Package 'dreamerr'

May 8, 2026

Type Package

Title Error Handling Made Easy

Version 1.5.0

Imports Formula, utils, stringmagic(>= 1.2.0)

Suggests knitr, rmarkdown, stats, graphics

Description Set of tools to facilitate package development and make R a more user-friendly place. Mostly for developers (or anyone who writes/shares functions). Provides a simple, powerful and flexible way to check the arguments passed to functions.
The developer can easily describe the type of argument needed. If the user provides a wrong argument, then an informative error message is prompted with the requested type and the problem clearly stated--saving the user a lot of time in debugging.

License GPL-3

Encoding UTF-8

VignetteBuilder knitr

BugReports https://github.com/lrberge/dreamerr/issues

RoxygenNote 7.3.2

NeedsCompilation no

Author Laurent Berge [aut, cre]

Maintainer Laurent Berge <laurent.berge@u-bordeaux.fr>

Repository CRAN

Date/Publication 2025-04-18 21:00:02 UTC

Contents

`dreamerr-package` *Error Handling Made Easy*

Description

The main purpose of this package is twofold: i) to facilitate the developer's life, and ii) to provide to the users meaningful, useful error messages. These objectives are accomplished with a single function: `check_arg`. That function checks the arguments given by the user: it offers a compact syntax such that complex arguments can be simply stated by the developer. In turn, if the user provides an argument of the wrong type then an informative error message will be buit, stating the expected type and where the error comes from–saving the user quite some time in debugging.

Details

Thus you can very easily make your package look professional with `check_arg` (checking arguments properly *is* professional).

It also offers a set of small tools to provide informative messages to the users. See `stop_up` and `warn_up` to throw errors and warnings in the appropriate location. There are many tools to form messages: `enumerate_items` to form textual list of elements (with many options including conjugating verbs, etc...), `plural` to conjugate verbs depending on the argument, and `n_letter`, `n_th`, `n_times` to write integers in words (which usually looks nicer).

To sum up in a few words, this package was created to enhance the user experience and facilitate package development.

Author(s)

Maintainer: Laurent Berge <laurent.berge@u-bordeaux.fr>

See Also

Useful links:

- Report bugs at https://github.com/lrberge/dreamerr/issues

---

check_arg *Checks arguments and informs the user appropriately*

Description

Full-fledged argument checking. Checks that the user provides arguments of the requested type (even complex) in a very simple way for the developer. Provides detailed and informative error messages for the user.

Usage

```
check_arg(
  .x,
  .type,
  .x1,
  .x2,
  .x3,
  .x4,
  .x5,
  .x6,
  .x7,
  .x8,
  .x9,
  ...,
  .message,
  .choices = NULL,
  .data = list(),
  .value,
  .env,
  .up = 0
)

check_set_arg(
  .x,
  .type,
  .x1,
  .x2,
  .x3,
  .x4,
  .x5,
  .x6,
  .x7,
```

```
  .x8,
  .x9,
  ...,
  .message,
  .choices = NULL,
  .data = list(),
  .value,
  .env,
  .up = 0
)

check_value(
  .x,
  .type,
  .message,
  .arg_name,
  .prefix,
  .choices = NULL,
  .data = list(),
  .value,
  .env,
  .up = 0
)

check_set_value(
  .x,
  .type,
  .message,
  .arg_name,
  .prefix,
  .choices = NULL,
  .data = list(),
  .value,
  .env,
  .up = 0
)

check_arg_plus

check_value_plus
```

