Package 'mlr3misc'

February 1, 2024

Title Helper Functions for 'mlr3'

Version 0.14.0

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 (>= 0.1.5), checkmate, data.table, digest, methods, R6

Suggests callr, evaluate, paradox, testthat (>= 3.0.0)

Config/testthat/edition 3

Config/testthat/parallel true

Encoding UTF-8

NeedsCompilation yes

RoxygenNote 7.2.3.9000

Collate 'Dictionary.R' 'named_list.R' 'Callback.R' 'Context.R' 'as_factor.R' 'as_short_string.R' 'assert_ro_binding.R' 'calculate_hash.R' 'capitalize.R' 'catn.R' 'check_packages_installed.R' 'chunk.R' 'compose.R' 'compute_mode.R' 'count_missing.R' 'crate.R' 'cross_join.R' 'dictionary_sugar.R' 'dictionary_sugar_safe.R' 'did_you_mean.R' 'distinct_values.R' 'encapsulate.R' 'enframe.R' 'extract_vars.R' 'format_bib.R' 'formulate.R' 'get_private.R' 'get_seed.R' 'has_element.R' 'ids.R' 'insert_named.R' 'invoke.R' 'is_scalar_na.R' 'keep_in_bounds.R' 'leanify.R' 'load_dataset.R' 'map_values.R' 'modify.R' 'named_vector.R' 'names2.R' 'nin.R' 'open_help.R' 'printf.R' 'probe.R'

'purrr_map.R' 'rcbind.R' 'rd_info.R' 'recycle_vector.R'
'register_namespace_callback.R' 'remove_named.R'
'reorder_vector.R' 'require_namespaces.R' 'rowwise_table.R'
'seq.R' 'set_class.R' 'set_names.R' 'set_params.R' 'shuffle.R'
'str_collapse.R' 'str_indent.R' 'str_trunc.R' 'to_decimal.R'
'topo_sort.R' 'transpose.R' 'unnest.R' 'which_max.R'
'with_package.R' 'zzz.R'

Author Michel Lang [cre, aut] (<https://orcid.org/0000-0001-9754-0393>),
Patrick Schratz [aut] (<https://orcid.org/0000-0003-0748-6624>)

Maintainer Michel Lang <michellang@gmail.com>

Repository CRAN

Date/Publication 2024-01-01 10:10:02 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> (ORCID)

Authors:

- Patrick Schratz <patrick.schratz@gmail.com> (ORCID)

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

assert_callback *Assertions for Callbacks*

Description

Assertions for Callback class.

Usage

```
assert_callback(callback)

assert_callbacks(callbacks)
```

Arguments

callback	(Callback).
callbacks	(list of Callback).

Value

Callback | List of Callbacks.

assert_ro_binding *Assertion for Active Bindings in R6 Classes*

Description

This assertion is intended to be called in active bindings of an R6::R6Class which does not allow assignment. If rhs is not missing, an exception is raised.

Usage

```
assert_ro_binding(rhs)
```

Arguments

rhs	(any) If not missing, an exception is raised.

Value

Nothing.

as_callback *Convert to a Callback*

Description

Convert object to a Callback or a list of Callback.

Usage

```
as_callback(x, ...)

## S3 method for class 'Callback'
as_callback(x, clone = FALSE, ...)

as_callbacks(x, clone = FALSE, ...)

## S3 method for class 'list'
as_callbacks(x, clone = FALSE, ...)

## S3 method for class 'Callback'
as_callbacks(x, clone = FALSE, ...)
```

Arguments

x	(any) Object to convert.
...	(any) Additional arguments.
clone	(logical(1)) If TRUE, ensures that the returned object is not the same as the input x.

Value

Callback.

