Package 'mlr3misc'

August 22, 2019

Title Helper Functions for 'mlr3'

Version 0.1.3

Description Frequently used helper functions and assertions
used in 'mlr3' and its companion packages. Comes with helper functions
for functional programming, for printing, to work with 'data.table',
as well as some generally useful 'R6' classes. This package also
supersedes the package 'BBmisc'.

License LGPL-3

URL https://mlr3misc.mlr-org.com, https://github.com/mlr-org/mlr3misc

BugReports https://github.com/mlr-org/mlr3misc/issues

Depends R (>= 3.1.0)

Imports backports, checkmate, data.table, R6

Suggests callr, evaluate, paradox, testthat

Encoding UTF-8

NeedsCompilation yes

RoxygenNote 6.1.1

Author Michel Lang [cre, aut] (<https://orcid.org/0000-0001-9754-0393>)

Maintainer Michel Lang <michellang@gmail.com>

Repository CRAN

Date/Publication 2019-08-22 20:50:02 UTC

R topics documented:

`mlr3misc-package` *mlr3misc: Helper Functions for 'mlr3'*

Description

Frequently used helper functions and assertions used in 'mlr3' and its companion packages. Comes with helper functions for functional programming, for printing, to work with 'data.table', as well as some generally useful 'R6' classes. This package also supersedes the package 'BBmisc'.

Author(s)

Maintainer: Michel Lang <michellang@gmail.com> (0000-0001-9754-0393)

See Also

Useful links:

- https://mlr3misc.mlr-org.com
- https://github.com/mlr-org/mlr3misc
- Report bugs at https://github.com/mlr-org/mlr3misc/issues

`as_factor` *Convert to Factor*

Description

Converts a vector to a `factor()` and ensures that levels are in the order of the provided levels.

Usage

```
as_factor(x, levels, ordered = is.ordered(x))
```

Arguments

`x`	:: atomic `vector()` Vector to convert to factor.
`levels`	:: `character()` Levels of the new factor.
`ordered`	:: `logical(1)` If `TRUE`, create an ordered factor.

Value

(`factor()`).

Examples

```
x = factor(c("a", "b"))
y = factor(c("a", "b"), levels = c("b", "a"))

# x with the level order of y
as_factor(x, levels(y))

# y with the level order of x
as_factor(y, levels(x))
```

`as_short_string` *Convert R Object to a Descriptive String*

Description

This function is intended to be convert any R object to a short descriptive string, e.g. in `base::print()` functions.

The following rules apply:

- if `x` is `atomic()` with length 0 or 1: printed as-is.
- if `x` is `atomic()` with length greater than 1, `x` is collapsed with `", "`, and the resulting string is truncated to `trunc_width` characters.
- if `x` is an expression: converted to character.
- Otherwise: the class is printed.

If `x` is a list, the above rules are applied (non-recursively) to its elements.

Usage

```
as_short_string(x, width = 30L, num_format = "%.4g")
```

Arguments

`x`	:: `any` Arbitrary object.
`width`	:: `integer(1)` Truncate strings to width `width`.
`num_format`	:: `character(1)` Used to format numerical scalars via `base::sprintf()`.

Value

(`character(1)`).

Examples

```
as_short_string(list(a = 1, b = NULL, "foo", c = 1:10))
```

chunk_vector *Chunk Vectors*

Description

Chunk atomic vectors into parts of roughly equal size. chunk() takes a vector length n and returns an integer with chunk numbers. chunk_vector() uses base::split() and chunk() to split an atomic vector into chunks.

Usage

```
chunk_vector(x, n_chunks = NULL, chunk_size = NULL, shuffle = TRUE)

chunk(n, n_chunks = NULL, chunk_size = NULL, shuffle = TRUE)
```

Arguments

x	:: vector() Vector to split into chunks.
n_chunks	:: integer(1) Requested number of chunks. Mutually exclusive with chunk_size and props.
chunk_size	:: integer(1) Requested number of elements in each chunk. Mutually exclusive with n_chunks and props.
shuffle	:: logical(1) If TRUE, permutes the order of x before chunking.
n	:: integer(1) Length of vector to split.

Value

chunk() returns a integer() of chunk indices, chunk_vector() a list() of integer vectors.