Arguments

`.x`	An argument to be checked. Must be an argument name. Can also be the type, see details/examples.
`.type`	A character string representing the requested type(s) of the arguments. This is a bit long so please look at the details section or the vignette for explanations. Each type is composed of one main class and restrictions (optional). Types can

be separated with pipes (|). The main classes are: i) "scalar" for scalars, i.e. vectors of length one, ii) "vector", iii) "matrix", iv) "data.frame", v) "list", vi) formula, vii) function, viii) charin, i.e. a character string in a set of choices, ix) "match", i.e. a character scalar that should partially match a vector of choices, x) path, a character scalar pointing to a file or directory, xi) "class(my_class1, my_class2)", i.e. an object whose class is any of the ones in parentheses, xii) "NA", something identical to NA. You can then add optional restrictions: 1) len(a, b), i.e. the object should be of length between a and b (you can leave a or b missing, len(a) means length *equal* to a), len(data) and len(value) are also possible (see details), 2) nrow(a,b) or ncol(a,b) to specify the expected number of rows or columns, 3) arg(a,b), only for functions, to retrict the number of arguments, 4) "na ok" to allow the object to have NAs (for "scalar" types), or "no na" to restrict the object to have no NA (for "data.frame", "vector", and "matrix" types), 5) GE, GT, LE and LT: for numeric scalars/vectors/matrices, GE{expr} restrics the object to have only values striclty greater than (greater or equal/strictly lower than/lower or equal) the value in curly brackets, 6) e.g. scalar(type1, type2), for scalars/vectors/matrices you can restrict the type of the object by adding the expected type in parentheses: should it be numeric, logical, etc.

.x1 An argument to be checked. Must be an argument name. Can also be the type, see details/examples.

.x2 An argument to be checked. Must be an argument name. Can also be the type, see details/examples.

.x3 An argument to be checked. Must be an argument name. Can also be the type, see details/examples.

.x4 An argument to be checked. Must be an argument name. Can also be the type, see details/examples.

.x5 An argument to be checked. Must be an argument name. Can also be the type, see details/examples.

.x6 An argument to be checked. Must be an argument name. Can also be the type, see details/examples.

.x7 An argument to be checked. Must be an argument name. Can also be the type, see details/examples.

.x8 An argument to be checked. Must be an argument name. Can also be the type, see details/examples.

.x9 An argument to be checked. Must be an argument name. Can also be the type, see details/examples.

... Only used to check '...' (dot-dot-dot) arguments.

.message A character string, optional. By default, if the user provides a wrong argument, the error message stating what type of argument is required is automatically formed. You can alternatively provide your own error message, maybe more tailored to your function. The reason of why there is a problem is appended in the end of the message. You can use the special character __ARG__ in the message. If found, __ARG__ will be replaced by the appropriate argument name. Note that this message is interpolated with string_magic().

`.choices` Only if one of the types (in argument `type`) is `"match"`. The values the argument can take. Note that even if the `type` is `"match"`, this argument is optional since you have other ways to declare the choices.

`.data` Must be a data.frame, a list or a vector. Used in three situations. 1) if the global keywords `eval` or `evalset` are present: the argument will also be evaluated in the data (i.e. the argument can be a variable name of the data set). 2) if the argument is expected to be a formula and `var(data)` is included in the type: then the formula will be expected to contain variables from `.data`. 3) if the keywords `len(data)`, `nrow(data)` or `ncol(data)` are requested, then the required length, number of rows/columns, will be based on the data provided in `.data`.

`.value` An integer scalar or a named list of integers scalars. Used when the keyword `value` is present (like for instance in `len(value)`). If several values are to be provided, then it must be a named list with names equal to the codes: for instance if `nrow(value)` and `ncol(value)` are both present in the type, you can use (numbers are an example) `.value = list(nrow = 5, ncol = 6)`. See Section IV) in the examples.

`.env` An environment defaults to the frame where the user called the original function. Only used in two situations. 1) if the global keywords `eval` or `evalset` are present: the argument will also be evaluated in this environment. 2) if the argument is expected to be a formula and `var(env)` is included in the type: then the formula will be expected to contain variables existing in `.env`.

`.up` Integer, default is 0. If the user provides a wrong argument, the error message will integrate the call of the function from which `check_arg` has been called. If `check_arg` is called in a non-user level sub function of a main user-level function, then use `.up = 1` to make the error message look like it occured in the main function (and not in the sub function). Of course you can have values higher than 1.

