# Package 'comparer'

February 20, 2023

**Type** Package

**Title** Compare Output and Run Time

**Version** 0.2.3

**Description** Quickly run experiments to compare the run time and output of
code blocks. The function mbc() can make fast comparisons of code,
and will calculate statistics comparing the resulting outputs.
It can be used to compare model fits to the same data or
see which function runs faster.
The R6 class ffexp$new() runs a function using all possible combinations
of selected inputs. This is useful for comparing the effect of
different parameter values. It can also run in parallel and
automatically save intermediate results, which is very useful
for long computations.

**License** GPL-3

**Encoding** UTF-8

**Imports** R6

**Suggests** plyr, progress, testthat (>= 2.1.0), covr, knitr, GauPro (>=
0.2.7), ggplot2, GGally, ggpubr, ContourFunctions, parallel,
snow, rmarkdown, reshape, tibble, lhs, DiceKriging, DiceOptim,
mixopt, reshape2, microbenchmark

**RoxygenNote** 7.2.3

**URL** <https://github.com/CollinErickson/comparer>

**BugReports** <https://github.com/CollinErickson/comparer/issues>

**VignetteBuilder** knitr

**Language** en-US

**NeedsCompilation** no

**Author** Collin Erickson [aut, cre]

**Maintainer** Collin Erickson <collinberickson@gmail.com>

**Repository** CRAN

**Date/Publication** 2023-02-19 23:00:07 UTC

# R **topics documented:**

---

| ffexp | *Full factorial experiment* |
|---|---|

---

## Description

A class for easily creating and evaluating full factorial experiments.

## Usage

```
e1 <- ffexp$new(eval_func=, )

e1$run_all()

e1$plot_run_times()

e1$save_self()
```

## Arguments

eval_func The function called to evaluate each design point.

... Factors and their levels to be evaluated at.

save_output Should the output be saved?

parallel If TRUE, function evaluations are done in parallel.

parallel_cores Number of cores to be used in parallel. If "detect", parallel::detectCores() is used to determine number. "detect-1" may be used so that the computer isn't running at full capacity, which can slow down other tasks.

## Methods

$new() Initialize an experiment. The preprocessing is done, but no function evaluations are run.

$run_all() Run all factor combinations.

$run_one() Run a single factor combination.

$add_result_of_one() Used to add result of evaluation to data set, don't manually call.

$plot_run_times() Plot the run times. Especially useful when they have been run in parallel.

$save_self() Save ffexp R6 object.

$recover_parallel_temp_save() If you ran the experiment using parallel with parallel_temp_save=TRUE and it crashes partway through, call this to recover the runs that were completed. Runs that were stopped mid-execution are not recoverable.

## Public fields

outrawdf Raw data frame of output.

outcleandf Clean output in data frame.

rungrid matrix specifying which inputs will be run for each experiment.

nvars Number of variables

allvars All variables

varlist Character vector of objects to pass to a parallel cluster.

arglist List of values for each argument

number_runs Total number of runs

completed_runs Logical vector of whether each run has been completed.

eval_func The function that is called for each experiment trial.

outlist A list of the output from each run.

save_output Logical of whether the output should be saved.

parallel Logical whether experiment runs should be run in parallel. Allows for massive speedup.

parallel_cores How many cores to use when running in parallel. Can be an integer, or 'detect' will detect how many cores are available, or 'detect-1' will do one less than that.

parallel_cluster The parallel cluster being used.

folder_path The path to the folder where output will be saved.

verbose How much should be printed when running. 0 is none, 2 is average.

extract_output_to_df A function to extract the raw output into a data frame. E.g., if the output is a list, but you want a single item to show up in the output data frame.

hashvalue A value used to make sure inputs match when reloading.

**Methods**

**Public methods:**

- ffexp$new()
- ffexp$run_all()
- ffexp$run_for_time()
- ffexp$run_superbatch()
- ffexp$run_one()
- ffexp$add_result_of_one()
- ffexp$plot_run_times()
- ffexp$plot_pairs()
- ffexp$plot()
- ffexp$calculate_effects()
- ffexp$calculate_effects2()
- ffexp$save_self()
- ffexp$create_save_folder_if_nonexistent()
- ffexp$rename_save_folder()
- ffexp$delete_save_folder_if_empty()
- ffexp$recover_parallel_temp_save()
- ffexp$rungrid2()
- ffexp$add_variable()
- ffexp$add_level()
- ffexp$remove_results()
- ffexp$print()
- ffexp$set_parallel_cores()
- ffexp$stop_cluster()
- ffexp$finalize()
- ffexp$clone()

**Method** new(): Create an 'ffexp' object.

*Usage:*

```
ffexp$new(
  ...,
  eval_func,
  save_output = FALSE,
  parallel = FALSE,
  parallel_cores = "detect",
  folder_path,
  varlist = NULL,
  verbose = 2,
  extract_output_to_df = NULL
)
```

*Arguments:*

... Input arguments for the experiment

eval_func  The function to be run. It must take named arguments matching the names of ...

save_output  Should output be saved to file?

parallel  Should a parallel cluster be used?

parallel_cores  When running in parallel, how many cores should be used. Not actually the number of cores used, actually the number of clusters created. Can be more than the computer has available, but will hurt performance. Can set to 'detect' to have it detect how many cores are available and use that, or 'detect-1' to use one fewer than there are.

folder_path  Where the data and files should be stored. If not given, a folder in the existing directory will be created.

varlist  Character vector of names of objects that need to be passed to the parallel environment.

verbose  How much should be printed when running. 0 is none, 2 is average.

extract_output_to_df  A function to extract the raw output into a data frame. E.g., if the output is a list, but you want a single item to show up in the output data frame.

**Method** run_all():  Run an experiment. The user can choose to run all rows, or just specified ones, if it should be run in parallel, and what files should be saved.

*Usage:*
```
ffexp$run_all(
  to_run = NULL,
  random_n = NULL,
  redo = FALSE,
  run_order,
  save_output = self$save_output,
  parallel = self$parallel,
  parallel_cores = self$parallel_cores,
  parallel_temp_save = save_output,
  write_start_files = save_output,
  write_error_files = save_output,
  delete_parallel_temp_save_after = FALSE,
  varlist = self$varlist,
  verbose = self$verbose,
  outfile,
  warn_repeat = TRUE
)
```

*Arguments:*

to_run  Which rows should be run? If NULL, then all that haven't been run yet.

random_n  Randomly selects n trials among those not yet completed and runs them.

redo  Should already completed rows be run again?

run_order  In what order should the rows by run? Options: random, in_order, and reverse.

save_output  Should the output be saved?

parallel  Should it be run in parallel?

parallel_cores  When running in parallel, how many cores should be used. Not actually the number of cores used, actually the number of clusters created. Can be more than the computer has available, but will hurt performance. Can set to 'detect' to have it detect how many cores are available and use that, or 'detect-1' to use one fewer than there are.

parallel_temp_save Should temp files be written when running in parallel? Prevents losing
    results if it crashes partway through.

write_start_files Should start files be written?

write_error_files Should error files be written for rows that fail?

delete_parallel_temp_save_after If using parallel temp save files, should they be deleted
    afterwards?

varlist A character vector of names of variables to be passed the the parallel cluster.

verbose How much should be printed when running. 0 is none, 2 is average.

outfile Where should master output file be saved when running in parallel?

warn_repeat Should warnings be given when repeating already completed rows?