Examples

```
x = 1:11

ch = chunk(length(x), n_chunks = 2)
table(ch)
split(x, ch)

chunk_vector(x, n_chunks = 2)

chunk_vector(x, n_chunks = 3, shuffle = TRUE)
```

compat-map *Apply Functions in the spirit of 'purrr'*

Description

map-like functions, similar to the ones implemented in **purrr**:

- `map()` returns the results of `.f` applied to `.x` as list. If `.f` is not a function, `map` will call `[[` on all elements of `.x` using the value of `.f` as index.
- `imap()` applies `.f` to each value of `.x` (passed as first argument) and its name (passed as second argument). If `.x` does not have names, a sequence along `.x` is passed as second argument instead.
- `pmap()` expects `.x` to be a list of vectors of equal length, and then applies `.f` to the first element of each vector of `.x`, then the second element of `.x`, and so on.
- `map_if()` applies `.f` to each element of `.x` where the predicate `.p` evaluates to `TRUE`.
- `map_at()` applies `.f` to each element of `.x` referenced by `.at`. All other elements remain unchanged.
- `keep()` keeps those elements of `.x` where predicate `.p` evaluates to `TRUE`.
- `discard()` discards those elements of `.x` where predicate `.p` evaluates to `TRUE`.
- `every()` is `TRUE` if predicate `.p` evaluates to `TRUE` for each `.x`.
- `some()` is `TRUE` if predicate `.p` evaluates to `TRUE` for at least one `.x`.
- `detect()` returns the first element where predicate `.p` evaluates to `TRUE`.

Additionally, the functions `map()`, `imap()` and `pmap` have type-safe variants with the following suffixes:

- `*_lgl()` returns a `logical(length(.x))`.
- `*_int()` returns a `integer(length(.x))`.
- `*_dbl()` returns a `double(length(.x))`.
- `*_chr()` returns a `character(length(.x))`.
- `*_dtr()` returns a `data.table::data.table()` where the results of `.f` are put together in an `base::rbind()` fashion.
- `*_dtc()` returns a `data.table::data.table()` where the results of `.f` are put together in an `base::cbind()` fashion.

Usage

```
map(.x, .f, ...)

map_lgl(.x, .f, ...)

map_int(.x, .f, ...)
```

```
map_dbl(.x, .f, ...)

map_chr(.x, .f, ...)

map_dtr(.x, .f, ..., .fill = FALSE)

map_dtc(.x, .f, ...)

pmap(.x, .f, ...)

pmap_lgl(.x, .f, ...)

pmap_int(.x, .f, ...)

pmap_dbl(.x, .f, ...)

pmap_chr(.x, .f, ...)

pmap_dtr(.x, .f, ..., .fill = FALSE)

pmap_dtc(.x, .f, ...)

imap(.x, .f, ...)

imap_lgl(.x, .f, ...)

imap_int(.x, .f, ...)

imap_dbl(.x, .f, ...)

imap_chr(.x, .f, ...)

imap_dtr(.x, .f, ..., .fill = FALSE)

imap_dtc(.x, .f, ...)

keep(.x, .f, ...)

discard(.x, .p, ...)

map_if(.x, .p, .f, ...)

map_at(.x, .at, .f, ...)

every(.x, .p, ...)

some(.x, .p, ...)
```

```
detect(.x, .p, ...)
```

Arguments

.x	:: (list() or atomic vector).
.f	:: (function | character() | integer()) Function to apply, or element to extract by name (if .f is character()) or position (if .f is integer()).
...	:: any Additional arguments passed down to .f or .p.
.fill	:: logical(1) Passed down to data.table::rbindlist().
.p	:: (function() | logical()) Predicate function.
.at	:: (character() | integer() | logical()) Index vector.

---

compute_mode *Compute The Mode*

Description

Computes the mode (most frequent value) of an atomic vector.

Usage

```
compute_mode(x, ties_method = "random", na_rm = TRUE)
```

Arguments

x	:: vector().
ties_method	:: character(1) Handling of ties. One of "first", "last" or "random" to return the first tied value, the last tied value, or a randomly selected tied value, respectively.
na_rm	:: logical(1) If TRUE, remove missing values prior to computing the mode.

Value

(vector(1)): mode value.