`.arg_name` A character scalar. If `.message` is not provided, an automatic error message will be generated using `.arg_name` as the argument name. The structure of the message will be `"Argument '[.arg_name]' must be [requested type]. Problem: [detail of the problem]"`.

`.prefix` A character scalar. If `.message` is not provided, an automatic error message will be generated. The structure of the message will be `"[.prefix] must be [requested type]. Problem: [detail of the problem]"`.

Format

An object of class `function` of length 1.

An object of class `function` of length 1.

Value

In case the `type` is `"match"`, it returns the matched value. In any other case, `NULL` is returned.

Functions

- `check_set_arg()`: Same as `check_arg`, but includes in addition: i) default setting, ii) type conversion, iii) partial matching, and iv) checking list elements. (Small drawback: cannot be turned off.)
- `check_value()`: Checks if a (single) value is of the appropriate type
- `check_set_value()`: Same as `check_value`, but includes in addition: i) default setting, ii) type conversion, iii) partial matching, and iv) checking list elements. (Small drawback: cannot be turned off.)

How to form a type

To write the expected type of an argument, you need to write the main class in combination with the class's options and restrictions (if any).

The syntax is: `"main_class option(s) restriction(s)"`

A type MUST have at least one main class. For example: in the type `"logical vector len(,2) no na"`, `vector` is the main class, `no na` is the option, and `logical` and `len(,2)` are restrictions

There are 14 main classes that can be checked. On the left the keyword, on the right what is expected from the argument, and in square brackets the related section in the examples:

- `scalar`: an atomic vector of length 1 [Section I)]
- `vector`: an atomic vector [Section IV)]
- `matrix`: a matrix [Section IV)]
- `vmatrix`: a matrix or vector [Section IV)]
- `data.frame`: a data.frame [Section VI)]
- `vdata.frame`: a data.frame or vector [Section VI)]
- `list`: a list [Section V)]
- `formula`: a formula [Section VIII)]
- `function`: a function [Section V)]
- `charin`: a character vector with values in a vector of choices [Section III)]
- `match`: a character vector with values in a vector of choices, partial matching enabled and only available in `check_set_arg` [Section III)]
- `path`: a character scalar pointing to a file or a directory
- `class`: a custom class [Section VI)]
- `NA`: a vector of length 1 equal to NA–does not support options nor restrictions, usually combined with other main classes (see Section on combining multiple types) [Section VI)]

There are eight type options, they are not available for each types. Here what they do and the types to which they are associated:

- `NA OK` (or `NAOK`): Tolerates the presence of NA values. Available for `scalar`.
- `NO NA` (or `NONA`): Throws an error if NAs are present. Available for `vector`, `matrix`, `vmatrix`, `data.frame`, and `vdata.frame`.
- `square`: Enforces the matrix to be square. Available for `matrix`, `vmatrix`.

- named: Enforces the object to have names. Available for vector, list.
- multi: Allows multiple matches. Available for charin, match.
- strict: Makes the matching case-sensitive. Available for match.
- os and ts: Available for formula. Option os (resp. ts) enforces that the formula is one-sided (resp. two-sided).
- dir, read and create: Available for path. Option dir checks whether the path points to a directory and not a file. Option read checks whether the file actually exists and has read permission. Option create creates up to the grand-parent folder if it was not yet existing.