---

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))
```

calculate_hash *Calculate a Hash for Multiple Objects*

Description

Calls `digest::digest()` using the 'xxhash64' algorithm after applying `hash_input` to each object. To customize the hashing behaviour, you can overwrite `hash_input` for specific classes. For `data.table` objects, `hash_input` is applied to all columns, so you can overwrite `hash_input` for columns of a specific class. Objects that don't have a specific method are hashed as is.

Usage

```
calculate_hash(...)
```

Arguments

`...`	(any) Objects to hash.

Value

`(character(1))`.

Examples

```
calculate_hash(iris, 1, "a")
```

Callback *Callback*

Description

Callbacks allow to customize the behavior of processes in mlr3 packages. The following packages implement callbacks:

- `CallbackOptimization` in **bbotk**.
- `CallbackTuning` in **mlr3tuning**.
- `CallbackTorch` in `mlr3torch`

Details

Callback is an abstract base class. A subclass inherits from Callback and adds stages as public members. Names of stages should start with `"on_"`. For each subclass a function should be implemented to create the callback. For an example on how to implement such a function see `callback_optimization()` in **bbotk**. Callbacks are executed at stages using the function `call_back()`. A Context defines which information can be accessed from the callback.

Public fields

`id` (`character(1)`)
: Identifier of the callback.

`label` (`character(1)`)
: Label for this object. Can be used in tables, plot and text output instead of the ID.

`man` (`character(1)`)
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. Defaults to `NA`, but can be set by child classes.

`state` (named `list()`)
: A callback can write data into the state.

Methods

Public methods:

- `Callback$new()`
- `Callback$format()`
- `Callback$print()`
- `Callback$help()`
- `Callback$call()`
- `Callback$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
Callback$new(id, label = NA_character_, man = NA_character_)
```

Arguments:

`id` (`character(1)`)
: Identifier for the new instance.

`label` (`character(1)`)
: Label for the new instance.

`man` (`character(1)`)
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

Method `format()`**:** Helper for print outputs.

Usage:

```
Callback$format(...)
```

Arguments:

`...` (ignored).

Method `print()`**:** Printer.

Usage:

```
Callback$print(...)
```

Arguments:

... (ignored).

Method `help()`**:** Opens the corresponding help page referenced by field `$man`.

Usage:

```
Callback$help()
```

Method `call()`**:** Call the specific stage for a given context.

Usage:

```
Callback$call(stage, context)
```

Arguments:

`stage` (`character(1)`)
stage.

`context` (`Context`)
Context.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Callback$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
library(R6)

# implement callback subclass
CallbackExample = R6Class("CallbackExample",
  inherit = mlr3misc::Callback,
  public = list(
    on_stage_a = NULL,
    on_stage_b = NULL,
    on_stage_c = NULL
  )
)
```

`capitalize` *Capitalize the First Letter of Strings*

Description

Takes a character vector and changes the first letter of each element to uppercase.

Usage

```
capitalize(str)
```

Arguments

str	(character()).

Value

Character vector, same length as str.

Examples

```
capitalize("foo bar")
```

---

catn *Function for Formatted Output*

Description

Wrapper around base::cat() with a line break. Elements are converted to character and concatenate with base::paste0(). If a vector is passed, elements are collapsed with line breaks.

Usage

```
catn(..., file = "")
```

Arguments

...	(any) Arguments passed down to base::paste0().
file	(character(1)) Passed to base::cat().

Examples

```
catn(c("Line 1", "Line 2"))
```

check_packages_installed *Check that packages are installed, without loading them*

Description

Calls `find.package()` to check if the all packages are installed.

Usage

```
check_packages_installed(
  pkgs,
  warn = TRUE,
  msg = "The following packages are required but not installed: %s"
)
```

Arguments

pkgs	(character()) Packages to check.
warn	(logical(1)) If TRUE, signals a warning of class "packageNotFoundWarning" about the missing packages.
msg	(character(1)) Format of the warning message. Use "%s" as placeholder for the list of packages.

Value

(logical()) named with package names. TRUE if the respective package is installed, FALSE otherwise.

Examples

```
check_packages_installed(c("mlr3misc", "foobar"), warn = FALSE)

# catch warning
tryCatch(check_packages_installed(c("mlr3misc", "foobaaar")),
  packageNotFoundWarning = function(w) as.character(w))
```

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)
```

clbk *Syntactic Sugar for Callback Construction*

Description

Functions to retrieve callbacks from mlr_callbacks and set parameters in one go.

Usage

```
clbk(.key, ...)

clbks(.keys)
```

Arguments

.key	(character(1)) Key of the object to construct.
...	(named list()) Named arguments passed to the state of the callback.
.keys	(character()) Keys of the objects to construct.

