Package 'valaddin'

February 10, 2019

Title Functional Input Validation

Version 1.0.0

Description A set of basic tools to transform functions into functions with input validation checks, in a manner suitable for both programmatic and interactive use.

License MIT + file LICENSE

Encoding UTF-8

LazyData true

Depends R (>= 3.1.0)

Imports lazyeval (>= 0.2.1)

Suggests magrittr, testthat, stringr, knitr, rmarkdown

VignetteBuilder knitr

URL https://github.com/egnha/valaddin

BugReports https://github.com/egnha/valaddin/issues

Collate 'utils.R' 'pipe.R' 'rawrd.R' 'formulas.R' 'checklist.R' 'components.R' 'call.R' 'firmly.R' 'validate.R' 'scope.R' 'checkers.R' 'valaddin.R'

RoxygenNote 6.1.1

NeedsCompilation no

Author Eugene Ha [aut, cre]

Maintainer Eugene Ha <eha@posteo.de>

Repository CRAN

Date/Publication 2019-02-10 13:30:03 UTC

R topics documented:

`checklist` *Is a formula a check formula?*

Description

`is_check_formula(x)` checks whether `x` is a check formula, while `is_checklist(x)` checks whether `x` is a *checklist*, i.e., a list of check formulae. (Neither function verifies logical consistency of the implied checks.)

Usage

```
is_check_formula(x)

is_checklist(x)
```

Arguments

`x`	Object to test.

Value

`is_check_formula`, resp. `is_checklist`, returns `TRUE` or `FALSE`, according to whether `x` is or is not a check formula, resp. checklist.

See Also

firmly (on the specification and use of check formulae)

Examples

```
is_check_formula(list(~x, ~y) ~ is.numeric)  # [1] TRUE
is_check_formula("Not positive" ~ {. > 0})   # [1] TRUE

is_checklist(list(list(~x, ~y) ~ is.numeric, "Not positive" ~ {. > 0}))
# [1] TRUE

# Invalid checklists
is_checklist("Not positive" ~ {. > 0})           # [1] FALSE (not a list)
is_checklist(list(is.numeric ~ list(~ x)))       # [1] FALSE (backwards)
is_checklist(list(list(log ~ x) ~ is.character)) # [1] FALSE (invalid check item)
```

`components` *Decompose a firmly applied function*

Description

Decompose a firmly applied function (i.e., a function created by `firmly`):

- `firm_core` extracts the underlying "core" function—the function that is called when all arguments are valid.
- `firm_checks` extracts the checks.
- `firm_error` extracts the subclass of the error condition that is signaled when an input validation error occurs.
- `firm_args` extracts the names of arguments whose presence is to be checked, i.e., those specified by the `.warn_missing` switch of `firmly`.

Usage

```
firm_core(x)

firm_checks(x)

firm_error(x)

firm_args(x)
```

Arguments

`x`	Object to decompose.

Value

If `x` is a firmly applied function:

- `firm_core` returns a function.
- `firm_checks` returns a data frame with components `expr` (language), `env` (environment), `string` (character), `msg` (character).
- `firm_error` returns a character vector.
- `firm_args` returns a character vector.

In the absence of the component to be extracted, these functions return `NULL`.

See Also

`firmly`

Examples

```
f <- function(x, y, ...) NULL
f_fm <- firmly(f, ~is.numeric, list(~x, ~y - x) ~ {. > 0})

identical(firm_core(f_fm), f)                  # [1] TRUE
firm_checks(f_fm)                              # 4 x 4 data frame
firm_error(f_fm)                               # [1] "simpleError"
firm_args(f_fm)                                # NULL
firm_args(firmly(f_fm, .warn_missing = "y"))   # [1] "y"
```

`firmly` *Apply a function firmly*

Description

`firmly` transforms a function into a function with input validation checks. `loosely` undoes the application of `firmly`, by returning the original function (without checks). `is_firm` is a predicate function that checks whether an object is a firmly applied function, i.e., a function created by `firmly`.

Use `%checkin%` to apply `firmly` as an operator. Since this allows you to keep checks and arguments adjacent, it is the preferred way to use `firmly` in scripts and packages.