**Method** run_for_time(): Run the experiment for a given time, not for a specified number of
trials. Runs 'batch_size' trials between checking the time elapsed, only needs to be more than 1
when running in parallel. It will complete the current batch before stopping, it does not quit in the
middle of the batch when reaching the time limit, so it will go over the time limit given.

*Usage:*

```
ffexp$run_for_time(
  sec,
  batch_size,
  show_time_in_bar = FALSE,
  save_output = self$save_output,
  parallel = self$parallel,
  parallel_cores = self$parallel_cores,
  parallel_temp_save = save_output,
  write_start_files = save_output,
  write_error_files = save_output,
  delete_parallel_temp_save_after = FALSE,
  varlist = self$varlist,
  verbose = self$verbose,
  warn_repeat = TRUE
)
```

*Arguments:*

sec Number of seconds to run for

batch_size Number of trials to run between checking the time elapsed.

show_time_in_bar The progress bar can show either the number of runs completed or the time
    elapsed.

save_output Should the output be saved?

parallel Should it be run in parallel?

parallel_cores When running in parallel, how many cores should be used. Not actually
    the number of cores used, actually the number of clusters created. Can be more than the
    computer has available, but will hurt performance. Can set to 'detect' to have it detect how
    many cores are available and use that, or 'detect-1' to use one fewer than there are.

parallel_temp_save Should temp files be written when running in parallel? Prevents losing
    results if it crashes partway through.

write_start_files Should start files be written?

`write_error_files` Should error files be written for rows that fail?

`delete_parallel_temp_save_after` If using parallel temp save files, should they be deleted afterwards?

`varlist` A character vector of names of variables to be passed the the parallel cluster.

`verbose` How much should be printed when running. 0 is none, 2 is average.

`warn_repeat` Should warnings be given when repeating already completed rows?

**Method** `run_superbatch()`: Run batches. Allows for better progress visualization and saving when running in parallel

*Usage:*
```
ffexp$run_superbatch(
  nsb,
  redo = FALSE,
  run_order,
  save_output = self$save_output,
  parallel = self$parallel,
  parallel_cores = self$parallel_cores,
  parallel_temp_save = save_output,
  write_start_files = save_output,
  write_error_files = save_output,
  delete_parallel_temp_save_after = FALSE,
  varlist = self$varlist,
  verbose = self$verbose,
  warn_repeat = TRUE
)
```

*Arguments:*

`nsb` Number of super batches

`redo` Should already completed rows be run again?

`run_order` In what order should the rows by run? Options: random, in_order, and reverse.

`save_output` Should the output be saved?

`parallel` Should it be run in parallel?

`parallel_cores` When running in parallel, how many cores should be used. Not actually the number of cores used, actually the number of clusters created. Can be more than the computer has available, but will hurt performance. Can set to 'detect' to have it detect how many cores are available and use that, or 'detect-1' to use one fewer than there are.

`parallel_temp_save` Should temp files be written when running in parallel? Prevents losing results if it crashes partway through.

`write_start_files` Should start files be written?

`write_error_files` Should error files be written for rows that fail?

`delete_parallel_temp_save_after` If using parallel temp save files, should they be deleted afterwards?

`varlist` A character vector of names of variables to be passed the the parallel cluster.

`verbose` How much should be printed when running. 0 is none, 2 is average.

`warn_repeat` Should warnings be given when repeating already completed rows?

`outfile` Where should master output file be saved when running in parallel?

**Method** `run_one()`: Run a single row of the experiment. You can specify which one to run. Generally this should not be used by users, use 'run_all' instead.

*Usage:*
```
ffexp$run_one(
  irow = NULL,
  save_output = self$save_output,
  write_start_files = save_output,
  write_error_files = save_output,
  warn_repeat = TRUE,
  is_parallel = FALSE,
  return_list_result_of_one = FALSE,
  verbose = self$verbose,
  force_this_as_output
)
```

*Arguments:*

`irow` Which row should be run?

`save_output` Should the output be saved?

`write_start_files` Should a file be written when starting the experiment?

`write_error_files` Should a file be written if there is an error?

`warn_repeat` Should a warning be given if repeating a row?

`is_parallel` Is this being run in parallel?

`return_list_result_of_one` Should the list of the result of this one be return?

`verbose` How much should be printed when running. 0 is none, 2 is average.

`force_this_as_output` Value to use instead of evaluating function.

**Method** `add_result_of_one()`: Add the result of a single experiment to the object. This shouldn't be used by users.

*Usage:*
```
ffexp$add_result_of_one(
  output,
  systime,
  irow,
  row_grid,
  row_df,
  start_time,
  end_time,
  save_output,
  hashvalue
)
```

*Arguments:*

`output` The output of the experiment.

`systime` The time it took to run

`irow` The row of inputs used.

`row_grid` The corresponding row in the run grid.

`row_df` The corresponding row data frame.

start_time  The start time of the experiment.

end_time  The end time of the experiment.

save_output  Should the output be saved?

hashvalue  Not used.

**Method** plot_run_times(): Plot the run times of each trial.

*Usage:*

ffexp$plot_run_times()

**Method** plot_pairs(): Plot pairs of inputs and outputs. Helps see correlations and distributions.

*Usage:*

ffexp$plot_pairs()

**Method** plot(): Calling 'plot' on an 'ffexp' object calls 'plot_pairs()'

*Usage:*

ffexp$plot()

**Method** calculate_effects(): Calculate the effects of each variable as if this was an experiment using a linear model.

*Usage:*

ffexp$calculate_effects()

**Method** calculate_effects2(): Calculate the effects of each variable as if this was an experiment using a linear model.

*Usage:*

ffexp$calculate_effects2()

**Method** save_self(): Save this R6 object

*Usage:*

ffexp$save_self(verbose = self$verbose)

*Arguments:*

verbose  How much should be printed when running. 0 is none, 2 is average.