See Also

Callback call_back

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`.
- `walk()`, `iwalk()` and `pwalk()` are the counterparts to `map()`, `imap()` and `pmap()`, but just visit (or change by reference) the elements of `.x`. They return input `.x` invisibly.

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))`.
- `*_br()` returns an object where the results of `.f` are put together with `base::rbind()`.
- `*_bc()` returns an object where the results of `.f` are put together with `base::cbind()`.
- `*_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_br(.x, .f, ...)

map_bc(.x, .f, ...)

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

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, .idcol = NULL)

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, .idcol = NULL)

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

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

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

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

## Default S3 method:
map_if(.x, .p, .f, ...)

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

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

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

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

walk(.x, .f, ...)

iwalk(.x, .f, ...)

pwalk(.x, .f, ...)
```

Arguments

`.x`	(`list()` | 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()`.
`.idcol`	(`logical(1)`) Passed down to `data.table::rbindlist()`.
`.p`	(`function()` | `logical()`) Predicate function.
`.at`	(`character()` | `integer()` | `logical()`) Index vector.

`compose` *Composition of Functions*

Description

Composes two or more functions into a single function. The returned function calls all provided functions in reverse order: The return value of the last function servers as input for the next to last function, and so on.

Usage

```
compose(...)
```

Arguments

`...`	(`functions`) Functions to compose.

Value

(`function()`) which calls the functions provided via `...` in reverse order.

Examples

```
f = compose(function(x) x + 1, function(x) x / 2)
f(10)
```

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")
```

Context *Context*

Description

Context objects allow Callback objects to access and modify data. The following packages implement context subclasses:

- ContextOptimization in **bbotk**.
- ContextEval in **mlr3tuning**.
- ContextTorch in mlr3torch

Details

Context is an abstract base class. A subclass inherits from Context. Data is stored in public fields. Access to the data can be restricted with active bindings (see example).

Public fields

`id (character(1))`
: Identifier of the object. Used in tables, plot and text output.

`label (character(1))`
: Label for this object. Can be used in tables, plot and text output instead of the ID.

Methods

Public methods:

- `Context$new()`
- `Context$format()`
- `Context$print()`
- `Context$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
Context$new(id, label = NA_character_)
```

Arguments:

`id (character(1))`
: Identifier for the new instance.

`label (character(1))`
: Label for the new instance.

Method `format()`**:** Format object as simple string.

Usage:

```
Context$format(...)
```

Arguments:

`...` (ignored).

Method `print()`**:** Print object.

Usage:

```
Context$print()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Context$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
library(data.table)
library(R6)

# data table with column x an y
data = data.table(x = runif(10), y = sample(c("A", "B"), 10, replace = TRUE))

# context only allows to access column y
ContextExample = R6Class("ContextExample",
  inherit = Context,
  public = list(
    data = NULL,

    initialize = function(data) {
        self$data = data
    }
  ),

  active = list(
    y = function(rhs) {
      if (missing(rhs)) return(self$data$y)
      self$data$y = rhs
    }
  )
)

context = ContextExample$new(data)

# retrieve content of column y
context$y

# change content of column y to "C"
context$y = "C"
```

`count_missing` *Count Missing Values in a Vector*

Description

Same as `sum(is.na(x))`, but without the allocation.

Usage

```
count_missing(x)
```

Arguments

x	`vector()` Supported are logical, integer, double, complex and string vectors.

Value

(integer(1)) number of missing values.

Examples

```
count_missing(c(1, 2, NA, 4, NA))
```

---

crate *Isolate a Function from its Environment*

Description

Put a function in a "lean" environment that does not carry unnecessary baggage with it (e.g. references to datasets).

Usage

```
crate(.fn, ..., .parent = topenv(), .compile = TRUE)
```

Arguments

.fn	(function()) function to crate
...	(any) The objects, which should be visible inside .fn.
.parent	(environment) Parent environment to look up names. Default to topenv().
.compile	(logical(1)) Whether to jit-compile the function. In case the function is already compiled. If the input function .fn is compiled, this has no effect, and the output function will always be compiled.

Examples