Examples

```
compute_mode(c(1, 1, 1, 2, 2, 2, 3))
compute_mode(c(1, 1, 1, 2, 2, 2, 3), ties_method = "last")
compute_mode(c(1, 1, 1, 2, 2, 2, 3), ties_method = "random")
```

cross_join *Cross-Join for data.table*

Description

A safe version of `data.table::CJ()` in case a column is called `sorted` or `unique`.

Usage

```
cross_join(dots, sorted = TRUE, unique = FALSE)
```

Arguments

`dots`	(named `list()`) Vectors to cross-join.
`sorted`	(`logical(1)`) See `data.table::CJ()`.
`unique`	(`logical(1)`) See `data.table::CJ()`.

Value

`data.table()`.

Examples

```
cross_join(dots = list(sorted = 1:3, b = letters[1:2]))
```

Dictionary *Key-Value Storage*

Description

A key-value store for R6::R6 objects. On retrieval of an object, the following applies:

- If the object is a `R6ClassGenerator`, it is initialized with `new()`.
- If the object is a function, it is called and must return an instance of a R6::R6 object.
- If the object is an instance of a R6 class, it is returned as-is.

Default argument required for construction can be stored alongside their constructors by passing them to `$add()`.

Format

R6::R6Class object.

Construction

```
d = Dictionary$new()
```

Methods

- `get(key,...)`
  (`character(1)`, ...) -> `any`
  Retrieves object with key `key` from the dictionary. Additional arguments must be named and are passed to the constructor of the stored object.
- `mget(keys,...)`
  (`character()`, ...) -> named `list()`
  Returns objects with keys `keys` in a list named with `keys`. Additional arguments must be named and are passed to the constructors of the stored objects.
- `has(keys)`
  `character()` -> `logical()`
  Returns a logical vector with `TRUE` at its i-th position if the i-th key exists.
- `keys(pattern = NULL)`
  `character(1)` -> `character()`
  Returns all keys which comply to the regular expression `pattern`. If `pattern` is `NULL` (default), all keys are returned.
- `add(key,value,...,required_args = character())`
  (`character(1)`, `any`, ..., `character()`) -> `self`
  Adds object `value` to the dictionary with key `key`, potentially overwriting a previously stored item. Additional arguments in `...` must be named and are passed as default arguments to `value` during construction. The names of all additional arguments which are mandatory for construction and missing in `...` should be listed in `required_args`.
- `remove(keys)`
  `character()` -> `self`
  Removes objects with keys `keys` from the dictionary.
- `required_args(key)`
  (`character(1)`) -> `character()`
  Returns the names of arguments required to construct the object.

S3 methods

- `as.data.table(d)`
  Dictionary -> `data.table::data.table()`
  Converts the dictionary to a `data.table::data.table()`.

Examples

```
library(R6)
item1 = R6Class("Item", public = list(x = 1))
item2 = R6Class("Item", public = list(x = 2))
d = Dictionary$new()
d$add("a", item1)
d$add("b", item2)
```

```
d$add("c", item1$new())
d$keys()
d$get("a")
d$mget(c("a", "b"))
```

`dictionary_sugar` *A Quick Way to Initialize Objects from Dictionaries*

Description

Given a Dictionary, retrieves the object with key `key`. Arguments in `...` must be named and are consumed in the following order:

1. All arguments whose names match the name of an argument of the constructor are passed to the `$get()` method of the Dictionary for construction.
2. All arguments whose names match the name of a parameter of the paradox::ParamSet of the constructed object are set as parameters. If there is no paradox::ParamSet in `obj$param_set`, this step is skipped.
3. All remaining arguments are assumed to be regular fields of the constructed R6 instance, and are assigned via `<-`.

Usage

```
dictionary_sugar(dict, .key, ...)
```

Arguments

`dict`	:: Dictionary.
`.key`	:: `character(1)` Key of the object to construct.
`...`	:: `any` See description.

Value

`R6::R6Class()`

Examples

```
library(R6)
item = R6Class("Item", public = list(x = 0))
d = Dictionary$new()
d$add("key", item)
dictionary_sugar(d, "key", x = 2)
```

did_you_mean *Suggest Alternatives*

Description

Helps to suggest alternatives from a list of strings, based on the string similarity in `utils::adist()`.

Usage

```
did_you_mean(str, candidates)
```

Arguments

`str`	:: `character(1)` String.
`candidates`	:: `character()` Candidate strings.