**Method** create_save_folder_if_nonexistent(): Create the save folder if it doesn't already exist.

*Usage:*

ffexp$create_save_folder_if_nonexistent()

**Method** rename_save_folder(): Rename the save folder

*Usage:*

ffexp$rename_save_folder(new_folder_path, new_folder_name)

*Arguments:*

new_folder_path  New path for the save folder

new_folder_name If you want the new save folder to be in the current directory, you can use this instead of 'new_folder_path' and just give the folder name.

**Method** `delete_save_folder_if_empty()`: Delete the save folder if it is empty. Used to prevent leaving behind empty folders.

*Usage:*

`ffexp$delete_save_folder_if_empty(verbose = self$verbose)`

*Arguments:*

verbose How much should be printed when running. 0 is none, 2 is average.

**Method** `recover_parallel_temp_save()`: Running this loads the information saved to files if 'save_parallel_temp_save=TRUE' was used when running. Useful when running long jobs in parallel so that you don't lose all results if it crashes before finishing.

*Usage:*

`ffexp$recover_parallel_temp_save(delete_after = FALSE, only_reload_new = FALSE)`

*Arguments:*

delete_after Should the temp files be deleted after they are recovered? If TRUE, make sure you save the ffexp object after running this function so you don't lose the data.

only_reload_new Will only reload output from runs that don't show as completed yet. Can make it much faster if there are many saved files, but most have already been loaded to this object.

**Method** `rungrid2()`: Display the input rows of the experiment. rungrid just gives integers, this gives the actual values.

*Usage:*

`ffexp$rungrid2(rows = 1:nrow(self$rungrid))`

*Arguments:*

rows Which rows to display the inputs for? On big experiments, specifying the rows can be much faster.

**Method** `add_variable()`: Add a variable to the experiment. You must specify the value of the variable for all existing rows, and then also the values of the variable which haven't been run yet.

*Usage:*

`ffexp$add_variable(name, existing_value, new_values, suppressMessage = FALSE)`

*Arguments:*

name Name of the variable being added.

existing_value Which existing argument is a level being added to?

new_values The values of the new variable which have not been run. This should not include 'arg_name', the name of the new variable at the existing values.

suppressMessage Should the message be suppressed? The message tells the user a new variable was added and it is being returned in a new object. Default FALSE.

**Method** `add_level()`: Add a level to one of the arguments. This returns a new object. The existing object is not changed.

*Usage:*

```
ffexp$add_level(arg_name, new_values, suppressMessage = FALSE)
```

*Arguments:*

arg_name  Which existing argument is a level being added to?

new_values  The value of the new levels to be added to 'arg_name'.

suppressMessage  Should the message be suppressed? The message tells the user a new level was added and it is being returned in a new object. Default FALSE.

**Method** `remove_results()`: Remove results of completed trials. They will be rerun next time $run_all() is called.

*Usage:*

```
ffexp$remove_results(to_remove)
```

*Arguments:*

to_remove  Indexes of trials to remove

**Method** `print()`: Printing the object shows some summary information.

*Usage:*

```
ffexp$print()
```

**Method** `set_parallel_cores()`: Set the number of parallel cores to be used when running in parallel. Needed in case user sets "detect"

*Usage:*

```
ffexp$set_parallel_cores(parallel_cores)
```

*Arguments:*

parallel_cores  When running in parallel, how many cores should be used. Not actually the number of cores used, actually the number of clusters created. Can be more than the computer has available, but will hurt performance. Can set to 'detect' to have it detect how many cores are available and use that, or 'detect-1' to use one fewer than there are.

**Method** `stop_cluster()`: Stop the parallel cluster.

*Usage:*

```
ffexp$stop_cluster()
```

**Method** `finalize()`: Cleanup after deleting object.

*Usage:*

```
ffexp$finalize()
```

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

*Usage:*

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

*Arguments:*

deep  Whether to make a deep clone.

**Examples**

```
# Two factors, both with two levels.
#  The evaluation function simply prints out the combination
cc <- ffexp$new(a=1:2,b=c("A","B"),
                eval_func=function(...) {c(...)})
# View the factor settings it will run (each row).
cc$rungrid
# Evaluate all four settings
cc$run_all()


cc <- ffexp$new(a=1:3,b=2, cd=data.frame(c=3:4,d=5:6),
                eval_func=function(...) {list(...)})
```

---

| hype | *Hyperparameter optimization* |
|------|-------------------------------|

---

**Description**

Hyperparameter optimization

**Usage**

```
hype(
  eval_func,
  ...,
  X0 = NULL,
  Z0 = NULL,
  n_lhs,
  extract_output_func,
  verbose = 1,
  model = "GauPro",
  covtype = "matern5_2",
  nugget.estim = TRUE
)
```

**Arguments**

| | |
|---|---|
| eval_func | The function we evaluate. |
| ... | Pass in hyperparameters, such as par_unif() as unnamed arguments. |
| X0 | A data frame of initial points to include. They must have the same names as the hyperparameters. If Z0 is also passed, it should match the points in X0. If Z0 is not passed, then X0 will be the first points evaluated. |
| Z0 | A vector whose values are the result of applying 'eval_func' to each row of X0. |
| n_lhs | The number of random points to start with. They are selected using a Latin hypercube sample. |

extract_output_func

>A function that takes in the output from 'eval_func' and returns the value we are trying to minimize.

verbose How much should be printed? 0 is none, 1 is standard, 2 is more, 5+ is a lot

model What kind of model to use.

covtype The covariance function to use for the Gaussian process model.

nugget.estim Whether a nugget should be estimated when fitting the Gaussian process model.

## Examples

```
# Have df output, but only use one value from it
h1 <- hype(
  eval_func = function(a, b) {data.frame(c=a^2+b^2, d=1:2)},
  extract_output_func = function(odf) {odf$c[1]},
  a = par_unif('a', -1, 2),
  b = par_unif('b', -10, 10),
  n_lhs = 10
)
h1$run_all()
h1$add_EI(n = 1)
h1$run_all()
#system.time(h1$run_EI_for_time(sec=3, batch_size = 1))
#system.time(h1$run_EI_for_time(sec=3, batch_size = 3))
h1$plotorder()
h1$plotX()
```

---

mbc *Model benchmark compare*

---

## Description

Compare the run time and output of various code chunks

## Usage

```
mbc(
  ...,
  times = 5,
  input,
  inputi,
  evaluator,
  post,
  target,
  targetin,
  metric = "rmse",
  paired,
  kfold
)
```