```
meta_f = function(z) {
  x = 1
  y = 2
  crate(function() {
    c(x, y, z)
  }, x)
}
x = 100
y = 200
z = 300
f = meta_f(1)
f()
```

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()`.

S3 methods

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

Public fields

`items (environment())`
: Stores the items of the dictionary

Methods

Public methods:

- `Dictionary$new()`
- `Dictionary$format()`
- `Dictionary$print()`
- `Dictionary$keys()`
- `Dictionary$has()`
- `Dictionary$get()`
- `Dictionary$mget()`
- `Dictionary$add()`
- `Dictionary$remove()`
- `Dictionary$prototype_args()`
- `Dictionary$clone()`

Method `new()`**:** Construct a new Dictionary.

Usage:

```
Dictionary$new()
```

Method `format()`**:** Format object as simple string.

Usage:

```
Dictionary$format(...)
```

Arguments:

`...` (ignored).

Method `print()`**:** Print object.

Usage:

```
Dictionary$print()
```

Method `keys()`**:** Returns all keys which comply to the regular expression `pattern`. If `pattern` is `NULL` (default), all keys are returned.

Usage:

```
Dictionary$keys(pattern = NULL)
```

Arguments:

`pattern (character(1))`.

Returns: `character()` of keys.

Method `has()`**:** Returns a logical vector with `TRUE` at its i-th position if the i-th key exists.

Usage:

```
Dictionary$has(keys)
```

Arguments:

keys (character()).

Returns: logical().

Method get(): Retrieves object with key key from the dictionary. Additional arguments must be named and are passed to the constructor of the stored object.

Usage:

```
Dictionary$get(key, ..., .prototype = FALSE)
```

Arguments:

key (character(1)).

... (any)
: Passed down to constructor.

.prototype (logical(1))
: Whether to construct a prototype object.

Returns: Object with corresponding key.

Method mget(): 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.

Usage:

```
Dictionary$mget(keys, ...)
```

Arguments:

keys (character()).

... (any)
: Passed down to constructor.

Returns: Named list() of objects with corresponding keys.

Method add(): 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.

Usage:

```
Dictionary$add(key, value, ..., .prototype_args = list())
```

Arguments:

key (character(1)).

value (any).

... (any)
: Passed down to constructor.

.prototype_args (list())
: List of arguments to construct a prototype object. Can be used when objects have construction arguments without defaults.

Returns: Dictionary.

Method remove(): Removes objects with from the dictionary.

Usage:

```
Dictionary$remove(keys)
```

Arguments:

`keys (character())`
: Keys of objects to remove.

Returns: `Dictionary`.

Method `prototype_args()`**:** Returns the arguments required to construct a simple prototype of the object.

Usage:

```
Dictionary$prototype_args(key)
```

Arguments:

`key (character(1))`
: Key of object to query for required arguments.

Returns: `list()` of prototype arguments

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Dictionary$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

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

Description

Given a Dictionary, retrieve objects with provided keys.

- `dictionary_sugar_get()` to retrieve a single object with key `.key`.
- `dictionary_sugar_mget()` to retrieve a list of objects with keys `.keys`.

- `dictionary_sugar()` is deprecated in favor of `dictionary_sugar_get()`.
- If `.key` or `.keys` is missing, the dictionary itself is returned.

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_get(dict, .key, ...)

dictionary_sugar(dict, .key, ...)

dictionary_sugar_mget(dict, .keys, ...)
```

Arguments

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

Value

`R6::R6Class()`

Examples

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

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

Description

Given a Dictionary, retrieve objects with provided keys.

- `dictionary_sugar_get_safe()` to retrieve a single object with key `.key`.
- `dictionary_sugar_mget_safe()` to retrieve a list of objects with keys `.keys`.
- If `.key` or `.keys` is missing, the dictionary itself is returned.
- Dictionary getters without the `_safe` suffix are discouraged as this sometimes caused unintended partial argument matching.

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_get_safe(.dict, .key, ...)

dictionary_sugar_mget_safe(.dict, .keys, ...)
```

Arguments

`.dict`	(Dictionary) The dictionary from which to retrieve the elements.
`.key`	(`character(1)`) Key of the object to construct.
`...`	(any) See description.
`.keys`	(`character()`) Keys of the objects to construct.

Value

`R6::R6Class()`

Examples

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

`dictionary_sugar_inc_get` *A Quick Way to Initialize Objects from Dictionaries with Incremented ID*

Description

Covenience wrapper around dictionary_sugar_get and dictionary_sugar_mget to allow easier avoidance of of ID clashes which is useful when the same object is used multiple times and the ids have to be unique. Let <key> be the key of the object to retrieve. When passing the <key>_<n> to this function, where <n> is any natural numer, the object with key <key> is retrieved and the suffix _<n> is appended to the id after the object is constructed.

Usage

