Package ‘favr’

December 15, 2025

Title Function Argument Validation

Version 1.0.0

Description Validate function arguments succinctly with informative error messages and optional automatic type casting and size recycling. Enable schema-based assertions by attaching reusable rules to data.frame and list objects for use throughout workflows.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.2

Imports cli, rlang, tidyselect, vctrs

Suggests testthat (>= 3.0.0)

Config/testthat/edition 3

URL `https://lj-jenkins.github.io/favr/`,
`https://github.com/LJ-Jenkins/favr`

Depends R (>= 4.1.0)

BugReports `https://github.com/LJ-Jenkins/favr/issues`

NeedsCompilation no

Author Luke Jenkins [aut, cre, cph] (ORCID:
<`https://orcid.org/0000-0002-7206-7242`>)

Maintainer Luke Jenkins <`luke-jenkins-dev@outlook.com`>

Repository CRAN

Date/Publication 2025-12-15 18:20:02 UTC

Contents

`abort_if_not` *Ensure the truth of R expressions*

Description

If any of the expressions in `...` are not all `TRUE`, abort is called for the first expression which was not (all) `TRUE`. The names of expressions can be used as the error message or a single default error message can be given using `.message`. Both are passed to format_inline for formatting.

Usage

```
abort_if_not(..., .message = NULL, .error_call = caller_env())

abort_if(..., .message = NULL, .error_call = caller_env())
```

Arguments

`...`	any number of R expressions, which should each evaluate to (a logical vector of all) `TRUE` for no error to occur. Positive numbers are not `TRUE`, even when they are coerced to `TRUE` inside `if()` or in arithmetic computations in R. If the expressions are named, the names will be used in the error message.
`.message`	single default error message for non-named expressions.
`.error_call`	the call environment to use for error messages (passed to abort).

Details

abort_if is the opposite of abort_if_not, i.e. expressions should evaluate to (all) `FALSE` for no error to occur. See enforce and schema for a non data-masked and data-masked version of abort_if_not with options for size recycling and type casting.

Value

NULL, called for side effects only.

Examples

```
# NB: Some of these examples are expected to produce an error. To
#     prevent them from terminating a run with example() they are
#     piped into a call to try().

abort_if_not(1 == 1, all.equal(pi, 3.14159265), 1 < 2) # all TRUE

m <- matrix(c(1, 3, 3, 1), 2, 2)
abort_if_not(m == t(m), diag(m) == rep(1, 2)) # all TRUE

abort_if_not(1) |> try()

# A custom error message can be given for each expression:
m[1, 2] <- 12
abort_if_not("{.var m} must be {.cls symmetric}" = m == t(m)) |>
  try()

# Alternatively, one error message can be used for all
# expressions:
abort_if_not(
  m[1, 1] == 1,
  diag(m) == rep(2, 2),
  .message = "{.var m} has a diagonal of: {diag(m)}"
) |> try()

# The `.error_call` argument can be used to specify where the
# error occurs, by default this is the caller environment:
myfunc <- function(x) abort_if_not(x)
myfunc(FALSE) |> try()

# abort_if() errors if any argument does not evaluate to
# (all) FALSE:
abort_if(1 == 1) |> try()

# Injection can be used:
x <- "my error"
abort_if_not({{ x }} := FALSE) |> try()
abort_if_not(!!x := FALSE) |> try()
abort_if_not(FALSE, .message = "{x}") |> try()

x <- list("my {.var bang-bang-bang} error" = FALSE)
abort_if_not(!!!x) |> try()
```

---

are-bare-type-predicates

Bare type predicates

Description

Wrappers around rlang type predicates that allow multiple objects to be passed. The following documentation is adapted from the rlang documentation:

These predicates check for a given type but only return `TRUE` for bare R objects. Bare objects have no class attributes. For example, a data frame is a list, but not a bare list.

- The predicates for vectors include the `.n` argument for pattern-matching on the vector length.
- Like `are_atomic()` and unlike base R `is.atomic()` for R < 4.4.0, `are_bare_atomic()` does not return `TRUE` for `NULL`. Starting in R 4.4.0, `is.atomic(NULL)` returns FALSE.
- Unlike base R `is.numeric()`, `are_bare_double()` only returns `TRUE` for floating point numbers.

