

Package ‘paradox’

March 17, 2023

Type Package

Title Define and Work with Parameter Spaces for Complex Algorithms

Version 0.11.1

Description Define parameter spaces, constraints and dependencies for arbitrary algorithms, to program on such spaces. Also includes statistical designs and random samplers. Objects are implemented as ‘R6’ classes.

License LGPL-3

URL <https://paradox.mlr-org.com>, <https://github.com/mlr-org/paradox>

BugReports <https://github.com/mlr-org/paradox/issues>

Imports backports, checkmate, data.table, methods, mlr3misc (>= 0.9.4), R6

Suggests knitr, lhs, spacefillr, testthat

Encoding UTF-8

Config/testthat/edition 3

Config/testthat/parallel false

NeedsCompilation no

RoxygenNote 7.2.3

Collate 'Condition.R' 'Design.R' 'NoDefault.R' 'Param.R' 'ParamDbf.R' 'ParamFct.R' 'ParamInt.R' 'ParamLgl.R' 'ParamSet.R' 'ParamSetCollection.R' 'ParamUty.R' 'Sampler.R' 'Sampler1D.R' 'SamplerHierarchical.R' 'SamplerJointIndep.R' 'SamplerUnif.R' 'asserts.R' 'default_values.R' 'helper.R' 'domain.R' 'generate_design_grid.R' 'generate_design_lhs.R' 'generate_design_random.R' 'generate_design_sobol.R' 'ps.R' 'reexports.R' 'to_tune.R' 'zzz.R'

Author Michel Lang [cre, aut] (<<https://orcid.org/0000-0001-9754-0393>>),
Bernd Bischl [aut] (<<https://orcid.org/0000-0001-6002-6980>>),
Jakob Richter [aut] (<<https://orcid.org/0000-0003-4481-5554>>),
Xudong Sun [aut] (<<https://orcid.org/0000-0003-3269-2307>>),
Martin Binder [aut],
Marc Becker [ctb] (<<https://orcid.org/0000-0002-8115-0400>>)

Maintainer Michel Lang <michellang@gmail.com>

Repository CRAN

Date/Publication 2023-03-17 14:30:03 UTC

R topics documented:

paradox-package	3
assert_param	3
Condition	4
default_values	6
Design	7
Domain	8
generate_design_grid	12
generate_design_lhs	13
generate_design_random	14
generate_design_sobol	15
NO_DEF	16
Param	16
ParamDbf	20
ParamFct	22
ParamInt	24
ParamLgl	26
ParamSet	27
ParamSetCollection	34
ParamUty	36
ps	38
psc	40
Sampler	40
Sampler1D	42
Sampler1DCateg	43
Sampler1DNormal	44
Sampler1DRfun	45
Sampler1DUnif	46
SamplerHierarchical	47
SamplerJointIndep	48
SamplerUnif	49
to_tune	50
Index	54

paradox-package	<i>paradox: Define and Work with Parameter Spaces for Complex Algorithms</i>
-----------------	--

Description

Define parameter spaces, constraints and dependencies for arbitrary algorithms, to program on such spaces. Also includes statistical designs and random samplers. Objects are implemented as 'R6' classes.

Author(s)

Maintainer: Michel Lang <michellang@gmail.com> ([ORCID](#))

Authors:

- Bernd Bischl <bernd_bischl@gmx.net> ([ORCID](#))
- Jakob Richter <jakob1richter@gmail.com> ([ORCID](#))
- Xudong Sun <smilesun.east@gmail.com> ([ORCID](#))
- Martin Binder <mlr.developer@mb706.com>

Other contributors:

- Marc Becker <marcbecker@posteo.de> ([ORCID](#)) [contributor]

See Also

Useful links:

- <https://paradox.mlr-org.com>
- <https://github.com/mlr-org/paradox>
- Report bugs at <https://github.com/mlr-org/paradox/issues>

assert_param	<i>Assertions for Params and ParamSets</i>
--------------	--

Description

Assertions for Params and ParamSets

Usage

```

assert_param(param, cl = "Param", no_untyped = FALSE, must_bounded = FALSE)

assert_param_set(
  param_set,
  cl = "Param",
  no_untyped = FALSE,
  must_bounded = FALSE,
  no_deps = FALSE
)

```

Arguments

param	(Param).
cl	(character()) Allowed subclasses.
no_untyped	(logical(1)) Are untyped Params allowed?
must_bounded	(logical(1)) Only bounded Params allowed?
param_set	(ParamSet).
no_deps	(logical(1)) Are dependencies allowed?

Value

The checked object, invisibly.

Condition

Dependency Condition

Description

Condition object, to specify the condition in a dependency.

Currently implemented simple conditions

- `CondEqual$new(rhs)`
Parent must be equal to rhs.
- `CondAnyOf$new(rhs)`
Parent must be any value of rhs.

Public fields

type (character(1))
Name / type of the condition.

rhs (any)
Right-hand-side of the condition.

Methods**Public methods:**

- [Condition\\$new\(\)](#)
- [Condition\\$test\(\)](#)
- [Condition\\$as_string\(\)](#)
- [Condition\\$format\(\)](#)
- [Condition\\$print\(\)](#)
- [Condition\\$clone\(\)](#)

Method `new()`: Creates a new instance of this [R6](#) class.

Usage:

`Condition$new(type, rhs)`

Arguments:

type (character(1))
Name / type of the condition.

rhs (any)
Right-hand-side of the condition.

Method `test()`: Checks if condition is satisfied. Called on a vector of parent param values.

Usage:

`Condition$test(x)`

Arguments:

x (vector()).

Returns: logical(1).

Method `as_string()`: Conversion helper for print outputs.

Usage:

`Condition$as_string(lhs_chr = "x")`

Arguments:

lhs_chr (character(1))

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

Usage:

`Condition$format(...)`

Arguments:

... (ignored).

Method print(): Printer.

Usage:

Condition\$print(...)

Arguments:

... (ignored).

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

Usage:

Condition\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

default_values

Extract Parameter Default Values

Description

Extract parameter default values.

Usage

```
default_values(x, ...)
```

```
## S3 method for class 'ParamSet'
default_values(x, ...)
```

Arguments

x	(any) Object to extract default values from.
...	(any) Additional arguments.

Value

list().

Description

A lightweight wrapper around a [ParamSet](#) and a `data.table::data.table()`, where the latter is a design of configurations produced from the former - e.g., by calling a `generate_design_grid()` or by sampling.

Public fields

`param_set` ([ParamSet](#)).
`data` ([data.table::data.table\(\)](#))
Stored data.

Methods

Public methods:

- [Design\\$new\(\)](#)
- [Design\\$format\(\)](#)
- [Design\\$print\(\)](#)
- [Design\\$transpose\(\)](#)
- [Design\\$clone\(\)](#)

Method `new()`: Creates a new instance of this [R6](#) class.

Usage:

```
Design$new(param_set, data, remove_dupl)
```

Arguments:

`param_set` ([ParamSet](#)).
`data` ([data.table::data.table\(\)](#))
Stored data.
`remove_dupl` (`logical(1)`)
Remove duplicates?

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

Usage:

```
Design$format(...)
```

Arguments:

... (ignored).

Method `print()`: Printer.

Usage:

```
Design$print(...)
```

Arguments:

... (ignored).

Method `transpose()`: Converts data into a list of lists of row-configurations, possibly removes NA entries of inactive parameter values due to unsatisfied dependencies, and possibly calls the `trafo` function of the [ParamSet](#).

Usage:

```
Design$transpose(filter_na = TRUE, trafo = TRUE)
```

Arguments:

`filter_na` (logical(1))

Should NA entries of inactive parameter values due to unsatisfied dependencies be removed?

`trafo` (logical(1))

Should the `trafo` function of the [ParamSet](#) be called?

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Domain

Domain: Parameter Range without an Id

Description

A Domain object is a representation of a single dimension of a [ParamSet](#). Domain objects are used to construct [ParamSets](#), either through the `ps()` short form, or through the `ParamSet$search_space()` mechanism (see `to_tune()`). Domain corresponds to a [Param](#) object, except it does not have an `$id`, and it *does* have a `trafo` and dependencies (`depends`) associated with it. For each of the basic [Param](#) classes ([ParamInt](#), [ParamDb1](#), [ParamLgl](#), [ParamFct](#), and [ParamUty](#)) there is a function constructing a Domain object (`p_int()`, `p_dbl()`, `p_lgl()`, `p_fct()`, `p_uty()`). They each have the same arguments as the corresponding [Param](#) `$new()` function, except without the `id` argument, and with the the additional parameters `trafo`, and `depends`.

Domain objects are representations of parameter ranges and are intermediate objects to be used in short form constructions in `to_tune()` and `ps()`. Because of their nature, they should not be modified by the user. The Domain object's internals are subject to change and should not be relied upon.

