(
  .fun = grim_map,
  .reported = "x",
  .name_test = "GRIM"
)
```

grim *The GRIM test (granularity-related inconsistency of means)*

Description

grim() checks if a reported mean value of integer data is mathematically consistent with the reported sample size and the number of items that compose the mean value.

Set percent to TRUE if x is a percentage. This will convert x to a decimal number and adjust the decimal count accordingly.

The function is vectorized, but it is recommended to use grim_map() for testing multiple cases.

Usage

```
grim(
  x,
  n,
  items = 1,
  percent = FALSE,
  show_rec = FALSE,
  rounding = "up_or_down",
  threshold = 5,
  symmetric = FALSE,
  tolerance = .Machine$double.eps^0.5
)
```

Arguments

`x`	String. The reported mean or percentage value.
`n`	Integer. The reported sample size.
`items`	Numeric. The number of items composing `x`. Default is 1, the most common case.
`percent`	Logical. Set `percent` to `TRUE` if `x` is a percentage. This will convert it to a decimal number and adjust the decimal count (i.e., increase it by 2). Default is `FALSE`.
`show_rec`	Logical. For internal use only. If set to `TRUE`, the output is a matrix that also contains intermediary values from GRIM-testing. Don't specify this manually; instead, use `show_rec` in `grim_map()`. Default is `FALSE`.
`rounding`	String. Rounding method or methods to be used for reconstructing the values to which `x` will be compared. Default is `"up_or_down"` (from 5).
`threshold`	Numeric. If `rounding` is set to `"up_from"`, `"down_from"`, or `"up_from_or_down_from"`, set `threshold` to the number from which the reconstructed values should then be rounded up or down. Otherwise, this argument plays no role. Default is `5`.
`symmetric`	Logical. Set `symmetric` to `TRUE` if the rounding of negative numbers with `"up"`, `"down"`, `"up_from"`, or `"down_from"` should mirror that of positive numbers so that their absolute values are always equal. Default is `FALSE`.
`tolerance`	Numeric. Tolerance of comparison between `x` and the possible mean or percentage values. Default is circa 0.000000015 (1.490116e-08), as in `dplyr::near()`.

Details

The `x` values need to be strings because only strings retain trailing zeros, which are as important for the GRIM test as any other decimal digits.

Use `restore_zeros()` on numeric values (or values that were numeric values at some point) to easily supply the trailing zeros they might once have had. See documentation there.

Browse the source code in the grim.R file. `grim()` is a vectorized version of the internal `grim_scalar()` function found there.

Value

Logical. `TRUE` if `x`, `n`, and `items` are mutually consistent, `FALSE` if not.

References

Brown, N. J. L., & Heathers, J. A. J. (2017). The GRIM Test: A Simple Technique Detects Numerous Anomalies in the Reporting of Results in Psychology. *Social Psychological and Personality Science*, 8(4), 363–369. https://journals.sagepub.com/doi/10.1177/1948550616673876

See Also

`grim_map()` applies `grim()` to any number of cases at once.

Examples

```
# A mean of 5.19 is not consistent with a sample size of 28:
grim(x = "5.19", n = 28)    # `x` in quotes!

# However, it is consistent with a sample size of 32:
grim(x = "5.19", n = 32)

# For a scale composed of two items:
grim(x = "2.84", n = 16, items = 2)

# With percentages instead of means -- here, 71%:
grim(x = "71", n = 43, percent = TRUE)
```

`grim-stats` *Possible GRIM inconsistencies*

Description

Even without GRIM-testing, means / proportions and sample sizes of granular distributions entail some key data:

- `grim_total()` returns the absolute number of GRIM-inconsistencies that are possible given the mean or percentage's number of decimal places (`D`) and the corresponding sample size.
- `grim_ratio()` returns a proportion that is normalized by `10^D`, and therefore comparable across mean or percentage values reported to varying `D`.
- `grim_ratio_upper()` returns the upper bound of `grim_ratio()` for a given `D`.

For discussion, see `vignette("grim")`, section *GRIM statistics*.

Usage

```
grim_total(x, n, items = 1, percent = FALSE)

grim_ratio(x, n, items = 1, percent = FALSE)

grim_ratio_upper(x, percent = FALSE)
```

Arguments

`x`	String or numeric (length 1). Mean or percentage value computed from data with integer units (e.g., mean scores on a Likert scale or percentage of study participants in some condition). *Note*: Numeric inputs don't include trailing zeros, but these are important for GRIM functions. See documentation for `grim()`.
`n`	Integer. Sample size corresponding to `x`.
`items`	Integer. Number of items composing the mean or percentage value in question. Default is `1`.
`percent`	Logical. Set `percent` to `TRUE` if `x` is expressed as a proportion of 100 rather than 1. The functions will then account for this fact through increasing the decimal count by 2. Default is `FALSE`.

Value

Integer or double. The number or proportion of possible GRIM inconsistencies.

References

Brown, N. J. L., & Heathers, J. A. J. (2017). The GRIM Test: A Simple Technique Detects Numerous Anomalies in the Reporting of Results in Psychology. *Social Psychological and Personality Science*, 8(4), 363–369. https://journals.sagepub.com/doi/10.1177/1948550616673876

See Also

`grim()` for the GRIM test itself; as well as `grim_map()` for applying it to many cases at once.

Examples

```
# Many value sets are inconsistent here:
grim_total(x = "83.29", n = 21)
grim_ratio(x = "83.29", n = 21)

# No sets are inconsistent in this case...
grim_total(x = "5.14", n = 83)
grim_ratio(x = "5.14", n = 83)

# ... but most would be if `x` was a percentage:
grim_total(x = "5.14", n = 83, percent = TRUE)
grim_ratio(x = "5.14", n = 83, percent = TRUE)
```

`grimmer` *The GRIMMER test (granularity-related inconsistency of means mapped to error repeats)*

Description

`grimmer()` checks if reported mean and SD values of integer data are mathematically consistent with the reported sample size and the number of items that compose the mean value. It works much like `grim()`.

The function is vectorized, but it is recommended to use `grimmer_map()` for testing multiple cases.

Usage

```
grimmer(
  x,
  sd,
  n,
  items = 1,
  show_reason = FALSE,
  rounding = "up_or_down",
  threshold = 5,
  symmetric = FALSE,
  tolerance = .Machine$double.eps^0.5
)
```

Arguments

`x`	String. The reported mean value.
`sd`	String. The reported standard deviation.
`n`	Integer. The reported sample size.
`items`	*(NOTE: Don't use the* `items` *argument. It currently contains a bug that will be fixed in the future.)* Integer. The number of items composing the `x` and `sd` values. Default is 1, the most common case.
`show_reason`	Logical. For internal use only. If set to `TRUE`, the output is a list of length-2 lists which also contain the reasons for inconsistencies. Don't specify this manually; instead, use `show_reason` in `grimmer_map()`. Default is `FALSE`.
`rounding`	String. Rounding method or methods to be used for reconstructing the values to which `x` will be compared. Default is `"up_or_down"` (from 5).
`threshold`	Numeric. If `rounding` is set to `"up_from"`, `"down_from"`, or `"up_from_or_down_from"`, set `threshold` to the number from which the reconstructed values should then be rounded up or down. Otherwise, this argument plays no role. Default is `5`.
`symmetric`	Logical. Set `symmetric` to `TRUE` if the rounding of negative numbers with `"up"`, `"down"`, `"up_from"`, or `"down_from"` should mirror that of positive numbers so that their absolute values are always equal. Default is `FALSE`.
`tolerance`	Numeric. Tolerance of comparison between `x` and the possible mean or percentage values. Default is circa 0.000000015 (1.490116e-08), as in `dplyr::near()`.

Details

GRIMMER was originally devised by Anaya (2016). The present implementation follows Allard's (2018) refined Analytic-GRIMMER (A-GRIMMER) algorithm. It adapts the R function `aGrimmer()` provided by Allard and modifies it to accord with scrutiny's standards, as laid out in `vignette("consistency-tests-in-depth")`, sections 1-2. The resulting `grimmer()` function, then, is a vectorized version of this basic implementation. For more context and variable name translations, see the top of the R/grimmer.R, the source file.

The present implementation can differ from Allard's in a small number of cases. In most cases, this means that the original flags a value set as inconsistent, but scrutiny's `grimmer*()` functions don't. For details, see the end of tests/testthat/test-grimmer.R, the `grimmer()` test file.

Value

Logical. `TRUE` if `x`, `sd`, `n`, and `items` are mutually consistent, `FALSE` if not.

References

Allard, A. (2018). Analytic-GRIMMER: a new way of testing the possibility of standard deviations. https://aurelienallard.netlify.app/post/anaytic-grimmer-possibility-standard-deviations/

Anaya, J. (2016). The GRIMMER test: A method for testing the validity of reported measures of variability. *PeerJ Preprints.* https://peerj.com/preprints/2400v1/

Examples

```
# A mean of 5.23 is not consistent with an SD of 2.55
# and a sample size of 35:
grimmer(x = "5.23", sd = "2.55", n = 35)

# However, mean and SD are consistent with a
# sample size of 31:
grimmer(x = "5.23", sd = "2.55", n = 31)

# For a scale composed of two items:
grimmer(x = "2.74", sd = "0.96", n = 63, items = 2)
```

---

grimmer_map — *GRIMMER-test many cases at once*

Description

Call `grimmer_map()` to GRIMMER-test any number of combinations of mean, standard deviation, sample size, and number of items. Mapping function for GRIMMER-testing.

For summary statistics, call `audit()` on the results. Visualize results using `grim_plot()`, as with GRIM results.

Usage

```
grimmer_map(
  data,
  items = 1,
  merge_items = TRUE,
  x = NULL,
  sd = NULL,
  n = NULL,
  show_reason = TRUE,
  rounding = "up_or_down",
  threshold = 5,
  symmetric = FALSE,
  tolerance = .Machine$double.eps^0.5
)
```

Arguments

`data`	Data frame with columns `x`, `sd`, `n`, and optionally `items` (see documentation for `grim()`). Any other columns in `data` will be returned alongside GRIMMER test results.
`items`	*(NOTE: Don't use the* `items` *argument. It currently contains a bug that will be fixed in the future.)* Integer. If there is no `items` column in `data`, this specifies the number of items composing the `x` and `sd` values. Default is 1, the most common case.

`merge_items`	Logical. If `TRUE` (the default), there will be no `items` column in the output. Instead, values from an `items` column or argument will be multiplied with values in the `n` column. This does not affect GRIM- or GRIMMER-testing.
`x`, `sd`, `n`	Optionally, specify these arguments as column names in `data`.
`show_reason`	Logical (length 1). Should there be a `reason` column that shows the reasons for inconsistencies (and `NA` for consistent values)? Default is `FALSE`.
`rounding`, `threshold`, `symmetric`, `tolerance`	Further parameters of GRIMMER-testing; see documentation for `grimmer()`.

Value

A tibble with these columns –

- `x`, `sd`, `n`: the inputs.
- `consistency`: GRIMMER consistency of `x`, `n`, and `items`.
- `<extra>`: any columns from `data` other than `x`, `n`, and `items`.

The tibble has the `scr_grimmer_map` class, which is recognized by the `audit()` generic. It also has the `scr_grim_map` class, so it can be visualized by `grim_plot()`.

Summaries with `audit()`

There is an S3 method for `audit()`, so you can call `audit()` following `grimmer_map()` to get a summary of `grimmer_map()`'s results. It is a tibble with a single row and these columns –

1. `incons_cases`: number of GRIMMER-inconsistent value sets.
2. `all_cases`: total number of value sets.
3. `incons_rate`: proportion of GRIMMER-inconsistent value sets.
4. `fail_grim`: number of value sets that fail the GRIM test.
5. `fail_test1`: number of value sets that fail the first GRIMMER test (sum of squares is a whole number).
6. `fail_test2`: number of value sets that fail the second GRIMMER test (matching SDs).
7. `fail_test3`: number of value sets that fail the third GRIMMER test (equal parity).

References

Allard, A. (2018). Analytic-GRIMMER: a new way of testing the possibility of standard deviations. https://aurelienallard.netlify.app/post/anaytic-grimmer-possibility-standard-deviations/

Anaya, J. (2016). The GRIMMER test: A method for testing the validity of reported measures of variability. *PeerJ Preprints.* https://peerj.com/preprints/2400v1/

Examples

```
# Use `grimmer_map()` on data like these:
pigs5

# The `consistency` column shows whether
# the values to its left are GRIMMER-consistent.
# If they aren't, the `reason` column says why:
pigs5 %>%
  grimmer_map()