```
dictionary_sugar_inc_get(dict, .key, ...)

dictionary_sugar_inc_mget(dict, .keys, ...)
```

Arguments

`dict`	(Dictionary) Dictionary from which to retrieve an element.
`.key`	(`character(1)`) Key of the object to construct - possibly with a suffix of the form _<n> which will be appended to the id.
`...`	(any) See description of dictionary_sugar.
`.keys`	(`character()`) Keys of the objects to construct - possibly with suffixes of the form _<n> which will be appended to the ids.

Value

An element from the dictionary.

Examples

```
d = Dictionary$new()
d$add("a", R6::R6Class("A", public = list(id = "a")))
d$add("b", R6::R6Class("B", public = list(id = "c")))
obj1 = dictionary_sugar_inc_get(d, "a_1")
obj1$id

obj2 = dictionary_sugar_inc_get(d, "b_1")
obj2$id

objs = dictionary_sugar_inc_mget(d, c("a_10", "b_2"))
map(objs, "id")
```

`dictionary_sugar_inc_get_safe` *A Quick Way to Initialize Objects from Dictionaries with Incremented ID*

Description

Covenience wrapper around dictionary_sugar_get_safe and dictionary_sugar_mget_safe to allow easier avoidance of of ID clashes which is useful when the same object is used multiple times and the ids have to be unique. Let <key> be the key of the object to retrieve. When passing the <key>_<n> to this function, where <n> is any natural numer, the object with key <key> is retrieved and the suffix _<n> is appended to the id after the object is constructed.

Usage

```
dictionary_sugar_inc_get_safe(.dict, .key, ...)

dictionary_sugar_inc_mget_safe(.dict, .keys, ...)
```

Arguments

`.dict`	(Dictionary) Dictionary from which to retrieve an element.
`.key`	(`character(1)`) Key of the object to construct - possibly with a suffix of the form _<n> which will be appended to the id.
`...`	(any) See description of dictionary_sugar_get_safe.
`.keys`	(`character()`) Keys of the objects to construct - possibly with suffixes of the form _<n> which will be appended to the ids.

Value

An element from the dictionary.

Examples

```
d = Dictionary$new()
d$add("a", R6::R6Class("A", public = list(id = "a")))
d$add("b", R6::R6Class("B", public = list(id = "c")))
obj1 = dictionary_sugar_inc_get_safe(d, "a_1")
obj1$id

obj2 = dictionary_sugar_inc_get_safe(d, "b_1")
obj2$id

objs = dictionary_sugar_inc_mget_safe(d, c("a_10", "b_2"))
map(objs, "id")
```

`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()`.
- `"try"`: Similar to `"none"`, but catches error. Output is printed to the console and not logged.
- `"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_,
  .timeout = Inf
)
```

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.
`.timeout`	(`numeric(1)`) Timeout in seconds. Uses `setTimeLimit()` for `"none"` and `"evaluate"` encapsulation. For `"callr"` encapsulation, the timeout is passed to `callr::r()`.

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)

if (requireNamespace("evaluate", quietly = TRUE)) {
  encapsulate("evaluate", f, list(n = 1), .seed = 1)
}

if (requireNamespace("callr", quietly = TRUE)) {
  encapsulate("callr", f, list(n = 1), .seed = 1)
}
```

---

enframe *Convert a Named Vector into a data.table and Vice Versa*

Description

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

`deframe()` converts a two-column data.frame to a named vector. If the data.frame only has a single column, an unnamed vector is returned.

Usage

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

deframe(x)
```

Arguments

x	(vector() (enframe()) or data.frame() (deframe())) 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() or named vector.