Usage

```
p_int(
  lower = -Inf,
  upper = Inf,
  special_vals = list(),
```



```
    default = NO_DEF,
    tags = character(),
    depends = NULL,
    trafo = NULL,
    logscale = FALSE
)

p_dbl(
  lower = -Inf,
  upper = Inf,
  special_vals = list(),
  default = NO_DEF,
  tags = character(),
  tolerance = sqrt(.Machine$double.eps),
  depends = NULL,
  trafo = NULL,
  logscale = FALSE
)

p_uty(
  default = NO_DEF,
  tags = character(),
  custom_check = NULL,
  depends = NULL,
  trafo = NULL
)

p_lgl(
  special_vals = list(),
  default = NO_DEF,
  tags = character(),
  depends = NULL,
  trafo = NULL
)

p_fct(
  levels,
  special_vals = list(),
  default = NO_DEF,
  tags = character(),
  depends = NULL,
  trafo = NULL
)
```

Arguments

lower	(numeric(1)) Lower bound, can be -Inf.
-------	---

upper	(numeric(1)) Upper bound can be +Inf.
special_vals	(list()) Arbitrary special values this parameter is allowed to take, to make it feasible. This allows extending the domain of the parameter. Note that these values are only used in feasibility checks, neither in generating designs nor sampling.
default	(any) Default value. Can be from the domain of the parameter or an element of special_vals. Has value <code>NO_DEF</code> if no default exists. NULL can be a valid default. The value has no effect on <code>ParamSet\$values</code> or the behavior of <code>ParamSet\$check()</code> , <code>\$test()</code> or <code>\$assert()</code> . The default is intended to be used for documentation purposes. ‘
tags	(character()) Arbitrary tags to group and subset parameters. Some tags serve a special purpose: <ul style="list-style-type: none"> • "required" implies that the parameters has to be given when setting values in <code>ParamSet</code>.
depends	(call expression) An expression indicating a requirement for the parameter that will be constructed from this. Can be given as an expression (using <code>quote()</code>), or the expression can be entered directly and will be parsed using NSE (see examples). The expression may be of the form <code><Param> == <value></code> or <code><Param> %in% <values></code> , which will result in dependencies according to <code>ParamSet\$add_dep(on = "<Param>", cond = CondEqual\$new(or <code>ParamSet\$add_dep(on = "<Param>", cond = CondAnyOf\$new(<values>))</code>, respectively (see <code>CondEqual</code>, <code>CondAnyOf</code>). The expression may also contain multiple conditions separated by <code>&&</code>.</code>
trafo	(function) Single argument function performing the transformation of a parameter. When the Domain is used to construct a <code>ParamSet</code> , this transformation will be applied to the corresponding parameter as part of the <code>\$trafo</code> function. Note that the trafo is <i>not</i> inherited by <code>TuneTokens</code> ! Defining a parameter with e.g. <code>p_dbl(..., trafo = ...)</code> will <i>not</i> automatically give the <code>to_tune()</code> assigned to it a transformation. trafo only makes sense for <code>ParamSets</code> that get used as search spaces for optimization or tuning, it is not useful when defining domains or hyperparameter ranges of learning algorithms, because these do not use trafos.
logscale	(logical(1)) Put numeric domains on a log scale. Default FALSE. Log-scale Domains represent parameter ranges where lower and upper bounds are logarithmized, and where a trafo is added that exponentiates sampled values to the original scale. This is <i>not</i> the same as setting <code>trafo = exp</code> , because <code>logscale = TRUE</code> will handle parameter bounds internally: a <code>p_dbl(1, 10, logscale = TRUE)</code> results in a <code>ParamDb1</code> that has lower bound 0, upper bound <code>log(10)</code> , and uses <code>exp</code> transformation on these. Therefore, the given bounds represent the bounds <i>after</i> the transformation. (see examples). <code>p_int()</code> with <code>logscale = TRUE</code> results in a <code>ParamDb1</code> , not a <code>ParamInt</code> , but

with bounds $\log(\max(\text{lower}, 0.5)) \dots \log(\text{upper} + 1)$ and a trafo similar to "as.integer(exp(x))" (with additional bounds correction). The lower bound is lifted to 0.5 if lower 0 to handle the lower == 0 case. The upper bound is increased to $\log(\text{upper} + 1)$ because the trafo would otherwise almost never generate a value of upper.

When logscale is TRUE, then upper bounds may be infinite, but lower bounds should be greater than 0 for p_dbl() or greater or equal 0 for p_int().

Note that "logscale" is *not* inherited by TuneTokens! Defining a parameter with p_dbl(... logscale = TRUE) will *not* automatically give the to_tune() assigned to it log-scale. logscale only makes sense for ParamSets that get used as search spaces for optimization or tuning, it is not useful when defining domains or hyperparameter ranges of learning algorithms, because these do not use trafos.

logscale happens on a natural ($e \approx 2.718282\dots$) basis. Be aware that using a different base ($\log_{10}()/10^{\wedge}$, $\log_2()/2^{\wedge}$) is completely equivalent and does not change the values being sampled after transformation.

tolerance	(numeric(1)) Initializes the \$tolerance field that determines the
custom_check	(function()) Custom function to check the feasibility. Function which checks the input. Must return 'TRUE' if the input is valid and a character(1) with the error message otherwise. This function should <i>not</i> throw an error. Defaults to NULL, which means that no check is performed.
levels	(character atomic list) Allowed categorical values of the parameter. If this is not a character, then a trafo is generated that converts the names (if not given: as.character()) of the values) of the levels argument to the values. This trafo is then performed <i>before</i> the function given as the trafo argument.

Details

The p_fct function admits a levels argument that goes beyond the levels accepted by ParamFct\$new(). Instead of a character vector, any atomic vector or list (optionally named) may be given. (If the value is a list that is not named, the names are inferred using as.character() on the values.) The resulting Domain will correspond to a range of values given by the names of the levels argument with a trafo that maps the character names to the arbitrary values of the levels argument.

Value

A Domain object.

See Also

Other ParamSet construction helpers: ps(), to_tune()

Examples

```
params = ps(
  unbounded_integer = p_int(),
```

```

    bounded_double = p_dbl(0, 10),
    half_bounded_integer = p_dbl(1),
    half_bounded_double = p_dbl(upper = 1),
    double_with_trafo = p_dbl(-1, 1, trafo = exp),
    extra_double = p_dbl(0, 1, special_vals = list("xxx"), tags = "tagged"),
    factor_param = p_fct(c("a", "b", "c")),
    factor_param_with_implicit_trafo = p_fct(list(a = 1, b = 2, c = list()))
  )
print(params)

params$trafo(list(
  bounded_double = 1,
  double_with_trafo = 1,
  factor_param = "c",
  factor_param_with_implicit_trafo = "c"
))

# logscale:
params = ps(x = p_dbl(1, 100, logscale = TRUE))

# The ParamSet has bounds log(1) .. log(100):
print(params)

# When generating a equidistant grid, it is equidistant within log values
grid = generate_design_grid(params, 3)
print(grid)

# But the values are on a log scale with desired bounds after trafo
print(grid$transpose())

# Integer parameters with logscale are `ParamDbl`s pre-trafo
params = ps(x = p_int(0, 10, logscale = TRUE))
print(params)

grid = generate_design_grid(params, 4)
print(grid)

# ... but get transformed to integers.
print(grid$transpose())

```

generate_design_grid *Generate a Grid Design*

Description

Generate a grid with a specified resolution in the parameter space. The resolution for categorical parameters is ignored, these parameters always produce a grid over all their valid levels. For number params the endpoints of the params are always included in the grid.

Usage

```
generate_design_grid(param_set, resolution = NULL, param_resolutions = NULL)
```

Arguments

param_set ([ParamSet](#)).

resolution ([integer\(1\)](#))
Global resolution for all [Params](#).

param_resolutions
 ([named integer\(\)](#))
Resolution per [Param](#), named by parameter ID.

Value

[Design](#).

See Also

Other generate_design: [generate_design_lhs\(\)](#), [generate_design_random\(\)](#), [generate_design_sobol\(\)](#)

Examples

```
ps = ParamSet$new(list(
  ParamDbl$new("ratio", lower = 0, upper = 1),
  ParamFct$new("letters", levels = letters[1:3])
))
generate_design_grid(ps, 10)
```

generate_design_lhs *Generate a Space-Filling LHS Design*

Description

Generate a space-filling design using Latin hypercube sampling. Dependent parameters whose constraints are unsatisfied generate NA entries in their respective columns.

Usage

```
generate_design_lhs(param_set, n, lhs_fun = NULL)
```

Arguments

param_set ([ParamSet](#)).

n ([integer\(1\)](#))
Number of points to sample.

lhs_fun ([function\(n, k\)](#))
Function to use to generate a LHS sample, with n samples and k values per param. LHS functions are implemented in package **lhs**, default is to use [lhs::maximinLHS\(\)](#).

Value

[Design](#).

See Also

Other generate_design: [generate_design_grid\(\)](#), [generate_design_random\(\)](#), [generate_design_sobol\(\)](#)

Examples

```
ps = ParamSet$new(list(
  ParamDbl$new("ratio", lower = 0, upper = 1),
  ParamFct$new("letters", levels = letters[1:3])
))

if (requireNamespace("lhs", quietly = TRUE)) {
  generate_design_lhs(ps, 10)
}
```

generate_design_random

Generate a Random Design

Description

Generates a design with randomly drawn points. Internally uses [SamplerUnif](#), hence, also works for [ParamSets](#) with dependencies. If dependencies do not hold, values are set to NA in the resulting data.table.

Usage

```
generate_design_random(param_set, n)
```

Arguments

param_set	(ParamSet).
n	(integer(1)) Number of points to draw randomly.

Value

[Design](#).