Usage

```
are_bare_list(..., .n = NULL, .all = FALSE)

are_bare_atomic(..., .n = NULL, .all = FALSE)

are_bare_vector(..., .n = NULL, .all = FALSE)

are_bare_integer(..., .n = NULL, .all = FALSE)

are_bare_double(..., .n = NULL, .all = FALSE)

are_bare_complex(..., .n = NULL, .all = FALSE)

are_bare_character(..., .n = NULL, .all = FALSE)

are_bare_string(..., .n = NULL, .all = FALSE)

are_bare_logical(..., .n = NULL, .all = FALSE)

are_bare_raw(..., .n = NULL, .all = FALSE)

are_bare_bytes(..., .n = NULL, .all = FALSE)

are_bare_numeric(..., .n = NULL, .all = FALSE)
```

Arguments

`...`	Objects to be tested.
`.n`	Expected lengths of the vectors.
`.all`	Whether to return if all arguments are TRUE.

Details

The optional input of `.n` can be given values that map to the arguments in `...`. If a unnamed vector/list, the input must either be the same length as the number of arguments given to `...`, or length 1: which is then recycled to the number number of arguments given to `...`. Alternatively, a named vector/list can be given, where the values for matching named elements are passed to the type predicate, but unmatched names are passed NULL.

Value

Named logical, or unnamed boolean if `.all` is `TRUE`.

See Also

are-type-predicates, are-scalar-type-predicates

Examples

```
x <- 1
y <- list()
class(y) <- c("my_class", class(y))
z <- mean

are_bare_list(x, y, z, list(1))

# `.all` can be given to test if all inputs
# evaluate to TRUE
are_bare_list(x, y, z, list(1), .all = TRUE)

# scalar inputs to `.n` are recycled to number of inputs
are_bare_list(x, y, z, list(1), .n = 2)

# inputs to `.n` matching the number of inputs
# are applied sequentially
are_bare_list(list(), y, list(1, 2, 3), list(1), .n = c(0, 0, 3, 1))

# named inputs to `.n` are applied to the matching input
# names, with the other inputs being given NULL
x <- list()
are_bare_list(x, y, list(1, 2, 3), list(1), .n = c(x = 5, "list(1)" = 2))
```

`are-scalar-type-predicates` *Scalar type predicates*

Description

Wrappers around rlang scalar type predicates that allow multiple objects to be passed. The following documentation is adapted from the rlang documentation:

These predicates check for a given type and whether the vector is "scalar", that is, of length 1.

In addition to the length check, `are_string()` and `are_bool()` return `FALSE` if their input is missing. This is useful for type-checking arguments, when your function expects a single string or a single `TRUE` or `FALSE`.

Usage

```
are_scalar_list(..., .all = FALSE)

are_scalar_atomic(..., .all = FALSE)

are_scalar_vector(..., .all = FALSE)

are_scalar_integer(..., .all = FALSE)

are_scalar_double(..., .all = FALSE)

are_scalar_complex(..., .all = FALSE)

are_scalar_character(..., .all = FALSE)

are_string(..., .string = NULL, .all = FALSE)

are_scalar_logical(..., .all = FALSE)

are_bool(..., .all = FALSE)

are_scalar_raw(..., .all = FALSE)

are_scalar_bytes(..., .all = FALSE)
```

Arguments

`...`	Objects to be tested.
`.all`	Whether to return if all arguments are TRUE.
`.string`	A string/character vector to compare to the inputs.

Details

The optional input of `.string` can be given character vectors that map to the arguments in `...`. If unnamed vector/list, the input must either be the same length as the number of arguments given to `...`, or length 1: which is then recycled to the number number of arguments given to `...`. Alternatively, a named vector/list can be given, where the values for matching named elements are passed to the type predicate, but unmatched names are passed NULL. List inputs can pass different character vectors for each dot argument. When a character vector is given for a single argument, `TRUE` is returned if at least one element is equal.

Value

Named logical, or unnamed boolean if `.all` is `TRUE`.