# Get summaries with `audit()`:
pigs5 %>%
  grimmer_map() %>%
  audit()
```

---

grimmer_map_seq *GRIMMER-testing with dispersed inputs*

Description

grimmer_map_seq() performs GRIMMER-testing with values surrounding the input values. This provides an easy and powerful way to assess whether small errors in computing or reporting may be responsible for GRIMMER inconsistencies in published statistics.

Call audit_seq() on the results for summary statistics.

Usage

```
grimmer_map_seq(
  data,
  x = NULL,
  sd = NULL,
  n = NULL,
  var = Inf,
  dispersion = 1:5,
  out_min = "auto",
  out_max = NULL,
  include_reported = FALSE,
  include_consistent = FALSE,
  ...
)
```

Arguments

data	A data frame that grimmer_map() could take.
x, sd, n	Optionally, specify these arguments as column names in data.
var	String. Names of the columns that will be dispersed. Default is c("x", "sd", "n").

`dispersion` Numeric. Sequence with steps up and down from the `var` inputs. It will be adjusted to these values' decimal levels. For example, with a reported `8.34`, the step size is `0.01`. Default is `1:5`, for five steps up and down.

`out_min`, `out_max`
If specified, output will be restricted so that it's not below `out_min` or above `out_max`. Defaults are `"auto"` for `out_min`, i.e., a minimum of one decimal unit above zero; and `NULL` for `out_max`, i.e., no maximum.

`include_reported`
Logical. Should the reported values themselves be included in the sequences originating from them? Default is `FALSE` because this might be redundant and bias the results.

`include_consistent`
Logical. Should the function also process consistent cases (from among those reported), not just inconsistent ones? Default is `FALSE` because the focus should be on clarifying inconsistencies.

`...` Arguments passed down to `grimmer_map()`. *(NOTE: Don't use the* `items` *argument. It currently contains a bug that will be fixed in the future.)*

Value

A tibble (data frame) with detailed test results.

Summaries with `audit_seq()`

You can call `audit_seq()` following `grimmer_map_seq()`. It will return a data frame with these columns:

- `x`, `sd`, and `n` are the original inputs, tested for `consistency` here.
- `hits_total` is the total number of GRIMMER-consistent value sets found within the specified `dispersion` range.
- `hits_x` is the number of GRIMMER-consistent value sets found by varying `x`.
- Accordingly with `sd` and `hits_sd` as well as `n` and `hits_n`.
- (Note that any consistent reported cases will be counted by the `hits_*` columns if both `include_reported` and `include_consistent` are set to `TRUE`.)
- `diff_x` reports the absolute difference between `x` and the next consistent dispersed value (in dispersion steps, not the actual numeric difference). `diff_x_up` and `diff_x_down` report the difference to the next higher or lower consistent value, respectively.
- `diff_sd`, `diff_sd_up`, and `diff_sd_down` do the same for `sd`.
- Likewise with `diff_n`, `diff_n_up`, and `diff_n_down`.

Call `audit()` following `audit_seq()` to summarize results even further. It's mostly self-explaining, but `na_count` and `na_rate` are the number and rate of times that a difference could not be computed because of a lack of corresponding hits within the `dispersion` range.

Examples

```
# `grimmer_map_seq()` can take any input
# that `grimmer_map()` can take:
pigs5

# All the results:
out <- grimmer_map_seq(pigs5, include_consistent = TRUE)
out

# Case-wise summaries with `audit_seq()`
# can be more important than the raw results:
out %>%
  audit_seq()
```

`grimmer_map_total_n` *GRIMMER-testing with hypothetical group sizes*

Description

When reporting group means, some published studies only report the total sample size but no group sizes corresponding to each mean. However, group sizes are crucial for GRIMMER-testing.

In the two-groups case, `grimmer_map_total_n()` helps in these ways:

- It creates hypothetical group sizes. With an even total sample size, it incrementally moves up and down from half the total sample size. For example, with a total sample size of 40, it starts at 20, goes on to 19 and 21, then to 18 and 22, etc. With odd sample sizes, it starts from the two integers around half.
- It GRIMMER-tests all of these values together with the group means.
- It reports all the scenarios in which both "dispersed" hypothetical group sizes are GRIMMER-consistent with the group means.

All of this works with one or more total sample sizes at a time. Call `audit_total_n()` for summary statistics.

Usage

```
grimmer_map_total_n(
  data,
  x1 = NULL,
  x2 = NULL,
  sd1 = NULL,
  sd2 = NULL,
  dispersion = 0:5,
  n_min = 1L,
  n_max = NULL,
  constant = NULL,
  constant_index = NULL,
  ...
)
```

Arguments

`data`	Data frame with string columns `x1`, `x2`, `sd1`, and `sd2`, as well as numeric column `n`. The first two are reported group means. `sd1` and `sd2` are reported group SDs. `n` is the reported total sample size. It is not very important whether a value is in `x1` or in `x2` because, after the first round of tests, the function switches roles between `x1` and `x2`, and reports the outcomes both ways. The same applies to `sd1` and `sd2`. However, do make sure the `x*` and `sd*` values are paired accurately, as reported.
`x1`, `x2`, `sd1`, `sd2`	Optionally, specify these arguments as column names in `data`.
`dispersion`	Numeric. Steps up and down from half the `n` values. Default is `0:5`, i.e., half `n` itself followed by five steps up and down.
`n_min`	Numeric. Minimal group size. Default is 1.
`n_max`	Numeric. Maximal group size. Default is `NULL`, i.e., no maximum.
`constant`	Optionally, add a length-2 vector or a list of length-2 vectors (such as a data frame with exactly two rows) to accompany the pairs of dispersed values. Default is `NULL`, i.e., no constant values.
`constant_index`	Integer (length 1). Index of `constant` or the first `constant` column in the output tibble. If `NULL` (the default), `constant` will go to the right of `n_change`.
`...`	Arguments passed down to `grimmer_map()`. *(NOTE: Don't use the* `items` *argument. It currently contains a bug that will be fixed in the future.)*

Value

A tibble with these columns:

- `x`, the group-wise reported input statistic, is repeated in row pairs.
- `n` is dispersed from half the input `n`, with `n_change` tracking the differences.
- `both_consistent` flags scenarios where both reported `x` values are consistent with the hypothetical `n` values.
- `case` corresponds to the row numbers of the input data frame.
- `dir` is `"forth"` in the first half of rows and `"back"` in the second half. `"forth"` means that `x2` from the input is paired with the larger dispersed `n`, whereas `"back"` means that `x1` is paired with the larger dispersed `n`.
- Other columns from `grimmer_map()` are preserved.

Summaries with `audit_total_n()`

You can call `audit_total_n()` following up on `grimmer_map_total_n()` to get a tibble with summary statistics. It will have these columns:

- `x1`, `x2`, `sd1`, `sd2`, and `n` are the original inputs.
- `hits_total` is the number of scenarios in which all of `x1`, `x2`, `sd1`, and `sd2` are GRIMMER-consistent. It is the sum of `hits_forth` and `hits_back` below.
- `hits_forth` is the number of both-consistent cases that result from pairing `x2` and `sd2` with the larger dispersed `n` value.

- `hits_back` is the same, except `x1` and `sd1` are paired with the larger dispersed `n` value.
- `scenarios_total` is the total number of test scenarios, whether or not both `x1` and `sd1` as well as `x2` and `sd2` are GRIMMER-consistent.
- `hit_rate` is the ratio of `hits_total` to `scenarios_total`.

References

Bauer, P. J., & Francis, G. (2021). Expression of Concern: Is It Light or Dark? Recalling Moral Behavior Changes Perception of Brightness. *Psychological Science*, 32(12), 2042–2043. https://journals.sagepub.com/doi/10.11

Allard, A. (2018). Analytic-GRIMMER: a new way of testing the possibility of standard deviations. https://aurelienallard.netlify.app/post/anaytic-grimmer-possibility-standard-deviations/

Bauer, P. J., & Francis, G. (2021). Expression of Concern: Is It Light or Dark? Recalling Moral Behavior Changes Perception of Brightness. *Psychological Science*, 32(12), 2042–2043. https://journals.sagepub.com/doi/10.11

See Also

`function_map_total_n()`, which created the present function using `grimmer_map()`.

Examples

```
# Run `grimmer_map_total_n()` on data like these:
df <- tibble::tribble(
  ~x1,    ~x2,    ~sd1,   ~sd2,   ~n,
  "3.43", "5.28", "1.09", "2.12", 70,
  "2.97", "4.42", "0.43", "1.65", 65
)
df

grimmer_map_total_n(df)

# `audit_total_n()` summaries can be more important than
# the detailed results themselves.
# The `hits_total` column shows all scenarios in
# which both divergent `n` values are GRIMMER-consistent
# with the `x*` values when paired with them both ways:
df %>%
  grimmer_map_total_n() %>%
  audit_total_n()

# By default (`dispersion = 0:5`), the function goes
# five steps up and down from `n`. If this sequence
# gets longer, the number of hits tends to increase:
df %>%
  grimmer_map_total_n(dispersion = 0:10) %>%
  audit_total_n()
```

`grim_granularity` *Granularity of non-continuous scales*

Description

`grim_granularity()` computes the minimal difference between two means or proportions of ordinal or interval data.

`grim_items()` is the reverse: It converts granularity values to the number of scale items, which might then be used for consistency testing functions such as `grim()`.

Usage

```
grim_granularity(n, items = 1)

grim_items(n, gran, tolerance = .Machine$double.eps^0.5)
```

Arguments

`n`	Numeric. Sample size.
`items`	Numeric. Number of items composing the scale. Default is 1, which will hold for most non-Likert scales.
`gran`	Numeric. Granularity.
`tolerance`	Numeric. In `grim_items()`, `tolerance` is the maximal amount by which results may differ from being whole numbers. If they exceed that amount, a warning will be shown.

Details

These two functions differ only in the names of their arguments — the underlying formula is the same (and it's very simple). However, for clarity, they are presented as distinct.

The output of `grim_items()` should be whole numbers, because scale items have a granularity of 1. If they differ from the next whole number by more than a numeric `tolerance` (which is determined by the argument by that name), a warning will be shown.

It would be wrong to determine a scale's granularity from the minimal distance between two values in a given distribution. This would only signify how those values actually do differ, not how they *can* differ *a priori* based on scale design. Also, keep in mind that continuous scales have no granularity at all.

Value

Numeric. Granularity or number of items.

References

Brown, N. J. L., & Heathers, J. A. J. (2017). The GRIM Test: A Simple Technique Detects Numerous Anomalies in the Reporting of Results in Psychology. *Social Psychological and Personality Science*, 8(4), 363–369. https://journals.sagepub.com/doi/10.1177/1948550616673876

Examples

```
# If a non-Likert scale ranges from 0 to 3
# and measures 16 cases:
grim_granularity(n = 16)   # `items = 1` by default

# Same but Likert scale with 2 items:
grim_granularity(n = 16, items = 2)

# If a scale is applied to a single case
# and has a granularity of 0.5:
grim_items(n = 1, gran = 0.5)

# With more cases, a warning appears
# because items can only be whole numbers:
grim_items(n = c(10, 15, 20), gran = 0.5)
```

---

grim_map *GRIM-test many cases at once*

Description

Call `grim_map()` to GRIM-test any number of combinations of mean/proportion, sample size, and number of items. Mapping function for GRIM-testing.

Set `percent` to `TRUE` if the `x` values are percentages. This will convert `x` values to decimals and adjust the decimal count accordingly.

Display intermediary numbers from GRIM-testing in columns by setting `show_rec` to `TRUE`.

For summary statistics, call `[audit()]` on the results.

Usage