Value

(`character(1)`). Either a phrase suggesting one or more candidates from `candidates`, or an empty string if no close match is found.

Examples

```
did_you_mean("yep", c("yes", "no"))
```

distinct_values *Get Distinct Values*

Description

Extracts the distinct values of an atomic vector, with the possibility to drop levels and remove missing values.

Usage

```
distinct_values(x, drop = TRUE, na_rm = TRUE)
```

Arguments

`x`	:: atomic `vector()`.
`drop`	:: `logical(1)` If `TRUE`, only returns values which are present in `x`. If `FALSE`, returns all levels for `factor()` and `ordered()`, as well as `TRUE` and `FALSE` for `logical()`s.
`na_rm`	:: `logical(1)` If `TRUE`, missing values are removed from the vector of distinct values.

Value

(atomic `vector()`) with distinct values in no particular order.

Examples

```
# for factors:
x = factor(c(letters[1:2], NA), levels = letters[1:3])
distinct_values(x)
distinct_values(x, na_rm = FALSE)
distinct_values(x, drop = FALSE)
distinct_values(x, drop = FALSE, na_rm = FALSE)

# for logicals:
distinct_values(TRUE, drop = FALSE)

# for numerics:
distinct_values(sample(1:3, 10, replace = TRUE))
```

---

encapsulate *Encapsulate Function Calls for Logging*

Description

Evaluates a function while both recording an output log and measuring the elapsed time. There are currently three different modes implemented to encapsulate a function call:

- `"none"`: Just runs the call in the current session and measures the elapsed time. Does not keep a log, output is printed directly to the console. Works well together with `traceback()`.
- `"evaluate"`: Uses the package **evaluate** to call the function, measure time and do the logging.
- `"callr"`: Uses the package **callr** to call the function, measure time and do the logging. This encapsulation spawns a separate R session in which the function is called. While this comes with a considerable overhead, it also guards your session from being teared down by segfaults.

Usage

```
encapsulate(method, .f, .args = list(), .opts = list(),
  .pkgs = character(), .seed = NA_integer_)
```

Arguments

`method`	:: `character(1)` One of `"none"`, `"evaluate"` or `"callr"`.
`.f`	:: `function()` Function to call.
`.args`	:: `list()` Arguments passed to `.f`.

`.opts` :: named `list()`
Options to set for the function call. Options get reset on exit.

`.pkgs` :: `character()`
Packages to load (not attach).

`.seed` :: `integer(1)`
Random seed to set before invoking the function call. Gets reset to the previous seed on exit.

Value

(named `list()`) with three fields:

- `"result"`: the return value of `.f`
- `"elapsed"`: elapsed time in seconds. Measured as `proc.time()` difference before/after the function call.
- `"log"`: `data.table()` with columns `"class"` (ordered factor with levels `"output"`, `"warning"` and `"error"`) and `"message"` (`character()`).

Examples

```
f = function(n) {
  message("hi from f")
  if (n > 5) {
    stop("n must be <= 5")
  }
  runif(n)
}
encapsulate("none", f, list(n = 1), .seed = 1)
encapsulate("evaluate", f, list(n = 1), .seed = 1)
encapsulate("callr", f, list(n = 1), .seed = 1)
```

---

enframe *Convert a Named Vector Into A data.table*

Description

Returns a `data.table::data.table()` with two columns: The names of `x` (or `seq_along(x)` if unnamed) and the values of `x`.

Usage

```
enframe(x, name = "name", value = "value")
```

Arguments

x	:: vector() Vector to convert to a data.table::data.table().
name	:: character(1) Name for the first column with names.
value	:: character(1) Name for the second column with values.

Value

data.table::data.table().

Examples

```
x = 1:3
enframe(x)

x = set_names(1:3, letters[1:3])
enframe(x, value = "x_values")
```

---

extract_vars *Extract Variables from a Formula*

Description

Given a formula() f, returns all variables used on the left-hand side and right-hand side of the formula.

Usage

```
extract_vars(f)
```

Arguments

f	:: formula().

Value

(list()) with elements "lhs" and "rhs", both character().

Examples

```
extract_vars(Species ~ Sepal.Width + Sepal.Length)
extract_vars(Species ~ .)
```

formulate *Create Formulas*

Description

Given the left-hand side and right-hand side as character vectors, generates a new `stats::formula()`.