Usage

```
firmly(.f, ..., .checklist = list(), .warn_missing = character(),
       .error_class = character())

.checks %checkin% .f

loosely(.f, .keep_check = FALSE, .keep_warning = FALSE)

is_firm(x)
```

Arguments

`.f`	Interpreted function, i.e., closure.
`...`	Input-validation check formula(e).
`.checklist`	List of check formulae. (These are combined with check formulae provided via `...`.)
`.warn_missing`	Arguments of `.f` whose absence should raise a warning (character).
`.error_class`	Subclass of the error condition to be raised when an input validation error occurs (character).
`.checks`	List of check formulae, optionally containing character vectors named `.warn_missing`, `.error_class`, corresponding to the similarly named arguments.

`.keep_check, .keep_warning`
: Should existing checks, resp. missing-argument warnings, be kept?

`x`
: Object to test.

Check Formulae

An **input validation check** is specified by a **check formula**, a special formula of the form

```
<scope> ~ <predicate>
```

where the right-hand side expresses *what* to check, and the left-hand side expresses *where* to check it.

The right-hand side `<predicate>` is a **predicate** function, i.e, a one-argument function that returns either `TRUE` or `FALSE`. It is the condition to check/enforce. The left-hand side `<scope>` is an expression specifying what the condition is to be applied to: whether the condition is to be applied to all (non-`...`) arguments of `.f` (the case of "global scope"), or whether the condition is to be selectively applied to certain expressions of the arguments (the case of "local scope").

According to **scope**, there are two classes of check formulae:

- **Check formulae of global scope**

```
<string> ~ <predicate>

~<predicate>

\item \strong{Check formulae of local scope}
  \preformatted{list(<check_item>, <check_item>, ...) ~ <predicate>}
```

Check Formulae of Global Scope: A **global check formula** is a succinct way of asserting that the function `<predicate>` returns `TRUE` when called on each (non-`...`) argument of `.f`. Each argument for which `<predicate>` *fails*—returns `FALSE` or is itself not evaluable—produces an error message, which is auto-generated unless a custom error message is supplied by specifying the string `<string>`.

```
\subsection{Example}{
  The condition that all (non-\code{...}) arguments of a function must
  be numerical can be enforced by the check formula
  \preformatted{~is.numeric}
  or
  \preformatted{"Not numeric" ~ is.numeric}
  if the custom error message \code{"Not numeric"} is to be used (in lieu
  of an auto-generated error message).
}
```

Check Formulae of Local Scope: A **local check formula** imposes argument-specific conditions. Each **check item** `<check_item>` is a formula of the form `~ <expression>` (one-sided) or `<string> ~ <expression>`; it imposes the condition that the function `<predicate>` is `TRUE` for the expression `<expression>`. As for global check formulae, each check item for which `<predicate>` fails produces an error message, which is auto-generated unless a custom error message is supplied by a string as part of the left-hand side of the check item (formula).

```
\subsection{Example}{
  The condition that \code{x} and \code{y} must differ for the function
  \code{function(x, y) {1 / (x - y)}} can be enforced by the local
  check formula
  \preformatted{list(~x - y) ~ function(.) abs(.) > 0}
  or
  \preformatted{list("x, y must differ" ~ x - y) ~ function(.) abs(.) > 0}
  if the custom error message \code{"x, y must differ"} is to be used (in
  lieu of an auto-generated error message).
}
```

Anonymous Predicate Functions: Following the **magrittr** package, an anonymous (predicate) function of a single argument `.` can be concisely expressed by enclosing the body of such a function within curly braces `{ }`.

```
\subsection{Example}{
  The (onsided, global) check formula
  \preformatted{~{. > 0}}
  is equivalent to the check formula \code{~function(.) {. > 0}}
}
```