```
grim_map(
  data,
  items = 1,
  merge_items = TRUE,
  percent = FALSE,
  x = NULL,
  n = NULL,
  show_rec = FALSE,
  show_prob = FALSE,
  rounding = "up_or_down",
  threshold = 5,
  symmetric = FALSE,
  tolerance = .Machine$double.eps^0.5,
  testables_only = FALSE,
  extra = Inf
)
```

Arguments

`data`	Data frame with columns `x`, `n`, and optionally `items` (see documentation for `grim()`. By default, any other columns in `data` will be returned alongside GRIM test results (see `extra` below).
`items`	Integer. If there is no `items` column in `data`, this specifies the number of items composing the `x` values. Default is 1, the most common case.
`merge_items`	Logical. If `TRUE` (the default), there will be no `items` column in the output. Instead, values from an `items` column or argument will be multiplied with values in the `n` column. This does not affect GRIM-testing.
`percent`	Logical. Set `percent` to `TRUE` if the `x` values are percentages. This will convert them to decimal numbers and adjust the decimal count (i.e., increase it by 2). It also affects the `ratio` column. Default is `FALSE`.
`x, n`	Optionally, specify these arguments as column names in `data`.
`show_rec`	Logical. If set to `TRUE`, the reconstructed numbers from GRIM-testing are shown as columns. See section *Reconstructed numbers* below. Default is `FALSE`.
`show_prob`	Logical. If set to `TRUE`, adds a `prob` column that contains the probability of GRIM inconsistency. This is simply the `ratio` column censored to range between 0 and 1. Default is `FALSE`.
`rounding, threshold, symmetric, tolerance`	Further parameters of GRIM-testing; see documentation for `grim()`.
`testables_only`	Logical. If `testables_only` is set to `TRUE`, only GRIM-testable cases (i.e., those with a positive GRIM ratio) are included. Default is `FALSE`.
`extra`	String or integer. The other column(s) from `data` to be returned in the output tibble alongside test results, referenced by their name(s) or number(s). Default is `Inf`, which returns all columns. To return none of them, set `extra` to 0.

Value

A tibble with these columns –

- `x, n`: the inputs.
- `consistency`: GRIM consistency of `x`, `n`, and `items`.
- `<extra>`: any columns from `data` other than `x`, `n`, and `items`.
- `ratio`: the GRIM ratio; see `grim_ratio()`.

  The tibble has the `scr_grim_map` class, which is recognized by the `audit()` generic.

Reconstructed numbers

If `show_rec` is set to `TRUE`, the output includes the following additional columns:

- `rec_sum`: the sum total from which the mean or proportion was ostensibly derived.
- `rec_x_upper`: the upper reconstructed `x` value.
- `rec_x_lower`: the lower reconstructed `x` value.
- `rec_x_upper_rounded`: the rounded `rec_x_upper` value.

- rec_x_lower_rounded: the rounded rec_x_lower value.

With the default for rounding, "up_or_down", each of the last two columns is replaced by two columns that specify the rounding procedures (i.e., "_up" and "_down").

Summaries with audit()

There is an S3 method for audit(), so you can call audit() following grim_map() to get a summary of grim_map()'s results. It is a tibble with one row and these columns –

1. incons_cases: number of GRIM-inconsistent value sets.
2. all_cases: total number of value sets.
3. incons_rate: proportion of GRIM-inconsistent value sets.
4. mean_grim_ratio: average of GRIM ratios.
5. incons_to_ratio: ratio of incons_rate to mean_grim_ratio.
6. testable_cases: number of GRIM-testable value sets (i.e., those with a positive ratio).
7. testable_rate: proportion of GRIM-testable value sets.

References

Brown, N. J. L., & Heathers, J. A. J. (2017). The GRIM Test: A Simple Technique Detects Numerous Anomalies in the Reporting of Results in Psychology. *Social Psychological and Personality Science*, 8(4), 363–369. https://journals.sagepub.com/doi/10.1177/1948550616673876

Examples

```
# Use `grim_map()` on data like these:
pigs1

# The `consistency` column shows
# whether the values to its left
# are GRIM-consistent:
pigs1 %>%
  grim_map()

# Display intermediary numbers from
# GRIM-testing with `show_rec = TRUE`:
pigs1 %>%
  grim_map(show_rec = TRUE)

# Get summaries with `audit()`:
pigs1 %>%
  grim_map() %>%
  audit()
```

grim_map_seq — *GRIM-testing with dispersed inputs*

Description

`grim_map_seq()` performs GRIM-testing with values surrounding the input values. This provides an easy and powerful way to assess whether small errors in computing or reporting may be responsible for GRIM inconsistencies in published statistics.

Call `audit_seq()` on the results for summary statistics.

Usage

```
grim_map_seq(
  data,
  x = NULL,
  n = NULL,
  var = Inf,
  dispersion = 1:5,
  out_min = "auto",
  out_max = NULL,
  include_reported = FALSE,
  include_consistent = FALSE,
  ...
)
```

Arguments

`data`	A data frame that `grim_map()` could take.
`x, n`	Optionally, specify these arguments as column names in `data`.
`var`	String. Names of the columns that will be dispersed. Default is `c("x", "n")`.
`dispersion`	Numeric. Sequence with steps up and down from the `var` inputs. It will be adjusted to these values' decimal levels. For example, with a reported `8.34`, the step size is `0.01`. Default is `1:5`, for five steps up and down.
`out_min, out_max`	If specified, output will be restricted so that it's not below `out_min` or above `out_max`. Defaults are `"auto"` for `out_min`, i.e., a minimum of one decimal unit above zero; and `NULL` for `out_max`, i.e., no maximum.
`include_reported`	Logical. Should the reported values themselves be included in the sequences originating from them? Default is `FALSE` because this might be redundant and bias the results.
`include_consistent`	Logical. Should the function also process consistent cases (from among those reported), not just inconsistent ones? Default is `FALSE` because the focus should be on clarifying inconsistencies.
`...`	Arguments passed down to `grim_map()`.

Value

A tibble (data frame) with detailed test results.

Summaries with `audit_seq()`

You can call `audit_seq()` following `grim_map_seq()`. It will return a data frame with these columns:

- `x` and `n` are the original inputs, tested for `consistency` here.
- `hits_total` is the total number of GRIM-consistent value sets found within the specified `dispersion` range.
- `hits_x` is the number of GRIM-consistent value sets found by varying `x`.
- Accordingly with `n` and `hits_n`.
- (Note that any consistent reported cases will be counted by the `hits_*` columns if both `include_reported` and `include_consistent` are set to `TRUE`.)
- `diff_x` reports the absolute difference between `x` and the next consistent dispersed value (in dispersion steps, not the actual numeric difference). `diff_x_up` and `diff_x_down` report the difference to the next higher or lower consistent value, respectively.
- `diff_n`, `diff_n_up`, and `diff_n_down` do the same for `n`.

Call `audit()` following `audit_seq()` to summarize results even further. It's mostly self-explaining, but `na_count` and `na_rate` are the number and rate of times that a difference could not be computed because of a lack of corresponding hits within the `dispersion` range.

Examples

```
# `grim_map_seq()` can take any input
# that `grim_map()` can take:
pigs1

# All the results:
out <- grim_map_seq(pigs1, include_consistent = TRUE)
out

# Case-wise summaries with `audit_seq()`
# can be more important than the raw results:
out %>%
  audit_seq()
```

`grim_map_total_n` *GRIM-testing with hypothetical group sizes*

Description

When reporting group means, some published studies only report the total sample size but no group sizes corresponding to each mean. However, group sizes are crucial for GRIM-testing.

In the two-groups case, `grim_map_total_n()` helps in these ways:

- It creates hypothetical group sizes. With an even total sample size, it incrementally moves up and down from half the total sample size. For example, with a total sample size of 40, it starts at 20, goes on to 19 and 21, then to 18 and 22, etc. With odd sample sizes, it starts from the two integers around half.
- It GRIM-tests all of these values together with the group means.
- It reports all the scenarios in which both "dispersed" hypothetical group sizes are GRIM-consistent with the group means.

All of this works with one or more total sample sizes at a time. Call `audit_total_n()` for summary statistics.

Usage

```
grim_map_total_n(
  data,
  x1 = NULL,
  x2 = NULL,
  dispersion = 0:5,
  n_min = 1L,
  n_max = NULL,
  constant = NULL,
  constant_index = NULL,
  ...
)
```

Arguments

`data`	Data frame with string columns `x1` and `x2`, and numeric column `n`. The first two are group mean or percentage values with unknown group sizes, and `n` is the total sample size. It is not very important whether a value is in `x1` or in `x2` because, after the first round of tests, the function switches roles between `x1` and `x2`, and reports the outcomes both ways.
`x1, x2`	Optionally, specify these arguments as column names in `data`.
`dispersion`	Numeric. Steps up and down from half the `n` values. Default is `0:5`, i.e., half `n` itself followed by five steps up and down.
`n_min`	Numeric. Minimal group size. Default is 1.
`n_max`	Numeric. Maximal group size. Default is `NULL`, i.e., no maximum.
`constant`	Optionally, add a length-2 vector or a list of length-2 vectors (such as a data frame with exactly two rows) to accompany the pairs of dispersed values. Default is `NULL`, i.e., no constant values.
`constant_index`	Integer (length 1). Index of `constant` or the first `constant` column in the output tibble. If `NULL` (the default), `constant` will go to the right of `n_change`.
`...`	Arguments passed down to `grim_map()`.

Value

A tibble with these columns:

- `x`, the group-wise reported input statistic, is repeated in row pairs.
- `n` is dispersed from half the input `n`, with `n_change` tracking the differences.
- `both_consistent` flags scenarios where both reported `x` values are consistent with the hypothetical `n` values.
- `case` corresponds to the row numbers of the input data frame.
- `dir` is `"forth"` in the first half of rows and `"back"` in the second half. `"forth"` means that `x2` from the input is paired with the larger dispersed `n`, whereas `"back"` means that `x1` is paired with the larger dispersed `n`.
- Other columns from `grim_map()` are preserved.

Summaries with `audit_total_n()`

You can call `audit_total_n()` following up on `grim_map_total_n()` to get a tibble with summary statistics. It will have these columns:

- `x1`, `x2`, and `n` are the original inputs.
- `hits_total` is the number of scenarios in which both `x1` and `x2` are GRIM-consistent. It is the sum of `hits_forth` and `hits_back` below.
- `hits_forth` is the number of both-consistent cases that result from pairing `x2` with the larger dispersed `n` value.
- `hits_back` is the same, except `x1` is paired with the larger dispersed `n` value.
- `scenarios_total` is the total number of test scenarios, whether or not both `x1` and `x2` are GRIM-consistent.
- `hit_rate` is the ratio of `hits_total` to `scenarios_total`.

Call `audit()` following `audit_total_n()` to summarize results even further.

References

Bauer, P. J., & Francis, G. (2021). Expression of Concern: Is It Light or Dark? Recalling Moral Behavior Changes Perception of Brightness. *Psychological Science*, 32(12), 2042–2043. https://journals.sagepub.com/doi/10.11

Brown, N. J. L., & Heathers, J. A. J. (2017). The GRIM Test: A Simple Technique Detects Numerous Anomalies in the Reporting of Results in Psychology. *Social Psychological and Personality Science*, 8(4), 363–369. https://journals.sagepub.com/doi/10.1177/1948550616673876

See Also

`function_map_total_n()`, which created the present function using `grim_map()`.

Examples

```
# Run `grim_map_total_n()` on data like these:
df <- tibble::tribble(
  ~x1,    ~x2,   ~n,
  "3.43", "5.28", 90,
  "2.97", "4.42", 103
)
df

grim_map_total_n(df)

# `audit_total_n()` summaries can be more important than
# the detailed results themselves.
# The `hits_total` column shows all scenarios in
# which both divergent `n` values are GRIM-consistent
# with the `x*` values when paired with them both ways:
df %>%
  grim_map_total_n() %>%
  audit_total_n()

# By default (`dispersion = 0:5`), the function goes
# five steps up and down from `n`. If this sequence
# gets longer, the number of hits tends to increase:
df %>%
  grim_map_total_n(dispersion = 0:10) %>%
  audit_total_n()
```

`grim_plot` *Visualize GRIM test results*

Description

`grim_plot()` visualizes summary data and their mutual GRIM consistency. Call this function only on a data frame that resulted from a call to `grim_map()`.

Consistent and inconsistent value pairs from the input data frame are shown in distinctive colors. By default, consistent value pairs are blue and inconsistent ones are red. These and other parameters of the underlying geoms can be controlled via arguments.

The background raster follows the `rounding` argument from the `grim_map()` call (unless any of the plotted mean or proportion values has more than 2 decimal places, in which case a gradient is shown, not a raster).

Usage