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 ~ .)
```

format_bib *Format Bibentries in Roxygen*

Description

Operates on a named list of `bibentry()` entries and formats them nicely for documentation with **roxygen2**.

- `format_bib()` is intended to be called in the `@references` section and prints the complete entry using `toRd()`.
- `cite_bib()` returns the family name of the first author (if available, falling back to the complete author name if not applicable) and the year in format `"[LastName] (YYYY)"`.

Usage

```
format_bib(..., bibentries = NULL, envir = parent.frame())

cite_bib(..., bibentries = NULL, envir = parent.frame())
```

Arguments

`...`	(`character()`) One or more names of bibentries.
`bibentries`	(named `list()`) Named list of bibentries.
`envir`	(`environment`) Environment to lookup `bibentries` if not provided.

Value

(`character(1)`).

Examples

```
bibentries = list(checkmate = citation("checkmate"), R = citation())
format_bib("checkmate")
format_bib("R")
cite_bib("checkmate")
cite_bib("checkmate", "R")
```

formulate *Create Formulas*

Description

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

Usage

```
formulate(lhs = character(), rhs = character(), env = NULL, quote = "right")
```

Arguments

lhs	(character()) Left-hand side of formula. Multiple elements will be collapsed with " + ".
rhs	(character()) Right-hand side of formula. Multiple elements will be collapsed with " + ".
env	(environment()) Environment for the new formula. Defaults to NULL.
quote	(character(1)) Which side of the formula to quote? Subset of ("left", "right"), defaulting to "right".

Value

`stats::formula()`.

Examples

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

get_private *Extract Private Fields of R6 Objects*

Description

Provides access to the private members of R6::R6Class objects.

Usage

```
get_private(x)
```

Arguments

x	(any) Object to extract the private members from.

Value

`environment()` of private members, or `NULL` if `x` is not an R6 object.

Examples

```
library(R6)
item = R6Class("Item", private = list(x = 1))$new()
get_private(item)$x
```

---

get_private<- *Assign Value to Private Field*

Description

Convenience function to assign a value to a private field of an R6 instance.

Usage

```
get_private(x, which) <- value
```

Arguments

x	(any) Object whose private field should be modified.
which	(character(1)) Private field that is being modified.
value	(any) Value to assign to the private field.

Value

The R6 instance x, modified in-place. If it is not an R6 instance, NULL is returned.

Examples

```
library(R6)
item = R6Class("Item", private = list(x = 1))$new()
get_private(item)$x
get_private(item, "x") = 2L
get_private(item)$x
```

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())
```

hash_input *Hash Input*

Description

Returns the part of an object to be used to calculate its hash.

Usage

```
hash_input(x)

## S3 method for class '`function`'
hash_input(x)

## S3 method for class 'data.table'
hash_input(x)

## Default S3 method:
hash_input(x)
```

Arguments

x	(any) Object for which to retrieve the hash input.

Methods (by class)

- `hash_input(`function`)`: The formals and the body are returned in a `list()`. This ensures that the bytecode or parent environment are not included. in the hash.
- `hash_input(data.table)`: The data.table is converted to a regular list and `hash_input()` is applied to all elements. The conversion to a list ensures that keys and indices are not included in the hash.
- `hash_input(default)`: Returns the object as is.

---

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()` | atomic `vector()`).
`.y`	(any) Object to test for.

Examples

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

ids *Extract ids from a List of Objects*

Description

None.

Usage

```
ids(xs)
```

Arguments

xs	(list()) Every element must have a slot 'id'.

Value

(character()).

Examples

```
xs = list(a = list(id = "foo", a = 1), bar = list(id = "bar", a = 2))
ids(xs)
```

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)

## S3 method for class '`NULL`'
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 not rely on positional argument matching.

Usage

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

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`	(named `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.
`.timeout`	(`numeric(1)`) Timeout in seconds. Uses `setTimeLimit()`. Note that timeouts are only triggered on a user interrupt, not in compiled code.

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)
```

leanify_r6 *Move all methods of an R6 Class to an environment*

Description

`leanify_r6` moves the content of an `R6::R6Class`'s functions to an environment, usually the package's namespace, to save space during serialization of R6 objects. `leanify_package` move all methods of *all* R6 Classes to an environment.

The function in the class (i.e. the object generator) is replaced by a stump function that does nothing except calling the original function that now resides somewhere else.

It is possible to call this function after the definition of an R6::R6 class inside a package, but it is preferred to use `leanify_package()` to just leanify all R6::R6 classes inside a package.

Usage

