Package 'ympes'

January 5, 2023

Type Package

Title Collection of Helper Functions

Version 0.3.0

Description Provides a collection of lightweight helper functions (imps) both for interactive use and for inclusion within other packages. These include minimal assertion functions with a focus on informative error messaging for both missing and incorrect function arguments as well as other functions for visualising colour palettes, quoting user input and working with age intervals.

License GPL-2

Encoding UTF-8

RoxygenNote 7.2.3

Suggests clipr, knitr, rmarkdown, tinytest

Depends R (>= 3.5.0)

LazyData true

VignetteBuilder knitr

URL https://timtaylor.github.io/ympes/

BugReports https://github.com/TimTaylor/ympes/issues

NeedsCompilation yes

Author Tim Taylor [aut, cre, cph] (<https://orcid.org/0000-0002-8587-7113>)

Maintainer Tim Taylor <tim.taylor@hiddenelephants.co.uk>

Repository CRAN

Date/Publication 2023-01-05 21:50:06 UTC

R topics documented:

`ageutils` *Utilities for Age Intervals*

Description

This help page documents the utility functions provided for working with age intervals:

`breaks_to_interval()` takes a specified set of breaks representing the left hand limits of a closed open interval, i.e [x, y), and returns the corresponding interval and upper bounds. The resulting intervals span from the minimum break through to `Inf`.

`cut_ages()` provides categorisation of ages based on specified breaks which represent the left-hand interval limits. The resultant groupings will span from the minimum break through to `Inf` and will always be closed on the left and open on the right. Ages below the minimum break will be returned as NA. As an example, if `breaks = c(0, 1, 10, 30)` the possible groupings would be [0, 1), [1, 10), [10, 30) and [30, Inf). This is roughly comparable to a call of `cut(ages, right = FALSE, breaks = c(limits, Inf))` but with both the resultant interval and the start and end points returned as entries in a list.

`split_interval_counts()` splits counts of a given age interval in to counts for individual years based on a given weighting. Age intervals are specified by their lower (closed) and upper (open) bounds, i.e. intervals of the form [lower, upper).

`aggregate_age_counts()` provides aggregation of counts across ages (in years). It is similar to a `cut()` and `tapply()` pattern but optimised for speed over flexibility. Groupings are the same as in `ages_to_interval()` and counts will be provided across all natural numbers grater than the minimum break. Missing values, and those less than the minimum break, are grouped as NA.

`reaggregate_interval_counts()` is equivalent to, but more efficient than, a call to `split_interval_counts()` followed by `aggregate_age_counts()`.

Usage

```
breaks_to_interval(breaks)

cut_ages(ages, breaks)

split_interval_counts(
  lower_bounds,
  upper_bounds,
  counts,
  max_upper = 100L,
  weights = NULL
)
```

```
aggregate_age_counts(counts, ages = 0:(length(counts) - 1L), breaks)

reaggregate_interval_counts(
  lower_bounds,
  upper_bounds,
  counts,
  breaks,
  max_upper = 100L,
  weights = NULL
)
```

Arguments

`breaks`	`[numeric]`. 1 or more non-negative cut points in increasing (strictly) order. These correspond to the left hand side of the desired intervals (e.g. the closed side of "[x, y)". Double values are coerced to integer prior to categorisation.
`ages`	`[numeric]`. Vector of age in years. Double values are coerced to integer prior to categorisation / aggregation. For `aggregate_age_counts()`, these must corresponding to the `counts` entry and will defaults to 0:(N-1) where `N` is the number of counts present. `ages` >= 200 are not permitted due to the internal implementation.
`lower_bounds, upper_bounds`	`[integerish]`. A pair of vectors representing the bounds of the intervals. `lower_bounds` must be strictly less than `upper_bounds` and greater than or equal to zero. Missing (NA) bounds are not permitted. Double vectors will be coerced to integer.
`counts`	`[numeric]`. Vector of counts to be aggregated.
`max_upper`	`[integerish]` Represents the maximum upper bounds permitted upon splitting the data. Used to replace `Inf` upper bounds prior to splitting. If any `upper_bound` is greater than `max_upper` the function will error. Double vectors will be coerced to integer.
`weights`	`[numeric]` Population weightings to apply for individual years. If `NULL` (default) counts will be split evenly based on interval size. If specified, must be of length `max_upper` and represent weights in the range 0:(max_upper - 1).

Value

`breaks_to_interval()`, `cut_ages()`:

A data frame with an ordered factor column (`interval`), as well as columns corresponding to the explicit bounds (`lower_bound` and `upper_bound`).

`split_interval_counts()`:

A data frame with entries `age` (in years) and `count`.

`aggregate_age_counts()` and `reaggregate_interval_counts()`:

A data frame with 4 entries; `interval`, `lower_bound`, `upper_bound` and an associated `count`.