Usage

```
formulate(lhs = NULL, rhs = NULL, env = NULL)
```

Arguments

`lhs`	:: `character(1)` Left-hand side of formula.
`rhs`	:: `character()` Right-hand side of formula. Multiple elements will be collapsed with `" + "`.
`env`	:: `environment()` Environment for the new formula. Defaults to `NULL`.

Value

`stats::formula()`.

Examples

```
formulate("Species", c("Sepal.Length", "Sepal.Width"))
formulate(rhs = c("Sepal.Length", "Sepal.Width"))
```

get_seed *Get the Random Seed*

Description

Retrieves the current random seed (`.Random.seed` in the global environment), and initializes the RNG first, if necessary.

Usage

```
get_seed()
```

Value

`integer()`. Depends on the `base::RNGkind()`.

Examples

```
str(get_seed())
```

has_element *Check if an Object is Element of a List*

Description

Simply checks if a list contains a given object.

- NB1: Objects are compared with identity.
- NB2: Only use this on lists with complex objects, for simpler structures there are faster operations.
- NB3: Clones of R6 objects are not detected.

Usage

```
has_element(.x, .y)
```

Arguments

.x	:: (list() or atomic vector).
.y	:: any Object to test for.

Examples

```
has_element(list(1, 2, 3), 1)
```

insert_named *Insert or Remove Named Elements*

Description

Insert elements from y into x by name, or remove elements from x by name. Works for vectors, lists, environments and data frames and data tables. Objects with reference semantic (environment() and data.table::data.table()) might be modified in-place.

Usage

```
insert_named(x, y)

## Default S3 method:
insert_named(x, y)

## S3 method for class 'environment'
insert_named(x, y)
```

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

## S3 method for class 'data.table'
insert_named(x, y)

remove_named(x, nn)

## S3 method for class 'environment'
remove_named(x, nn)

## S3 method for class 'data.frame'
remove_named(x, nn)

## S3 method for class 'data.table'
remove_named(x, nn)
```

Arguments

x	:: (vector() | list() | environment() | data.table::data.table()) Object to insert elements into, or remove elements from. Changes are by-reference for environments and data tables.
y	:: list() List of elements to insert into x.
nn	:: character() Character vector of elements to remove.

Value

Modified object.

Examples

```
x = list(a = 1, b = 2)
insert_named(x, list(b = 3, c = 4))
remove_named(x, "b")
```

---

invoke *Invoke a Function Call*

Description

An alternative interface for do.call(), similar to the deprecated function in **purrr**. This function tries hard to not evaluate the passed arguments too eagerly which is important when working with large R objects.

It is recommended to pass all arguments named in order to to not rely on on positional argument matching.

Usage

```
invoke(.f, ..., .args = list(), .opts = list(), .seed = NA_integer_)
```

Arguments

`.f`	:: `function()` Function to call.
`...`	:: `any` Additional function arguments passed to `.f`.
`.args`	:: `list()` Additional function arguments passed to `.f`, as (named) `list()`. These arguments will be concatenated to the arguments provided via `...`.
`.opts`	:: `list()` List of options which are set before the `.f` is called. Options are reset to their previous state afterwards.
`.seed`	:: `integer(1)` Random seed to set before invoking the function call. Gets reset to the previous seed on exit.

Examples

```
invoke(mean, .args = list(x = 1:10))
invoke(mean, na.rm = TRUE, .args = list(1:10))
```

`is_scalar_na` *Check for a Single Scalar Value*

Description

Check for a Single Scalar Value

Usage

```
is_scalar_na(x)
```

Arguments

`x`	:: `any` Argument to check.

Value

(`logical(1)`).

keep_in_bounds *Remove All Elements Out Of Bounds*

Description

Filters vector `x` to only keep elements which are in bounds `[lower,upper]`. This is equivalent to the following, but tries to avoid unnecessary allocations:

```
x[!is.na(x) & x >= lower & x <= upper]
```

Currently only works for integer `x`.

Usage

```
keep_in_bounds(x, lower, upper)
```

Arguments

`x`	:: `integer()` Vector to filter.
`lower`	:: `integer(1)` Lower bound.
`upper`	:: `integer(1)` Upper bound.

Value

(integer()) with only values in `[lower,upper]`.

Examples