## Arguments

| | |
|---|---|
| `...` | Functions to run |
| `times` | Number of times to run |
| `input` | Object to be passed as input to each function |
| `inputi` | Function to be called with the replicate number then passed to each function. |
| `evaluator` | An expression that the ... expressions will be passed as "." for evaluation. |
| `post` | Function or expression (using ".") to post-process results. |
| `target` | Values the functions are expected to (approximately) return. |
| `targetin` | Values that will be given to the result of the run to produce output. |
| `metric` | c("rmse", "t", "mis90", "sr27") Metric used to compare output values to target. mis90 is the mean interval score for 90% confidence, see Gneiting and Raftery (2007). sr27 is the scoring rule given in Equation 27 of Gneiting and Raftery (2007). |
| `paired` | Should the results be paired for comparison? |
| `kfold` | First element should be the number of elements that are being split into groups. If the number of folds is different from 'times', then the second argument is the number of folds. Use 'ki' in 'inputi' and 'targeti' to select elements in the current fold. |

## Value

Data frame of comparison results

## References

Gneiting, T., & Raftery, A. E. (2007). Strictly proper scoring rules, prediction, and estimation. Journal of the American Statistical Association, 102(477), 359-378.

## Examples

```
# Compare distribution of mean for different sample sizes
mbc(mean(rnorm(1e2)),
    mean(rnorm(1e4)),
    times=20)

# Compare mean and median on same data
mbc(mean(x),
    median(x),
    inputi={x=rexp(1e2)})

# input given, no post
mbc({Sys.sleep(rexp(1, 30));mean(x)},
     {Sys.sleep(rexp(1, 5));median(x)},
    inputi={x=runif(100)})

# input given with post
mbc(mean={Sys.sleep(rexp(1, 30));mean(x)},
```

```
       med={Sys.sleep(rexp(1, 5));median(x)},
       inputi={x=runif(100)},
       post=function(x){c(x+1, x^2)})

# input given with post, 30 times
mbc(mean={Sys.sleep(rexp(1, 30));mean(x)+runif(1)},
    med={Sys.sleep(rexp(1, 50));median(x)+runif(1)},
    inputi={x=runif(100)},
    post=function(x){c(x+1, x^2)}, times=10)

# Name one function and post
mbc({mean(x)+runif(1)},
    a1={median(x)+runif(1)},
    inputi={x=runif(100)},
    post=function(x){c(rr=x+1, gg=x^2)}, times=10)

# No input
m1 <- mbc(mean={x <- runif(100);Sys.sleep(rexp(1, 30));mean(x)},
          med={x <- runif(100);Sys.sleep(rexp(1, 50));median(x)})
```

---

par_discretenum          *Parameter with uniform distribution for hyperparameter optimization*

---

### Description

Parameter with uniform distribution for hyperparameter optimization

### Usage

```
par_discretenum(name, values)
```

### Arguments

| | |
|---|---|
| name | Name of the parameter, must match the input to 'eval_func'. |
| values | Values, discrete numeric |

### Examples

```
p1 <- par_discretenum('x1', 0:2)
class(p1)
print(p1)
```

---

| par_integer | *Parameter with uniform distribution over integer range for hyperparameter optimization* |
|---|---|

---

### Description

Parameter with uniform distribution over integer range for hyperparameter optimization

### Usage

```
par_integer(name, lower, upper)
```

### Arguments

| | |
|---|---|
| name | Name of the parameter, must match the input to 'eval_func'. |
| lower | Lower bound of the parameter |
| upper | Upper bound of the parameter |

### Examples

```
p1 <- par_integer('x1', 3, 8)
class(p1)
print(p1)
table(p1$generate(runif(1000)))
```

---

| par_log10 | *Hyperparameter on log10 scale* |
|---|---|

---

### Description

Hyperparameter on log10 scale

### Usage

```
par_log10(name, lower, upper)
```

### Arguments

| | |
|---|---|
| name | Name of the parameter, must match the input to 'eval_func'. |
| lower | Lower bound of the parameter |
| upper | Upper bound of the parameter |

### Examples

```
p1 <- par_log10('x1', 1e-4, 1e4)
class(p1)
print(p1)
```

---

par_ordered                    *Hyperparameter of discrete (factor) variable*

---

### Description

Hyperparameter of discrete (factor) variable

### Usage

```
par_ordered(name, values)
```

### Arguments

| | |
|---|---|
| name | Name of the parameter, must match the input to 'eval_func'. |
| values | Vector of values |

### Examples

```
p1 <- par_ordered('x1', c('a', 'b', 'c'))
class(p1)
print(p1)
```

---

par_unif                       *Uniform parameter*

---

### Description

Parameter with uniform distribution for hyperparameter optimization

### Usage

```
par_unif(name, lower, upper)
```

### Arguments

| | |
|---|---|
| name | Name of the parameter, must match the input to 'eval_func'. |
| lower | Lower bound of the parameter |
| upper | Upper bound of the parameter |

### Value

Returns an R6 class generated by R6_par_unif.

### Examples

```
p1 <- par_unif('x1', 1, 10)
class(p1)
print(p1)
```

---

par_unordered              *Hyperparameter of discrete (factor) variable*

---

### Description

Hyperparameter of discrete (factor) variable

### Usage

```
par_unordered(name, values)
```

### Arguments

| | |
|---|---|
| name | Name of the parameter, must match the input to 'eval_func'. |
| values | Vector of values |

### Examples

```
p1 <- par_unordered('x1', c('a', 'b', 'c'))
class(p1)
print(p1)
```

---

plot.mbc              *Plot mbc class*

---

### Description

Plot mbc class

### Usage

```
## S3 method for class 'mbc'
plot(x, ...)
```

### Arguments

| | |
|---|---|
| x | Object of class mbc |
| ... | Additional parameters |

### Value

None

## Examples

```
m1 <- mbc(mn= {Sys.sleep(rexp(1, 30));mean(x)},
          med={Sys.sleep(rexp(1, 5));median(x)},
          input=runif(100))
plot(m1)
```

---

print.mbc                        *Print mbc class*

---

## Description

Print mbc class

## Usage

```
## S3 method for class 'mbc'
print(x, ...)
```

## Arguments

| | |
|---|---|
| x | Object of class mbc |
| ... | Additional parameters |

## Value

None

## Examples

```
m1 <- mbc({Sys.sleep(rexp(1, 30));mean(x)},
          {Sys.sleep(rexp(1, 5));median(x)},
          input=runif(100))
print(m1)
```

---

R6_hype                        *Hyperparameter optimization*

---

## Description

Hyperparameter optimization

Hyperparameter optimization

**Public fields**

    X  Data frame of inputs that have been evaluated or will be evaluated next.