Value

`firmly`**:** `firmly` does nothing when there is nothing to do: `.f` is returned, unaltered, when both `.checklist` and `.warn_missing` are empty, or when `.f` has no named argument and `.warn_missing` is empty.

```
Otherwise, \code{firmly} again returns a function that behaves
\emph{identically} to \code{.f}, but also performs input validation:
before a call to \code{.f} is attempted, its inputs are checked, and if
any check fails, an error halts further execution with a message
tabulating every failing check. (If all checks pass, the call to
\code{.f} respects lazy evaluation, as usual.)

\subsection{Subclass of the input-validation error object}{
  The subclass of the error object is \code{.error_class}, unless
  \code{.error_class} is \code{character()}. In the latter case, the
  subclass of the error object is that of the existing error object, if
  \code{.f} is itself a firmly applied function, or it is
  \code{"simpleError"}, otherwise.
}

\subsection{Formal Arguments and Attributes}{
  \code{firmly} preserves the attributes and formal arguments of
  \code{.f} (except that the \code{"class"} attribute gains the component
  \code{"firm_closure"}, unless it already contains it).
}
```

`%checkin%`**:** `%checkin%` applies the check formula(e) in the list `.checks` to `.f`, using `firmly`. The `.warn_missing` and `.error_class` arguments of `firmly` may be specified as named components of `.checks`.

`loosely`: `loosely` returns `.f`, unaltered, when `.f` is not a firmly applied function, or both `.keep_check` and `.keep_warning` are `TRUE`.

Otherwise, \code{loosely} returns the underlying (original) function, stripped of any input validation checks imposed by \code{firmly}, unless one of the flags \code{.keep_check}, \code{.keep_warning} is switched on: if \code{.keep_check}, resp. \code{.keep_warning}, is \code{TRUE}, \code{loosely} retains any existing checks, resp. missing-argument warnings, of \code{.f}.

`is_firm`: `is_firm` returns `TRUE` if x is a firmly applied function (i.e., has class `"firm_closure"`), and `FALSE`, otherwise.

See Also

`firmly` is enhanced by a number of helper functions:

- To verify that a check formula is syntactically correct, use the predicates `is_check_formula`, `is_checklist`.
- To make custom check-formula generators, use `localize`.
- Pre-made check-formula generators are provided to facilitate argument checks for types, scalar objects, and other common data structures and input assumptions. These functions are prefixed by `vld_`, for convenient browsing and look-up in editors and IDE's that support name completion.
- To access the components of a firmly applied function, use `firm_core`, `firm_checks`, `firm_error`, `firm_args`, (or simply `print` the function to display its components).