See Also

are-type-predicates, are-bare-type-predicates

Examples

```
x <- 1
y <- list()
z <- mean

are_scalar_list(x, y, z, list(1))

# `.all` can be given to test if all inputs
# evaluate to TRUE
are_list(x, y, z, list(1), .all = TRUE)
```

`are-type-predicates` *Type predicates*

Description

Wrappers around rlang type predicates that allow multiple objects to be passed. The following documentation is adapted from the rlang documentation:

These type predicates aim to make type testing in R more consistent. They are wrappers around `base::typeof()`, so operate at a level beneath S3/S4 etc.

Compared to base R functions:

- The predicates for vectors include the `.n` argument for pattern-matching on the vector length.
- Unlike `is.atomic()` in R < 4.4.0, `are_atomic()` does not return `TRUE` for `NULL`. Starting in R 4.4.0 `is.atomic(NULL)` returns FALSE.
- Unlike `is.vector()`, `are_vector()` tests if an object is an atomic vector or a list. `is.vector` checks for the presence of attributes (other than name).

Usage

```
are_list(..., .n = NULL, .all = FALSE)

are_atomic(..., .n = NULL, .all = FALSE)

are_vector(..., .n = NULL, .all = FALSE)

are_integer(..., .n = NULL, .all = FALSE)

are_double(..., .n = NULL, .finite = NULL, .all = FALSE)

are_complex(..., .n = NULL, .finite = NULL, .all = FALSE)

are_character(..., .n = NULL, .all = FALSE)

are_logical(..., .n = NULL, .all = FALSE)
```

```
are_raw(..., .n = NULL, .all = FALSE)

are_bytes(..., .n = NULL, .all = FALSE)

are_null(..., .all = FALSE)
```

Arguments

`...`	Objects to be tested.
`.n`	Expected lengths of the vectors.
`.all`	If `TRUE`, return boolean of whether all arguments returned `TRUE`.
`.finite`	Whether all values of the vectors are finite. The non-finite values are `NA`, `Inf`, `-Inf` and `NaN`. Setting this to something other than `NULL` can be expensive because the whole vector needs to be traversed and checked.

Details

The optional inputs of `.n` and `.finite` can be given inputs that map to the arguments in `...`. If a unnamed vector/list, the input must either be the same length as the number of arguments given to `...`, or length 1: which is then recycled to the number number of arguments given to `...`. Alternatively, a named vector/list can be given, where the values for matching named elements are passed to the type predicate, but unmatched names are passed NULL.

Value

Named logical, or unnamed boolean if `.all` is `TRUE`.

See Also

are-bare-type-predicates are-scalar-type-predicates

Examples

```
x <- 1
y <- list()
z <- mean

are_list(x, y, z, list(1))

# `.all` can be given to test if all inputs
# evaluate to TRUE
are_list(x, y, z, list(1), .all = TRUE)

# scalar inputs to `.n` and `.finite` are
# recycled to number of inputs
are_list(x, y, z, list(1), .n = 1)

# inputs to `.n` and `.finite` matching the
# number of inputs are applied sequentially
are_list(x, y, z, list(1), .n = c(1, 0, 1, 2))
```

```
# named inputs to `.n` and `.finite` are applied
# to the matching input names, with the other inputs
# being given NULL
are_list(x, y, z, list(1), .n = c(y = 1, "list(1)" = 2))
```

are_empty *Are objects empty vectors or NULL?*

Description

Are objects empty vectors or NULL?

Usage

```
are_empty(..., .all = FALSE)
```

Arguments

`...`	Objects to be tested.
`.all`	Whether to return if all arguments are TRUE.

Value

Named logical, or unnamed boolean if `.all` is `TRUE`.

See Also

is_empty

Examples

```
x <- 1
y <- NULL
z <- list()

are_empty(x, y, z, NULL)

are_empty(x, y, z, NULL, .all = TRUE)

are_empty(list(NULL))
```

`are_integerish` *Are vectors integer-like?*

Description

Wrappers around rlang type predicates that allow multiple objects to be passed. The following documentation is adapted from the rlang documentation:

These predicates check whether R considers a number vector to be integer-like, according to its own tolerance check (which is in fact delegated to the C library). This function is not adapted to data analysis, see the help for `base::is.integer()` for examples of how to check for whole numbers.