You can further add restrictions. There are roughly six types of restrictions. Here what they do and the types to which they are associated:

- sub-type restriction: For atomic types (scalar, vector, matrix or vmatrix), you can restrict the underlying data to be of a specific sub-type. The simple sub-types are: i) integer (numeric without decimals and logicals), i') strict integer (numeric that can be converted to integer with as.integer, and not logicals), ii) numeric, iii) factor, iv) logical and iv') loose logical (0/1 are also OK). Simply add the sub-type in the type string (e.g. "integer scalar"), or if you allow multiple types, put them in parentheses rigth after the main class: e.g. "scalar(character, integer)". See Section XI) in the examples. See also the section below for more information on the sub-types. Some types (character, integer, numeric, logical and factor) also support the keyword "conv" in check_set_arg.
- GE/GT/LE/LT: For atomic types with numeric data, you can check the values in the object. The GE/GT/LE/LT mean respectively greater or equal/greater than/lower or equal/lower than. The syntax is GE{expr}, with expr any expression. See Section IV) in the examples.
- len(a, b): You can restrict the length of objects with len(a, b) (with a and b integers). Available for vector and list. Then the length must be in between a and b. Either a or b can be missing which means absence of restriction. If len(a), this means must be equal to a. You can also use the keywords len(data) which ensures that the length is the same as the length of the object given in the argument .data, or len(value) which ensures the length is equal to the value given in .value. See Section IV) in the examples.
- nrow(a, b), ncol(a, b): To restrict the number of rows and columns. Available for matrix, vmatrix, data.frame, vdata.frame. Tolerates the data and value keywords (see in len). See Section IV) in the examples.
- var(data, env): Available only for formula. var(data) ensures that the variables in the formula are present in the data set given by the extra argument .data. var(env) ensures they are present in the environment, and var(data, env) in either the environment or the data set. See Section VIII) in the examples.
- arg(a, b): Available only for function. Ensures that the function has a number of arguments between a and b, both integers (possibly missing). Tolerates the value keyword (see in len). See Section V) in the examples.
- left(a, b) and right(a, b): Only available for formula. Restricts the number of parts in the left-hand-side or in the right-hand-side of the formula. Tolerates the value keyword (see in len). See Section VIII) in the examples.

[Section I)]: R:Section%20I) [Section IV)]: R:Section%20IV) [Section IV)]: R:Section%20IV) [Section IV)]: R:Section%20IV) [Section VI)]: R:Section%20VI) [Section VI)]: R:Section%20VI) [Section V)]: R:Section%20V) [Section VIII)]: R:Section%20VIII) [Section V)]: R:Section%20V)

[Section III)]: R:Section%20III) [Section III)]: R:Section%20III) [Section VI)]: R:Section%20VI) [Section VI)]: R:Section%20VI)

Global keywords

There are eight global keywords that can be placed anywhere in the type. They are described in Section II) in the examples.

- `NULL`: allows the argument to be equal to `NULL`.
- `safe NULL`: allows the argument to be equal to `NULL`, but an error is thrown if the argument is of the type `base$variable` or `base[["variable"]]`. This is to prevent oversights from the user, especially useful when the main class is a vector.
- `NULL{expr}`: allows the argument to be equal to `NULL`, if the argument is `NULL`, then it assigns the value of expr to the argument.
- `MBT`: (means "must be there") an error is thrown if the argument is not provided by the user.
- `eval`: used in combination with the extra argument `.data`. Evaluates the value of the argument both in the data set and in the environment (this means the argument can be a variable name).
- `evalset`: like `eval`, but after evaluation, assigns the obtained value to the argument. Only available in `check_set_arg`.
- `dotnames`: only when checking `'...'` argument (see the related section below). Enforces that each object in `'...'` has a name.

The `match` and `charin` types

The main classes `match` and `charin` are similar to `match.arg`. These two types are detailed in the examples Section III).

By default, the main class `match` expects a single character string whose value is in a set of choices. By default, there is no case sensitity (which can be turned on with the option `strict`) and there is always partial matching. It can expect a vector (instead of a single element) if the option `multi` is present.

You have three different ways to set the choices:

- by setting the argument default: e.g. `fun = function(x = c("Tom", "John")) check_arg(x, "match")`
- by providing the argument `.choices`: e.g. `fun = function(x) check_arg(x, "match", .choices = c("Tom", "John"))`
- by writing the choices in parentheses: e.g. `fun = function(x) check_arg(x, "match(Tom, John)")`

When the user doesn't provide the argument, the default is set to the first choice. Since the main class `match` performs a re-assignment of the variable, it is only available in `check_set_arg`.