Examples

```
cut_ages(ages = 0:9, breaks = c(0L, 3L, 5L, 10L))
cut_ages(ages = 0:9, breaks = 5L)

split_interval_counts(
    lower_bounds = c(0, 5, 10),
    upper_bounds = c(5, 10, 20),
    counts = c(5, 10, 30)
)

# default ages generated if only counts provided (here ages will be 0:64)
aggregate_age_counts(counts = 1:65, breaks = c(0L, 1L, 5L, 15L, 25L, 45L, 65L))
aggregate_age_counts(counts = 1:65, breaks = 50L)

# NA ages are handled with their own grouping
ages <- 1:65;
ages[1:44] <- NA
aggregate_age_counts(
    counts = 1:65,
    ages = ages,
    breaks = c(0L, 1L, 5L, 15L, 25L, 45L, 65L)
)

reaggregate_interval_counts(
    lower_bounds = c(0, 5, 10),
    upper_bounds = c(5, 10, 20),
    counts = c(5, 10, 30),
    breaks = c(0L, 1L, 5L, 15L, 25L, 45L, 65L)
)
```

---

assertions *Argument assertions*

Description

Assertions for function arguments. Motivated by `vctrs::vec_assert()` but with lower overhead at a cost of less informative error messages. Designed to make it easy to identify the top level calling function whether used within a user facing function or internally.

Usage

```
assert_integer(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_int(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_double(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_dbl(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_numeric(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_num(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_logical(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_lgl(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_character(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_chr(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_data_frame(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_list(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_scalar_integer(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_scalar_int(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_scalar_double(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_scalar_dbl(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_scalar_numeric(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_scalar_num(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_scalar_logical(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_scalar_lgl(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_bool(x, arg = deparse(substitute(x)), call = sys.call(-1L))
```

```
assert_boolean(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_scalar_character(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_scalar_chr(x, arg = deparse(substitute(x)), call = sys.call(-1L))

assert_string(x, arg = deparse(substitute(x)), call = sys.call(-1L))
```

Arguments

x	Argument to check.
arg	[character]
	Name of argument being checked (used in error message).
call	[call]
	Call to use in error message.

Value

NULL if the assertion succeeds (error otherwise).

Examples

```
# Use in a user facing function
fun <- function(i, d, l, chr, b) {
    assert_scalar_int(i)
    TRUE
}
fun(i=1L)
try(fun())
try(fun(i="cat"))

# Use in an internal function
internal_fun <- function(a) {
    assert_string(a, arg = deparse(substitute(a)), call = sys.call(-1L))
    TRUE
}
external_fun <- function(b) {
    internal_fun(a=b)
}
external_fun(b="cat")
try(external_fun())
try(external_fun(b = letters))
```

cc *Quote names*

Description

`cc()` quotes comma separated names whilst trimming outer whitespace. It is intended for interactive use only.

Usage

```
cc(..., .clip = getOption("imp.clipboard", FALSE))
```

Arguments

`...`	Unquoted names (separated by commas) that you wish to quote. Empty arguments (e.g. third item in `one,two,,four`) will be returned as `""`.
`.clip`	`[bool]` Should the code to generate the constructed character vector be copied to your system clipboard. Defaults to FALSE unless the option "imp.clipboard" is set to TRUE. Note that copying to clipboard requires the availability of package clipr.

Value

A character vector of the quoted input.

Examples

```
cc(dale, audrey, laura, hawk)
```

greprows *Pattern matching on data frame rows*

Description

`greprows()` searches for pattern matches within a data frames columns and returns the related rows or row indices.

Usage

```
greprows(
  dat,
  pattern,
  cols = NULL,
  value = TRUE,
  ignore.case = FALSE,
  perl = FALSE,
  fixed = FALSE,
  invert = FALSE
)
```

Arguments

dat	Data frame
pattern	character string containing a regular expression (or character string for fixed = TRUE) to be matched in the given character vector. Coerced by as.character to a character string if possible. If a character vector of length 2 or more is supplied, the first element is used with a warning. Missing values are allowed except for regexpr, gregexpr and regexec.
cols	[character] Character vector of columns to search. If NULL (default) all character and factor columns will be searched.
value	[logical] Should a data frame of rows be returned. If FALSE row indices will be returned instead of the rows themselves.
ignore.case	if FALSE, the pattern matching is *case sensitive* and if TRUE, case is ignored during matching.
perl	logical. Should Perl-compatible regexps be used?
fixed	logical. If TRUE, pattern is a string to be matched as is. Overrides all conflicting arguments.
invert	logical. If TRUE return indices or values for elements that do *not* match.

Value

A data frame of the corresponding rows or, if value = FALSE, the corresponding row numbers.

See Also

grep

Examples

```
dat <- data.frame(
    first = letters,
    second = factor(rev(LETTERS)),
```

```
    third = "Q"
)
greprows(dat, "A|b")
greprows(dat, "A|b", ignore.case = TRUE)
greprows(dat, "c", value = FALSE)
```

`plot_palette` *Plot a colour palette*

Description

`plot_palette()` plots a palette from a vector of colour values (name or hex).

Usage

```
plot_palette(values, label = TRUE, square = FALSE)
```

Arguments

`values`	`[character]`
	Vector of named or hex colours.
`label`	`[bool]`
	Do you want to label the plot or not?
	If `values` is a named vector the names are used for labels, otherwise, the values.
`square`	`[bool]`
	Display palette as square?

Value

The input (invisibly).

Examples

```
plot_palette(c("#5FE756", "red", "black"))
plot_palette(c("#5FE756", "red", "black"), square=TRUE)
```

pop_dat *Aggregated population data*

Description

A dataset derived from the 2021 UK census containing population for different age categories across England and Wales.

Usage

```
pop_dat
```

Format

A data frame with 200 rows and 6 variables:

area_code Unique area identifier

area_name Unique area name

age_category Left-closed and right-open age interval

value count of individ

Source

https://github.com/TimTaylor/census_pop_2021

Index