Things to consider when checking for integer-like doubles:

- This check can be expensive because the whole double vector has to be traversed and checked.
- Large double values may be integerish but may still not be coercible to integer. This is because integers in R only support values up to `2^31 - 1` while numbers stored as double can be much larger.

Usage

```
are_integerish(..., .n = NULL, .finite = NULL, .all = FALSE)

are_scalar_integerish(..., .n = NULL, .finite = NULL, .all = FALSE)

are_bare_integerish(..., .n = NULL, .finite = NULL, .all = FALSE)
```

Arguments

`...`	Objects to be tested.
`.n`	Expected lengths of the vectors.
`.finite`	Whether all values of the vectors are finite. The non-finite values are `NA`, `Inf`, `-Inf` and `NaN`. Setting this to something other than `NULL` can be expensive because the whole vector needs to be traversed and checked.
`.all`	If `TRUE`, return boolean of whether all arguments returned `TRUE`.

Details

The optional inputs of `.n` and `.finite` can be given inputs that map to the arguments in `...`. If a unnamed vector/list, the input must either be the same length as the number of arguments given to `...`, or length 1: which is then recycled to the number number of arguments given to `...`. Alternatively, a named vector/list can be given, where the values for matching named elements are passed to the type predicate, but unmatched names are passed NULL.

Value

Named logical, or unnamed boolean if `.all` is `TRUE`.

See Also

are_bare_numeric for testing whether an object is a base numeric type (a bare double or integer vector).

Examples

```
x <- 10L
y <- 10.0
z <- 10.000001

are_integerish(x, y, z, TRUE)

#' # `.all` can be given to test if all inputs
# evaluate to TRUE
are_integerish(x, y, z, TRUE, .all = TRUE)

# scalar inputs to `.n` and `.finite` are
# recycled to number of inputs
are_integerish(x, y, z, TRUE, .n = 2)

# inputs to `.n` and `.finite` matching the
# number of inputs are applied sequentially
are_integerish(x, y, z, TRUE, .n = c(1, 2, 1, 1))

# named inputs to `.n` and `.finite` are applied
# to the matching input names, with the other inputs
# being given NULL
are_integerish(x, y, z, TRUE, .n = c(y = 2, "TRUE" = 1))
```

`are_named` *Are objects named?*

Description

Wrappers around rlang predicates that allow multiple objects to be passed. The following documentation is adapted from the rlang documentation:

- `are_named()` is a scalar predicate that checks that objects in `...` have a `names` attribute and that none of the names are missing or empty (`NA` or `""`).
- `are_named2()` is like `are_named()` but always returns `TRUE` for empty vectors, even those that don't have a `names` attribute. In other words, it tests for the property that each element of a vector is named. `are_named2()` composes well with `names2()` whereas `are_named()` composes with `names()`.
- `have_names()` is a vectorised variant.

Usage

```
are_named(..., .all = FALSE)

are_named2(..., .all = FALSE)

have_names(..., .all = FALSE)
```

Arguments

`...`	Objects to be tested.
`.all`	Whether to return if all arguments are TRUE.

Value

`are_named()` and `are_named2()` return a named logical, or unnamed boolean if `.all` is `TRUE`. `have_names()` is vectorised and returns a list of logical vectors whhere each is as long as the input object. When `.all` is `TRUE` for `have_names()`, all logical vectors are collapsed and a boolean is returned.

See Also

are-bare-type-predicates rlang::is_named

Examples

```
# are_named() is a scalar predicate about the whole vector of names:
x <- c(a = 1, b = 2)
are_named(x, c(a = 1, 2))
are_named(x, c(a = 1, 2), .all = TRUE)

# Unlike are_named2(), are_named() returns `FALSE` for empty vectors
# that don't have a `names` attribute.
are_named(list(), vector())
are_named2(list(), vector())

# have_names() is vectorised
y <- c(a = 1, 2)
have_names(x, y, c(a = 1, 2, 3))
have_names(x, y, c(a = 1, 2, 3), .all = TRUE)

# Empty and missing names are treated as invalid:
invalid <- setNames(letters[1:5], letters[1:5])
names(invalid)[1] <- ""
names(invalid)[3] <- NA

are_named(invalid)
have_names(invalid)

# A data frame normally has valid, unique names
# but a matrix usually doesn't because the names
# are stored in a different attribute.
```

```
mat <- matrix(1:4, 2)
colnames(mat) <- c("a", "b")
are_named(mtcars, mat)
have_names(mtcars, mat)
```

---

are_true *Are objects TRUE or FALSE?*

Description

Test if any number of inputs are TRUE or FALSE. Inputs are passed to isTRUE or isFALSE.