```
grim_plot(
  data = NULL,
  show_data = TRUE,
  show_raster = TRUE,
  show_gradient = TRUE,
  n = NULL,
  digits = NULL,
  rounding = "up_or_down",
  color_cons = "royalblue1",
  color_incons = "red",
  tile_alpha = 1,
  tile_size = 1.5,
  raster_alpha = 1,
  raster_color = "grey75"
)
```

Arguments

`data`	Data frame. Result of a call to `grim_map()`.
`show_data`	Logical. If set to `FALSE`, test results from the data are not displayed. Choose this if you only want to show the background raster. You can then control plot parameters directly via the `n`, `digits`, and `rounding` arguments. Default is `TRUE`.
`show_raster`	Logical. If `TRUE` (the default), the plot has a background raster.
`show_gradient`	Logical. If the number of decimal places is 3 or greater, should a gradient be shown to signal the overall probability of GRIM inconsistency? Default is `TRUE`.
`n`	Integer. Maximal value on the x-axis. Default is `NULL`, in which case `n` becomes `10 ^ digits` (e.g., `100` if `digits` is `2`).
`digits`	Integer. Only relevant if `show_data` is set to `FALSE`. The plot will then be constructed as it would be for data where all `x` values have this many decimal places. Default is `2`.
`rounding`	String. Only relevant if `show_data` is set to `FALSE`. The plot will then be constructed as it would be for data rounded in this particular way. Default is `"up_or_down"`.
`color_cons, color_incons`	Strings. Fill colors of the consistent and inconsistent scatter points. Defaults are `"royalblue1"` (consistent) and `"red"` (inconsistent).
`tile_alpha, tile_size`	Numeric. Further parameters of the scatter points: opacity and, indirectly, size. Defaults are `1` and `1.5`.
`raster_alpha, raster_color`	Numeric and string, respectively. Parameters of the background raster: opacity and fill color. Defaults are `1` and `"grey75"`.

Value

A ggplot object.

Background raster

The background raster shows the probability of GRIM-inconsistency for random means or proportions, from 0 (all inconsistent) to the greatest number on the x-axis (all consistent). If the number of decimal places in the inputs – means or percentages – is 3 or greater, individual points would be too small to display. In these cases, there will not be a raster but a gradient, showing the overall trend.

As any raster only makes sense with respect to one specific number of decimal places, the function will throw an error if these numbers differ among input `x` values (and `show_raster` is `TRUE`). You can avoid the error and force plotting by specifying `digits` as the number of decimal places for which the raster or gradient should be displayed.

For 1 or 2 decimal places, the raster will be specific to the rounding procedure. As the raster varies by rounding procedure, it will automatically correspond to the `rounding` argument specified in the preceding `grim_map()` call. This works fast because the raster is based on data saved in the package itself, so these data don't need to be generated anew every time the function is called. Inconsistent value sets are marked with dark boxes. All other places in the raster denote consistent value sets. The raster is independent of the data – it only follows the `rounding` specification in the `grim_map()` call and the `digits` argument in `grim_plot()`.

Display an "empty" plot, one without empirical test results, by setting `show_data` to `FALSE`. You can then control key parameters of the plot with `digits` and `rounding`.

With `grim_map()`'s default for `rounding`, `"up_or_down"`, strikingly few values are flagged as inconsistent for sample sizes 40 and 80 (or 4 and 8). This effect disappears if `rounding` is set to any other value (see `vignette("rounding-options")`).

The 4/8 leniency effect arises because accepting values rounded either up or down is more careful and conservative than any other rounding procedure. In any case, `grim_plot()` doesn't cause this effect — it only reveals it.

References

Brown, N. J. L., & Heathers, J. A. J. (2017). The GRIM Test: A Simple Technique Detects Numerous Anomalies in the Reporting of Results in Psychology. *Social Psychological and Personality Science*, 8(4), 363–369. https://journals.sagepub.com/doi/10.1177/1948550616673876

Examples

```
# Call `grim_plot()` following `grim_map()`:
pigs1 %>%
  grim_map() %>%
  grim_plot()

# If you change the rounding procedure
# in `grim_map()`, the plot will
# follow automatically if there is
# a difference:
pigs1 %>%
  grim_map(rounding = "ceiling") %>%
  grim_plot()

# For percentages, the y-axis
# label also changes automatically:
pigs2 %>%
  grim_map(percent = TRUE) %>%
  grim_plot()
```

`is_numeric_like` *Test whether a vector is numeric or coercible to numeric*

Description

`is_numeric_like()` tests whether an object is "coercible to numeric" by the particular standards of scrutiny. This means:

- Integer and double vectors are `TRUE`.
- Logical vectors are `FALSE`, as are non-vector objects.
- Other vectors (most likely strings) are `TRUE` if all their non-`NA` values can be coerced to non-`NA` numeric values, and `FALSE` otherwise.
- Factors are first coerced to string, then tested.
- Lists are tested like atomic vectors unless any of their elements have length greater 1, in which case they are always `FALSE`.
- If all values are non-numeric, non-logical `NA`, the output is also `NA`.

See details for discussion.

Usage

```
is_numeric_like(x)
```

Arguments

`x`	Object to be tested.

Details

The scrutiny package often deals with "number-strings", i.e., strings that can be coerced to numeric without introducing new `NA`s. This is a matter of displaying data in a certain way, as opposed to their storage mode.

`is_numeric_like()` returns `FALSE` for logical vectors simply because these are displayed as strings, not as numbers, and the usual coercion rules would be misleading in this context. Likewise, the function treats factors like strings because that is how they are displayed: the fact that factors are stored as integers is irrelevant.

Why store numbers as strings or factors? Only these data types can preserve trailing zeros, and only if the data were originally entered as strings. See `vignette("wrangling")`, section *Trailing zeros*.

Value

Logical (length 1).

See Also

The vctrs package, which provides a serious typing framework for R; in contrast to this rather ad-hoc function.

Examples

```
# Numeric vectors are `TRUE`:
is_numeric_like(x = 1:5)
is_numeric_like(x = 2.47)

# Logical vectors are always `FALSE`:
is_numeric_like(x = c(TRUE, FALSE))

# Strings are `TRUE` if all of their non-`NA`
# values can be coerced to non-`NA` numbers,
# and `FALSE` otherwise:
is_numeric_like(x = c("42", "0.7", NA))
is_numeric_like(x = c("42", "xyz", NA))

# Factors are treated like their
# string equivalents:
is_numeric_like(x = as.factor(c("42", "0.7", NA)))
is_numeric_like(x = as.factor(c("42", "xyz", NA)))

# Lists behave like atomic vectors if all of their
# elements have length 1...
is_numeric_like(x = list("42", "0.7", NA))
is_numeric_like(x = list("42", "xyz", NA))

# ...but if they don't, they are `FALSE`:
is_numeric_like(x = list("42", "0.7", NA, c(1, 2, 3)))

# If all values are `NA`, so is the output...
is_numeric_like(x = as.character(c(NA, NA, NA)))

# ...unless the `NA`s are numeric or logical:
is_numeric_like(x = as.numeric(c(NA, NA, NA)))
is_numeric_like(x = c(NA, NA, NA))
```

`manage_helper_col` *Helper column operations*

Description

If your consistency test mapper function supports helper columns, call `manage_helper_col()` internally; once for every such column. It will check whether a helper column is compatible with its eponymous argument, i.e., if the argument was not specified by the user but has its default value.

By default (`affix = TRUE`), the function will add the column to the mapper's input data frame. It returns the input data frame, so reassign its output to that variable.

All of this only works in mapper functions that were "handwritten" using `function()`, as opposed to those produced by `function_map()`. See `vignette("consistency-tests-in-depth")`, section *Writing mappers manually*.

Usage

```
manage_helper_col(data, var_arg, default, affix = TRUE)
```

Arguments

`data`	The data frame that is the mapper function's first argument.
`var_arg`	The argument to the mapper function that has the same name as the helper column you want to manage.
`default`	The default for the argument that was specified in `var_arg`.
`affix`	Logical (length 1). If `data` doesn't include the helper column already, should `var_arg` be added to `data`, bearing its proper name? Default is `TRUE`.

Value

`data`, possibly modified (see `affix` argument).

`manage_key_colnames` *Enable name-independent key column identification*

Description

A handwritten mapper function for consistency tests, such as `grim_map()`, may include arguments named after the key columns in its input data frame. When such an argument is specified by the user as a column name of the input data frame, it identifies a differently-named column as that key column.

Create such functionality in three steps:

1. Add arguments to your mapper function named after the respective key columns. They should be `NULL` by default; e.g., `x = NULL, n = NULL`.
2. Within the mapper, capture the user input by quoting it using `rlang::enexpr()`. Reassign these values to the argument variables; e.g., `x <- rlang::enexpr(x)` and `n <- rlang::enexpr(n)`.
3. For every such argument, call `manage_key_colnames()` and reassign its value to the input data frame variable, adding a short description; e.g.,`data <- manage_key_colnames(data, x, "mean/proportion")` and `data <- manage_key_colnames(data, n, "sample size")`.

Usage

```
manage_key_colnames(data, arg, description = NULL)
```

Arguments

`data`	The mapper function's input data frame.
`arg`	Symbol. The quoted input variable, captured by `rlang::enexpr()`.
`description`	String (length 1). Short description of the column in question, to be inserted into an error message.

Value

The input data frame, `data`, possibly modified.

See Also

`vignette("consistency-tests-in-depth")`, for context.

parens-extractors *Extract substrings from before and inside parentheses*

Description

`before_parens()` and `inside_parens()` extract substrings from before or inside parentheses, or similar separators like brackets or curly braces.

See `split_by_parens()` to split some or all columns in a data frame into both parts.

Usage

```
before_parens(string, sep = "parens")

inside_parens(string, sep = "parens")
```

Arguments

`string`	Vector of strings with parentheses or similar.
`sep`	String. What to split by. Either `"parens"`, `"brackets"`, `"braces"`, or a length-2 vector of custom separators. See examples for `split_by_parens()`. Default is `"parens"`.

Value

String vector of the same length as `string`. The part of `string` before or inside `sep`, respectively.

Examples

```
x <- c(
  "3.72 (0.95)",
  "5.86 (2.75)",
  "3.06 (6.48)"
)

before_parens(string = x)

inside_parens(string = x)
```

pigs1 — *Means and sample sizes for GRIM-testing*

Description

A fictional dataset with means and sample sizes of flying pigs. It can be used to demonstrate the functionality of `grim_map()` and functions building up on it.

Usage

```
pigs1
```

Format

A tibble (data frame) with 12 rows and 2 columns. The columns are:

x String. Means.

n Numeric. Sample sizes.

Value

A tibble (data frame).

See Also

`pigs2` for GRIM-testing percentages instead of means, `pigs3` for DEBIT-testing, and `pigs4` for detecting duplicates.

`pigs2` for GRIM-testing percentages instead of means, `pigs3` for DEBIT-testing, `pigs4` for detecting duplicates, and `pigs5` for GRIMMER-testing.

pigs2 — *Percentages and sample sizes for GRIM-testing*

Description

A fictional dataset with percentages and sample sizes of flying pigs. It can be used to demonstrate the functionality of `grim_map()`, particularly its `percent` argument, and functions building up on it.

Usage

```
pigs2
```

Format

A tibble (data frame) with 6 rows and 2 columns. The columns are:

x String. Percentages.

n Numeric. Sample sizes.

Value

A tibble (data frame).

See Also

`pigs1` for GRIM-testing means instead of percentages, `pigs3` for DEBIT-testing, and `pigs4` for detecting duplicates.

`pigs1` for GRIM-testing means, `pigs3` for DEBIT-testing, `pigs4` for detecting duplicates, and `pigs5` for GRIMMER-testing.

`pigs3` *Binary means and standard deviations for using DEBIT*

Description

A fictional dataset with means and standard deviations from a binary distribution related to flying pigs. It can be used to demonstrate the functionality of `debit_map()` and functions building up on it.

Usage

```
pigs3
```

Format

A tibble (data frame) with 7 rows and 3 columns. The columns are:

x String. Means.

sd String. Standard deviations.

n Numeric. Sample sizes.

Value

A tibble (data frame).

See Also

`pigs1` for GRIM-testing means, `pigs2` for GRIM-testing percentages, and `pigs4` for detecting duplicates.

`pigs1` for GRIM-testing means, `pigs2` for GRIM-testing percentages instead of means, `pigs4` for detecting duplicates, and `pigs5` for GRIMMER-testing.

`pigs4` *Data with duplications*

Description

A fictional dataset with observations of flying pigs. It contains multiple duplicates. The dataset can be used to demonstrate the functionality of `duplicate_*()` functions such as `duplicate_count()`.

Usage

```
pigs4
```

Format

A tibble (data frame) with 5 rows and 3 columns, describing various body measures of the fictional pigs. The columns are:

snout String. Snout width.

tail String. Tail length.

wings String. Wingspan.

Value

A tibble (data frame).

See Also

`pigs1` for GRIM-testing means, `pigs2` for GRIM-testing percentages, `pigs3` for using DEBIT, and `pigs5` for GRIMMER-testing.

`pigs5` *Means, SDs, and sample sizes for GRIMMER-testing*

Description

A fictional dataset with means, standard deviations (SDs), and sample sizes of flying pigs. It can be used to demonstrate the functionality of `grimmer_map()` and functions building up on it.

Usage