```
keep_in_bounds(sample(20), 5, 10)
```

load_dataset *Retrieve a Single Data Set*

Description

Loads a data set with name `id` from package `package` and returns it. If the package is not installed, an error with condition "packageNotFoundError" is raised. The name of the missing packages is stored in the condition as `packages`.

Usage

```
load_dataset(id, package, keep_rownames = FALSE)
```

Arguments

id	:: character(1) Name of the data set.
package	:: character(1) Package to load the data set from.
keep_rownames	:: logical(1) Keep possible row names (default: FALSE).

Examples

```
head(load_dataset("iris", "datasets"))
```

---

map_values *Replace Elements of Vectors with New Values*

Description

Replaces all values in x which match old with values in new. Values are matched with base::match().

Usage

```
map_values(x, old, new)
```

Arguments

x	:: vector().
old	:: vector() Vector with values to replace.
new	:: vector() Values to replace with. Will be forced to the same length as old with base::rep_len().

Value

(vector()) of the same length as x.

Examples

```
x = letters[1:5]

# replace all "b" with "_b_", and all "c" with "_c_"
old = c("b", "c")
new = c("_b_", "_c_")
map_values(x, old, new)
```

`modify_if` *Selectively Modify Elements of a Vector*

Description

Modifies elements of a vector selectively, similar to the functions in **purrr**.

`modify_if()` applies a predicate function `.p` to all elements of `.x` and applies `.f` to those elements of `.x` where `.p` evaluates to `TRUE`.

`modify_at()` applies `.f` to those elements of `.x` selected via `.at`.

Usage

```
modify_if(.x, .p, .f, ...)

modify_at(.x, .at, .f, ...)
```

Arguments

`.x`	:: `vector()`.
`.p`	:: `function()` Predicate function.
`.f`	:: `function()` Function to apply on `.x`.
`...`	:: `any` Additional arguments passed to `.f`.
`.at`	:: (`integer()` | `character()`) Index vector to select elements from `.x`.

Examples

```
x = modify_if(iris, is.factor, as.character)
str(x)

x = modify_at(iris, 5, as.character)
x = modify_at(iris, "Sepal.Length", sqrt)
str(x)
```

named_list *Create a Named List*

Description

Create a Named List

Usage

```
named_list(nn = character(0L), init = NULL)
```

Arguments

nn	:: character() Names of new list.
init	:: any All list elements are initialized to this value.

Value

(named list()).

Examples

```
named_list(c("a", "b"))
named_list(c("a", "b"), init = 1)
```

named_vector *Create a Named Vector*

Description

Creates a simple atomic vector with init as values.

Usage

```
named_vector(nn = character(0L), init = NA)
```

Arguments

nn	:: character() Names of new vector
init	:: atomic All vector elements are initialized to this value.

Value

(named `vector()`).

Examples

```
named_vector(c("a", "b"), NA)
named_vector(character())
```

`names2` *A Type-Stable names() Replacement*

Description

A simple wrapper around `base::names()`. Returns a character vector even if no names attribute is set. Values `NA` and `""` are treated as missing and replaced with the value provided in `missing_val`.

Usage

```
names2(x, missing_val = NA_character_)
```

Arguments

`x`	:: `any` Object.
`missing_val`	:: `atomic(1)` Value to set for missing names. Default is `NA_character_`.

Value

(`character(length(x))`).

Examples

```
x = 1:3
names(x)
names2(x)

names(x)[1:2] = letters[1:2]
names(x)
names2(x, missing_val = "")
```

printf *Functions for Formatted Output and Conditions*

Description

`catf()`, `messagef()`, `warningf()` and `stopf()` are wrappers around `base::cat()`, `base::message()`, `base::warning()` and `base::stop()`, respectively.

Usage

```
catf(msg, ..., file = "")

messagef(msg, ...)

warningf(msg, ...)

stopf(msg, ...)
```

Arguments

`msg`	:: `character(1)` Format string passed to `base::sprintf()`.
`...`	:: `any` Arguments passed down to `base::sprintf()`.
`file`	:: `character(1)` Passed to `base::cat()`.

rcbind *Bind Columns by Reference*

Description

Performs `base::cbind()` on data.tables, possibly by reference.

Usage