See Also

Other generate_design: [generate_design_grid\(\)](#), [generate_design_lhs\(\)](#), [generate_design_sobol\(\)](#)

Examples

```
ps = ParamSet$new(list(
  ParamDbf$new("ratio", lower = 0, upper = 1),
  ParamFct$new("letters", levels = letters[1:3])
))
generate_design_random(ps, 10)
```

generate_design_sobol *Generate a Space-Filling Sobol Sequence Design*

Description

Generate a space-filling design using a Sobol sequence. Dependent parameters whose constraints are unsatisfied generate NA entries in their respective columns.

Uses [spacefillr::generate_sobol_set](#).

Note that non determinism is achieved by sampling the seed argument via `sample(.Machine$integer.max, size = 1L)`.

Usage

```
generate_design_sobol(param_set, n)
```

Arguments

param_set	(ParamSet).
n	(integer(1)) Number of points to sample.

Value

[Design](#).

See Also

Other generate_design: [generate_design_grid\(\)](#), [generate_design_lhs\(\)](#), [generate_design_random\(\)](#)

Examples

```
ps = ParamSet$new(list(
  ParamDbf$new("ratio", lower = 0, upper = 1),
  ParamFct$new("letters", levels = letters[1:3])
))

if (requireNamespace("spacefillr", quietly = TRUE)) {
  generate_design_sobol(ps, 10)
}
```

NO_DEF	<i>Extra data type for "no default value"</i>
--------	---

Description

Special new data type for no-default. Not often needed by the end-user, mainly internal.

- NoDefault: R6 factory.
- NO_DEF: R6 Singleton object for type, used in [Param](#).
- `is_noddefault()`: Is an object of type 'no default'?

Param	<i>Param Class</i>
-------	--------------------

Description

This is the abstract base class for parameter objects like [ParamDbl](#) and [ParamFct](#).

S3 methods

- `as.data.table()`
[Param](#) -> `data.table::data.table()`
 Converts param to `data.table::data.table()` with 1 row. See [ParamSet](#).

Public fields

`id` (character(1))
 Identifier of the object.

`description` (character(1))
 String to describe this parameter. Used, for example, in `mlr3misc::rd_info()` to automatically generate documentation for parameter sets.

`special_vals` (list())
 Arbitrary special values this parameter is allowed to take.

`default` (any)
 Default value.

`tags` (character())
 Arbitrary tags to group and subset parameters.

Active bindings

class (character(1))
R6 class name. Read-only.

is_number (logical(1))
TRUE if the parameter is of type "dbl" or "int".

is_categ (logical(1))
TRUE if the parameter is of type "fct" or "lgl".

has_default (logical(1))
Is there a default value?

Methods**Public methods:**

- [Param\\$new\(\)](#)
- [Param\\$check\(\)](#)
- [Param\\$assert\(\)](#)
- [Param\\$test\(\)](#)
- [Param\\$rep\(\)](#)
- [Param\\$format\(\)](#)
- [Param\\$print\(\)](#)
- [Param\\$qunif\(\)](#)
- [Param\\$convert\(\)](#)
- [Param\\$clone\(\)](#)

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

Note that this object is typically constructed via derived classes, e.g., [ParamDb1](#).

Usage:

```
Param$new(id, special_vals, default, tags)
```

Arguments:

id (character(1))

Identifier of the object.

special_vals (list())

Arbitrary special values this parameter is allowed to take, to make it feasible. This allows extending the domain of the parameter. Note that these values are only used in feasibility checks, neither in generating designs nor sampling.

default (any)

Default value. Can be from the domain of the parameter or an element of `special_vals`. Has value `NO_DEF` if no default exists. `NULL` can be a valid default. The value has no effect on `ParamSet$values` or the behavior of `ParamSet$check()`, `$test()` or `$assert()`. The default is intended to be used for documentation purposes. ‘

tags (character())

Arbitrary tags to group and subset parameters. Some tags serve a special purpose:

- "required" implies that the parameters has to be given when setting values in [ParamSet](#).

Method `check()`: **checkmate**-like check-function. Take a value from the domain of the parameter, and check if it is feasible. A value is feasible if it is of the same `storage_type`, inside of the bounds or element of `special_vals`.

Usage:

`Param$check(x)`

Arguments:

`x` (any).

Returns: If successful TRUE, if not a string with the error message.

Method `assert()`: **checkmate**-like assert-function. Take a value from the domain of the parameter, and assert if it is feasible. A value is feasible if it is of the same `storage_type`, inside of the bounds or element of `special_vals`.

Usage:

`Param$assert(x)`

Arguments:

`x` (any).

Returns: If successful `x` invisibly, if not an exception is raised.

Method `test()`: **checkmate**-like test-function. Take a value from the domain of the parameter, and test if it is feasible. A value is feasible if it is of the same `storage_type`, inside of the bounds or element of `special_vals`.

Usage:

`Param$test(x)`

Arguments:

`x` (any).

Returns: If successful TRUE, if not FALSE.

Method `rep()`: Repeats this parameter `n`-times (by cloning). Each parameter is named "`[id]rep[k]`" and gets the additional tag "`[id]_rep`".

Usage:

`Param$rep(n)`

Arguments:

`n` (integer(1)).

Returns: [ParamSet](#).

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

Usage:

`Param$format(...)`

Arguments:

`...` (ignored).

Method print(): Printer.

Usage:

```
Param#print(  
    ...,  
    hide_cols = c("nlevels", "is_bounded", "special_vals", "tags", "storage_type")  
)
```

Arguments:

... (ignored).

hide_cols (character())

Which fields should not be printed? Default is "nlevels", "is_bounded", "special_vals", "tags", and "storage_type".

Method qunif(): Takes values from [0,1] and maps them, regularly distributed, to the domain of the parameter. Think of: quantile function or the use case to map a uniform-[0,1] random variable into a uniform sample from this param.

Usage:

```
Param$qunif(x)
```

Arguments:

x (numeric(1)).

Returns: Value of the domain of the parameter.

Method convert(): Converts a value to the closest valid param. Only for values that pass \$check() and mostly used internally.

Usage:

```
Param$convert(x)
```

Arguments:

x (any).

Returns: x converted to a valid type for the Param.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Other Params: [ParamDb1](#), [ParamFct](#), [ParamInt](#), [ParamLgl](#), [ParamUty](#)

ParamDb1	<i>Numerical Parameter</i>
----------	----------------------------

Description

A [Param](#) to describe real-valued parameters.

Super class

[paradox::Param](#) -> ParamDb1

Public fields

lower (numeric(1))

Lower bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

upper (numeric(1))

Upper bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

tolerance (numeric(1))

tolerance of values to accept beyond \$lower and \$upper. Used both for relative and absolute tolerance.

Active bindings

levels (character() | NULL)

Set of allowed levels. Always NULL for [ParamDb1](#), [ParamInt](#) and [ParamUty](#). Always c(TRUE, FALSE) for [ParamLgl](#).

nlevels (integer(1) | Inf)

Number of categorical levels. Always Inf for [ParamDb1](#) and [ParamUty](#). The number of integers in the range [lower, upper], or Inf if unbounded for [ParamInt](#). Always 2 for [ParamLgl](#).

is_bounded (logical(1))

Are the bounds finite? Always TRUE for [ParamFct](#) and [ParamLgl](#). Always FALSE for [ParamUty](#).

storage_type (character(1))

Data type when values of this parameter are stored in a data table or sampled. Always "numeric" for [ParamDb1](#). Always "character" for [ParamFct](#). Always "integer" for [ParamInt](#). Always "logical" for [ParamLgl](#). Always "list" for [ParamUty](#).

Methods**Public methods:**

- [ParamDb1\\$new\(\)](#)
- [ParamDb1\\$convert\(\)](#)
- [ParamDb1\\$clone\(\)](#)

Method [new\(\)](#): Creates a new instance of this R6 class.

Usage:

```
ParamDbl$new(
  id,
  lower = -Inf,
  upper = Inf,
  special_vals = list(),
  default = NO_DEF,
  tags = character(),
  tolerance = sqrt(.Machine$double.eps)
)
```

Arguments:

id (character(1))

Identifier of the object.

lower (numeric(1))

Lower bound, can be -Inf.

upper (numeric(1))

Upper bound can be +Inf.

special_vals (list())

Arbitrary special values this parameter is allowed to take, to make it feasible. This allows extending the domain of the parameter. Note that these values are only used in feasibility checks, neither in generating designs nor sampling.

default (any)

Default value. Can be from the domain of the parameter or an element of special_vals. Has value `NO_DEF` if no default exists. NULL can be a valid default. The value has no effect on ParamSet\$values or the behavior of ParamSet\$check(), \$test() or \$assert(). The default is intended to be used for documentation purposes. ‘

tags (character())

Arbitrary tags to group and subset parameters. Some tags serve a special purpose:

- "required" implies that the parameters has to be given when setting values in [Param-Set](#).

tolerance (numeric(1))

Initializes the \$tolerance field that determines the

Method `convert()`: Restrict the value to within the allowed range. This works in conjunction with \$tolerance, which accepts values slightly out of this range.

Usage:

```
ParamDbl$convert(x)
```

Arguments:

x (numeric(1))

Value to convert.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Note

The upper and lower bounds in `$check()` are expanded by `sqrt(.Machine$double.eps)` to prevent errors due to the precision of double values.

See Also

Other Params: [ParamFct](#), [ParamInt](#), [ParamLgl](#), [ParamUty](#), [Param](#)

Examples

```
ParamDbl$new("ratio", lower = 0, upper = 1, default = 0.5)
```

 ParamFct

Factor Parameter

Description

A [Param](#) to describe categorical (factor) parameters.