Usage

```
are_true(..., .all = FALSE)

are_false(..., .all = FALSE)
```

Arguments

`...`	Objects to be tested.
`.all`	Whether to return if all arguments are TRUE.

Value

Named logical, or unnamed boolean if `.all` is `TRUE`.

See Also

isTRUE isFALSE

Examples

```
x <- TRUE
y <- 1
z <- mean

are_true(x, y, z, TRUE, 0)

are_true(x, y, z, TRUE, 0, .all = TRUE)

are_false(x, y, z, TRUE, 0)

are_false(x, y, z, TRUE, 0, .all = TRUE)
```

`cast_if_not` *Cast objects to a given type*

Description

The names of the `...` expressions, which should be variables within the `.env` envrionment, are attempted to be casted to the type specified in the expression: e.g., `name_of_object_to_cast = object_of_type_to_cast_to`. Expressions are evaluated in the environment specified and objects are assigned back into that same environment. Lossy casting can be undertaken by wrapping the expression in a call to lossy, e.g., `x = lossy(integer())`. The type conversion is from the vctrs package and thus sticks to the vctrs type conversion rules.

Usage

```
cast_if_not(..., .env = caller_env(), .error_call = caller_env())
```

Arguments

`...`	any number of named R expressions.
`.env`	the environment to use for the evaluation of the casting expressions and the assignment of the casted objects.
`.error_call`	the call environment to use for error messages (passed to abort).

Details

See abort_if_not for general validation, recycle_if_not for recycling, and enforce and schema for non data-masked and data-masked validations, recycling and casting.

Value

NULL, but objects named in `...` will be changed in the `.env` environment specified.

Examples

```
# NB: Some of these examples are expected to produce an error. To
#     prevent them from terminating a run with example() they are
#     piped into a call to try().

x <- 1L
cast_if_not(x = double())
class(x) # numeric

# By default, lossy casting is not allowed:
x <- c(1, 1.5)
cast_if_not(x = integer()) |> try()

# lossy casting can be enabled using `lossy()` call:
cast_if_not(x = lossy(integer()))
```

```
class(x) # integer

# Other objects can be used as the type to cast to, e.g.:
x <- 1L
y <- 2.3
cast_if_not(x = y)
class(x) # numeric

# Changed objects are available immediately:
x <- y <- 1L
cast_if_not(x = double(), y = x)
cat(class(x), class(y), sep = ", ") # numeric, numeric

myfunc <- function(x) {
  cast_if_not(x = double())
  class(x)
}
x <- 1L
myfunc(x) # x is cast to double within the function
class(x) # x is still an integer outside the function

# The `.env` argument determines the expression and assignment
# environment:
x <- 1L
e <- new.env()
e$x <- 1L
cast_if_not(x = 1.5, .env = e)
cat(
  "environment 'e'", class(e$x), "local environment", class(x),
  sep = ", "
) # environment 'e', numeric, local environment, integer

# Named objects (lhs) are checked to be in the `.env` environment,
# throwing an error if not found:
x <- 1L
e <- new.env()
cast_if_not(x = 1.5, .env = e) |> try()

# For expressions (rhs), the `.env` argument is preferentially
# chosen, but if not found then the normal R scoping rules
# apply:
x <- 1.5
e <- new.env()
e$z <- 1L
cast_if_not(z = x, .env = e)
class(e$z) # numeric

# The `.error_call` argument can be used to specify where the
# error occurs, by default this is the caller environment:
myfunc <- function(x) cast_if_not(x = character())
myfunc(FALSE) |> try()

# Injection can be used:
```

```
y <- 1L
x <- "y"
cast_if_not(!!x := double()) |> try()
class(y) # numeric

y <- 1L
x <- list(y = double())
cast_if_not(!!!x)
class(y) # numeric

# Objects are reverted to their original values if an error
# occur:
x <- y <- 1L
cast_if_not(x = double(), y = character()) |> try()
class(x) # integer
```