```
pigs5
```

Format

A tibble (data frame) with 12 rows and 3 columns. The columns are:

x String. Means.

sd String. Standard deviations.

n Numeric. Sample sizes.

Value

A tibble (data frame).

See Also

`pigs1` for (only) GRIM-testing the same means as here, `pigs2` for GRIM-testing percentages instead of means, `pigs3` for DEBIT-testing, and `pigs4` for detecting duplicates.

`reround` *General interface to reconstructing rounded numbers*

Description

`reround()` takes one or more intermediate reconstructed values and rounds them in some specific way – namely, the way they are supposed to have been rounded originally, in the process that generated the reported values.

This function provides an interface to all of scrutiny's rounding functions as well as `base::round()`. It is used as a helper within `grim()`, `grimmer()`, and `debit()`; and it might find use in other places for consistency testing or reconstruction of statistical analyses.

Usage

```
reround(
  x,
  digits = 0L,
  rounding = "up_or_down",
  threshold = 5,
  symmetric = FALSE
)
```

Arguments

`x`	Numeric. Vector of possibly original values.
`digits`	Integer. Number of decimal places in the reported key values (i.e., mean or percentage within `grim()`, or standard deviation within `grimmer()`).
`rounding`	String. The rounding method that is supposed to have been used originally. See `vignette("rounding-options")`. Default is `"up_or_down"`, which returns two values: `x` rounded up *and* down.

threshold	Integer. If rounding is set to "up_from", "down_from", or "up_from_or_down_from", threshold must be set to the number from which the reconstructed values should then be rounded up or down. Otherwise irrelevant. Default is 5.
symmetric	Logical. Set symmetric to TRUE if the rounding of negative numbers with "up_or_down", "up", "down", "up_from_or_down_from", "up_from", or "down_from" should mirror that of positive numbers so that their absolute values are always equal. Otherwise irrelevant. Default is FALSE.

Details

reround() internally calls the appropriate rounding function(s) determined by the rounding argument. See vignette("rounding-options") for a complete list of values that rounding can take.

For the specific rounding functions themselves, see documentation at round_up(), round_ceiling(), and base::round().

Value

Numeric vector of length 1 or 2. (It has length 1 unless rounding is "up_or_down", "up_from_or_down_from", or"ceiling_or_floor", in which case it has length 2.)

restore_zeros *Restore trailing zeros*

Description

restore_zeros() takes a vector with values that might have lost trailing zeros, most likely from being registered as numeric. It turns each value into a string and adds trailing zeros until the mantissa hits some limit.

The default for that limit is the number of digits in the longest mantissa of the vector's values. The length of the integer part plays no role.

Don't rely on the default limit without checking: The original width could have been larger because the longest extant mantissa might itself have lost trailing zeros.

restore_zeros_df() is a variant for data frames. It wraps restore_zeros() and, by default, applies it to all columns that are coercible to numeric.

Usage

```
restore_zeros(
  x,
  width = NULL,
  sep_in = "\\.",
  sep_out = sep_in,
  sep = deprecated()
)
```

```
restore_zeros_df(
  data,
  cols = everything(),
  check_numeric_like = TRUE,
  check_decimals = FALSE,
  width = NULL,
  sep_in = "\\.",
  sep_out = NULL,
  sep = deprecated(),
  ...
)
```

Arguments

`x`	Numeric (or string coercible to numeric). Vector of numbers that might have lost trailing zeros.
`width`	Integer. Number of decimal places the mantissas should have, including the restored zeros. Default is `NULL`, in which case the number of characters in the longest mantissa will be used instead.
`sep_in`	Substring that separates the input's mantissa from its integer part. Default is `"\\."`, which renders a decimal point.
`sep_out`	Substring that will be returned in the output to separate the mantissa from the integer part. By default, `sep_out` is the same as `sep_in`.
`sep`	[Deprecated] Use `sep_in`, not `sep`. If `sep` is specified nonetheless, `sep_in` takes on `sep`'s value.
`data`	Data frame or matrix. Only in `restore_zeros_df()`, and instead of `x`.
`cols`	Only in `restore_zeros_df()`. Select columns from `data` using tidyselect. Default is `everything()`, which selects all columns that pass the test of `check_numeric_like`.
`check_numeric_like`	Logical. Only in `restore_zeros_df()`. If `TRUE` (the default), the function will skip columns that are not numeric or coercible to numeric, as determined by `is_numeric_like()`.
`check_decimals`	Logical. Only in `restore_zeros_df()`. If set to `TRUE`, the function will skip columns where no values have any decimal places. Default is `FALSE`.
`...`	Only in `restore_zeros_df()`. These dots must be empty.

Details

These functions exploit the fact that groups of summary values such as means or percentages are often reported to the same number of decimal places. If such a number is known but values were not entered as strings, trailing zeros will be lost. In this case, `restore_zeros()` or `restore_zeros_df()` will be helpful to prepare data for consistency testing functions such as `grim_map()` or `grimmer_map()`.

Value

- For `restore_zeros()`, a string vector. At least some of the strings will have newly restored zeros, unless (1) all input values had the same number of decimal places, and (2) `width` was not specified as a number greater than that single number of decimal places.
- For `restore_zeros_df()`, a data frame.

Displaying decimal places

You might not see all decimal places of numeric values in a vector, and consequently wonder if `restore_zeros()`, when applied to the vector, adds too many zeros. That is because displayed numbers, unlike stored numbers, are often rounded.

For a vector `x`, you can count the characters of the longest mantissa from among its values like this:

`x %>% decimal_places() %>% max()`

See Also

Wrapped functions: `sprintf()`.

Examples

```
# By default, the target width is that of
# the longest mantissa:
vec <- c(212, 75.38, 4.9625)
vec %>%
  restore_zeros()

# Alternatively, supply a number via `width`:
vec %>%
  restore_zeros(width = 6)

# For better printing:
iris <- tibble::as_tibble(iris)

# Apply `restore_zeros()` to all numeric
# columns, but not to the factor column:
iris %>%
  restore_zeros_df()

# Select columns as in `dplyr::select()`:
iris %>%
  restore_zeros_df(starts_with("Sepal"), width = 3)
```

`reverse_map_seq` *Reverse the `*_map_seq()` process*

Description

`reverse_map_seq()` takes the output of a function created by `function_map_seq()` and reconstructs the original data frame.

See `audit_seq()`, which takes `reverse_map_seq()` as a basis.

Usage

```
reverse_map_seq(data)
```

Arguments

`data`	Data frame that inherits the `"scr_map_seq"` class.

Value

The reconstructed tibble (data frame) which a factory-made `*_map_seq()` function took as its `data` argument.

Examples

```
# Originally reported summary data...
pigs1

# ...GRIM-tested with varying inputs...
out <- grim_map_seq(pigs1, include_consistent = TRUE)

# ...and faithfully reconstructed:
reverse_map_seq(out)
```

`reverse_map_total_n` *Reverse the* `*_map_total_n()` *process*

Description

`reverse_map_total_n()` takes the output of a function created by `function_map_total_n()` and reconstructs the original data frame.

See `audit_total_n()`, which takes `reverse_map_total_n()` as a basis.

Usage

```
reverse_map_total_n(data)
```

Arguments

`data`	Data frame that inherits the `"scr_map_total_n"` class.

Value

The reconstructed tibble (data frame) which a factory-made `*_map_total_n()` function took as its `data` argument.

Examples

```
# Originally reported summary data...
df <- tibble::tribble(
  ~x1,    ~x2,   ~n,
  "3.43", "5.28", 90,
  "2.97", "4.42", 103
)
df

# ...GRIM-tested with dispersed `n` values...
out <- grim_map_total_n(df)
out

# ...and faithfully reconstructed:
reverse_map_total_n(out)
```

---

rounding-common   *Common rounding procedures*

Description

`round_up()` rounds up from 5, `round_down()` rounds down from 5. Otherwise, both functions work like `base::round()`.

`round_up()` and `round_down()` are special cases of `round_up_from()` and `round_down_from()`, which allow users to choose custom thresholds for rounding up or down, respectively.

Usage

```
round_up_from(x, digits = 0L, threshold, symmetric = FALSE)

round_down_from(x, digits = 0L, threshold, symmetric = FALSE)

round_up(x, digits = 0L, symmetric = FALSE)

round_down(x, digits = 0L, symmetric = FALSE)
```

Arguments

`x`	Numeric. The decimal number to round.
`digits`	Integer. Number of digits to round `x` to. Default is `0`.
`threshold`	Integer. Only in `round_up_from()` and `round_down_from()`. Threshold for rounding up or down, respectively. Value is 5 in `round_up()`'s internal call to `round_up_from()` and in `round_down()`'s internal call to `round_down_from()`.

`symmetric`	Logical. Set `symmetric` to `TRUE` if the rounding of negative numbers should mirror that of positive numbers so that their absolute values are equal. Default is `FALSE`.

Details

These functions differ from `base::round()` mainly insofar as the decision about rounding 5 up or down is not based on the integer portion of `x` (i.e., no "rounding to even"). Instead, in `round_up_from()`, that decision is determined by the `threshold` argument for rounding up, and likewise with `round_down_from()`. The threshold is constant at 5 for `round_up()` and `round_down()`.

As a result, these functions are more predictable and less prone to floating-point number quirks than `base::round()`. Compare `round_down()` and `base::round()` in the data frame for rounding 5 created in the Examples section below: `round_down()` yields a continuous sequence of final digits from 0 to 9, whereas `base::round()` behaves in a way that can only be explained by floating point issues.

However, this surprising behavior on the part of `base::round()` is not necessarily a flaw (see its documentation, or this vignette: https://rpubs.com/maechler/Rounding). In the present version of R (4.0.0 or later), `base::round()` works fine, and the functions presented here are not meant to replace it. Their main purpose as helpers within scrutiny is to reconstruct the computations of researchers who might have used different software. See `vignette("rounding-options")`.

Value

Numeric. `x` rounded to `digits`.

See Also

`round_ceiling()` always rounds up, `round_floor()` always rounds down, `round_trunc()` always rounds toward 0, and `round_anti_trunc()` always round away from 0.

Examples

```
# Both `round_up()` and `round_down()` work like
# `base::round()` unless the closest digit to be
# cut off by rounding is 5:

   round_up(x = 9.273, digits = 1)     # 7 cut off
 round_down(x = 9.273, digits = 1)     # 7 cut off
base::round(x = 9.273, digits = 1)     # 7 cut off

   round_up(x = 7.584, digits = 2)     # 4 cut off
 round_down(x = 7.584, digits = 2)     # 4 cut off
base::round(x = 7.584, digits = 2)     # 4 cut off


# Here is the borderline case of 5 rounded by
# `round_up()`, `round_down()`, and `base::round()`:

original <- c(    # Define example values
  0.05, 0.15, 0.25, 0.35, 0.45,
  0.55, 0.65, 0.75, 0.85, 0.95
)
tibble::tibble(   # Output table
  original,
  round_up = round_up(x = original, digits = 1),
  round_down = round_down(x = original, digits = 1),
  base_round = base::round(x = original, digits = 1)
)

# (Note: Defining `original` as `seq(0.05:0.95, by = 0.1)`
# would lead to wrong results unless `original` is rounded
# to 2 or so digits before it's rounded to 1.)
```

rounding-uncommon *Uncommon rounding procedures*

Description

Always round up, down, toward zero, or away from it:

- `round_ceiling()` always rounds up.
- `round_floor()` always rounds down.
- `round_trunc()` always rounds toward zero.
- `round_anti_trunc()` always rounds away from zero. (`0` itself is rounded to `1`.)
- `anti_trunc()` does not round but otherwise works like `round_anti_trunc()`.

Despite not being widely used, they are featured here in case they are needed for reconstruction.

Usage

```
round_ceiling(x, digits = 0L)

round_floor(x, digits = 0L)

round_trunc(x, digits = 0L)

anti_trunc(x)

round_anti_trunc(x, digits = 0L)
```

Arguments

x	Numeric. The decimal number to round.
digits	Integer. Number of digits to round `x` to. Default is `0`.

Details

round_ceiling(), round_floor(), and round_trunc() generalize the base R functions ceiling(), floor(), and trunc(), and include them as special cases: With the default value for digits, 0, these round_* functions are equivalent to their respective base counterparts.

The last round_* function, round_anti_trunc(), generalizes another function presented here: anti_trunc() works like trunc() except it moves away from 0, rather than towards it. That is, whereas trunc() minimizes the absolute value of x (as compared to the other rounding functions), anti_trunc() maximizes it. anti_trunc(x) is therefore equal to trunc(x) + 1 if x is positive, and to trunc(x) - 1 if x is negative.

round_anti_trunc(), then, generalizes anti_trunc() just as round_ceiling() generalizes ceiling(), etc.

Moreover, round_trunc() is equivalent to round_floor() for positive numbers and to round_ceiling() for negative numbers. The reverse is again true for round_anti_trunc(): It is equivalent to round_ceiling() for positive numbers and to round_floor() for negative numbers.

Value

Numeric. x rounded to digits (except for anti_trunc(), which has no digits argument).

See Also

round_up() and round_down() round up or down from 5, respectively. round_up_from() and round_down_from() allow users to specify custom thresholds for rounding up or down.

Examples