Super class

```
paradox::Param -> ParamFct
```

Public fields

```
levels (character() | NULL)
  Set of allowed levels. Always NULL for ParamDbl, ParamInt and ParamUty. Always c(TRUE, FALSE) for ParamLgl.
```

Active bindings

```
lower (numeric(1))
  Lower bound. Always NA for ParamFct, ParamLgl and ParamUty.
upper (numeric(1))
  Upper bound. Always NA for ParamFct, ParamLgl and ParamUty.
nlevels (integer(1) | Inf)
  Number of categorical levels. Always Inf for ParamDbl and ParamUty. The number of integers in the range [lower, upper], or Inf if unbounded for ParamInt. Always 2 for ParamLgl.
is_bounded (logical(1))
  Are the bounds finite? Always TRUE for ParamFct and ParamLgl. Always FALSE for ParamUty.
storage_type (character(1))
  Data type when values of this parameter are stored in a data table or sampled. Always "numeric" for ParamDbl. Always "character" for ParamFct. Always "integer" for ParamInt. Always "logical" for ParamLgl. Always "list" for ParamUty.
```

Methods

Public methods:

- [ParamFct\\$new\(\)](#)
- [ParamFct\\$clone\(\)](#)

Method `new()`: Creates a new instance of this [R6](#) class.

Usage:

```
ParamFct$new(
  id,
  levels,
  special_vals = list(),
  default = NO_DEF,
  tags = character()
)
```

Arguments:

`id` (`character(1)`)

Identifier of the object.

`levels` (`character()`)

Set of allowed levels.

`special_vals` (`list()`)

Arbitrary special values this parameter is allowed to take, to make it feasible. This allows extending the domain of the parameter. Note that these values are only used in feasibility checks, neither in generating designs nor sampling.

`default` (`any`)

Default value. Can be from the domain of the parameter or an element of `special_vals`. Has value `NO_DEF` if no default exists. `NULL` can be a valid default. The value has no effect on `ParamSet$values` or the behavior of `ParamSet$check()`, `$test()` or `$assert()`. The default is intended to be used for documentation purposes. ‘

`tags` (`character()`)

Arbitrary tags to group and subset parameters. Some tags serve a special purpose:

- "required" implies that the parameters has to be given when setting values in [ParamSet](#).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Params: [ParamDb1](#), [ParamInt](#), [ParamLg1](#), [ParamUty](#), [Param](#)

Examples

```
ParamFct$new("f", levels = letters[1:3])
```

ParamInt	<i>Integer Parameter</i>
----------	--------------------------

Description

A [Param](#) to describe integer parameters.

Methods

See [Param](#).

Super class

[paradox::Param](#) -> ParamInt

Public fields

`lower` (numeric(1))
Lower bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

`upper` (numeric(1))
Upper bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

Active bindings

`levels` (character() | NULL)
Set of allowed levels. Always NULL for [ParamDbl](#), [ParamInt](#) and [ParamUty](#). Always c(TRUE, FALSE) for [ParamLgl](#).

`nlevels` (integer(1) | Inf)
Number of categorical levels. Always Inf for [ParamDbl](#) and [ParamUty](#). The number of integers in the range [lower, upper], or Inf if unbounded for [ParamInt](#). Always 2 for [ParamLgl](#).

`is_bounded` (logical(1))
Are the bounds finite? Always TRUE for [ParamFct](#) and [ParamLgl](#). Always FALSE for [ParamUty](#).

`storage_type` (character(1))
Data type when values of this parameter are stored in a data table or sampled. Always "numeric" for [ParamDbl](#). Always "character" for [ParamFct](#). Always "integer" for [ParamInt](#). Always "logical" for [ParamLgl](#). Always "list" for [ParamUty](#).

Methods

Public methods:

- [ParamInt\\$new\(\)](#)
- [ParamInt\\$convert\(\)](#)
- [ParamInt\\$clone\(\)](#)

Method `new()`: Creates a new instance of this [R6](#) class.

Usage:

```
ParamInt$new(
  id,
  lower = -Inf,
  upper = Inf,
  special_vals = list(),
  default = NO_DEF,
  tags = character()
)
```

Arguments:

`id` (`character(1)`)

Identifier of the object.

`lower` (`numeric(1)`)

Lower bound, can be `-Inf`.

`upper` (`numeric(1)`)

Upper bound can be `+Inf`.

`special_vals` (`list()`)

Arbitrary special values this parameter is allowed to take, to make it feasible. This allows extending the domain of the parameter. Note that these values are only used in feasibility checks, neither in generating designs nor sampling.

`default` (`any`)

Default value. Can be from the domain of the parameter or an element of `special_vals`. Has value `NO_DEF` if no default exists. `NULL` can be a valid default. The value has no effect on `ParamSet$values` or the behavior of `ParamSet$check()`, `$test()` or `$assert()`. The default is intended to be used for documentation purposes. ‘

`tags` (`character()`)

Arbitrary tags to group and subset parameters. Some tags serve a special purpose:

- "required" implies that the parameters has to be given when setting values in [Param-Set](#).

Method `convert()`: Converts a value to an integer.

Usage:

```
ParamInt$convert(x)
```

Arguments:

`x` (`numeric(1)`)

Value to convert.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Params: [ParamDbl](#), [ParamFct](#), [ParamLgl](#), [ParamUty](#), [Param](#)

Examples

```
ParamInt$new("count", lower = 0, upper = 10, default = 1)
```

 ParamLgl

Logical Parameter

Description

A [Param](#) to describe logical parameters.

Super class

[paradox::Param](#) -> ParamLgl

Active bindings

lower (numeric(1))

Lower bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

upper (numeric(1))

Upper bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

levels (character() | NULL)

Set of allowed levels. Always NULL for [ParamDbl](#), [ParamInt](#) and [ParamUty](#). Always c(TRUE, FALSE) for [ParamLgl](#).

nlevels (integer(1) | Inf)

Number of categorical levels. Always Inf for [ParamDbl](#) and [ParamUty](#). The number of integers in the range [lower, upper], or Inf if unbounded for [ParamInt](#). Always 2 for [ParamLgl](#).

is_bounded (logical(1))

Are the bounds finite? Always TRUE for [ParamFct](#) and [ParamLgl](#). Always FALSE for [ParamUty](#).

storage_type (character(1))

Data type when values of this parameter are stored in a data table or sampled. Always "numeric" for [ParamDbl](#). Always "character" for [ParamFct](#). Always "integer" for [ParamInt](#). Always "logical" for [ParamLgl](#). Always "list" for [ParamUty](#).

Methods**Public methods:**

- [ParamLgl\\$new\(\)](#)
- [ParamLgl\\$clone\(\)](#)

Method [new\(\)](#): Creates a new instance of this R6 class.

Usage:

```
ParamLgl$new(id, special_vals = list(), default = NO_DEF, tags = character())
```

Arguments:

`id` (character(1))

Identifier of the object.

`special_vals` (list())

Arbitrary special values this parameter is allowed to take, to make it feasible. This allows extending the domain of the parameter. Note that these values are only used in feasibility checks, neither in generating designs nor sampling.

`default` (any)

Default value. Can be from the domain of the parameter or an element of `special_vals`. Has value `NO_DEF` if no default exists. `NULL` can be a valid default. The value has no effect on `ParamSet$values` or the behavior of `ParamSet$check()`, `$test()` or `$assert()`. The default is intended to be used for documentation purposes. ‘

`tags` (character())

Arbitrary tags to group and subset parameters. Some tags serve a special purpose:

- "required" implies that the parameters has to be given when setting values in [ParamSet](#).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Params: [ParamDb1](#), [ParamFct](#), [ParamInt](#), [ParamUty](#), [Param](#)

Examples

```
ParamLgl$new("flag", default = TRUE)
```

 ParamSet

ParamSet

Description

A set of [Param](#) objects. Please note that when creating a set or adding to it, the parameters of the resulting set have to be uniquely named with IDs with valid R names. The set also contains a member variable values which can be used to store an active configuration / or to partially fix some parameters to constant values (regarding subsequent sampling or generation of designs).

S3 methods and type converters

- `as.data.table()`
`ParamSet` -> `data.table::data.table()`
 Compact representation as datatable. Col types are:
 - `id`: character
 - `lower`, `upper`: double
 - `levels`: list col, with NULL elements
 - `special_vals`: list col of list
 - `is_bounded`: logical
 - `default`: list col, with NULL elements
 - `storage_type`: character
 - `tags`: list col of character vectors

Public fields

`assert_values` (logical(1))
 Should values be checked for validity during assignment to active binding \$values? Default is TRUE, only switch this off if you know what you are doing.

Active bindings

`params` (named list())
 List of `Param`, named with their respective ID.

`params_unid` (named list())
 List of `Param`, named with their true ID. However, this field has the `Param`'s `$id` value set to a potentially invalid value. This active binding should only be used internally.

`deps` (`data.table::data.table()`)
 Table has cols `id` (character(1)) and `on` (character(1)) and `cond` (`Condition`). Lists all (direct) dependency parents of a param, through parameter IDs. Internally created by a call to `add_dep`. Settable, if you want to remove dependencies or perform other changes.

`set_id` (character(1))
 ID of this param set. Default `""`. Settable.

`length` (integer(1))
 Number of contained `Params`.

`is_empty` (logical(1))
 Is the `ParamSet` empty?