The main class `charin` is similar to `match` in that it expects a single character string in a set of choices. The main differences are: i) there is no partial matching, ii) the choices cannot be set by setting the argument default, and iii) its checking can be turned off with setDreamer_check(FALSE) (that's the main difference between `check_arg` and `check_set_arg`).

Combining multiple types

You can combine multiple types with a pipe: '|'. The syntax is as follows:

`"main_type option(x) restriction(s) | main_type option(x) restriction(s) | main_type option(x) restriction(s)"`

You can combine as many types as you want. The behavior is as follows: if the argument matches any of the types, then that's fine.

For example, say you require an argument to be either a logical scalar, either a data.frame, then you can write: `check_arg(x, "logical scalar | data.frame")`. See Section X) in the examples for a more complex example.

Tips on the type

The type MUST be a character string of length 1. Two main classes must be separated by a pipe. Otherwise the order of the keywords, the spaces, or the case don't matter. Further the global keywords can be placed anywhere and need not be separated by a pipe.

Note that a rare but problematic situation is when you set a default with the global `NULL{default}` and that default contains a keyword. For example in the type `"NULL{list()} numeric matrix"` `list` should not be considered as a main class, but only `matrix`. To be on the safe side, then just separate them with a pipe: `"NULL{list()} | numeric matrix"` would work appropriately.

Checking multiple arguments

You can check multiple arguments at once provided they are of the same type. Say variables `x1` to `x5` should be logical scalars. Just use: `check_arg(x1, x2, x3, x4, x5, "logical scalar")`. It is always more efficient to check multiple arguments of the same type *at once*.

It is important to note that in case of multiple arguments, you can place the type anywhere you want provided it is a character literal (and not in a variable!). This means that `check_arg("logical scalar", x1, x2, x3, x4, x5)` would also work.

If your type is in a variable, then you must explicitly provide the argument `.type` (like in `check_arg(x, .type = my_type)`).

Nesting argument checking (`.up`)

When you develop several functions that share common features, it is usually good practice to pool the common computations into an internal function (to avoid code duplication).

When you do so, you can do all the argument checking in the internal function. Then use the argument `.up = 1` so that if the user provdes a wrong argument, the error message will refer to the user-level function and NOT to the internal function, making it much clearer for the user.

This is detailed in Section XII) in the examples.

Checking the `...` (dot-dot-dot) argument

`check_arg` offers the possibility to check the `...`, provided each expected object in `...` should be of the same type. To do that, just add `...` as the first argument in `check_arg`, that's it! For example, you want all elements of `...` to be numeric vectors, then use `check_arg(..., "numeric vector")`.

When checking ..., you have the special global argument dotnames which enforces that each element in ... has a name. Further, the other global MBT (must be there) now means that at least one element in ... must be provided.

This is detailed in Section XIV) in the examples.

What's the difference between check_arg and check_set_arg?

The function check_set_arg extends check_arg in several ways. First it offers new keywords:

- evalset: evaluates the argument in a data set (i.e. the argument can be variables names of a data set), then re-assigns back its value.
- NULL{default}: if the argument is NULL, then the value in curly brackets is assigned to the argument.
- match: if the argument partially matches the choices, then the matches are assigned to the argument.
- conv: in atomic main classes (scalar, vector and matrix), the data can be converted to a given sub-type (currently integer, numeric, logical, character and factor), then assigned back to the argument.

As you can see, it's all about assignment: these special keywords of check_set_arg will modify the arguments *in place*. You have such examples in Section II), III) and XI) of the examples.

Second, it allows to check arguments that are themselves list of arguments (note that conv also works in that case). For example, one argument of your function is plot.opts, a list of arguments to be passed to plot. You can check the elements of plot.opts (e.g. plot.opts$main) with check_set_arg. It also re-assigns the values of the list given the special keywords just described. List element checking is described in Section XIII) of the examples.