```
rcbind(x, y)
```

Arguments

`x`	:: `data.table::data.table()` `data.table::data.table()` to add columns to.
`y`	:: `data.table::data.table()` `data.table::data.table()` to take columns from.

Value

(data.table::data.table()): Updated x .

---

require_namespaces *Require Multiple Namespaces*

Description

Packages are loaded (not attached) via base::requireNamespace(). If at least on package can not be loaded, an exception of class "packageNotFoundError" is raised. The character vector of missing packages is stored in the condition as packages.

Usage

```
require_namespaces(pkgs,
  msg = "The following packages could not be loaded: %s")
```

Arguments

pkgs	:: character() Packages to load.
msg	:: character(1) Message to print on error. "%s" is placeholder for the list of packages.

Examples

```
require_namespaces("mlr3misc")

# catch condition, return missing packages
tryCatch(require_namespaces(c("mlr3misc", "foobaaar")),
  packageNotFoundError = function(e) e$packages)
```

---

rowwise_table *Row-Wise Constructor for 'data.table'*

Description

Similar to the **tibble** function tribble(), this function allows to construct tabular data in a row-wise fashion.

The first arguments passed as formula will be interpreted as column names. The remaining arguments will be put into the resulting table.

Usage

```
rowwise_table(..., .key = NULL)
```

Arguments

`...`	:: any Arguments: Column names in first rows as formulas (with empty left hand side), then the tabular data in the following rows.
`.key`	:: `character(1)` If not `NULL`, set the key via `data.table::setkeyv()` after constructing the table.

Value

`data.table::data.table()`.

Examples

```
rowwise_table(
  ~a, ~b,
  1, "a",
  2, "b"
)
```

---

sequence_helpers *Sequence Construction Helpers*

Description

`seq_row()` creates a sequence along the number of rows of `x`, `seq_col()` a sequence along the number of columns of `x`. `seq_len0()` and `seq_along0()` are the 0-based counterparts to `base::seq_len()` and `base::seq_along()`.

Usage

```
seq_row(x)

seq_col(x)

seq_len0(n)

seq_along0(x)
```

Arguments

`x`	:: any Arbitrary object. Used to query its rows, cols or length.
`n`	:: `integer(1)` Length of the sequence.

Examples

```
seq_len0(3)
```

`set_class` *Set the Class*

Description

Simple wrapper for `class(x) = classes`.

Usage

```
set_class(x, classes)
```

Arguments

`x`	:: `any`.
`classes`	:: `character(1)` Vector of new class names.

Value

Object `x`, with updated class attribute.

Examples

```
set_class(list(), c("foo1", "foo2"))
```

`set_names` *Set Names*

Description

Sets the names (or colnames) of `x` to `nm`. If `nm` is a function, it is used to transform the already existing names of `x`.

Usage

```
set_names(x, nm = x, ...)

set_col_names(x, nm, ...)
```

Arguments

x	:: any. Object to set names for.
nm	:: (character() | function()) New names, or a function which transforms already existing names.
...	:: any Passed down to nm if nm is a function.

Value

x with updated names.

Examples

```
x = letters[1:3]

# name x with itself:
x = set_names(x)
print(x)

# convert names to uppercase
x = set_names(x, toupper)
print(x)
```

---

shuffle *Safe Version of Sample*

Description

A version of sample() which does not treat positive scalar integer x differently. See example.

Usage

```
shuffle(x, n = length(x), ...)
```

Arguments

x	:: vector() Vector to sample elements from.
n	:: integer() Number of elements to sample.
...	:: any Arguments passed down to base::sample.int().

Examples

```
x = 2:3
sample(x)
shuffle(x)

x = 3
sample(x)
shuffle(x)
```

`str_collapse` *Collapse Strings*

Description

Collapse multiple strings into a single string.

Usage

```
str_collapse(str, sep = ", ", quote = "", n = Inf,
  ellipsis = "[...]")
```

Arguments

`str`	:: `character()` Vector of strings.
`sep`	:: `character(1)` String used to collapse the elements of `x`.
`quote`	:: `character(1)` Quotes to use around each element of `x`.
`n`	:: `integer(1)` Number of elements to keep from `x`. See `utils::head()`.
`ellipsis`	:: `character(1)` If the string has to be shortened, this is signaled by appending `ellipsis` to `str`. Default is `" [...]"`.

Value

(`character(1)`).

Examples

```
str_collapse(letters, quote = "'", n = 5)
```

str_indent *Indent Strings*

Description

Formats a text block for printing.

Usage