```
leanify_r6(cls, env = cls$parent_env)

leanify_package(pkg_env = parent.frame(), skip_if = function(x) FALSE)
```

Arguments

`cls`	(R6::R6Class) Class generator to modify.
`env`	(`environment`) The target environment where the function should be stored. This should be either `cls$parent_env` (default) or one of its parent environments, otherwise the stump function will not find the moved (original code) function.
`pkg_env`	:: `environment` The namespace from which to leanify all R6 classes. Does not have to be a package namespace, but this is the intended usecase.
`skip_if`	:: `function` Function with one argument: Is called for each individual `R6::R6Class`. If it returns `TRUE`, the class is skipped. Default function evaluating to `FALSE` always (i.e. skipping no classes).

Value

`NULL`.

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)
```

---

mlr_callbacks *Dictionary of Callbacks*

Description

A simple Dictionary storing objects of class Callback. Each callback has an associated help page, see `mlr_callbacks_[id]`.

This dictionary can get populated with additional callbacks by add-on packages. As a convention, the key should start with the name of the package, i.e. `package.callback`.

For a more convenient way to retrieve and construct learners, see `clbk()`/`clbks()`.

Usage

```
mlr_callbacks
```

Format

An object of class `DictionaryCallbacks` (inherits from `Dictionary`, `R6`) of length 13.

---

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 = "")
```

---

open_help *Opens a Manual Page*

Description

Simply opens a manual page specified in "package::topic" syntax.

Usage

```
open_help(man)
```

Arguments

man	(character(1)) Manual page to open in "package::topic" syntax.

Value

Nothing.

---

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. The call is not included for warnings and errors.

Usage

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

messagef(msg, ..., wrap = FALSE)

warningf(msg, ..., wrap = FALSE)

stopf(msg, ..., wrap = FALSE)
```

Arguments

msg	(character(1)) Format string passed to base::sprintf().
...	(any) Arguments passed down to base::sprintf().
file	(character(1)) Passed to base::cat().
wrap	(integer(1) | logical(1)) If set to a positive integer, base::strwrap() is used to wrap the string to the provided width. If set to TRUE, the width defaults to 0.9 * getOption("width"). If set to FALSE, wrapping is disabled (default). If wrapping is enabled, all whitespace characters ([[:space:]]) are converted to spaces, and consecutive spaces are converted to a single space.

Examples

```
messagef("
  This is a rather long %s
  on multiple lines
  which will get wrapped.
", "string", wrap = 15)
```

---

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 .

Examples

```
x = data.table::data.table(a = 1:3, b = 3:1)
y = data.table::data.table(c = runif(3))
rcbind(x, y)
```

---

rd_info *Helpers to Create Manual Pages*

Description

rd_info() is an internal generic to generate Rd or markdown code to be used in manual pages. rd_format_string() and rd_format_range() are string functions to assist generating proper Rd code.

Usage

```
rd_info(obj, ...)

rd_format_range(lower, upper)

rd_format_string(str, quote = c("\\dQuote{", "}"))

rd_format_packages(packages)
```

Arguments

obj	(any) Object of the respective class.
...	(any)) Additional arguments.
lower	(numeric(1)) Lower bound.
upper	(numeric(1)) Upper bound.

str	(character()) Vector of strings.
quote	(character()) Quotes to use around each element of x. Will be replicated to lenght 2.
packages	(character()) Vector of package names.

Value

character(), possibly with markdown code.

---

recycle_vectors *Recycle List of Vectors to Common Length*

Description

Repeats all vectors of a list .x to the length of the longest vector using rep() with argument length.out. This operation will only work if the length of the longest vectors is an integer multiple of all shorter vectors, and will throw an exception otherwise.

Usage

```
recycle_vectors(.x)
```

Arguments

.x	(list()).

Value

(list()) with vectors of same size.

Examples

```
recycle_vectors(list(a = 1:3, b = 2))
```

register_namespace_callback *Registers a Callback on Namespace load/unLoad Events*

Description

Register a function `callback` to be called after a namespace is loaded. Calls `callback` once if the namespace has already been loaded before and also adds an unload-hook that removes the load hook.

Usage

```
register_namespace_callback(pkgname, namespace, callback)
```

Arguments

`pkgname`	(`character(1)`) Name of the package which registers the callback.
`namespace`	(`character(1)`) Namespace to react on.
`callback`	(`function()`) Function to call on namespace load.