Then why creating two functions? If the user runs a function in which the arguments were checked with check_arg and it works, then argument checking can be safely disabled, and it would also work. On the other hand, since check_set_arg does value re-assignment, it cannot be safely turned-off–therefore cannot be disabled with setDreamerr_check. Distinguishing between the two allows the user to disable argument checking and gain (although very modest) perfomance in large loops. Therefore, when you create functions, I suggest to use always check_arg, unless you need the extra features of check_set_arg.

check_value

The functions check_value and check_set_value are almost identical to the respective functions check_arg and check_set_arg. The key differences are as follows:

- They can check values instead of arguments. Indeed, if you try to check a value with check_arg, nothing will happen (provided the name of the value is not an argument). Why? Because it will consider it as a missing argument. Therefore, you are can check anything with check_value.
- You can check only one item at a time (whereas you can check up to 10 arguments in check_arg).

The main reason for using check_value is that sometimes you only know if an argument is valid after having perfomed some modifications on it. For instance, the argument may be a formula, but you also require that the variables in the formula are numeric. You cannot check all that at once with

check_arg, but you can first check the formula with it, then extract the values from the formula and use check_value to ensure that the variables from the formula are numeric.

check_value is detailed in Section XVI) in the examples.

Disabling argument checking

Although the argument checking offered by check_arg is highly optimized and fast (it depends on the type (and your computer), but it is roughly of the order of 80 micro seconds for non-missing arguments, 20 micro seconds for missing arguments), you may want to disable it for small functions in large loops (>100K iterations although this practice is not really common in R). If so, just use the function setDreamerr_check, by typing setDreamerr_check(FALSE). This will disable any call to check_arg.

Note that the argument checking of check_set_arg cannot be disabled because the special types it allows perform reassignment in the upper frame. That's the main difference with check_arg.

The developer mode

If you're new to check_arg, given the many types available, it's very common to make mistakes when creating check_arg calls. But no worry, the developer mode is here to help!

The developer mode ensures that any problematic call is spotted and the problem is clearly stated. It also refers to the related section in the examples if appropriate. To turn the developer mode on, use setDreamerr_dev.mode(TRUE).

Note that since this mode ensures a detailed cheking of the call it is thus a strain on performance and should be always turned off otherwise needed. See Section XV) in the examples.

Author(s)