```
str_indent(initial, str, width = 0.9 * getOption("width"), exdent = 2L,
  ...)
```

Arguments

initial	:: character(1) Initial string, passed to strwrap().
str	:: character() Vector of strings.
width	:: integer(1) Width of the output.
exdent	:: integer(1) Indentation of subsequent lines in paragraph.
...	:: any Additional parameters passed to str_collapse().

Value

(character()).

Examples

```
cat(str_indent("Letters:", str_collapse(letters), width = 25), sep = "\n")
```

str_trunc *Truncate Strings*

Description

str_trunc() truncates a string to a given width.

Usage

```
str_trunc(str, width = 0.9 * getOption("width"), ellipsis = "[...]")
```

Arguments

`str`	:: `character()` Vector of strings.
`width`	:: `integer(1)` Width of the output.
`ellipsis`	:: `character(1)` If the string has to be shortened, this is signaled by appending `ellipsis` to `str`. Default is `"[...]"`.

Value

(`character()`).

Examples

```
str_trunc("This is a quite long string", 20)
```

`topo_sort` *Topological Sorting of Dependency Graphs*

Description

Topologically sort a graph, where we are passed node labels and a list of direct parents for each node, as labels, too. A node can be 'processed' if all its parents have been 'processed', and hence occur at previous indices in the resulting sorting. Returns a table, in topological row order for IDs, and an entry `depth`, which encodes the topological layer, starting at 0. So nodes with `depth == 0` are the ones with no dependencies, and the one with maximal `depth` are the ones on which nothing else depends on.

Usage

```
topo_sort(nodes)
```

Arguments

`nodes`	:: `data.table::data.table()` Has 2 columns: • `id` of type `character`, contains all node labels. • `parents` of type `list` of `character`, contains all direct parents label of `id`.

Value

(`data.table::data.table()`) with columns `id`, `depth`, sorted topologically for IDs.

Examples

```
nodes = rowwise_table(
  ~id, ~parents,
  "a", "b",
  "b", "c",
  "c", character()
)
topo_sort(nodes)
```

---

transpose_list *Transpose lists of lists*

Description

Transposes a list of list, and turns it inside out, similar to the function `transpose()` in package **purrr**.

Usage

```
transpose_list(.l)
```

Arguments

`.l`	:: `list()` of `list()`.

Value

`list()`.

Examples

```
x = list(list(a = 2, b = 3), list(a = 5, b = 10))
str(x)
str(transpose_list(x))

# list of data frame rows:
transpose_list(iris[1:2, ])
```

unnest *Unnest List Data Table Columns*

Description

Transforms each element of a list columns into its own column, possibly by reference.

Usage

```
unnest(x, cols, prefix = NULL)
```

Arguments

x	:: data.table::data.table() data.table::data.table() with columns to unnest.
cols	:: character() Column names of list columns to operate on.
prefix	:: character(1) String to prefix the new column names with.

Value

(data.table::data.table()).

Examples

```
x = data.table::data.table(
  id = 1:2,
  value = list(list(a = 1, b = 2), list(a = 2, b = 2))
)
print(x)
unnest(x, "value")
```

which_min *Index of the Minimum/Maximum Value, with ties correction*

Description

Works similar to base::which.min()/base::which.max(), but corrects for ties. Missing values are set to Inf for which_min and to -Inf for which_max().

Usage

```
which_min(x, ties_method = "random")

which_max(x, ties_method = "random")
```

Arguments

x	:: numeric() Numeric vector.
ties_method	:: character(1) Handling of ties. One of "first", "last" or "random" (default) to return the first index, the last index, or a random index of the minimum/maximum values. Passed down to base::max.col().

Value

(integer()): Index of the minimum/maximum value. Returns an empty integer vector for empty input vectors and vectors with no non-missing values.

Examples

```
x = c(2, 3, 1, 3, 5, 1, 1)
which_min(x, ties_method = "first")
which_min(x, ties_method = "last")
which_min(x, ties_method = "random")

which_max(x)
which_max(integer(0))
which_max(NA)
which_max(c(NA, 1))
```

%nin% *Negated in-operator*

Description

This operator is equivalent to !(x %in% y).

Usage

```
x %nin% y
```

Arguments

x	:: vector() Values that should not be in y.
y	:: vector() Values to match against.

Index