    Z  Output at X

    runtime  The time it took to evaluate each row of X

    parnames  Names of the parameters

    parlowerraw  Lower bounds for each parameter on raw scale

    parupperraw  Upper bounds for each parameter on raw scale

    parlowertrans  Lower bounds for each parameter on transformed scale

    paruppertrans  Upper bounds for each parameter on transformed scale

    parlist  List of all parameters

    modlist  A list with details about the model. The user shouldn't ever edit this directly.

    ffexp  An ffexp R6 object used to run the experiment and store the results.

    eval_func  The function we evaluate.

    extract_output_func  A function that takes in the output from 'eval_func' and returns the value
        we are trying to minimize.

    par_all_cts  Are all the parameters continuous?

    verbose  How much should be printed? 0 is none, 1 is standard, 2 is more, 5+ is a lot

**Active bindings**

    mod  Gaussian process model used to predict what the output will be.

**Methods**

    **Public methods:**

- R6_hype$new()
- R6_hype$add_data()
- R6_hype$add_X()
- R6_hype$add_LHS()
- R6_hype$convert_trans_to_raw()
- R6_hype$convert_raw_to_trans()
- R6_hype$change_par_bounds()
- R6_hype$add_EI()
- R6_hype$fit_mod()
- R6_hype$run_all()
- R6_hype$run_EI_for_time()
- R6_hype$plot()
- R6_hype$pairs()
- R6_hype$plotorder()
- R6_hype$plotX()
- R6_hype$plotXorder()

- `R6_hype$plotinteractions()`
- `R6_hype$print()`
- `R6_hype$best_params()`
- `R6_hype$update_mod_userspeclist()`
- `R6_hype$clone()`

**Method** `new()`: Create hype R6 object.

*Usage:*
```
R6_hype$new(
  eval_func,
  ...,
  X0 = NULL,
  Z0 = NULL,
  n_lhs,
  extract_output_func,
  verbose = 1,
  model = "GauPro",
  covtype = "matern5_2",
  nugget.estim = TRUE
)
```

*Arguments:*

`eval_func` The function used to evaluate new points.

`...` Hyperparameters to optimize over.

`X0` Data frame of initial points to run, or points already evaluated. If already evaluated, give in outputs in "Z0"

`Z0` Evaluated outputs at "X0".

`n_lhs` The number that should initially be run using a maximin Latin hypercube.

`extract_output_func` A function that takes in the output from 'eval_func' and returns the value we are trying to minimize.

`verbose` How much should be printed? 0 is none, 1 is standard, 2 is more, 5+ is a lot

`model` What package to fit the Gaussian process model with. Either "GauPro" or "DiceKriging"/"DK".

`covtype` Covariance/correlation/kernel function for the GP model.

`nugget.estim` Should the nugget be estimated when fitting the GP model?

**Method** `add_data()`: Add data to the experiment results.

*Usage:*
```
R6_hype$add_data(X, Z)
```

*Arguments:*

`X` Data frame with names matching the input parameters

`Z` Output at rows of X matching the experiment output.

**Method** `add_X()`: Add new inputs to run. This allows the user to specify what they want run next.

*Usage:*

```
R6_hype$add_X(X)
```

*Arguments:*

X  Data frame with names matching the input parameters.

**Method** add_LHS(): Add new input points using a maximin Latin hypercube. Latin hypercubes are usually more spacing than randomly picking points.

*Usage:*
```
R6_hype$add_LHS(n, just_return_df = FALSE)
```

*Arguments:*

n  Number of points to add.

just_return_df  Instead of adding to experiment, should it just return the new set of values?

**Method** convert_trans_to_raw(): Convert parameters from transformed scale to raw scale.

*Usage:*
```
R6_hype$convert_trans_to_raw(Xtrans)
```

*Arguments:*

Xtrans  Parameters on the transformed scale

**Method** convert_raw_to_trans(): Convert parameters from raw scale to transformed scale.

*Usage:*
```
R6_hype$convert_raw_to_trans(Xraw)
```

*Arguments:*

Xraw  Parameters on the raw scale

**Method** change_par_bounds(): Change lower/upper bounds of a parameter

*Usage:*
```
R6_hype$change_par_bounds(parname, lower, upper)
```

*Arguments:*

parname  Name of the parameter

lower  New lower bound. Leave empty if not changing.

upper  New upper bound. Leave empty if not changing.

**Method** add_EI(): Add new inputs to run using the expected information criteria

*Usage:*
```
R6_hype$add_EI(
  n,
  covtype = NULL,
  nugget.estim = NULL,
  model = NULL,
  eps,
  just_return = FALSE,
  calculate_at
)
```

*Arguments:*

n Number of points to add.

covtype Covariance function to use for the Gaussian process model.

nugget.estim Should a nugget be estimated?

model Which package should be used to fit the model and calculate the EI? Use "DK" for DiceKriging or "GauPro" for GauPro.

eps Exploration parameter. The minimum amount of improvement you care about.

just_return Just return the EI info, don't actually add the points to the design.

calculate_at Calculate the EI at a specific point.

**Method** fit_mod(): Fit model to the data collected so far

*Usage:*
```
R6_hype$fit_mod(covtype = NULL, nugget.estim = NULL, model = NULL)
```

*Arguments:*

covtype Covariance function to use for the Gaussian process model.

nugget.estim Should a nugget be estimated?

model Which package should be used to fit the model and calculate the EI? Use "DK" for DiceKriging or "GauPro" for GauPro.

**Method** run_all(): Run all unevaluated input points.

*Usage:*
```
R6_hype$run_all(...)
```

*Arguments:*

... Passed into 'ffexp$run_all'. Can set 'parallel=TRUE' to evaluate multiple points simultaneously as long as all needed variables have been passed to 'varlist'

**Method** run_EI_for_time(): Add points using the expected information criteria, evaluate them, and repeat until a specified amount of time has passed.

*Usage:*
```
R6_hype$run_EI_for_time(
  sec,
  batch_size,
  covtype = "matern5_2",
  nugget.estim = TRUE,
  verbose = 0,
  model = "GauPro",
  eps = 0,
  ...
)
```

*Arguments:*

sec Number of seconds to run for. It will go over this time limit, finish the current iteration, then stop.

batch_size Number of points to run at once.

covtype Covariance function to use for the Gaussian process model.

nugget.estim  Should a nugget be estimated?

verbose  Verbose parameter to pass to ffexp$

model  Which package should be used to fit the model and calculate the EI? Use "DK" for
   DiceKriging or "GauPro" for GauPro.

eps  Exploration parameter. The minimum amount of improvement you care about.

...  Passed into 'ffexp$run_all'.