Value

`NULL`.

reorder_vector *Reorder Vector According to Second Vector*

Description

Returns an integer vector to order vector `x` according to vector `y`.

Usage

```
reorder_vector(x, y, na_last = NA)
```

Arguments

`x`	(`vector()`).
`y`	(`vector()`).
`na_last`	(`logical(1)`) What to do with values in `x` which are not in `y`? • `NA`: Extra values are removed. • `FALSE`: Extra values are moved to the beginning of the new vector. • `TRUE`: Extra values are moved to the end of the new vector.

Value

(integer()).

Examples

```
# x subset of y
x = c("b", "a", "c", "d")
y = letters
x[reorder_vector(x, y)]

# y subset of x
y = letters[1:3]
x[reorder_vector(x, y)]
x[reorder_vector(x, y, na_last = TRUE)]
x[reorder_vector(x, y, na_last = FALSE)]
```

---

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",
  quietly = FALSE
)
```

Arguments

pkgs	(character()) Packages to load.
msg	(character(1)) Message to print on error. Use "%s" as placeholder for the list of packages.
quietly	(logical(1)) If TRUE then returns TRUE if all packages are loaded, otherwise FALSE.

Value

(character()) of loaded packages (invisibly).

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)
```

set_params *Modify Values of a Parameter Set*

Description

Convenience function to modfiy (or overwrite) the values of a paradox::ParamSet.

Usage

```
set_params(.ps, ..., .values = list(), .insert = TRUE)
```

Arguments

.ps	(paradox::ParamSet) The parameter set whose values are changed.
...	(any) Named parameter values.
.values	(list()) Named list with parameter values.
.insert	(logical(1)) Whether to insert the values (old values are being kept, if not overwritten), or to discard the old values. Is TRUE by default.

Examples

```
if (requireNamespace("paradox")) {
  param_set = paradox::ps(a = paradox::p_dbl(), b = paradox::p_dbl())
  param_set$values$a = 0
  set_params(param_set, a = 1, .values = list(b = 2), .insert = TRUE)
  set_params(param_set, a = 3, .insert = FALSE)
  set_params(param_set, b = 4, .insert = TRUE)
}
```

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 = character(), n = Inf, ellipsis = "[...]")
```

Arguments

str	(character()) Vector of strings.
sep	(character(1)) String used to collapse the elements of x.
quote	(character()) Quotes to use around each element of x. Will be replicated to lenght 2.
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)
```

---

to_decimal *Convert a Vector of Bits to a Decimal Number*

Description

Converts a logical vector from binary to decimal. The bit vector may have any length, the last position is the least significant, i.e. bits are multiplied with 2^(n-1), 2^(n-2), ..., 2^1, 2^0 where n is the length of the bit vector.

Usage

```
to_decimal(bits)
```

Arguments

bits	(logical()) Logical vector of input values. Missing values are treated as being FALSE. If bits is longer than 30 elements, an exception is raised.

Value

(integer(1)).

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 Columns*

Description

Transforms list columns to separate columns, possibly by reference. The original columns are removed from the returned table. All non-atomic objects in the list columns are expand to new list column.

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	(logical(1) | character(1)) String to prefix the new column names with. Use "{col}" (without the quotes) as placeholder for the original column name.

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(data.table::copy(x), "value")
unnest(data.table::copy(x), "value", prefix = "{col}.")
```

---

which_min *Index of the Minimum/Maximum Value, with Correction for Ties*

Description

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

Usage

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

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

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.
na_rm	(logical(1)) Remove NAs before computation?

Value

(integer()): Index of the minimum/maximum value. Returns an empty integer vector for empty input vectors and vectors with no non-missing values (if na_rm is TRUE). Returns NA if na_rm is FALSE and at least one NA is found in x.

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))
```

---

with_package *Execture code with a modified search path*

Description

Attaches a package to the search path (if not already attached), executes code and eventually removes the package from the search path again, restoring the previous state.

Note that this function is deprecated in favor of the (now fixed) version in **withr**.

Usage

```
with_package(package, code, ...)
```

Arguments

package	(character(1)) Name of the package to attach.
code	(expression) Code to run.
...	(any) Additional arguments passed to library().

Value

Result of the evaluation of code.

See Also

withr package.

`%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