Laurent Berge

Examples

```
# check_arg is only used within functions

#
# I) Example for the main class "scalar"
#

test_scalar = function(xlog, xnum, xint, xnumlt, xdate){
  # when forming the type: you can see that case, order and spaces don't matter
  check_arg(xlog, "scalarLogical")
  check_arg(xnum, "numeric scalar")
  check_arg(xint, "  scalar Integer GE{0}  ")
  check_arg(xnumlt, "numeric scalar lt{0.15}")

  # Below it is critical that there's no space between scalar and the parenthesis
  check_arg(xdate, "scalar(Date)")
  invisible(NULL)
}

# Following is OK
```

```
test_scalar()
test_scalar(xlog = FALSE, xnum = 55, xint = 5, xnumlt = 0.11, xdate = Sys.Date())

#
# Now errors, all the following are wrong arguments, leading to errors
# Please note the details in the error messages.

# logical
try(test_scalar(xlog = NA))
try(test_scalar(xlog = 2))
try(test_scalar(xlog = sum))
try(test_scalar(xlog = faefeaf5))
try(test_scalar(xlog = c(TRUE, FALSE)))
try(test_scalar(xlog = c()))

# numeric
try(test_scalar(xnum = NA))
try(test_scalar(xnum = 1:5))
try(test_scalar(xnum = Sys.Date()))

# integer
try(test_scalar(xint = 5.5))
try(test_scalar(xint = -1))

# num < 0.15
try(test_scalar(xnumlt = 0.15))
try(test_scalar(xnumlt = 0.16))
try(test_scalar(xnumlt = Sys.Date()))

# Date
try(test_scalar(xdate = 0.15))


#
# II) Examples for the globals: NULL, MBT, eval, evalset
#

test_globals = function(xnum, xlog = TRUE, xint){

  # Default setting with NULL is only available in check_set_arg
  # MBT (must be there) throws an error if the user doesn't provide the argument
  check_set_arg(xnum, "numeric vector NULL{1} MBT")

  # NULL allows NULL values
  check_arg(xlog, "logical scalar safe NULL")

  check_arg(xint, "integer vector")

  list(xnum = xnum, xlog = xlog)
}

# xnum is required because of MBT option
try(test_globals())
```

```
# NULL{expr} sets the value of xnum to expr if xnum = NULL
# Here NULL{1} sets xnum to 1
test_globals(xnum = NULL)

# NULL (not NULL{expr}) does not reassign: xlog remains NULL
test_globals(xnum = NULL, xlog = NULL)

# safe NULL: doesn't accept NULL from data.frame (DF) subselection
# ex: the variable 'log' does not exist in the iris DF
try(test_globals(5, xlog = iris$log))
# but xnum accepts it
test_globals(iris$log)


#
# eval and evalset
#

test_eval = function(x1, x2, data = list(), i = c()){
  check_arg(x1, "eval numeric vector", .data = data)

  # evalset is in check_set_arg
  check_set_arg(x2, "evalset numeric vector", .data = data)

 # We show the variables
 if(1 %in% i){
   cat("x1:\n")
   print(as.character(try(x1, silent = TRUE)))
 }

 if(2 %in% i){
   cat("x2:\n")
   print(as.character(try(x2, silent = TRUE)))
 }

}

# eval: evaluates the argument both in the environment and the data
test_eval(x1 = Sepal.Length, data = iris) # OK
# if we use a variable not in the environment nor in the data => error
try(test_eval(x1 = Sopal.Length, data = iris))

# but eval doesn't reassign back the value of the argument:
test_eval(x1 = Sepal.Length, data = iris, i = 1)

# evaset does the same as eval, but also reasssigns the value obtained:
test_eval(x2 = Sepal.Length, data = iris, i = 2)


#
# III) Match and charin
#
```

```
# match  => does partial matching, only available in check_set_arg
# charin => no partial matching, exact values required, but in check_arg

#
# match
#

# Note the three different ways to provide the choices
#
# If the argument has no default, it is kept that way (see x2)
# If the argument is not provided by the user,
#  it is left untouched (see x3)

test_match = function(x1 = c("bonjour", "Au revoir"), x2, x3 = "test"){
  # 1) choices set thanks to the argument default (like in match.arg)
  check_set_arg(x1, "strict match")

  # 2) choices set with the argument .choices
  check_set_arg(x2, "match", .choices = c("Sarah", "Santa", "Santa Fe", "SANTA"))

  # 3) choices set with the parentheses
  check_set_arg(x3, "multi match(Orange, Juice, Good)")

 cat("x1:", x1, "\nx2:", tryCatch(x2, error = function(e) "[missing]"), "\nx3:", x3, "\n")