**Method** `plot()`:  Make a plot to summarize the experiment.

*Usage:*
```
R6_hype$plot()
```

**Method** `pairs()`:  Plot pairs of inputs and output

*Usage:*
```
R6_hype$pairs()
```

**Method** `plotorder()`:  Plot the output of the points evaluated in order.

*Usage:*
```
R6_hype$plotorder()
```

**Method** `plotX()`:  Plot the output as a function of each input.

*Usage:*
```
R6_hype$plotX(
  addlines = TRUE,
  addEIlines = TRUE,
  covtype = NULL,
  nugget.estim = NULL,
  model = NULL
)
```

*Arguments:*

addlines  Should prediction mean and 95% interval be plotted?

addEIlines  Should expected improvement lines be plotted?

covtype  Covariance function to use for the Gaussian process model.

nugget.estim  Should a nugget be estimated?

model  Which package should be used to fit the model and calculate the EI? Use "DK" for
   DiceKriging or "GauPro" for GauPro.

**Method** `plotXorder()`:  Plot each input in the order they were chosen. Colored by quality.

*Usage:*
```
R6_hype$plotXorder()
```

**Method** `plotinteractions()`:  Plot the 2D plots from inputs to the output. All other variables
are held at their values for the best input.

*Usage:*
```
R6_hype$plotinteractions(covtype = "matern5_2", nugget.estim = TRUE)
```

*Arguments:*

covtype Covariance function to use for the Gaussian process model.

nugget.estim Should a nugget be estimated?

**Method** print(): Print details of the object.

*Usage:*

R6_hype$print(...)

*Arguments:*

... not used

**Method** best_params(): Returns the best parameters evaluated so far.

*Usage:*

R6_hype$best_params()

**Method** update_mod_userspeclist(): Updates the specifications for the GP model.

*Usage:*

```
R6_hype$update_mod_userspeclist(
  model = NULL,
  covtype = NULL,
  nugget.estim = NULL
)
```

*Arguments:*

model What package to fit the Gaussian process model with. Either "GauPro" or "DiceKriging"/"DK".

covtype Covariance/correlation/kernel function for the GP model.

nugget.estim Should the nugget be estimated when fitting the GP model?

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

*Usage:*

R6_hype$clone(deep = FALSE)

*Arguments:*

deep Whether to make a deep clone.

# Examples

```
# Have df output, but only use one value from it
h1 <- hype(
  eval_func = function(a, b) {data.frame(c=a^2+b^2, d=1:2)},
  extract_output_func = function(odf) {odf$c[1]},
  a = par_unif('a', -1, 2),
  b = par_unif('b', -10, 10),
  n_lhs = 10
)
h1$run_all()
h1$add_EI(n = 1)
h1$run_all()
```

```
#system.time(h1$run_EI_for_time(sec=3, batch_size = 1))
#system.time(h1$run_EI_for_time(sec=3, batch_size = 3))
h1$plotorder()
h1$plotX()
```

R6_par_discretenum          *R6 object for discrete numeric*

### Description

R6 object for discrete numeric

R6 object for discrete numeric

### Details

Parameter with uniform distribution for hyperparameter optimization

### Super class

`comparer::par_hype` -> `par_discretenum`

### Public fields

name  Name of the parameter, must match the input to 'eval_func'.

values  Values, discrete numeric

ggtrans  Transformation for ggplot, see ggplot2::scale_x_continuous()

### Methods

#### Public methods:

- `R6_par_discretenum$fromraw()`
- `R6_par_discretenum$toraw()`
- `R6_par_discretenum$generate()`
- `R6_par_discretenum$getseq()`
- `R6_par_discretenum$isvalid()`
- `R6_par_discretenum$convert_to_mopar()`
- `R6_par_discretenum$new()`
- `R6_par_discretenum$print()`
- `R6_par_discretenum$clone()`

**Method** `fromraw()`: Function to convert from raw scale to transformed scale

*Usage:*

`R6_par_discretenum$fromraw(x)`

*Arguments:*

x  Value of raw scale

**Method** `toraw()`: Function to convert from transformed scale to raw scale

*Usage:*
`R6_par_discretenum$toraw(x)`

*Arguments:*
x  Value of transformed scale

**Method** `generate()`: Generate values in the raw space based on quantiles.

*Usage:*
`R6_par_discretenum$generate(q)`

*Arguments:*
q  In [0,1].

**Method** `getseq()`: Get a sequence, uniform on the transformed scale

*Usage:*
`R6_par_discretenum$getseq(n)`

*Arguments:*
n  Number of points. Ignored for discrete.

**Method** `isvalid()`: Check if input is valid for parameter

*Usage:*
`R6_par_discretenum$isvalid(x)`

*Arguments:*
x  Parameter value

**Method** `convert_to_mopar()`: Convert this to a parameter for the mixopt R package.

*Usage:*
`R6_par_discretenum$convert_to_mopar(raw_scale = FALSE)`

*Arguments:*
raw_scale  Should it be on the raw scale?

**Method** `new()`: Create a hyperparameter with uniform distribution

*Usage:*
`R6_par_discretenum$new(name, values)`

*Arguments:*
name  Name of the parameter, must match the input to 'eval_func'.
values  Numeric values, must be in ascending order

**Method** `print()`: Print details of the object.

*Usage:*
`R6_par_discretenum$print(...)`

*Arguments:*

... not used

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

*Usage:*

`R6_par_discretenum$clone(deep = FALSE)`

*Arguments:*

deep  Whether to make a deep clone.

## Examples

```
p1 <- R6_par_discretenum$new('x1', 0:2)
class(p1)
print(p1)
```

---

R6_par_hype                     *Parameter for hyperparameter optimization*

---

## Description

Parameter for hyperparameter optimization

Parameter for hyperparameter optimization

## Public fields

partrans  The transformation type.

## Methods

### Public methods:

- [R6_par_hype$getseq()](R6_par_hype$getseq())
- [R6_par_hype$clone()](R6_par_hype$clone())

**Method** `getseq()`: Get a sequence, uniform on the transformed scale

*Usage:*

`R6_par_hype$getseq(n)`

*Arguments:*

n  Number of points. Ignored for discrete.

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

*Usage:*

`R6_par_hype$clone(deep = FALSE)`

*Arguments:*

deep  Whether to make a deep clone.