Examples

```
## Not run:

dlog <- function(x, h) (log(x + h) - log(x)) / h

# Require all arguments to be numeric (auto-generated error message)
dlog_fm <- firmly(dlog, ~is.numeric)
dlog_fm(1, .1)    # [1] 0.9531018
dlog_fm("1", .1)  # Error: "FALSE: is.numeric(x)"

# Require all arguments to be numeric (custom error message)
dlog_fm <- firmly(dlog, "Not numeric" ~ is.numeric)
dlog_fm("1", .1)  # Error: "Not numeric: `x`"

# Alternatively, "globalize" a localized checker (see ?localize, ?globalize)
dlog_fm <- firmly(dlog, globalize(vld_numeric))
dlog_fm("1", .1)  # Error: "Not double/integer: `x`"

# Predicate functions can be specified anonymously or by name
dlog_fm <- firmly(dlog, list(~x, ~x + h, ~abs(h)) ~ function(x) x > 0)
dlog_fm <- firmly(dlog, list(~x, ~x + h, ~abs(h)) ~ {. > 0})
is_positive <- function(x) x > 0
dlog_fm <- firmly(dlog, list(~x, ~x + h, ~abs(h)) ~ is_positive)
```

```
dlog_fm(1, 0)  # Error: "FALSE: is_positive(abs(h))"

# Describe checks individually using custom error messages
dlog_fm <-
  firmly(dlog,
         list("x not positive" ~ x, ~x + h, "Division by 0 (=h)" ~ abs(h)) ~
           is_positive)
dlog_fm(-1, 0)
# Errors: "x not positive", "FALSE: is_positive(x + h)", "Division by 0 (=h)"

# Specify checks more succinctly by using a (localized) custom checker
req_positive <- localize("Not positive" ~ is_positive)
dlog_fm <- firmly(dlog, req_positive(~x, ~x + h, ~abs(h)))
dlog_fm(1, 0)  # Error: "Not positive: abs(h)"

# Combine multiple checks
dlog_fm <- firmly(dlog,
                  "Not numeric" ~ is.numeric,
                  list(~x, ~x + h, "Division by 0" ~ abs(h)) ~ {. > 0})
dlog_fm("1", 0)  # Errors: "Not numeric: `x`", check-eval error, "Division by 0"

# Any check can be expressed using isTRUE
err_msg <- "x, h differ in length"
dlog_fm <- firmly(dlog, list(err_msg ~ length(x) - length(h)) ~ {. == 0L})
dlog_fm(1:2, 0:2)  # Error: "x, h differ in length"
dlog_fm <- firmly(dlog, list(err_msg ~ length(x) == length(h)) ~ isTRUE)
dlog_fm(1:2, 0:2)  # Error: "x, h differ in length"

# More succinctly, use vld_true
dlog_fm <- firmly(dlog, vld_true(~length(x) == length(h), ~all(abs(h) > 0)))
dlog_fm(1:2, 0:2)
# Errors: "Not TRUE: length(x) == length(h)", "Not TRUE: all(abs(h) > 0)"

dlog_fm(1:2, 1:2)  # [1] 0.6931472 0.3465736

# loosely recovers the underlying function
identical(loosely(dlog_fm), dlog)  # [1] TRUE

# Use .warn_missing when you want to ensure an argument is explicitly given
# (see vignette("valaddin") for an elaboration of this particular example)
as_POSIXct <- firmly(as.POSIXct, .warn_missing = "tz")
Sys.setenv(TZ = "EST")
as_POSIXct("2017-01-01 03:14:16")  # [1] "2017-01-01 03:14:16 EST"
                                   # Warning: "Argument(s) expected ... `tz`"
as_POSIXct("2017-01-01 03:14:16", tz = "UTC")  # [1] "2017-01-01 03:14:16 UTC"
loosely(as_POSIXct)("2017-01-01 03:14:16")     # [1] "2017-01-01 03:14:16 EST"

# Use firmly to constrain undesirable behavior, e.g., long-running computations
fib <- function(n) {
  if (n <= 1L) return(1L)
  Recall(n - 1) + Recall(n - 2)
}
fib <- firmly(fib, list("`n` capped at 30" ~ ceiling(n)) ~ {. <= 30L})
```

```
fib(21)  # [1] 17711 (NB: Validation done only once, not for every recursive call)
fib(31)  # Error: `n` capped at 30

# Apply fib unrestricted
loosely(fib)(31)  # [1] 2178309 (may take several seconds to finish)

# firmly won't force an argument that's not involved in checks
g <- firmly(function(x, y) "Pass", list(~x) ~ is.character)
g(c("a", "b"), stop("Not signaled"))  # [1] "Pass"

# In scripts and packages, it is recommended to use the operator %checkin%
vec_add <- list(
  ~is.numeric,
  list(~length(x) == length(y)) ~ isTRUE,
  .error_class = "inputError"
) %checkin%
  function(x, y) {
    x + y
  }

# Or call firmly with .f explicitly assigned to the function
vec_add2 <- firmly(
  ~is.numeric,
  list(~length(x) == length(y)) ~ isTRUE,
  .f = function(x, y) {
    x + y
  },
  .error_class = "inputError"
)

all.equal(vec_add, vec_add2)  # [1] TRUE

## End(Not run)
```

`input-validators` *Generate input-validation checks*

Description

`localize` derives a function that *generates* check formulae of local scope from a check formula of global scope. `globalize` takes such a check-formula generator and returns the underlying global check formula. These operations are mutually invertible.

Usage

```
localize(chk)

globalize(chkr)
```

Arguments

chk	Check formula of global scope *with* custom error message, i.e., a formula of the form <string> ~ <predicate>.
chkr	Function of class "check_maker", i.e., a function created by localize.

Value

localize returns a function of class "check_maker" and call signature function(...):

- The ... are **check items** (see *Check Formulae of Local Scope* in the documentation page firmly).
- The return value is the check formula of local scope whose scope is comprised of these check items, and whose predicate function is that of chk (i.e., the right-hand side of chk). Unless a check item has its own error message, the error message is derived from that of chk (i.e., the left-hand side of chk).

globalize returns the global-scope check formula from which the function chkr is derived.