`enforce` *Ensure the truth of R expressions and cast/recycle objects.*

Description

If any of the expressions in `...` are not all `TRUE`, abort is called for the first which was not (all) `TRUE`. Alternatively, rlang formulas can be used to pass multiple objects to validation formulas/functions, and/or attempt safe type casting and size recycling using the cast, recycle and coerce functions. The rhs of formulas can be given in a list to pass multiple functions/formulas/calls. Expressions are evaluated in the environment specified and objects are assigned back into that environment. Type casting and recycling are undertaken using the vctrs package and thus apply vctrs type and size rules.

Usage

```
enforce(..., .env = caller_env(), .error_call = caller_env())
```

Arguments

`...`	any number of R expressions or formulas to be evaluated. Expressions must evaluate to logical whilst formulas can use c on the lhs and either functions or formulas that evaluate to logical, or one of the type/size functions: cast, recycle or coerce on the rhs. The rhs of a formula can also be a list of multiple functions/formulas/calls. If an expression is named, or if the list element on the rhs of a formula is named, the name is passed to format_inline and is used in the error message.
`.env`	the environment to use for the evaluation of the expressions and the assignment of the objects.
`.error_call`	the call environment to use for error messages (passed to abort).

Details

See abort_if_not for only validations and schema for a data-masked version of this function.

Value

NULL, but objects casted/recycled in ... will be changed in the .env environment specified.

Examples

```
# NB: Some of these examples are expected to produce an error. To
#     prevent them from terminating a run with example() they are
#     piped into a call to try().

x <- 1L
y <- "hi"
z <- \(x) x > 1
enforce(x == 1, is.character(y), is.function(z)) # all TRUE

enforce(x == 2) |> try()

# A custom error message can be given for each expression by
# naming it:
enforce(
  "{.var y} must be {.cls numeric}, check input" = is.numeric(y)
) |> try()

# Formulas can be used to take pass multiple objects
# on the lhs, with functions/additional formulas required on
# the rhs:
enforce(
  "multiple objects using: {.fn c}" = c(x, y) ~ is.integer
) |> try()

# Formulas can also be used with `cast()`, `recycle()`, and
# `coerce()` on the rhs to safely cast or recycle objects:
enforce(x ~ cast(double()))
class(x) # x is now numeric
enforce(x ~ recycle(5))
length(x) # x is now length 5
enforce(y ~ coerce(type = factor(), size = 5))
print(y) # y is now factor and length 5

# Multiple calls can be used with formulas by wrapping them
# in `list()`, with the names of list elements being
# preferentially chosen for error messaging and the error
# message also showing which formula/function/call caused the
# error:
enforce(
  "generic message" = c(x, y, z) ~ list(
    Negate(is.null),
    "{.var specific} message" = Negate(is.function)
  )
) |> try()

# Changed elements are available immediately:
x <- y <- 1L
```

```
enforce(x ~ cast(double()), y ~ cast(x))
cat(class(x), class(y)) # both now numeric

# The `.error_call` argument can be used to specify where the
# error occurs, by default this is the caller environment:
myfunc <- function(...) enforce(...)
myfunc(x > 4) |> try()

# rlang injection can be used:
msg <- "{.var injection} msg"
cols <- quote(c(x, y))
enforce(!!msg := !!cols ~ is.integer) |> try()

# Objects are reverted to their original values if an error
# occur:
x <- y <- 1L
enforce(
  x ~ cast(double()), y ~ recycle(5), y ~ is.function
) |> try() # errors
class(x) # integer
length(y) # 1
```

recycle_if_not *Recycle objects to a given size*

Description

The names of the `...` expressions, which should be variables within the `.env` envrionment, are attempted to be recycled to the size specified in the expression: e.g., `name_of_object_to_recycle = size_to_recycle_to`. Expressions are evaluated in the environment specified and objects are assigned back into that same environment. The object recycling is from the vctrs package and thus stick to the vctrs recycling rules.