```
# Always round up:
round_ceiling(x = 4.52, digits = 1)        # 2 cut off

# Always round down:
round_floor(x = 4.67, digits = 1)          # 7 cut off

# Always round toward 0:
round_trunc(8.439, digits = 2)             # 9 cut off
round_trunc(-8.439, digits = 2)            # 9 cut off

# Always round away from 0:
round_anti_trunc(x = 8.421, digits = 2)    # 1 cut off
round_anti_trunc(x = -8.421, digits = 2)   # 1 cut off
```

---

rounding_bias *Compute rounding bias*

Description

Rounding often leads to bias, such that the mean of a rounded distribution is different from the mean of the original distribution. Call rounding_bias() to compute the amount of this bias.

Usage

```
rounding_bias(
  x,
  digits,
  rounding = "up",
  threshold = 5,
  symmetric = FALSE,
  mean = TRUE
)
```

Arguments

`x`	Numeric or string coercible to numeric.
`digits`	Integer. Number of decimal digits to which `x` will be rounded.
`rounding`	String. Rounding procedure that will be applied to `x`. See `vignette("rounding-options")`. Default is `"up"`.
`threshold, symmetric`	Further arguments passed down to `reround()`.
`mean`	Logical. If `TRUE` (the default), the mean total of bias will be returned. Set `mean` to `FALSE` to get a vector of individual biases the length of `x`.

Details

Bias is calculated by subtracting the original vector, `x`, from a vector rounded in the specified way.

The function passes all arguments except for `mean` down to `reround()`. Other than there, however, `rounding` is `"up"` by default, and it can't be set to `"up_or_down"`, `"up_from_or_down_from"`, or`"ceiling_or_floor"`.

Value

Numeric. By default of `mean`, the length is 1; otherwise, it is the same length as `x`.

Examples

```
# Define example vector:
vec <- seq_distance(0.01, string_output = FALSE)
vec

# The default rounds `x` up from 5:
rounding_bias(x = vec, digits = 1)

# Other rounding procedures are supported,
# such as rounding down from 5...
rounding_bias(x = vec, digits = 1, rounding = "down")

# ...or rounding to even with `base::round()`:
rounding_bias(x = vec, digits = 1, rounding = "even")
```

`row_to_colnames` *Turn row values into column names*

Description

Data frames sometimes have wrong column names, while the correct column names are stored in one or more rows in the data frame itself. To remedy this issue, call `row_to_colnames()` on the data frame: It replaces the column names by the values of the specified rows (by default, only the first one). These rows are then dropped by default.

Usage

```
row_to_colnames(data, row = 1L, collapse = " ", drop = TRUE)
```

Arguments

`data`	Data frame or matrix.
`row`	Integer. Position of the rows (one or more) that jointly contain the correct column names. Default is `1`.
`collapse`	String. If the length of `row` is greater than 1, each new column name will be that many row values pasted together. `collapse`, then, is the substring between two former row values in the final column names. Default is `" "` (a space).
`drop`	Logical. If `TRUE` (the default), the rows specified with `row` are removed.

Details

If multiple rows are specified, the row values for each individual column are pasted together. Some special characters might then be missing.

This function might be useful when importing tables from PDF, e.g. with tabulizer. In R, these data frames (converted from matrices) do sometimes have the issue described above.

Value

A tibble (data frame).

See Also

`unheadr::mash_colnames()`, a more sophisticated solution to the same problem.

sd-binary *Standard deviation of binary data*

Description

Compute the sample SD of binary data (i.e., only 0 and 1 values) in either of four ways, each based on different inputs:

- `sd_binary_groups()` takes the cell sizes of both groups, those coded as 0 and those coded as 1.
- `sd_binary_0_n()` takes the cell size of the group coded as 0 and the total sample size.
- `sd_binary_1_n()` takes the cell size of the group coded as 1 and the total sample size.
- `sd_binary_mean_n()` takes the mean and the total sample size.

These functions are used as helpers inside `debit()`, and consequently `debit_map()`.

Usage

```
sd_binary_groups(group_0, group_1)

sd_binary_0_n(group_0, n)

sd_binary_1_n(group_1, n)

sd_binary_mean_n(mean, n)
```

Arguments

`group_0`	Integer. Cell size of the group coded as 0.
`group_1`	Integer. Cell size of the group coded as 1.
`n`	Integer. Total sample size.
`mean`	Numeric. Mean of the binary data.

Value

Numeric. Sample standard deviation.

References

Heathers, James A. J., and Brown, Nicholas J. L. 2019. DEBIT: A Simple Consistency Test For Binary Data. https://osf.io/5vb3u/.

See Also

`is_subset_of_vals(x, 0, 1)` checks whether `x` (a list or atomic vector) contains nothing but binary numbers.

Examples

```
# If 127 values are coded as 0 and 153 as 1...
sd_binary_groups(group_0 = 127, group_1 = 153)

# ...so that n = 280:
sd_binary_0_n(group_0 = 127, n = 280)
sd_binary_1_n(group_1 = 153, n = 280)

# If only the mean and total sample size are
# given, or these are more convenient to use,
# they still lead to the same result as above
# if the mean is given with a sufficiently
# large number of decimal places:
sd_binary_mean_n(mean = 0.5464286, n = 280)
```

`seq-decimal` *Sequence generation at decimal level*

Description

Functions that provide a smooth interface to generating sequences based on the input values' decimal depth. Each function creates a sequence with a step size of one unit on the level of the input values' ultimate decimal digit (e.g., `2.45`, `2.46`, `2.47`, ...):

- `seq_endpoint()` creates a sequence from one input value to another. For step size, it goes by the value with more decimal places.
- `seq_distance()` only takes the starting point and, instead of the endpoint, the desired output length. For step size, it goes by the starting point by default.

`seq_endpoint_df()` and `seq_distance_df()` are variants that create a data frame. Further columns can be added as in `tibble::tibble()`. Regular arguments are the same as in the respective non-df function, but with a dot before each.

Usage

```
seq_endpoint(from, to, offset_from = 0L, offset_to = 0L, string_output = TRUE)

seq_distance(
  from,
  by = NULL,
  length_out = 10L,
  dir = 1,
  offset_from = 0L,
  string_output = TRUE
)

seq_endpoint_df(
  .from,
  .to,
  ...,
  .offset_from = 0L,
  .offset_to = 0L,
  .string_output = TRUE
)

seq_distance_df(
  .from,
  .by = NULL,
  ...,
  .length_out = 10L,
  .dir = 1,
  .offset_from = 0L,
  .string_output = TRUE
)
```

Arguments

`from`, `.from`	Numeric (or string coercible to numeric). Starting point of the sequence.
`to`, `.to`	Numeric (or string coercible to numeric). Endpoint of the sequence. Only in `seq_endpoint()` and `seq_endpoint_df()`.
`offset_from`, `.offset_from`	Integer. If set to a non-zero number, the starting point will be offset by that many units on the level of the last decimal digit. Default is `0`.
`offset_to`, `.offset_to`	Integer. If set to a non-zero number, the endpoint will be offset by that many units on the level of the last decimal digit. Default is `0`. Only in `seq_endpoint()` and `seq_endpoint_df()`.
`string_output`, `.string_output`	Logical or string. If `TRUE` (the default), the output is a string vector. Decimal places are then padded with zeros to match `from`'s (or `to`'s) number of decimal places. `"auto"` works like `TRUE` if and only if `from` (`.from`) is a string.
`by`, `.by`	Numeric. Only in `seq_distance()` and `seq_distance_df()`. Step size of the sequence. If not set, inferred automatically. Default is `NULL`.
`length_out`, `.length_out`	Integer. Length of the output vector (i.e., the number of its values). Default is `10`. Only in `seq_distance()` and `seq_distance_df()`.
`dir`, `.dir`	Integer. If set to `-1`, the sequence goes backward. Default is `1`. Only in `seq_distance()` and `seq_distance_df()`.
`...`	Further columns, added as in `tibble::tibble()`. Only in `seq_endpoint_df()` and `seq_distance_df()`.

Details

If either `from` or `to` ends on zero, be sure to enter that value as a string! This is crucial because trailing zeros get dropped from numeric values. A handy way to format numeric values or number-strings correctly is `restore_zeros()`. The output of the present functions is like that by default (of `string_output`).

In `seq_endpoint()` and `seq_endpoint_df()`, the step size is determined by `from` and `to`, whichever has more decimal places. In `seq_distance()` and `seq_distance_df()`, it's determined by the decimal places of `from`.

These functions are scrutiny's take on `base::seq()`, and themselves wrappers around it.

Value

String by default of `string_output`, numeric otherwise.

See Also

`seq_disperse()` for sequences centered around the input.

Examples

```
# Sequence between two points:
seq_endpoint(from = 4.7, to = 5)

# Sequence of some length; default is 10:
seq_distance(from = 0.93)
seq_distance(from = 0.93, length_out = 5)

# Both of these functions can offset the
# starting point...
seq_endpoint(from = 14.2, to = 15, offset_from = 4)
seq_distance(from = 14.2, offset_from = 4)

# ...but only `seq_endpoint()` can offset the
# endpoint, because of its `to` argument:
seq_endpoint(from = 9.5, to = 10, offset_to = 2)

# In return, `seq_distance()` can reverse its direction:
seq_distance(from = 20.03, dir = -1)

# Both functions have a `_df` variant that returns
# a data frame. Arguments are the same but with a
# dot, and further columns can be added as in
# `tibble::tibble()`:
seq_endpoint_df(.from = 4.7, .to = 5, n = 20)
seq_distance_df(.from = 0.43, .length_out = 5, sd = 0.08)
```

seq-predicates *Is a vector a certain kind of sequence?*

Description

Predicate functions that test whether `x` is a numeric vector (or coercible to numeric) with some special properties:

- `is_seq_linear()` tests whether every two consecutive elements of `x` differ by some constant amount.
- `is_seq_ascending()` and `is_seq_descending()` test whether the difference between every two consecutive values is positive or negative, respectively. `is_seq_dispersed()` tests whether `x` values are grouped around a specific central value, `from`, with the same distance to both sides per value pair. By default (`test_linear = TRUE`), these functions also test for linearity, like `is_seq_linear()`.

`NA` elements of `x` are handled in a nuanced way. See *Value* section below and the examples in `vignette("devtools")`, section *NA handling*.

Usage

```
is_seq_linear(x, tolerance = .Machine$double.eps^0.5)

is_seq_ascending(x, test_linear = TRUE, tolerance = .Machine$double.eps^0.5)

is_seq_descending(x, test_linear = TRUE, tolerance = .Machine$double.eps^0.5)

is_seq_dispersed(
  x,
  from,
  test_linear = TRUE,
  tolerance = .Machine$double.eps^0.5
)
```

Arguments

`x`	Numeric or coercible to numeric, as determined by `is_numeric_like()`. Vector to be tested.
`tolerance`	Numeric. Tolerance of comparison between numbers when testing. Default is circa 0.000000015 (1.490116e-08), as in `dplyr::near()`.
`test_linear`	Logical. In functions other than `is_seq_linear()`, should `x` also be tested for linearity? Default is `TRUE`.
`from`	Numeric or coercible to numeric. Only in `is_seq_dispersed()`. It will test whether `from` is at the center of `x`, and if every pair of other values is equidistant to it.

Value

A single logical value. If `x` contains at least one `NA` element, the functions return either `NA` or `FALSE`:

- If all elements of `x` are `NA`, the functions return `NA`.

- If some but not all elements are NA, they check if x *might* be a sequence of the kind in question: Is it a linear (and / or ascending, etc.) sequence after the NAs were replaced by appropriate values? If so, they return NA; otherwise, they return FALSE.

See Also

validate::is_linear_sequence(), which is much like is_seq_linear() but more permissive with NA values. It comes with some additional features, such as support for date-times.

Examples

```
# These are linear sequences...
is_seq_linear(x = 3:7)
is_seq_linear(x = c(3:7, 8))