See Also

The notion of "scope" is explained in the *Check Formulae* section of firmly.

Ready-made checkers for types, scalar objects, and miscellaneous predicates are provided as a convenience, and as a model for creating families of check makers.

Examples

```
chk_pos_gbl <- "Not positive" ~ {. > 0}
chk_pos_lcl <- localize(chk_pos_gbl)
chk_pos_lcl(~x, "y not greater than x" ~ x - y)
# list("Not positive: x" ~ x, "y not greater than x" ~ x - y) ~ {. > 0}

# localize and globalize are mutual inverses
identical(globalize(localize(chk_pos_gbl)), chk_pos_gbl)  # [1] TRUE
all.equal(localize(globalize(chk_pos_lcl)), chk_pos_lcl)  # [1] TRUE

## Not run:

pass <- function(x, y) "Pass"

# Impose local positivity checks
f <- firmly(pass, chk_pos_lcl(~x, "y not greater than x" ~ x - y))
f(2, 1)  # [1] "Pass"
f(2, 2)  # Error: "y not greater than x"
f(0, 1)  # Errors: "Not positive: x", "y not greater than x"

# Or just check positivity of x
g <- firmly(pass, chk_pos_lcl(~x))
g(1, 0)  # [1] "Pass"
g(0, 0)  # Error: "Not positive: x"
```

```
# In contrast, chk_pos_gbl checks positivity for all arguments
h <- firmly(pass, chk_pos_gbl)
h(2, 2)  # [1] "Pass"
h(1, 0)  # Error: "Not positive: `y`"
h(0, 0)  # Errors: "Not positive: `x`", "Not positive: `y`"

# Alternatively, globalize the localized checker
h2 <- firmly(pass, globalize(chk_pos_lcl))
all.equal(h, h2)  # [1] TRUE

# Use localize to make parameterized checkers
chk_lte <- function(n, ...) {
  err_msg <- paste("Not <=", as.character(n))
  localize(err_msg ~ {. <= n})(...)
}
fib <- function(n) {
  if (n <= 1L) return(1L)
  Recall(n - 1) + Recall(n - 2)
}
capped_fib <- firmly(fib, chk_lte(30, ~ ceiling(n)))
capped_fib(19)  # [1] 6765
capped_fib(31)  # Error: "Not <= 30: ceiling(n)"

## End(Not run)
```

`misc-checkers` *Miscellaneous checkers*

Description

These functions make check formulae of local scope based on the correspondingly named **base** R predicates `is.*` (e.g., `vld_data_frame` corresponds to the predicate `is.data.frame`), with the following exceptions:

- `vld_empty` is based on the predicate `length(.) == 0`
- `vld_formula` is based on the predicate `typeof(.) == "language" && inherits(., "formula")`
- `vld_closure` is based on the predicate `typeof(.) == "closure"`
- `vld_true` and `vld_false` are based on the predicates `identical(., TRUE)` and `identical(., FALSE)`, resp.

The checkers `vld_true` and `vld_false` are all-purpose checkers to specify *arbitrary* input validation checks.

Usage

```
vld_all(...)

vld_any(...)
```

```
vld_array(...)

vld_atomic(...)

vld_call(...)

vld_closure(...)

vld_data_frame(...)

vld_empty(...)

vld_environment(...)

vld_expression(...)

vld_factor(...)

vld_false(...)

vld_formula(...)

vld_function(...)

vld_language(...)

vld_list(...)

vld_matrix(...)

vld_na(...)

vld_name(...)

vld_nan(...)

vld_null(...)

vld_numeric(...)

vld_ordered(...)

vld_pairlist(...)

vld_primitive(...)

vld_recursive(...)
```

```
vld_symbol(...)

vld_table(...)

vld_true(...)

vld_unsorted(...)

vld_vector(...)
```

Arguments

...	Check items, i.e., formulae that are one-sided or have a string as left-hand side (see *Check Formulae of Local Scope* in the documentation page firmly). These are the expressions to check.

Details

Each function `vld_*` is a function of class `"check_maker"`, generated by `localize`.