`class` (named character())
 Classes of contained parameters, named with parameter IDs.

`lower` (named double())
 Lower bounds of parameters (NA if parameter is not numeric). Named with parameter IDs.

`upper` (named double())
 Upper bounds of parameters (NA if parameter is not numeric). Named with parameter IDs.

- `levels` (named `list()`)
List of character vectors of allowed categorical values of contained parameters. NULL if the parameter is not categorical. Named with parameter IDs.
- `nlevels` (named `integer()`)
Number of categorical levels per parameter, Inf for double parameters or unbounded integer parameters. Named with param IDs.
- `is_bounded` (named `logical()`)
Do all parameters have finite bounds? Named with parameter IDs.
- `special_vals` (named `list()` of `list()`)
Special values for all parameters. Named with parameter IDs.
- `default` (named `list()`)
Default values of all parameters. If no default exists, element is not present. Named with parameter IDs.
- `tags` (named `list()` of `character()`)
Can be used to group and subset parameters. Named with parameter IDs.
- `storage_type` (`character()`)
Data types of parameters when stored in tables. Named with parameter IDs.
- `is_number` (named `logical()`)
Position is TRUE for [ParamDbf](#) and [ParamInt](#). Named with parameter IDs.
- `is_categ` (named `logical()`)
Position is TRUE for [ParamFct](#) and [ParamLgl](#). Named with parameter IDs.
- `all_numeric` (`logical(1)`)
Is TRUE if all parameters are [ParamDbf](#) or [ParamInt](#).
- `all_categorical` (`logical(1)`)
Is TRUE if all parameters are [ParamFct](#) and [ParamLgl](#).
- `trafo` (`function(x, param_set)`)
Transformation function. Settable. User has to pass a `function(x, param_set)`, of the form (named `list()`, [ParamSet](#)) -> named `list()`.
The function is responsible to transform a feasible configuration into another encoding, before potentially evaluating the configuration with the target algorithm. For the output, not many things have to hold. It needs to have unique names, and the target algorithm has to accept the configuration. For convenience, the self-paramset is also passed in, if you need some info from it (e.g. tags). Is NULL by default, and you can set it to NULL to switch the transformation off.
- `has_trafo` (`logical(1)`)
Has the set a trafo function?
- `values` (named `list()`)
Currently set / fixed parameter values. Settable, and feasibility of values will be checked when you set them. You do not have to set values for all parameters, but only for a subset. When you set values, all previously set values will be unset / removed.
- `has_deps` (`logical(1)`)
Has the set parameter dependencies?

Methods

Public methods:

- [ParamSet\\$new\(\)](#)
- [ParamSet\\$add\(\)](#)
- [ParamSet\\$ids\(\)](#)
- [ParamSet\\$get_values\(\)](#)
- [ParamSet\\$set_values\(\)](#)
- [ParamSet\\$subset\(\)](#)
- [ParamSet\\$search_space\(\)](#)
- [ParamSet\\$check\(\)](#)
- [ParamSet\\$test\(\)](#)
- [ParamSet\\$assert\(\)](#)
- [ParamSet\\$check_dt\(\)](#)
- [ParamSet\\$test_dt\(\)](#)
- [ParamSet\\$assert_dt\(\)](#)
- [ParamSet\\$add_dep\(\)](#)
- [ParamSet\\$format\(\)](#)
- [ParamSet\\$print\(\)](#)
- [ParamSet\\$clone\(\)](#)

Method `new()`: Creates a new instance of this [R6](#) class.

Usage:

```
ParamSet$new(params = named_list())
```

Arguments:

`params` (`list()`)

List of [Param](#), named with their respective ID. Parameters are cloned.

Method `add()`: Adds a single param or another set to this set, all params are cloned.

Usage:

```
ParamSet$add(p)
```

Arguments:

`p` ([Param](#) | [ParamSet](#)).

Method `ids()`: Retrieves IDs of contained parameters based on some filter criteria selections, NULL means no restriction. Only returns IDs of parameters that satisfy all conditions.

Usage:

```
ParamSet$ids(class = NULL, is_bounded = NULL, tags = NULL)
```

Arguments:

`class` (`character()`).

`is_bounded` (`logical(1)`).

`tags` (`character()`).

Returns: `character()`.

Method `get_values()`: Retrieves parameter values based on some selections, NULL means no restriction and is equivalent to `$values`. Only returns values of parameters that satisfy all conditions.

Usage:

```
ParamSet$get_values(
  class = NULL,
  is_bounded = NULL,
  tags = NULL,
  type = "with_token",
  check_required = TRUE
)
```

Arguments:

```
class (character()).
is_bounded (logical(1)).
tags (character()).
type (character(1))
  Return values with_token, without_token or only_token?
check_required (logical(1))
  Check if all required parameters are set?
```

Returns: Named list().

Method `set_values()`: Allows to to modify (and overwrite) or replace the parameter values. Per default already set values are being kept unless new values are being provided.

Usage:

```
ParamSet$set_values(..., .values = list(), .insert = TRUE)
```

Arguments:

```
... (any)
  Named parameter values.
.values (named list())
  Named list with parameter values. Names must not already appear in ....
.insert (logical(1))
  Whether to insert the values (old values are being kept, if not overwritten), or to replace all
  values. Default is TRUE.
```

Method `subset()`: Changes the current set to the set of passed IDs.

Usage:

```
ParamSet$subset(ids)
```

Arguments:

```
ids (character()).
```

Method `search_space()`: Construct a [ParamSet](#) to tune over. Constructed from [TuneToken](#) in `$values`, see [to_tune\(\)](#).

Usage:

```
ParamSet$search_space(values = self$values)
```

Arguments:

values (named list): optional named list of [TuneToken](#) objects to convert, in place of \$values.

Method `check()`: **checkmate**-like check-function. Takes a named list. A point x is feasible, if it configures a subset of params, all individual param constraints are satisfied and all dependencies are satisfied. Params for which dependencies are not satisfied should not be part of x .

Usage:

```
ParamSet$check(xs)
```

Arguments:

xs (named list()).

Returns: If successful TRUE, if not a string with the error message.

Method `test()`: **checkmate**-like test-function. Takes a named list. A point x is feasible, if it configures a subset of params, all individual param constraints are satisfied and all dependencies are satisfied. Params for which dependencies are not satisfied should not be part of x .

Usage:

```
ParamSet$test(xs)
```

Arguments:

xs (named list()).

Returns: If successful TRUE, if not FALSE.

Method `assert()`: **checkmate**-like assert-function. Takes a named list. A point x is feasible, if it configures a subset of params, all individual param constraints are satisfied and all dependencies are satisfied. Params for which dependencies are not satisfied should not be part of x .

Usage:

```
ParamSet$assert(xs, .var.name = vname(xs))
```

Arguments:

xs (named list()).

.var.name (character(1))

Name of the checked object to print in error messages.

Defaults to the heuristic implemented in [vname](#).

Returns: If successful xs invisibly, if not an error message.

Method `check_dt()`: **checkmate**-like check-function. Takes a [data.table::data.table](#) where rows are points and columns are parameters. A point x is feasible, if it configures a subset of params, all individual param constraints are satisfied and all dependencies are satisfied. Params for which dependencies are not satisfied should be set to NA in xdt.

Usage:

```
ParamSet$check_dt(xdt)
```

Arguments:

xdt ([data.table::data.table](#) | `data.frame()`).

Returns: If successful TRUE, if not a string with the error message.

Method `test_dt()`: **checkmate**-like test-function (s. `$check_dt()`).

Usage:

```
ParamSet$test_dt(xdt)
```

Arguments:

`xdt` ([data.table::data.table](#)).

Returns: If successful TRUE, if not FALSE.

Method `assert_dt()`: **checkmate**-like assert-function (s. `$check_dt()`).

Usage:

```
ParamSet$assert_dt(xdt, .var.name = vname(xdt))
```

Arguments:

`xdt` ([data.table::data.table](#)).

`.var.name` (`character(1)`)

Name of the checked object to print in error messages.

Defaults to the heuristic implemented in [vname](#).

Returns: If successful `xs` invisibly, if not an error message.

Method `add_dep()`: Adds a dependency to this set, so that param `id` now depends on param `on`.

Usage:

```
ParamSet$add_dep(id, on, cond)
```

Arguments:

`id` (`character(1)`).

`on` (`character(1)`).

`cond` ([Condition](#)).

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

Usage:

```
ParamSet$format(...)
```

Arguments:

`...` (ignored).

Method `print()`: Printer.

Usage:

```
ParamSet$print(
  ...,
  hide_cols = c("levels", "is_bounded", "special_vals", "tags", "storage_type")
)
```

Arguments:

`...` (ignored).

`hide_cols` (`character()`)

Which fields should not be printed? Default is "levels", "is_bounded", "special_vals", "tags", and "storage_type".