# ...but these aren't:
is_seq_linear(x = c(3:7, 9))
is_seq_linear(x = c(10, 3:7))

# All other `is_seq_*()` functions
# also test for linearity by default:
is_seq_ascending(x = c(2, 7, 9))
is_seq_ascending(x = c(2, 7, 9), test_linear = FALSE)

is_seq_descending(x = c(9, 7, 2))
is_seq_descending(x = c(9, 7, 2), test_linear = FALSE)

is_seq_dispersed(x = c(2, 3, 5, 7, 8), from = 5)
is_seq_dispersed(x = c(2, 3, 5, 7, 8), from = 5, test_linear = FALSE)

# These fail their respective
# individual test even
# without linearity testing:
is_seq_ascending(x = c(1, 7, 4), test_linear = FALSE)
is_seq_descending(x = c(9, 15, 3), test_linear = FALSE)
is_seq_dispersed(1:10, from = 5, test_linear = FALSE)
```

---

seq_disperse *Sequence generation with dispersion at decimal level*

Description

seq_disperse() creates a sequence around a given number. It goes a specified number of steps up and down from it. Step size depends on the number's decimal places. For example, 7.93 will be surrounded by values like 7.91, 7.92, and 7.94, 7.95, etc.

seq_disperse_df() is a variant that creates a data frame. Further columns can be added as in tibble::tibble(). Regular arguments are the same as in seq_disperse(), but with a dot before each.

Usage

```
seq_disperse(
  from,
  by = NULL,
  dispersion = 1:5,
  offset_from = 0L,
  out_min = "auto",
  out_max = NULL,
  string_output = TRUE,
  include_reported = TRUE,
  track_diff_var = FALSE,
  track_var_change = deprecated()
)

seq_disperse_df(
  .from,
  .by = NULL,
  ...,
  .dispersion = 1:5,
  .offset_from = 0L,
  .out_min = "auto",
  .out_max = NULL,
  .string_output = TRUE,
  .include_reported = TRUE,
  .track_diff_var = FALSE,
  .track_var_change = FALSE
)
```

Arguments

`from`, `.from`	Numeric (or string coercible to numeric). Starting point of the sequence.
`by`, `.by`	Numeric. Step size of the sequence. If not set, inferred automatically. Default is `NULL`.
`dispersion`, `.dispersion`	Numeric. Vector that determines the steps up and down, starting at `from` (or `.from`, respectively) and proceeding on the level of its last decimal place. Default is `1:5`, i.e., five steps up and down.
`offset_from`, `.offset_from`	Integer. If set to a non-zero number, the starting point will be offset by that many units on the level of the last decimal digit. Default is `0`.
`out_min`, `.out_min`, `out_max`, `.out_max`	If specified, output will be restricted so that it's not below `out_min` or above `out_max`. Defaults are `"auto"` for `out_min`, i.e., a minimum of one decimal unit above zero; and `NULL` for `out_max`, i.e., no maximum.
`string_output`, `.string_output`	Logical or string. If `TRUE` (the default), the output is a string vector. Decimal places are then padded with zeros to match `from`'s number of decimal places. `"auto"` works like `TRUE` if and only if `from` (`.from`) is a string.

`include_reported, .include_reported`
: Logical. Should `from` (`.from`) itself be part of the sequence built around it? Default is `TRUE` for the sake of continuity, but this can be misleading if the focus is on the dispersed values, as opposed to the input.

`track_diff_var, .track_diff_var`
: Logical. In `seq_disperse()`, ignore this argument. In `seq_disperse_df()`, default is `TRUE`, which creates the `"diff_var"` output column.

`track_var_change, .track_var_change`
: **[Deprecated]** Renamed to `track_diff_var` / `.track_diff_var`.

`...`
: Further columns, added as in `tibble::tibble()`. Only in `seq_disperse_df()`.

Details

Unlike `seq_endpoint()` and friends, the present functions don't necessarily return continuous or even regular sequences. The greater flexibility is due to the `dispersion` (`.dispersion`) argument, which takes any numeric vector. By default, however, the output sequence is regular and continuous.

Underlying this difference is the fact that `seq_disperse()` and `seq_disperse_df()` do not wrap around `base::seq()`, although they are otherwise similar to `seq_endpoint()` and friends.

Value

- `seq_disperse()` returns a string vector by default (`string_output` = `TRUE`) and a numeric vector otherwise.
- `seq_disperse_df()` returns a tibble (data frame). The sequence is stored in the `x` column. `x` is string by default (`.string_output` = `TRUE`), numeric otherwise. Other columns might have been added via the dots (`...`).

See Also

Conceptually, `seq_disperse()` is a blend of two function families: those around `seq_endpoint()` and those around `disperse()`. The present functions were originally conceived for `seq_disperse_df()` to be a helper within the `function_map_seq()` implementation.

Examples

```
# Basic usage:
seq_disperse(from = 4.02)

# If trailing zeros don't matter,
# the output can be numeric:
seq_disperse(from = 4.02, string_output = FALSE)

# Control steps up and down with
# `dispersion` (default is `1:5`):
seq_disperse(from = 4.02, dispersion = 1:10)

# Sequences might be discontinuous...
disp1 <- seq(from = 2, to = 10, by = 2)
seq_disperse(from = 4.02, dispersion = disp1)
```

```
# ...or even irregular:
disp2 <- c(2, 3, 7)
seq_disperse(from = 4.02, dispersion = disp2)

# The data fame variant supports further
# columns added as in `tibble::tibble()`:
seq_disperse_df(.from = 4.02, n = 45)
```

---

seq_length *Set sequence length*

Description

`seq_length()` seamlessly extends or shortens a linear sequence using the sequence's own step size.

Alternatively, you can directly set the length of a linear sequence in this way: `seq_length(x) <- value`.

Usage

```
seq_length(x, value)

seq_length(x) <- value
```

Arguments

`x`	Numeric or coercible to numeric. `x` must be linear, i.e., each of its elements must differ from the next by the same amount.
`value`	Numeric (whole number, length 1). The new length for `x`.

Value

A vector of the same type as `x`, with length `value`.

- If `value` > `length(x)`, all original element of `x` are preserved. A number of new elements equal to the difference is appended at the end.
- If `value` == `length(x)`, nothing changes.
- If `value` < `length(x)`, a number of elements of `x` equal to the difference is removed from the end.

Examples

```
x <- 3:7

# Increase the length of `x` from 5 to 10:
seq_length(x, 10)

# Modify `x` directly (but get
# the same results otherwise):
seq_length(x) <- 10
x

# Likewise, decrease the length:
x <- 3:7
seq_length(x, 2)

seq_length(x) <- 2
x

# The functions are sensitive to decimal levels.
# They also return a string vector if (and only if)
# `x` is a string vector:
x <- seq_endpoint(from = 0, to = 0.5)
x

seq_length(x, 10)

seq_length(x) <- 10
x

# Same with decreasing the length:
seq_length(x, 2)

seq_length(x) <- 2
x
```

seq_test_ranking *Rank sequence test results*

Description

Run this function after generating a sequence with `seq_endpoint_df()` or `seq_distance_df()` and testing it with one of scrutiny's mapping functions, such as `grim_map()`. It will rank the test's consistent and inconsistent results by their positions in the sequence.

Usage

```
seq_test_ranking(x, explain = TRUE)
```

Arguments

`x`	Data frame.
`explain`	If `TRUE` (the default), results come with an explanation.

Details

The function checks the provenance of the test results and throws a warning if it's not correct.

Value

A tibble (data frame). The function will also print an explanation of the results. See examples.

Examples

```
seq_distance_df(.from = "0.00", n = 50) %>%
  grim_map() %>%
  seq_test_ranking()
```

---

split_by_parens *Split columns by parentheses, brackets, braces, or similar*

Description

Summary statistics are often presented like `"2.65 (0.27)"`. When working with tables copied into R, it can be tedious to separate values before and inside parentheses. `split_by_parens()` does this automatically.

By default, it operates on all columns. Output can optionally be pivoted into a longer format by setting `transform` to `TRUE`.

Choose separators other than parentheses with the `sep` argument.

Usage

```
split_by_parens(
  data,
  cols = everything(),
  check_sep = TRUE,
  keep = FALSE,
  transform = FALSE,
  sep = "parens",
  end1 = "x",
  end2 = "sd",
  ...
)
```

Arguments

`data`	Data frame.
`cols`	Select columns from `data` using tidyselect. Default is `everything()`, which selects all columns that pass `check_sep`.
`check_sep`	Logical. If `TRUE` (the default), columns are excluded if they don't contain the `sep` elements.
`keep`	Logical. If set to `TRUE`, the originally selected columns that were split by the function also appear in the output. Default is `FALSE`.
`transform`	Logical. If set to `TRUE`, the output will be pivoted to be better suitable for typical follow-up tasks. Default is `FALSE`.

`sep`	String. What to split by. Either `"parens"`, `"brackets"`, or `"braces"`; or a length-2 vector of custom separators (see Examples). Default is `"parens"`.
`end1, end2`	Strings. Endings of the two column names that result from splitting a column. Default is `"x"` for `end1` and `"sd"` for `end2`.
`...`	These dots must be empty.

Value

Data frame.

See Also

- `before_parens()` and `inside_parens()` take a string vector and extract values from the respective position.
- `dplyr::across()` powers the application of the two above functions within split_by_parens()‘, including the creation of new columns.
- `tidyr::separate_wider_delim()` is a more general function, but it does not recognize closing elements such as closed parentheses.

Examples

```
# Call `split_by_parens()` on data like these:
df1 <- tibble::tribble(
  ~drone,           ~selfpilot,
  "0.09 (0.21)",    "0.19 (0.13)",
  "0.19 (0.28)",    "0.53 (0.10)",
  "0.62 (0.16)",    "0.50 (0.11)",
  "0.15 (0.35)",    "0.57 (0.16)",
)

# Basic usage:
df1 %>%
  split_by_parens()

# Name specific columns with `cols` to only split those:
df1 %>%
  split_by_parens(cols = drone)

# Pivot the data into a longer format
# by setting `transform` to `TRUE`:
df1 %>%
  split_by_parens(transform = TRUE)

# Choose different column names or
# name suffixes with `end1` and `end2`:
df1 %>%
  split_by_parens(end1 = "beta", end2 = "se")

df1 %>%
  split_by_parens(
    transform = TRUE,
    end1 = "beta", end2 = "se"
  )

# With a different separator...
df2 <- tibble::tribble(
  ~drone,           ~selfpilot,
  "0.09 [0.21]",    "0.19 [0.13]",
  "0.19 [0.28]",    "0.53 [0.10]",
  "0.62 [0.16]",    "0.50 [0.11]",
  "0.15 [0.35]",    "0.57 [0.16]",
)

# ... specify `sep`:
df2 %>%
  split_by_parens(sep = "brackets")

# (Accordingly with `{}` and `"braces"`.)

# If the separator is yet a different one...
df3 <- tibble::tribble(
  ~drone,           ~selfpilot,
  "0.09 <0.21>",    "0.19 <0.13>",
  "0.19 <0.28>",    "0.53 <0.10>",
  "0.62 <0.16>",    "0.50 <0.11>",
  "0.15 <0.35>",    "0.57 <0.16>",
)

# ... `sep` should be a length-2 vector
# that contains the separating elements:
df3 %>%
  split_by_parens(sep = c("<", ">"))
```

subset-superset *Test for subsets, supersets, and equal sets*

Description

Predicate functions that take a vector and test whether it has some particular relation to another vector. That second vector is entered in either of three ways –

Enter it directly (basic functions):

`is_subset_of()` tests if a vector is a subset of another vector; i.e., if all its elements are contained in the second one. `is_superset_of()` does the reverse: It tests if the first vector contains all elements of the second one. `is_equal_set()` tests if both vectors have exactly the same values.

Enter its values:

`is_subset_of_vals()`, `is_superset_of_vals()`, and `is_equal_set_vals()` are variants that each take a single vector plus any number of other arguments. These are treated like elements of the second vector in the basic functions above.

Enter multiple vectors that jointly contain its values:

Finally, `is_subset_of_vecs()`, `is_superset_of_vecs()`, and `is_equal_set_vecs()` take one vector plus any number of other vectors and treat their elements (!) like elements of a second vector in the basic functions above.

Each `is_subset*()` function has an `is_proper_subset*()` variant. These variants also test whether the sets are unequal, so that `x` is a subset of `y` but `y` is not a subset of `x`. The same applies to `is_superset*()` functions and their `is_proper_superset*()` variants.

Usage

```
is_subset_of(x, y)

is_superset_of(x, y)

is_equal_set(x, y)

is_proper_subset_of(x, y)

is_proper_superset_of(x, y)

is_subset_of_vals(x, ...)

is_superset_of_vals(x, ...)

is_equal_set_vals(x, ...)

is_proper_subset_of_vals(x, ...)