Value

Check formula of local scope.

See Also

Corresponding predicates: `all`, `any`, `is.array`, `is.atomic`, `is.call`, `is.data.frame`, `is.environment`, `is.expression`, `is.factor`, `is.function`, `is.language`, `is.list`, `is.matrix`, `is.na`, `is.name`, `is.nan`, `is.null`, `is.numeric`, `is.ordered`, `is.pairlist`, `is.primitive`, `is.recursive`, `is.symbol`, `is.table`, `is.unsorted`, `is.vector`

`globalize` recovers the underlying check formula of global scope.

The notions of "scope" and "check item" are explained in the *Check Formulae* section of firmly.

Other checkers: type-checkers, scalar-checkers

Examples

```
## Not run:

f <- function(x, y) "Pass"

# Impose the condition that x is a formula
g <- firmly(f, vld_formula(~x))
g(z ~ a + b, 0)  # [1] "Pass"
g(0, 0)          # Error: "Not formula: x"

# Impose the condition that x and y are disjoint (assuming they are vectors)
h <- firmly(f, vld_empty(~intersect(x, y)))
h(letters[1:3], letters[4:5])  # [1] "Pass"
h(letters[1:3], letters[3:5])  # Error: "Not empty: intersect(x, y)"
```

```
# Use a custom error message
h <- firmly(f, vld_empty("x, y must be disjoint" ~ intersect(x, y)))
h(letters[1:3], letters[3:5])  # Error: "x, y must be disjoint"

# vld_true can be used to implement any kind of input validation
ifelse_f <- firmly(ifelse, vld_true(~typeof(yes) == typeof(no)))
(w <- {set.seed(1); rnorm(5)})
# [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
ifelse_f(w > 0, 0, "1")  # Error: "Not TRUE: typeof(yes) == typeof(no)"
ifelse_f(w > 0, 0, 1)    # [1] 1 0 1 0 0

## End(Not run)
```

`scalar-checkers` *Scalar checkers*

Description

These functions make check formulae of local scope based on the correspondingly named scalar type predicate from **base** R. For example, `vld_scalar_logical` creates check formulae (of local scope) for the predicate `is.logical(.) && length(.) == 1`. The function `vld_singleton` is based on the predicate `length(.) == 1`.

The functions `vld_boolean`, `vld_number`, `vld_string` are aliases for `vld_scalar_logical`, `vld_scalar_numeric`, `vld_scalar_character`, resp. (with appropriately modified error messages).

Usage

```
vld_boolean(...)

vld_number(...)

vld_scalar_atomic(...)

vld_scalar_character(...)

vld_scalar_complex(...)

vld_scalar_double(...)

vld_scalar_integer(...)

vld_scalar_list(...)

vld_scalar_logical(...)

vld_scalar_numeric(...)
```

```
vld_scalar_raw(...)

vld_scalar_vector(...)

vld_singleton(...)

vld_string(...)
```

Arguments

`...`	Check items, i.e., formulae that are one-sided or have a string as left-hand side (see *Check Formulae of Local Scope* in the documentation page firmly). These are the expressions to check.

Details

Each function `vld_*` is a function of class `"check_maker"`, generated by `localize`.

Value

Check formula of local scope.

See Also

Corresponding predicates: `is.atomic`, `is.character`, `is.complex`, `is.double`, `is.integer`, `is.list`, `is.logical`, `is.numeric`, `is.raw`, `is.vector`

`globalize` recovers the underlying check formula of global scope.

The notions of "scope" and "check item" are explained in the *Check Formulae* section of firmly.