Usage

```
recycle_if_not(..., .env = caller_env(), .error_call = caller_env())
```

Arguments

`...`	any number of named R expressions.
`.env`	the environment to use for the evaluation of the recycling expressions and the assignment of the recycled objects.
`.error_call`	the call environment to use for error messages (passed to abort).

Details

See abort_if_not for general validation, recycle_if_not for recycling, and enforce and schema for non data-masked and data-masked validations, recycling and casting.

Value

NULL, but objects named in `...` will be changed in the `.env` environment specified.

Examples

```
# NB: Some of these examples are expected to produce an error. To
#     prevent them from terminating a run with example() they are
#     piped into a call to try().

x <- 1
recycle_if_not(x = 5)
length(x) # 5

# recycle_if_not() follows `vctrs` recycling rules:
x <- c(1, 1)
recycle_if_not(x = 6) |> try()

# Beware when using other objects as the size argument, e.g.:
x <- 1L
y <- c(1, 1, 1)
recycle_if_not(x = y) |> try()

# When using other objects, call vctrs::vec_size() on them first:
recycle_if_not(x = vctrs::vec_size(y))
length(x) # 3

# Changed objects are available immediately:
x <- y <- 1
recycle_if_not(x = 3, y = vctrs::vec_size(x))
cat(length(x), length(y), sep = ", ") # 3, 3

myfunc <- function(x) {
  recycle_if_not(x = 3)
  length(x)
}
x <- 1L
myfunc(x) # x is recycled to length 3 within the function
length(x) # x is still scalar outside the function

# The `.env` argument determines the expression and assignment
# environment:
x <- 1
e <- new.env()
e$x <- 1
recycle_if_not(x = 3, .env = e)
cat(
  "environment 'e'", length(e$x), "local environment", length(x),
  sep = ", "
) # environment 'e', 3, local environment, 1

# Named objects (lhs) are checked to be in the `.env` environment,
# throwing an error if not found:
```

```
x <- 1
e <- new.env()
recycle_if_not(x = 3, .env = e) |> try()

# For expressions (rhs), the `.env` argument is preferentially
# chosen, but if not found then the normal R scoping rules
# apply:
x <- 3
e <- new.env()
e$z <- 1
recycle_if_not(z = x, .env = e)
length(e$z) # 3

# The `.error_call` argument can be used to specify where the
# error occurs, by default this is the caller environment:
myfunc <- function(x) recycle_if_not(x = -5)
myfunc(1) |> try()

#' # Injection can be used:
y <- 1L
x <- "y"
recycle_if_not(!!x := 5) |> try()
length(y) # 5

y <- 1L
x <- list(y = 5)
recycle_if_not(!!!x)
length(y) # 5

# Objects are reverted to their original values if an error
# occur:
x <- y <- 1L
recycle_if_not(x = 5, y = -5) |> try()
length(x) # 1
```

`schema` *Ensure the truth of data-masked R expressions and cast/recycle named elements.*

Description

If any of the expressions in `...`, evaluated within the data mask `'data'` (see data masking), are not all `TRUE`, abort is called for the first which was not (all) `TRUE`. Alternatively, rlang formulas can be used to take advantage of tidyselect features and pass multiple named elements in `data` to validation formulas/functions, and/or attempt safe type casting and size recycling using the cast, recycle and coerce functions. The rhs of formulas can be given in a list to pass multiple functions/formulas/calls. The `.names` and `.size` arguments can also be used to check for given names and size of the data.frame/list itself. Type casting, size checking, and recycling are undertaken using the vctrs package and thus apply vctrs type and size rules.