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
ps = ParamSet$new(
  params = list(
    ParamDbf$new("d", lower = -5, upper = 5, default = 0),
    ParamFct$new("f", levels = letters[1:3])
  )
)

ps$trafo = function(x, param_set) {
  x$d = 2^x$d
  return(x)
}

ps$add(ParamInt$new("i", lower = 0L, upper = 16L))

ps$check(list(d = 2.1, f = "a", i = 3L))
```

ParamSetCollection *ParamSetCollection*

Description

A collection of multiple [ParamSet](#) objects.

- The collection is basically a light-weight wrapper / container around references to multiple sets.
- In order to ensure unique param names, every param in the collection is referred to with "`<set_id>.<param_id>`". Parameters from ParamSets with empty (i.e. `""`) `$set_id` are referenced directly. Multiple ParamSets with `$set_id ""` can be combined, but their parameter names must be unique.
- Operation subset is currently not allowed.
- Operation add currently only works when adding complete sets not single params.
- When you either ask for 'values' or set them, the operation is delegated to the individual, contained param set references. The collection itself does not maintain a values state. This also implies that if you directly change values in one of the referenced sets, this change is reflected in the collection.
- Dependencies: It is possible to currently handle dependencies
 - regarding parameters inside of the same set - in this case simply add the dependency to the set, best before adding the set to the collection

- across sets, where a param from one set depends on the state of a param from another set
- in this case add call `add_dep` on the collection.

If you call `deps` on the collection, you are returned a complete table of dependencies, from sets and across sets.

Super class

`paradox::ParamSet` -> ParamSetCollection

Active bindings

`params` (named `list()`)

List of `Param`, named with their respective ID.

`params_unid` (named `list()`)

List of `Param`, named with their true ID. However, this field has the `Param`'s `$id` value set to a potentially invalid value. This active binding should only be used internally.

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

Table has cols `id` (`character(1)`) and `on` (`character(1)`) and `cond` (`Condition`). Lists all (direct) dependency parents of a param, through parameter IDs. Internally created by a call to `add_dep`. Settable, if you want to remove dependencies or perform other changes.

`values` (named `list()`)

Currently set / fixed parameter values. Settable, and feasibility of values will be checked when you set them. You do not have to set values for all parameters, but only for a subset. When you set values, all previously set values will be unset / removed.

Methods

Public methods:

- `ParamSetCollection$new()`
- `ParamSetCollection$add()`
- `ParamSetCollection$remove_sets()`
- `ParamSetCollection$subset()`
- `ParamSetCollection$clone()`

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

Usage:

```
ParamSetCollection$new(sets)
```

Arguments:

`sets` (`list()` of `ParamSet`)

Parameter objects are cloned.

Method `add()`: Adds a set to this collection.

Usage:

```
ParamSetCollection$add(p)
```

Arguments:

p ([ParamSet](#)).

Method `remove_sets()`: Removes sets of given ids from collection.

Usage:

`ParamSetCollection$remove_sets(ids)`

Arguments:

ids (`character()`).

Method `subset()`: Only included for consistency. Not allowed to perform on [ParamSetCollections](#).

Usage:

`ParamSetCollection$subset(ids)`

Arguments:

ids (`character()`).

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

Usage:

`ParamSetCollection$clone(deep = FALSE)`

Arguments:

deep Whether to make a deep clone.

ParamUty

Untyped Parameter

Description

A [Param](#) to describe untyped parameters.

Super class

`paradox:Param -> ParamUty`

Public fields

`custom_check (function())`

Custom function to check the feasibility.

`repr (character(1))`

Custom field for printing the parameter table.

Active bindings

- `lower` (numeric(1))
Lower bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).
- `upper` (numeric(1))
Upper bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).
- `levels` (character() | NULL)
Set of allowed levels. Always NULL for [ParamDbf](#), [ParamInt](#) and [ParamUty](#). Always `c(TRUE, FALSE)` for [ParamLgl](#).
- `nlevels` (integer(1) | Inf)
Number of categorical levels. Always Inf for [ParamDbf](#) and [ParamUty](#). The number of integers in the range `[lower, upper]`, or Inf if unbounded for [ParamInt](#). Always 2 for [ParamLgl](#).
- `is_bounded` (logical(1))
Are the bounds finite? Always TRUE for [ParamFct](#) and [ParamLgl](#). Always FALSE for [ParamUty](#).
- `storage_type` (character(1))
Data type when values of this parameter are stored in a data table or sampled. Always "numeric" for [ParamDbf](#). Always "character" for [ParamFct](#). Always "integer" for [ParamInt](#). Always "logical" for [ParamLgl](#). Always "list" for [ParamUty](#).

Methods**Public methods:**

- [ParamUty\\$new\(\)](#)
- [ParamUty\\$clone\(\)](#)

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

Usage:

```
ParamUty$new(
  id,
  default = NO_DEF,
  tags = character(),
  custom_check = NULL,
  repr = substitute(default)
)
```

Arguments:

- `id` (character(1))
Identifier of the object.
- `default` (any)
Default value. Can be from the domain of the parameter or an element of `special_vals`. Has value `NO_DEF` if no default exists. NULL can be a valid default. The value has no effect on `ParamSet$values` or the behavior of `ParamSet$check()`, `$test()` or `$assert()`. The default is intended to be used for documentation purposes. ‘
- `tags` (character())
Arbitrary tags to group and subset parameters. Some tags serve a special purpose:

- "required" implies that the parameters has to be given when setting values in [ParamSet](#).

`custom_check` (function())

Custom function to check the feasibility. Function which checks the input. Must return 'TRUE' if the input is valid and a string with the error message otherwise. Defaults to NULL, which means that no check is performed.

`repr` (character(1))

Custom representation string. Used for parameter table in help pages.

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

Usage:

`ParamUty$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Params: [ParamDb1](#), [ParamFct](#), [ParamInt](#), [ParamLgl](#), [Param](#)

Examples

```
ParamUty$new("untyped", default = Inf)
```

ps

Construct a ParamSet using Short Forms

Description

The `ps()` short form constructor uses [Domain](#) objects (`p_db1`, `p_fct`, ...) to construct [ParamSets](#) in a succinct and readable way.

For more specifics also see the documentation of [Domain](#).

Usage

```
ps(..., .extra_trafo = NULL, .allow_dangling_dependencies = FALSE)
```

Arguments

`...` ([Domain](#) | [Param](#))
 Named arguments of [Domain](#) or [Param](#) objects. The [ParamSet](#) will be constructed of the given [Params](#), or of [Params](#) constructed from the given domains. The names of the arguments will be used as `$id` (the `$id` of [Param](#) arguments are ignored).

`.extra_trafo` (function(x, param_set))
 Transformation to set the resulting [ParamSet](#)'s `$trafo` value to. This is in addition to any `trafo` of [Domain](#) objects given in `...`, and will be run *after* transformations of individual parameters were performed.

```
.allow_dangling_dependencies
  (logical)
```

Whether dependencies depending on parameters that are not present should be allowed. A parameter `x` having `depends = y == 0` if `y` is not present in the `ps()` call would usually throw an error, but if dangling dependencies are allowed, the dependency is added regardless. This is usually a bad idea and mainly for internal use. Dependencies between `ParamSets` when using `to_tune()` can be realized using this.

Value

A `ParamSet` object.

See Also

Other `ParamSet` construction helpers: [Domain](#), [to_tune\(\)](#)

Examples

```
pars = ps(
  a = p_int(0, 10),
  b = p_int(upper = 20),
  c = p_dbl(),
  e = p_fct(letters[1:3]),
  f = p_uty(custom_check = checkmate::check_function)
)
print(pars)

pars = ps(
  a = p_dbl(0, 1, trafo = exp),
  b = p_dbl(0, 1, trafo = exp),
  .extra_trafo = function(x, ps) {
    x$c <- x$a + x$b
    x
  }
)

# See how the addition happens after exp()ing:
pars$trafo(list(a = 0, b = 0))

pars$values = list(
  a = to_tune(ps(x = p_int(0, 1),
    .extra_trafo = function(x, param_set) list(a = x$x)
  )),
  # make 'y' depend on 'x', but they are defined in different ParamSets
  # Therefore we need to allow dangling dependencies here.
  b = to_tune(ps(y = p_int(0, 1, depends = x == 1),
    .extra_trafo = function(x, param_set) list(b = x$y),
    .allow_dangling_dependencies = TRUE
  ))
)
```

```
pars$search_space()
```

psc *Create a ParamSet Collection*

Description

Creates a [ParamSetCollection](#).

Usage

```
psc(...)
```

Arguments

... (any)
The [ParamSets](#) from which to create the collection.

Sampler *Sampler Class*

Description

This is the abstract base class for sampling objects like [Sampler1D](#), [SamplerHierarchical](#) or [SamplerJointIndep](#).

Public fields

param_set ([ParamSet](#))
Domain / support of the distribution we want to sample from.

Methods

Public methods:

- [Sampler\\$new\(\)](#)
- [Sampler\\$sample\(\)](#)
- [Sampler\\$format\(\)](#)
- [Sampler\\$print\(\)](#)
- [Sampler\\$clone\(\)](#)

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

Note that this object is typically constructed via derived classes, e.g., [Sampler1D](#).

Usage:

```
Sampler$new(param_set)
```


Arguments:

param_set ([ParamSet](#))

Domain / support of the distribution we want to sample from. ParamSet is cloned on construction.

Method sample(): Sample n values from the distribution.

Usage:

Sampler\$sample(n)

Arguments:

n (integer(1)).

Returns: [Design](#).

Method format(): Helper for print outputs.

Usage:

Sampler\$format(...)

Arguments:

... (ignored).

Method print(): Printer.

Usage:

Sampler\$print(...)

Arguments:

... (ignored).

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

Usage:

Sampler\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DCateg](#), [Sampler1DNormal](#), [Sampler1DRfun](#), [Sampler1DUnif](#), [Sampler1D](#), [SamplerHierarchical](#), [SamplerJointIndep](#), [SamplerUnif](#)

Sampler1D

*Sampler1D Class***Description**

1D sampler, abstract base class for Sampler like [Sampler1DUnif](#), [Sampler1DRfun](#), [Sampler1DCateg](#) and [Sampler1DNormal](#).

Super class

[paradox::Sampler](#) -> Sampler1D

Active bindings

param ([Param](#))
Returns the one Parameter that is sampled from.

Methods**Public methods:**

- [Sampler1D\\$new\(\)](#)
- [Sampler1D\\$clone\(\)](#)

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

Note that this object is typically constructed via derived classes, e.g., [Sampler1DUnif](#).

Usage:

```
Sampler1D$new(param)
```

Arguments:

param ([Param](#))

Domain / support of the distribution we want to sample from.

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

Usage:

```
Sampler1D$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DCateg](#), [Sampler1DNormal](#), [Sampler1DRfun](#), [Sampler1DUnif](#), [SamplerHierarchical](#), [SamplerJointIndep](#), [SamplerUnif](#), [Sampler](#)

Sampler1DCateg	<i>Sampler1DCateg Class</i>
----------------	-----------------------------

Description

Sampling from a discrete distribution, for a [ParamFct](#) or [ParamLgl](#).