## Examples

```
p1 <- R6_par_hype$new()
class(p1)
print(p1)
```

---

R6_par_integer *Parameter with uniform distribution over integer range for hyperparameter optimization*

---

### Description

Parameter with uniform distribution over integer range for hyperparameter optimization

Parameter with uniform distribution over integer range for hyperparameter optimization

### Super class

`comparer::par_hype` -> `par_integer`

### Public fields

name  Name of the parameter, must match the input to 'eval_func'.

lower  Lower bound of the parameter

upper  Upper bound of the parameter

ggtrans  Transformation for ggplot, see ggplot2::scale_x_continuous()

### Methods

#### Public methods:

- `R6_par_integer$fromraw()`
- `R6_par_integer$toraw()`
- `R6_par_integer$generate()`
- `R6_par_integer$getseq()`
- `R6_par_integer$isvalid()`
- `R6_par_integer$convert_to_mopar()`
- `R6_par_integer$new()`
- `R6_par_integer$print()`
- `R6_par_integer$clone()`

**Method** `fromraw()`: Function to convert from raw scale to transformed scale

*Usage:*

`R6_par_integer$fromraw(x)`

*Arguments:*

x  Value of raw scale

**Method** `toraw()`:  Function to convert from transformed scale to raw scale

*Usage:*

`R6_par_integer$toraw(x)`

*Arguments:*

x  Value of transformed scale

**Method** `generate()`:  Generate values in the raw space based on quantiles.

*Usage:*

`R6_par_integer$generate(q)`

*Arguments:*

q  In [0,1].

**Method** `getseq()`:  Get a sequence, uniform on the transformed scale

*Usage:*

`R6_par_integer$getseq(n)`

*Arguments:*

n  Number of points. Ignored for discrete.

**Method** `isvalid()`:  Check if input is valid for parameter

*Usage:*

`R6_par_integer$isvalid(x)`

*Arguments:*

x  Parameter value

**Method** `convert_to_mopar()`:  Convert this to a parameter for the mixopt R package.

*Usage:*

`R6_par_integer$convert_to_mopar(raw_scale = FALSE)`

*Arguments:*

raw_scale  Should it be on the raw scale?

**Method** `new()`:  Create a hyperparameter with uniform distribution

*Usage:*

`R6_par_integer$new(name, lower, upper)`

*Arguments:*

name  Name of the parameter, must match the input to 'eval_func'.

lower  Lower bound of the parameter

upper  Upper bound of the parameter

**Method** `print()`:  Print details of the object.

*Usage:*

`R6_par_integer$print(...)`

*Arguments:*

... not used,

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

*Usage:*

`R6_par_integer$clone(deep = FALSE)`

*Arguments:*

deep  Whether to make a deep clone.

## Examples

```
p1 <- R6_par_integer$new('x1', 0, 2)
class(p1)
print(p1)
```

---

R6_par_log10                    *R6 class for hyperparameter on log10 scale*

---

## Description

R6 class for hyperparameter on log10 scale

R6 class for hyperparameter on log10 scale

## Super class

`comparer::par_hype` -> `par_log10`

## Public fields

name  Name of the parameter, must match the input to 'eval_func'.

lower  Lower bound of the parameter

upper  Upper bound of the parameter

ggtrans  Transformation for ggplot, see ggplot2::scale_x_continuous()

## Methods

### Public methods:

- `R6_par_log10$fromraw()`
- `R6_par_log10$toraw()`
- `R6_par_log10$generate()`
- `R6_par_log10$isvalid()`
- `R6_par_log10$convert_to_mopar()`
- `R6_par_log10$new()`
- `R6_par_log10$print()`
- `R6_par_log10$clone()`

**Method** `fromraw()`:  Function to convert from raw scale to transformed scale

*Usage:*

`R6_par_log10$fromraw(x)`

*Arguments:*

x  Value of raw scale

**Method** `toraw()`:  Function to convert from transformed scale to raw scale

*Usage:*

`R6_par_log10$toraw(x)`

*Arguments:*

x  Value of transformed scale

**Method** `generate()`:  Generate values in the raw space based on quantiles.

*Usage:*

`R6_par_log10$generate(q)`

*Arguments:*

q  In [0,1].

**Method** `isvalid()`:  Check if input is valid for parameter

*Usage:*

`R6_par_log10$isvalid(x)`

*Arguments:*

x  Parameter value

**Method** `convert_to_mopar()`:  Convert this to a parameter for the mixopt R package.

*Usage:*

`R6_par_log10$convert_to_mopar(raw_scale = FALSE)`

*Arguments:*

`raw_scale`  Should it be on the raw scale?

**Method** `new()`:  Create a hyperparameter with uniform distribution

*Usage:*

`R6_par_log10$new(name, lower, upper)`

*Arguments:*

name  Name of the parameter, must match the input to 'eval_func'.

`lower`  Lower bound of the parameter

`upper`  Upper bound of the parameter

**Method** `print()`:  Print details of the object.

*Usage:*

`R6_par_log10$print(...)`

*Arguments:*

... not used

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

*Usage:*

`R6_par_log10$clone(deep = FALSE)`

*Arguments:*

deep Whether to make a deep clone.

## Examples

```
p1 <- par_log10('x1', 1e-4, 1e4)
class(p1)
print(p1)
```

---

R6_par_ordered *R6 class for hyperparameter of discrete (factor) variable*

---

## Description

R6 class for hyperparameter of discrete (factor) variable

R6 class for hyperparameter of discrete (factor) variable

## Super class

`comparer::par_hype -> par_ordered`

## Public fields

name Name of the parameter, must match the input to 'eval_func'.

values Vector of values

ggtrans Transformation for ggplot, see ggplot2::scale_x_continuous()

lower Lower bound of the parameter

upper Upper bound of the parameter

## Methods

**Public methods:**

- `R6_par_ordered$fromraw()`
- `R6_par_ordered$toraw()`
- `R6_par_ordered$fromint()`
- `R6_par_ordered$toint()`
- `R6_par_ordered$generate()`
- `R6_par_ordered$getseq()`
- `R6_par_ordered$isvalid()`

- `R6_par_ordered$convert_to_mopar()`
- `R6_par_ordered$new()`
- `R6_par_ordered$print()`
- `R6_par_ordered$clone()`

**Method** `fromraw()`: Function to convert from raw scale to transformed scale

*Usage:*
`R6_par_ordered$fromraw(x)`

*Arguments:*

x Value of raw scale

**Method** `toraw()`: Function to convert from transformed scale to raw scale

*Usage:*
`R6_par_ordered$toraw(x)`

*Arguments:*

x Value of transformed scale

**Method** `fromint()`: Convert from integer index to actual value

*Usage:*
`R6_par_ordered$fromint(x)`

*Arguments:*

x Integer index

**Method** `toint()`: Convert from value to integer index

*Usage:*
`R6_par_ordered$toint(x)`

*Arguments:*

x Value

**Method** `generate()`: Generate values in the raw space based on quantiles.

*Usage:*
`R6_par_ordered$generate(q)`

*Arguments:*

q In [0,1].