Usage

```
schema(data, ...)

## S3 method for class 'list'
schema(
  data,
  ...,
  .names = NULL,
  .size = NULL,
  .error_call = caller_env(),
  .darg = caller_arg(data)
)

## S3 method for class 'data.frame'
schema(
  data,
  ...,
  .names = NULL,
  .size = NULL,
  .error_call = caller_env(),
  .darg = caller_arg(data)
)

enforce_schema(data, ...)

## S3 method for class 'with_schema'
enforce_schema(data, ..., .error_call = caller_env(), .darg = caller_arg(data))

add_to_schema(data, ...)

## S3 method for class 'with_schema'
add_to_schema(
  data,
  ...,
  .names = NULL,
  .size = NULL,
  .error_call = caller_env(),
  .darg = caller_arg(data)
)
```

Arguments

`data`	a data.frame or list to use as the data mask.
`...`	any number of R expressions or formulas to be evaluated using `data` as a data mask. Formulas can use tidyselect syntax on the lhs and either functions or formulas that evaluate to logical, or one of the type/size functions: cast, recycle and coerce on the rhs. The rhs of a formula can also be a list of multiple functions/formulas/calls. If an expression is named, or if the list element on the rhs

	of a formula is named, the name is passed to format_inline and is used in the error message.
`.names`	character vector of names which must be present in the `data` data.frame/list.
`.size`	positive scalar integerish value for the size that the `data` data.frame/list must be.
`.error_call`	the call environment to use for error messages (passed to abort).
`.darg`	the argument name of `data` to use in error messages.

Details

See abort_if_not for a non-data-masked validation tool and enforce for a non-data-masked version of this function.

Value

`data` is returned with attached class `with_schema` and attribute `schema` containing the schema call to be enforced later.

Examples

```
# NB: Some of these examples are expected to produce an error. To
#     prevent them from terminating a run with example() they are
#     piped into a call to try().

li <- list(x = 1L, y = "hi", z = \(x) x > 1)
li <- li |>
  schema(x == 1, is.character(y), is.function(z)) # all TRUE

# The schema call is attached to the returned object and
# can be re-evaluated using enforce_schema():
li <- enforce_schema(li) # no error
li2 <- li
li2$x <- 2L
enforce_schema(li2) |> try()

# Calling `schema()` again overwrites any existing schema.
# Alternatively use `add_to_schema()` to add arguments to
# an existing schema (.size overwrites, other args append):
li <- li |>
  add_to_schema(is.numeric(x), .names = c("x", "y"), .size = 3)

# A custom error message can be given for each expression by
# naming it:
schema(li,
  "{.var y} must be {.cls numeric}, check input" = is.numeric(y)
) |> try()

# Formulas can be used to take advantage of tidyselect features
# on the lhs, with functions/additional formulas required on
# the rhs:
schema(li,
  "multiple columns: {.pkg tidyselect}" = c(x, y) ~ is.integer
) |> try()

# Formulas can also be used with `cast()`, `recycle()`, and
# `coerce()` on the rhs to safely cast or recycle named
# elements:
class(schema(li, x ~ cast(double()))$x) # x is now numeric
length(schema(li, x ~ recycle(5))$x) # x is now length 5
schema(
  li,
  y ~ coerce(type = factor(), size = 5)
)$y # y is now factor and length 5

# Multiple calls can be used with formulas by wrapping them
# in `list()`, with the names of list elements being
# preferentially chosen for error messaging and the error
# message also showing which formula/function/call caused the
# error:
schema(
  li,
  "generic message" = c(x, y, z) ~ list(
    Negate(is.null),
    "{.var specific} message" = Negate(is.function)
  )
) |> try()

# Changed elements are available immediately:
df <- data.frame(x = 1L, y = 1L)
lapply(schema(df, x ~ cast(double()), y ~ cast(x)), class)
# both now numeric

# `.names` and `.size` arguments can be used to check that given
# names are present and that the data has the desired size:
schema(li, .names = c("a", "x", "y", "b")) |> try()
schema(li, .size = 5) |> try()

# The `.error_call` argument can be used to specify where the
# error occurs, by default this is the caller environment:
myfunc <- function(x, ...) schema(x, ...)
myfunc(li, x > 4) |> try()

# rlang pronouns and injection can be used, but care must be
# taken when using `.env` and `enforce_schema()` as the
# caller environment may have changed:
msg <- "{.var injection} msg"
cols <- quote(c(x, y))
schema(li, !!msg := !!cols ~ is.integer) |> try()

x <- 1L
li <- schema(li, x == .env$x) # no error

x <- 2
enforce_schema(li) |>
  try() # error as the environmental variable has changed
```

Index