Super classes

[paradox::Sampler](#) -> [paradox::Sampler1D](#) -> Sampler1DCateg

Public fields

prob (numeric() | NULL)
Numeric vector of param\$nlevels probabilities.

Methods**Public methods:**

- [Sampler1DCateg\\$new\(\)](#)
- [Sampler1DCateg\\$clone\(\)](#)

Method new(): Creates a new instance of this [R6](#) class.

Usage:

```
Sampler1DCateg$new(param, prob = NULL)
```

Arguments:

param ([Param](#))

Domain / support of the distribution we want to sample from.

prob (numeric() | NULL)

Numeric vector of param\$nlevels probabilities, which is uniform by default.

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

Usage:

```
Sampler1DCateg$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DNormal](#), [Sampler1DRfun](#), [Sampler1DUnif](#), [Sampler1D](#), [SamplerHierarchical](#), [SamplerJointIndep](#), [SamplerUnif](#), [Sampler](#)

Sampler1DNormal *Sampler1DNormal Class*

Description

Normal sampling (potentially truncated) for [ParamDbl](#).

Super classes

[paradox::Sampler](#) -> [paradox::Sampler1D](#) -> [paradox::Sampler1DRfun](#) -> [Sampler1DNormal](#)

Active bindings

mean (numeric(1))
Mean parameter of the normal distribution.

sd (numeric(1))
SD parameter of the normal distribution.

Methods

Public methods:

- [Sampler1DNormal\\$new\(\)](#)
- [Sampler1DNormal\\$clone\(\)](#)

Method [new\(\)](#): Creates a new instance of this [R6](#) class.

Usage:

```
Sampler1DNormal$new(param, mean = NULL, sd = NULL)
```

Arguments:

param ([Param](#))

Domain / support of the distribution we want to sample from.

mean (numeric(1))

Mean parameter of the normal distribution. Default is `mean(c(param$lower, param$upper))`.

sd (numeric(1))

SD parameter of the normal distribution. Default is `(param$upper - param$lower)/4`.

Method [clone\(\)](#): The objects of this class are cloneable with this method.

Usage:

```
Sampler1DNormal$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DCateg](#), [Sampler1DRfun](#), [Sampler1DUnif](#), [Sampler1D](#), [SamplerHierarchical](#), [SamplerJointIndep](#), [SamplerUnif](#), [Sampler](#)

Sampler1DRfun	<i>Sampler1DRfun Class</i>
---------------	----------------------------

Description

Arbitrary sampling from 1D RNG functions from R.

Super classes

[paradox::Sampler](#) -> [paradox::Sampler1D](#) -> Sampler1DRfun

Public fields

`rfun` (function())
 Random number generator function.

`trunc` (logical(1))
 TRUE enables naive rejection sampling, so we stay inside of [lower, upper].

Methods**Public methods:**

- [Sampler1DRfun\\$new\(\)](#)
- [Sampler1DRfun\\$clone\(\)](#)

Method `new()`: Creates a new instance of this [R6](#) class.

Usage:

`Sampler1DRfun$new(param, rfun, trunc = TRUE)`

Arguments:

`param` ([Param](#))

Domain / support of the distribution we want to sample from.

`rfun` (function())

Random number generator function, e.g. `rexp` to sample from exponential distribution.

`trunc` (logical(1))

TRUE enables naive rejection sampling, so we stay inside of [lower, upper].

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

Usage:

`Sampler1DRfun$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DCateg](#), [Sampler1DNormal](#), [Sampler1DUnif](#), [Sampler1D](#), [SamplerHierarchical](#), [SamplerJointIndep](#), [SamplerUnif](#), [Sampler](#)

Sampler1DUnif	<i>Sampler1DUnif Class</i>
---------------	----------------------------

Description

Uniform random sampler for arbitrary (bounded) parameters.

Super classes

[paradox::Sampler](#) -> [paradox::Sampler1D](#) -> Sampler1DUnif

Methods

Public methods:

- [Sampler1DUnif\\$new\(\)](#)
- [Sampler1DUnif\\$clone\(\)](#)

Method `new()`: Creates a new instance of this [R6](#) class.

Usage:

```
Sampler1DUnif$new(param)
```

Arguments:

param ([Param](#))

Domain / support of the distribution we want to sample from.

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

Usage:

```
Sampler1DUnif$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DCateg](#), [Sampler1DNormal](#), [Sampler1DRfun](#), [Sampler1D](#), [SamplerHierarchical](#), [SamplerJointIndep](#), [SamplerUnif](#), [Sampler](#)

SamplerHierarchical *SamplerHierarchical Class*

Description

Hierarchical sampling for arbitrary param sets with dependencies, where the user specifies 1D samplers per param. Dependencies are topologically sorted, parameters are then sampled in topological order, and if dependencies do not hold, values are set to NA in the resulting data . table.

Super class

`paradox::Sampler` -> SamplerHierarchical

Public fields

`samplers (list())`
List of [Sampler1D](#) objects that gives a Sampler for each [Param](#) in the `param_set`.

Methods

Public methods:

- `SamplerHierarchical$new()`
- `SamplerHierarchical$clone()`

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

Usage:

```
SamplerHierarchical$new(param_set, samplers)
```

Arguments:

`param_set` ([ParamSet](#))

Domain / support of the distribution we want to sample from. ParamSet is cloned on construction.

`samplers (list())`

List of [Sampler1D](#) objects that gives a Sampler for each [Param](#) in the `param_set`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DCateg](#), [Sampler1DNormal](#), [Sampler1DRfun](#), [Sampler1DUnif](#), [Sampler1D](#), [SamplerJointIndep](#), [SamplerUnif](#), [Sampler](#)

SamplerJointIndep *SamplerJointIndep Class*

Description

Create joint, independent sampler out of multiple other samplers.

Super class

`paradox::Sampler` -> SamplerJointIndep

Public fields

`samplers (list())`
List of `Sampler` objects.

Methods

Public methods:

- `SamplerJointIndep$new()`
- `SamplerJointIndep$clone()`

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

Usage:

```
SamplerJointIndep$new(samplers)
```

Arguments:

`samplers (list())`
List of `Sampler` objects.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: `Sampler1DCateg`, `Sampler1DNormal`, `Sampler1DRfun`, `Sampler1DUnif`, `Sampler1D`, `SamplerHierarchical`, `SamplerUnif`, `Sampler`

`SamplerUnif`*SamplerUnif Class*

Description

Uniform random sampling for an arbitrary (bounded) [ParamSet](#). Constructs 1 uniform sampler per [Param](#), then passes them to [SamplerHierarchical](#). Hence, also works for [ParamSets](#) sets with dependencies.

Super classes

```
paradox::Sampler -> paradox::SamplerHierarchical -> SamplerUnif
```

Methods

Public methods:

- [SamplerUnif\\$new\(\)](#)
- [SamplerUnif\\$clone\(\)](#)

Method `new()`: Creates a new instance of this [R6](#) class.

Usage:

```
SamplerUnif$new(param_set)
```

Arguments:

`param_set` ([ParamSet](#))

Domain / support of the distribution we want to sample from. [ParamSet](#) is cloned on construction.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DCateg](#), [Sampler1DNormal](#), [Sampler1DRfun](#), [Sampler1DUnif](#), [Sampler1D](#), [SamplerHierarchical](#), [SamplerJointIndep](#), [Sampler](#)

to_tune

*Indicate that a Parameter Value should be Tuned***Description**

to_tune() creates a TuneToken object which can be assigned to the \$values slot of a ParamSet as an alternative to a concrete value. This indicates that the value is not given directly but should be tuned using **bbotk** or **mlr3tuning**. If the thus parameterized object is invoked directly, without being wrapped by or given to a tuner, it will give an error.

The tuning range ParamSet that is constructed from the TuneToken values in a ParamSet's \$values slot can be accessed through the ParamSet\$search_space() method. This is done automatically by tuners if no tuning range is given, but it is also possible to access the \$search_space() method, modify it further, and give the modified ParamSet to a tuning function (or do anything else with it, nobody is judging you).

A TuneToken represents the range over which the parameter whose \$values slot it occupies should be tuned over. It can be constructed via the to_tune() function in one of several ways:

- to_tune(): Indicates a parameter should be tuned over its entire range. Only applies to finite parameters (i.e. discrete or bounded numeric parameters)
- to_tune(lower, upper, logscale): Indicates a numeric parameter should be tuned in the inclusive interval spanning lower to upper, possibly on a log scale if logscale is set to TRUE. All parameters are optional, and the parameter's own lower / upper bounds are used without log scale, by default. Depending on the parameter, integer (if it is a ParamInt) or real values (if it is a ParamDbf) are used. lower, upper, and logscale can be given by position, except when only one of them is given, in which case it must be named to disambiguate from the following cases. When logscale is TRUE, then a trafo is generated automatically that transforms to the given bounds. The bounds are log()'d pre-trafo (see examples). See the logscale argument of Domain functions for more info. Note that "logscale" is *not* inherited from the Param that the TuneToken belongs to! Defining a parameter with p_dbl(... logscale = TRUE) will *not* automatically give the to_tune() assigned to it log-scale.
- to_tune(levels): Indicates a parameter should be tuned through the given discrete values. levels can be any named or unnamed atomic vector or list (although in the unnamed case it must be possible to construct a corresponding character vector with distinct values using as.character).
- to_tune(<Domain>): The given Domain object (constructed e.g. with p_int() or p_fct()) indicates the range which should be tuned over. The supplied trafo function is used for parameter transformation.
- to_tune(<Param>): The given Param object indicates the range which should be tuned over.
- to_tune(<ParamSet>): The given ParamSet is used to tune over a single Param. This is useful for cases where a single evaluation-time parameter value (e.g. ParamUty) is constructed from multiple tuner-visible parameters (which may not be ParamUty). The supplied ParamSet should always contain a \$trafo function, which must always return a list with a single entry.

The TuneToken object's internals are subject to change and should not be relied upon. TuneToken objects should only be constructed via `to_tune()`, and should only be used by giving them to `$values` of a [ParamSet](#).

Usage

```
to_tune(...)
```

Arguments

... if given, restricts the range to be tuning over, as described above.

Value

A TuneToken object.

See Also

Other ParamSet construction helpers: [Domain](#), [ps\(\)](#)

Examples

```
params = ParamSet$new(list(
  ParamInt$new("int", 0, 10),
  ParamInt$new("int_unbounded"),
  ParamDbl$new("dbl", 0, 10),
  ParamDbl$new("dbl_unbounded"),
  ParamDbl$new("dbl_bounded_below", lower = 1),
  ParamFct$new("fct", c("a", "b", "c")),
  ParamUty$new("uty1"),
  ParamUty$new("uty2"),
  ParamUty$new("uty3"),
  ParamUty$new("uty4"),
  ParamUty$new("uty5")
))

params$values = list(

  # tune over entire range of `int`, 0..10:
  int = to_tune(),

  # tune over 2..7:
  int_unbounded = to_tune(2, 7),

  # tune on a log scale in range 1..10;
  # recognize upper bound of 10 automatically, but restrict lower bound to 1:
  dbl = to_tune(lower = 1, logscale = TRUE),
  ## This is equivalent to the following:
  # dbl = to_tune(p_dbl(log(1), log(10), trafo = exp)),

  # nothing keeps us from tuning a dbl over integer values
  dbl_unbounded = to_tune(p_int(1, 10)),
```

```

# tune over values "a" and "b" only
fct = to_tune(c("a", "b")),

# tune over integers 2..8.
# ParamUty needs type information in form of p_xxx() in to_tune.
uty1 = to_tune(p_int(2, 8)),

# tune uty2 like a factor, trying 1, 10, and 100:
uty2 = to_tune(c(1, 10, 100)),

# tune uty3 like a factor. The factor levels are the names of the list
# ("exp", "square"), but the trafo will generate the values from the list.
# This way you can tune an objective that has function-valued inputs.
uty3 = to_tune(list(exp = exp, square = function(x) x^2)),

# tune through multiple parameters. When doing this, the ParamSet in tune()
# must have the trafo that generates a list with one element and the right
# name:
uty4 = to_tune(ps(
  base = p_dbl(0, 1),
  exp = p_int(0, 3),
  .extra_trafo = function(x, param_set) {
    list(uty4 = x$base ^ x$exp)
  }
)),

# not all values need to be tuned!
uty5 = 100
)

print(params$values)

print(params$search_space())

# Change `values` directly and generate new `search_space()` to play around
params$values$uty3 = 8
params$values$uty2 = to_tune(c(2, 4, 8))

print(params$search_space())

# Notice how `logscale` applies `log()` to lower and upper bound pre-trafo:
params = ParamSet$new(list(ParamDbl$new("x")))

params$values$x = to_tune(1, 100, logscale = TRUE)

print(params$search_space())

grid = generate_design_grid(params$search_space(), 3)

# The grid is equidistant within log-bounds pre-trafo:
print(grid)

```

```
# But the values are on a log scale scale with desired bounds after trafo:  
print(grid$transpose())
```

Index

- * **ParamSet construction helpers**
 - Domain, 8
 - ps, 38
 - to_tune, 50
- * **Params**
 - Param, 16
 - ParamDb1, 20
 - ParamFct, 22
 - ParamInt, 24
 - ParamLgl, 26
 - ParamUty, 36
- * **Sampler**
 - Sampler, 40
 - Sampler1D, 42
 - Sampler1DCateg, 43
 - Sampler1DNormal, 44
 - Sampler1DRfun, 45
 - Sampler1DUnif, 46
 - SamplerHierarchical, 47
 - SamplerJointIndep, 48
 - SamplerUnif, 49
- * **generate_design**
 - generate_design_grid, 12
 - generate_design_lhs, 13
 - generate_design_random, 14
 - generate_design_sobol, 15
- assert_param, 3
- assert_param_set (assert_param), 3
- CondAnyOf, 10
- CondAnyOf (Condition), 4
- CondEqual, 10
- CondEqual (Condition), 4
- Condition, 4, 28, 33, 35
- data.table::data.table, 32, 33
- data.table::data.table(), 7, 16, 28, 35
- default_values, 6
- Design, 7, 13–15, 41
- Domain, 8, 38, 39, 50, 51
- generate_design_grid, 12, 14, 15
- generate_design_grid(), 7
- generate_design_lhs, 13, 13, 14, 15
- generate_design_random, 13, 14, 14, 15
- generate_design_sobol, 13, 14, 15
- is_noddefault (NO_DEF), 16
- lhs::maximinLHS(), 13
- mlr3misc::rd_info(), 16
- NO_DEF, 10, 16, 17, 21, 23, 25, 27, 37
- NoDefault (NO_DEF), 16
- p_db1 (Domain), 8
- p_fct (Domain), 8
- p_fct(), 50
- p_int (Domain), 8
- p_int(), 50
- p_lgl (Domain), 8
- p_uty (Domain), 8
- paradox (paradox-package), 3
- paradox-package, 3
- paradox::Param, 20, 22, 24, 26, 36
- paradox::ParamSet, 35
- paradox::Sampler, 42–49
- paradox::Sampler1D, 43–46
- paradox::Sampler1DRfun, 44
- paradox::SamplerHierarchical, 49
- Param, 4, 8, 13, 16, 16, 20, 22–24, 26–28, 30, 35, 36, 38, 42–47, 49, 50
- ParamDb1, 8, 10, 16, 17, 19, 20, 20, 22–24, 26, 27, 29, 37, 38, 44, 50
- ParamFct, 8, 11, 16, 19, 20, 22, 22, 24, 26, 27, 29, 37, 38, 43
- ParamInt, 8, 10, 19, 20, 22–24, 24, 26, 27, 29, 37, 38, 50

ParamLgl, [8](#), [19](#), [20](#), [22–24](#), [26](#), [26](#), [29](#), [37](#), [38](#),
[43](#)

ParamSet, [4](#), [7](#), [8](#), [10](#), [11](#), [13–16](#), [18](#), [21](#), [23](#),
[25](#), [27](#), [27](#), [28–31](#), [34–36](#), [38–41](#), [47](#),
[49–51](#)

ParamSetCollection, [34](#), [36](#), [40](#)

ParamUty, [8](#), [19](#), [20](#), [22–24](#), [26](#), [27](#), [36](#), [37](#), [50](#)

ps, [11](#), [38](#), [51](#)

ps(), [8](#)

psc, [40](#)

R6, [5](#), [7](#), [17](#), [20](#), [23](#), [25](#), [26](#), [30](#), [35](#), [37](#), [40](#),
[42–49](#)

Sampler, [40](#), [42–49](#)

Sampler1D, [40](#), [41](#), [42](#), [43–49](#)

Sampler1DCateg, [41](#), [42](#), [43](#), [44–49](#)

Sampler1DNormal, [41–43](#), [44](#), [45–49](#)

Sampler1DRfun, [41–44](#), [45](#), [46–49](#)

Sampler1DUnif, [41–45](#), [46](#), [47–49](#)

SamplerHierarchical, [40–46](#), [47](#), [48](#), [49](#)

SamplerJointIndep, [40–47](#), [48](#), [49](#)

SamplerUnif, [14](#), [41–48](#), [49](#)

spacefillr::generate_sobol_set, [15](#)

to_tune, [11](#), [39](#), [50](#)

to_tune(), [8](#), [31](#), [39](#)

TuneToken, [10](#), [11](#), [31](#), [32](#)

TuneToken (to_tune), [50](#)

vname, [32](#), [33](#)