**Method** `getseq()`: Get a sequence, uniform on the transformed scale

*Usage:*
`R6_par_ordered$getseq(n)`

*Arguments:*

n Number of points. Ignored for discrete.

**Method** `isvalid()`: Check if input is valid for parameter

*Usage:*

```
R6_par_ordered$isvalid(x)
```

*Arguments:*

x Parameter value

**Method** `convert_to_mopar()`: Convert this to a parameter for the mixopt R package.

*Usage:*

```
R6_par_ordered$convert_to_mopar(raw_scale = FALSE)
```

*Arguments:*

`raw_scale` Should it be on the raw scale?

**Method** `new()`: Create a hyperparameter with uniform distribution

*Usage:*

```
R6_par_ordered$new(name, values)
```

*Arguments:*

name Name of the parameter, must match the input to 'eval_func'.

values The values the variable can take on.

**Method** `print()`: Print details of the object.

*Usage:*

```
R6_par_ordered$print(...)
```

*Arguments:*

... not used

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

*Usage:*

```
R6_par_ordered$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

## Examples

```
p1 <- par_ordered('x1', c('a', 'b', 'c'))
class(p1)
print(p1)
```

R6_par_unif                    *R6 class for Uniform parameter*

### Description

R6 class for Uniform parameter

R6 class for Uniform parameter

### Details

Parameter with uniform distribution for hyperparameter optimization

### Super class

`comparer::par_hype` -> `par_unif`

### Public fields

name  Name of the parameter, must match the input to 'eval_func'.

lower  Lower bound of the parameter

upper  Upper bound of the parameter

ggtrans  Transformation for ggplot, see ggplot2::scale_x_continuous()

### Methods

#### Public methods:

- `R6_par_unif$fromraw()`
- `R6_par_unif$toraw()`
- `R6_par_unif$generate()`
- `R6_par_unif$isvalid()`
- `R6_par_unif$convert_to_mopar()`
- `R6_par_unif$new()`
- `R6_par_unif$print()`
- `R6_par_unif$clone()`

**Method** `fromraw()`: Function to convert from raw scale to transformed scale

*Usage:*

`R6_par_unif$fromraw(x)`

*Arguments:*

x  Value of raw scale

**Method** `toraw()`: Function to convert from transformed scale to raw scale

*Usage:*

`R6_par_unif$toraw(x)`

*Arguments:*

x  Value of transformed scale

**Method** `generate()`: Generate values in the raw space based on quantiles.

*Usage:*
`R6_par_unif$generate(q)`

*Arguments:*

q  In [0,1].

**Method** `isvalid()`: Check if input is valid for parameter

*Usage:*
`R6_par_unif$isvalid(x)`

*Arguments:*

x  Parameter value

**Method** `convert_to_mopar()`: Convert this to a parameter for the mixopt R package.

*Usage:*
`R6_par_unif$convert_to_mopar(raw_scale = FALSE)`

*Arguments:*

`raw_scale`  Should it be on the raw scale?

**Method** `new()`: Create a hyperparameter with uniform distribution

*Usage:*
`R6_par_unif$new(name, lower, upper)`

*Arguments:*

name  Name of the parameter, must match the input to 'eval_func'.

lower  Lower bound of the parameter

upper  Upper bound of the parameter

**Method** `print()`: Print details of the object.

*Usage:*
`R6_par_unif$print(...)`

*Arguments:*

`...`  not used,

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

*Usage:*
`R6_par_unif$clone(deep = FALSE)`

*Arguments:*

deep  Whether to make a deep clone.

---

R6_par_unordered                *R6 class for hyperparameter of discrete (factor) variable*

---

### Description

R6 class for hyperparameter of discrete (factor) variable

R6 class for hyperparameter of discrete (factor) variable

### Super class

`comparer::par_hype` -> `par_unordered`

### Public fields

`name`  Name of the parameter, must match the input to 'eval_func'.

`values`  Vector of values

`ggtrans`  Transformation for ggplot, see ggplot2::scale_x_continuous()

`lower`  Lower bound of the parameter

`upper`  Upper bound of the parameter

### Methods

#### Public methods:

- `R6_par_unordered$fromraw()`
- `R6_par_unordered$toraw()`
- `R6_par_unordered$fromint()`
- `R6_par_unordered$toint()`
- `R6_par_unordered$generate()`
- `R6_par_unordered$getseq()`
- `R6_par_unordered$isvalid()`
- `R6_par_unordered$convert_to_mopar()`
- `R6_par_unordered$new()`
- `R6_par_unordered$print()`
- `R6_par_unordered$clone()`

**Method** `fromraw()`: Function to convert from raw scale to transformed scale

*Usage:*

`R6_par_unordered$fromraw(x)`

*Arguments:*

`x`  Value of raw scale

**Method** `toraw()`: Function to convert from transformed scale to raw scale

*Usage:*

```
R6_par_unordered$toraw(x)
```

*Arguments:*

x Value of transformed scale

**Method** `fromint()`: Convert from integer index to actual value

*Usage:*

```
R6_par_unordered$fromint(x)
```

*Arguments:*

x Integer index

**Method** `toint()`: Convert from value to integer index

*Usage:*

```
R6_par_unordered$toint(x)
```

*Arguments:*

x Value

**Method** `generate()`: Generate values in the raw space based on quantiles.

*Usage:*

```
R6_par_unordered$generate(q)
```

*Arguments:*

q In [0,1].

**Method** `getseq()`: Get a sequence, uniform on the transformed scale

*Usage:*

```
R6_par_unordered$getseq(n)
```

*Arguments:*

n Number of points. Ignored for discrete.

**Method** `isvalid()`: Check if input is valid for parameter

*Usage:*

```
R6_par_unordered$isvalid(x)
```

*Arguments:*

x Parameter value

**Method** `convert_to_mopar()`: Convert this to a parameter for the mixopt R package.

*Usage:*

```
R6_par_unordered$convert_to_mopar(raw_scale = FALSE)
```

*Arguments:*

raw_scale Should it be on the raw scale?

**Method** `new()`: Create a hyperparameter with uniform distribution

*Usage:*

```
R6_par_unordered$new(name, values)
```

*Arguments:*

name  Name of the parameter, must match the input to 'eval_func'.

values  The values the variable can take on.

**Method** `print()`: Print details of the object.

*Usage:*
```
R6_par_unordered$print(...)
```
*Arguments:*

...  not used

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

*Usage:*
```
R6_par_unordered$clone(deep = FALSE)
```
*Arguments:*

deep  Whether to make a deep clone.

### Examples

```
p1 <- par_unordered('x1', c('a', 'b', 'c'))
class(p1)
print(p1)
```

# Index