is_proper_superset_of_vals(x, ...)

is_subset_of_vecs(x, ...)

is_superset_of_vecs(x, ...)

is_equal_set_vecs(x, ...)

is_proper_subset_of_vecs(x, ...)

is_proper_superset_of_vecs(x, ...)
```

Arguments

`x`	A vector.
`y`	A vector. Only in the basic functions, not those with `*_vals()` or `*_vecs()`.
`...`	In the `*_vals()` functions, any number of values `x` might contain; in the `*_vecs()` functions, any number of vectors the elements of which `x` might contain.

Details

The `*_vals()` variants are meant for flexible, interactive subset/superset testing. That is, in order to test whether certain values collectively fulfill the role of the second vector, you can just add them to the function call.

The `*_vecs()` variants likewise offer flexibility, but also bridge the gap between vectors and values contained in them.

All functions simply check if values are present, regardless of how often a value occurs. In other words, they look for types but don't count tokens.

Value

A single logical value. `TRUE` if the respective test was passed, `FALSE` otherwise.

Examples

```
# Define example vectors:
ab <- c("a", "b")
abc <- c("a", "b", "c")
abcde <- c("a", "b", "c", "d", "e")

# `is_subset_of()` tests if a vector is
# completely covered by another one:
abc %>% is_subset_of(ab)
abc %>% is_subset_of(abc)
abc %>% is_subset_of(abcde)

# To the contrary, `is_superset_of()` tests if the
# first vector completely covers the second one:
abc %>% is_superset_of(ab)
abc %>% is_superset_of(abc)
abc %>% is_superset_of(abcde)

# `is_equal_set()` tests both of the above --
# i.e., if both vectors have exactly the
# same values:
abc %>% is_equal_set(ab)
abc %>% is_equal_set(abc)
abc %>% is_equal_set(abcde)

# Each of the three functions has a `*_vals()` variant
# that doesn't take a second vector like the first
# one, but any number of other arguments. These are
# jointly treated like the elements of the second
# vector in the basic functions:
abc %>% is_subset_of_vals("a", "b")
abc %>% is_subset_of_vals("a", "b", "c")
abc %>% is_subset_of_vals("a", "b", "c", "d", "e")

# (... and likewise for supersets and equal sets.)
```

unnest_consistency_cols

Unnest a test result column

Description

Within a consistency test mapper function, it may become necessary to unpack a column resulting from a basic `*_scalar()` testing function. That will be the case if a `show_*` argument of the mapper function like `show_rec` in `grim_map()` is `TRUE`, and the `*_scalar()` function returns a list of values, not just a single value.

At the point where such as list is stored in a data frame column (most likely `"consistency"`), call `unnest_consistency_cols()` to unnest the results into multiple columns.

Usage

```
unnest_consistency_cols(results, col_names, index = FALSE, col = "consistency")
```

Arguments

`results`	Data frame containing a list-column by the name passed to `col`.
`col_names`	String vector of new names for the unnested columns. It should start with the same string that was given for `col`.
`index`	Logical. Should the list-column be indexed into? Default is `FALSE`.
`col`	String (length 1). Name of the list-column within `results` to operate on. Default is `"consistency"`.

Details

This function is a custom workaround in place of `tidyr::unnest_wider()`, mirroring some of the latter's functionality. It was created because `unnest_wider()` can be too slow for use as a helper function.

Value

Data frame. The column names are determined by `col_names`.

See Also

`vignette("consistency-tests-in-depth")`, for context.

`unround` *Reconstruct rounding bounds*

Description

`unround()` takes a rounded number and returns the range of the original value: lower and upper bounds for the hypothetical earlier number that was later rounded to the input number. It also displays a range with inequation signs, showing whether the bounds are inclusive or not.

By default, the presumed rounding method is rounding up (or down) from 5. See the `Rounding` section for other methods.

Usage

```
unround(x, rounding = "up_or_down", threshold = 5, digits = NULL)
```

Arguments

`x`	String or numeric. Rounded number. `x` must be a string unless `digits` is specified (most likely by a function that uses `unround()` as a helper).
`rounding`	String. Rounding method presumably used to create `x`. Default is `"up_or_down"`. For more, see section `Rounding`.
`threshold`	Integer. Number from which to round up or down. Other rounding methods are not affected. Default is `5`.
`digits`	Integer. This argument is meant to make `unround()` more efficient to use as a helper function so that it doesn't need to redundantly count decimal places. Don't specify it otherwise. Default is `NULL`, in which case decimal places really are counted internally and `x` must be a string.

Details

The function is vectorized over `x` and `rounding`. This can be useful to unround multiple numbers at once, or to check how a single number is unrounded with different assumed rounding methods.

If both vectors have a length greater than 1, it must be the same length. However, this will pair numbers with rounding methods, which can be confusing. It is recommended that at least one of these input vectors has length 1.

Why does `x` need to be a string if `digits` is not specified? In that case, `unround()` must count decimal places by itself. If `x` then was numeric, it wouldn't have any trailing zeros because these get dropped from numerics.

Trailing zeros are as important for reconstructing boundary values as any other trailing digits would be. Strings don't drop trailing zeros, so they are used instead.

Value

A tibble with seven columns: `range`, `rounding`, `lower`, `incl_lower`, `x`, `incl_upper`, and `upper`. The `range` column is a handy representation of the information stored in the columns from `lower` to `upper`, in the same order.

Rounding

Depending on how x was rounded, the boundary values can be inclusive or exclusive. The `incl_lower` and `incl_upper` columns in the resulting tibble are `TRUE` in the first case and `FALSE` in the second. The `range` column reflects this with equation and inequation signs.

However, these ranges are based on assumptions about the way x was rounded. Set `rounding` to the rounding method that hypothetically lead to x:

Value of `rounding`	**Corresponding range**
`"up_or_down"` (default)	`lower <= x <= upper`
`"up"`	`lower <= x < upper`
`"down"`	`lower < x <= upper`
`"even"`	(no fix range)
`"ceiling"`	`lower < x = upper`
`"floor"`	`lower = x < upper`
`"trunc"` (positive x)	`lower = x < upper`
`"trunc"` (negative x)	`lower < x = upper`
`"trunc"` (zero x)	`lower < x < upper`
`"anti_trunc"` (positive x)	`lower < x = upper`
`"anti_trunc"` (negative x)	`lower = x < upper`
`"anti_trunc"` (zero x)	(undefined; `NA`)

Base R's own `round()` (R version >= 4.0.0), referenced by `rounding = "even"`, is reconstructed in the same way as `"up_or_down"`, but whether the boundary values are inclusive or not is hard to predict. Therefore, `unround()` checks if they are, and informs you about it.

See Also

For more about rounding `"up"`, `"down"`, or to `"even"`, see `round_up()`.

For more about the less likely `rounding` methods, `"ceiling"`, `"floor"`, `"trunc"`, and `"anti_trunc"`, see `round_ceiling()`.

Examples

```
# By default, the function assumes that `x`
# was either rounded up or down:
unround(x = "2.7")

# If `x` was rounded up, run this:
unround(x = "2.7", rounding = "up")

# Likewise with rounding down...
unround(x = "2.7", rounding = "down")

# ...and with `base::round()` which, broadly
# speaking, rounds to the nearest even number:
unround(x = "2.7", rounding = "even")

# Multiple input number-strings return
# multiple rows in the output data frame:
unround(x = c(3.6, "5.20", 5.174))
```

write_doc_audit *Documentation template for* audit()

Description

write_doc_audit() creates a roxygen2 block section to be inserted into the documentation of a mapper function such as grim_map() or debit_map(): functions for which there are, or should be, audit() methods. The section informs users about the ways in which audit() summarizes the results of the respective mapper function.

Copy the output from your console and paste it into the roxygen2 block of your *_map() function. To preserve the numbered list structure when indenting roxygen2 comments with Ctrl+Shift+/, leave empty lines between the pasted output and the rest of the block.

Usage

```
write_doc_audit(sample_output, name_test)
```

Arguments

sample_output	Data frame. Result of a call to audit() on a data frame that resulted from a call to the mapper function for which you wrote the audit() method, such as audit(grim_map(pigs1)) or audit(debit_map(pigs3)).
name_test	String (length 1). Name of the consistency test which the mapper function applies, such as "GRIM" or "DEBIT".

Value

A string vector formatted by glue::glue().

Examples

```
# Start by running `audit()`:
out_grim  <- audit(grim_map(pigs1))
out_debit <- audit(debit_map(pigs3))

out_grim
out_debit

# Documenting the `audit()` method for `grim_map()`:
write_doc_audit(sample_output = out_grim, name_test = "GRIM")

# Documenting the `audit()` method for `debit_map()`:
write_doc_audit(sample_output = out_debit, name_test = "DEBIT")
```

write_doc_audit_seq *Documentation template for* audit_seq()

Description

write_doc_audit_seq() creates a roxygen2 block section to be inserted into the documentation of functions created with function_map_seq(). The section informs users about the ways in which audit_seq() summarizes the results of the manufactured *_map_seq() function.

Copy the output from your console and paste it into the roxygen2 block of your *_map_seq() function. To preserve the bullet-point structure when indenting roxygen2 comments with Ctrl+Shift+/, leave empty lines between the pasted output and the rest of the block.

Usage

```
write_doc_audit_seq(key_args, name_test)
```

Arguments

key_args	String vector with the names of the key columns that are tested for consistency by the *_map_seq() function. The values need to have the same order as in that function's output.
name_test	String (length 1). Name of the consistency test which the *_map_seq() function applies, such as "GRIM".

Value

A string vector formatted by glue::glue().

See Also

The sister function write_doc_audit_total_n() and, for context, vignette("consistency-tests-in-depth").

Examples

```
# For GRIM and `grim_map_seq()`:
write_doc_audit_seq(key_args = c("x", "n"), name_test = "GRIM")

# For DEBIT and `debit_map_seq()`:
write_doc_audit_seq(key_args = c("x", "sd", "n"), name_test = "DEBIT")
```

write_doc_audit_total_n

Documentation template for audit_total_n()

Description

write_doc_audit_total_n() creates a roxygen2 block section to be inserted into the documentation of functions created with function_map_total_n(). The section informs users about the ways in which audit_seq() summarizes the results of the manufactured *_map_total_n() function.

Copy the output from your console and paste it into the roxygen2 block of your *_map_total_n() function. To preserve the bullet-point structure when indenting roxygen2 comments with Ctrl+Shift+/, leave empty lines between the pasted output and the rest of the block.

Usage

```
write_doc_audit_total_n(key_args, name_test)
```

Arguments

key_args	String vector with the names of the key columns that are tested for consistency by the *_map_seq() function. (These are the original variable names, without "1" and "2" suffixes.) The values need to have the same order as in that function's output.
name_test	String (length 1). Name of the consistency test which the *_map_seq() function applies, such as "GRIM".

Value

A string vector formatted by glue::glue().

See Also

The sister function write_doc_audit_seq() and, for context, vignette("consistency-tests-in-depth").

Examples

```
# For GRIM and `grim_map_total_n()`:
write_doc_audit_total_n(key_args = c("x", "n"), name_test = "GRIM")

# For DEBIT and `debit_map_total_n()`:
write_doc_audit_total_n(key_args = c("x", "sd", "n"), name_test = "DEBIT")
```

`write_doc_factory_map_conventions` *Documentation template for function factory conventions*

Description

`write_doc_factory_map_conventions()` creates a roxygen2 block section to be inserted into the documentation of a function factory such as `function_map_seq()` or `function_map_total_n()`. It lays out the naming guidelines that users of your function factory should follow when creating new manufactured functions.

Copy the output from your console and paste it into the roxygen2 block of your function factory.

Usage

```
write_doc_factory_map_conventions(
  ending,
  name_test1 = "GRIM",
  name_test2 = "GRIMMER",
  scrutiny_prefix = FALSE
)
```

Arguments

`ending`	String (length 1). The part of your function factory's name after `function_map_`. To
`name_test1, name_test2`	Strings (length 1 each). Plain-text names of example consistency tests. Defaults are `"GRIM"` and `"GRIMMER"`, respectively.
`scrutiny_prefix`	Logical (length 1). Should the scrutiny functions mentioned in the output have a `scrutiny::` namespace specification? Set this to `TRUE` if the output will go into another package's documentation. Default is `FALSE`.

Value

A string vector formatted by `glue::glue()`.

See Also

For context, see *Implementing consistency tests*.

Examples

```
# For `function_map_seq()`:
write_doc_factory_map_conventions(ending = "seq")

# For `function_map_total_n()`:
write_doc_factory_map_conventions(ending = "total_n")
```

Index