Other checkers: type-checkers, misc-checkers

Examples

```
## Not run:

f <- function(x, y) "Pass"

# Impose a check on x: ensure it's boolean (i.e., a scalar logical vector)
f_firm <- firmly(f, vld_boolean(~x))
f_firm(TRUE, 0)           # [1] "Pass"
f_firm(c(TRUE, TRUE), 0)  # Error: "Not boolean: x"

# Use a custom error message
f_firm <- firmly(f, vld_boolean("x is not TRUE/FALSE/NA" ~ x))
f_firm(c(TRUE, TRUE), 0)  # Error: "x is not TRUE/FALSE/NA"

# To impose the same check on all arguments, apply globalize
f_firmer <- firmly(f, globalize(vld_boolean))
f_firmer(TRUE, FALSE)    # [1] "Pass"
f_firmer(TRUE, 0)        # Error: "Not boolean: `y`"
```

```
f_firmer(logical(0), 0)  # Errors: "Not boolean: `x`", "Not boolean: `y`"

## End(Not run)
```

---

type-checkers *Type checkers*

Description

These functions make check formulae of local scope based on the correspondingly named (atomic) type predicate from **base** R.

Usage

```
vld_character(...)

vld_complex(...)

vld_double(...)

vld_integer(...)

vld_logical(...)

vld_raw(...)
```

Arguments

`...`	Check items, i.e., formulae that are one-sided or have a string as left-hand side (see *Check Formulae of Local Scope* in the documentation page firmly). These are the expressions to check.

Details

Each function `vld_*` is a function of class `"check_maker"`, generated by `localize`.

Value

Check formula of local scope.

See Also

Corresponding predicates: `is.character`, `is.complex`, `is.double`, `is.integer`, `is.logical`, `is.raw`

`globalize` recovers the underlying check formula of global scope.

The notions of "scope" and "check item" are explained in the *Check Formulae* section of firmly.

Other checkers: scalar-checkers, misc-checkers

Examples

```
## Not run:

f <- function(x, y) "Pass"

# Impose a check on x: ensure it's of type "logical"
f_firm <- firmly(f, vld_logical(~x))
f_firm(TRUE, 0)  # [1] "Pass"
f_firm(1, 0)     # Error: "Not logical: x"

# Use a custom error message
f_firm <- firmly(f, vld_logical("x should be a logical vector" ~ x))
f_firm(1, 0)     # Error: "x should be a logical vector"

# To impose the same check on all arguments, apply globalize()
f_firmer <- firmly(f, globalize(vld_logical))
f_firmer(TRUE, FALSE)  # [1] "Pass"
f_firmer(TRUE, 0)      # Error: "Not logical: `y`"
f_firmer(1, 0)         # Errors: "Not logical: `x`", "Not logical: `y`"

## End(Not run)
```

`valaddin` *valaddin: Functional Input Validation*

Description

valaddin provides a functional operator, `firmly`, that enhances functions with input validation. You supply a function `f` along with input validation requirements, and `firmly` returns a function that applies `f` "firmly": before a call to `f` is attempted, its inputs are checked, and if any check fails, an error halts further execution with a message tabulating every failing check. Because `firmly` implements input validation by operating on whole functions rather than values, it is suitable for both programming and interactive use.

Using `firmly` to add input validation to your functions improves the legibility, reusability, and reliability of your code:

- Emphasize the core logic of your functions by excising validation boilerplate.
- Reduce duplication by reusing common checks across functions with common input requirements.
- Make function outputs more predictable by constraining their inputs.
- Vary the strictness of a function according to need and circumstance.

Details

For an example-oriented overview of valaddin, see `vignette("valaddin")`.

validate *Validate objects*

Description

Validate objects

Usage

```
validate(., ..., .checklist = list(), .error_class = "validationError")

.f %checkout% .checks
```

Arguments

.	Object to validate.
...	Input-validation check formula(e).
.checklist	List of check formulae. (These are combined with check formulae provided via ....)
.error_class	Subclass of the error condition to be raised when an input validation error occurs (character).
.f	Interpreted function, i.e., closure.
.checks	List of check formulae, optionally containing a character vector named .error_class, corresponding to the similarly named argument.

Examples

```
## Not run:
library(magrittr)

# Valid assertions: data frame returned (invisibly)
mtcars %>%
  validate(
    vld_all(~sapply(., is.numeric)),
    ~{nrow(.) > 10},
    vld_all(~c("mpg", "cyl") %in% names(.))
  )

# Invalid assertions: error raised
mtcars %>%
  validate(
    vld_all(~sapply(., is.numeric)),
    ~{nrow(.) > 1000},
    vld_all(~c("mpg", "cylinders") %in% names(.))
  )

## End(Not run)
```

Index