}

# Everything below is OK
test_match()
test_match(x1 = "Au", x2 = "sar", x3 = c("GOOD", "or"))
test_match(x2 = "Santa")

# Errors caught:
try(test_match(x1 = c("Au", "revoir")))
try(test_match(x1 = "au"))
try(test_match(x1 = sum))
try(test_match(x1 = list(a = 1:5)))

try(test_match(x2 = "san"))
try(test_match(x2 = "santa"))

# Same value as x3's default, but now provided by the user
try(test_match(x3 = "test"))
try(test_match(x3 = c("or", "ju", "bad")))

# You can check multiple arguments at once
# [see details for multiple arguments in Section X)]
# Note that now the choices must be set in the argument
#  and they must have the same options (ie multi, strict)

test_match_multi = function(x1 = c("bonjour", "Au revoir"), x2 = c("Sarah", "Santa"),
                            x3 = c("Orange", "Juice", "Good")){
```

```
  # multiple arguments at once
  check_set_arg(x1, x2, x3, "match")

  cat("x1:", x1, "\nx2:", x2, "\nx3:", x3, "\n")
}

test_match_multi()

#
# charin
#

# charin is similar to match but requires the user to provide the exact value
# only the multi option is available

test_charin = function(x1 = "bonjour", x2  = "Sarah"){

  # 1) set the choices with .choices
  check_arg(x1, "charin", .choices = c("bonjour", "au revoir"))

  # 2) set the choices with the parentheses
  check_arg(x2, "multi charin(Sarah, Santa, Santa Fe)")

  cat("x1:", x1, "\nx2:", x2, "\n")
}

# Now we need the exact values
test_charin("au revoir", c("Santa", "Santa Fe"))

# Errors when partial matching tried
try(test_charin("au re"))


#
# IV) Vectors and marices, equalities, dimensions and lengths
#

# You can restrict the length of objects with len(a, b)
#    - if len(a, b) length must be in between a and b
#    - if len(a, ) length must be at least a
#    - if len(, b) length must be at most b
#    - if len(a) length must be equal to a
# You can also use the special keywords len(data) or len(value),
#  but then the argument .data or .value must also be provided.
#  (the related example comes later)
#
# You can restrict the number of rows/columns with nrow(a, b) and ncol(a, b)
#
# You can restrict a matrix to be square with the 'square' keyword
#
# You can restrict the values an element can take with GE/GT/LE/LT,
# respectively greater or equal/greater than/lower or equal/lower than
# The syntax is GE{expr}, with expr any expression
```

```
# Of course, it only works for numeric values
#
# By default NAs are tolerated in vector, matrix and data.frame.
# You can refuse NAs using the keyword: 'no na' or 'nona'
#

test_vmat = function(xvec, xmat, xvmat, xstmat, xnamed){
  # vector of integers with values between 5 and exp(3)
  check_arg(xvec, "integer Vector GE{5} LT{exp(3)}")

  # logical matrix with at least two rows and with 3 columns
  check_arg(xmat, "logicalMatrix NROW(2,) NCOL(3)")

  # vector or matrix (vmatrix) of integers or character strings
  # with at most 3 observations
  # NAs are not allowed
  check_arg(xvmat, "vmatrix(character, integer) nrow(,3) no na")

  # square matrix of integers, logicals reports errors
  check_arg(xstmat, "strict integer square Matrix")

  # A vector with names of length 2
  check_arg(xnamed, "named Vector len(2)")
  invisible(NULL)
}

# OK
test_vmat(xvec = 5:20, xmat = matrix(TRUE, 3, 3), xvmat = c("abc", 4, 3),
          xstmat = matrix(1:4, 2, 2), xnamed = c(bon=1, jour=2))

# Vector checks:
try(test_vmat(xvec = 2))
try(test_vmat(xvec = 21))
try(test_vmat(xvec = 5.5))

# Matrix checks:
try(test_vmat(xmat = matrix(TRUE, 3, 4)))
try(test_vmat(xmat = matrix(2, 3, 3)))
try(test_vmat(xmat = matrix(FALSE, 1, 3)))
try(test_vmat(xmat = iris))

try(test_vmat(xvmat = iris))
try(test_vmat(xvmat = c(NA, 5)))

try(test_vmat(xstmat = matrix(1, 1, 3)))
try(test_vmat(xstmat = matrix(c(TRUE, FALSE, NA), 3, 3)))

# Named vector checks:
try(test_vmat(xnamed = 1:3))
try(test_vmat(xnamed = c(bon=1, jour=2, les=3)))

#
# Illustration of the keywords 'data', 'value'
```

```
#

# 'value'
# Matrix multiplication X * Y * Z
test_dynamic_restriction = function(x, y, z){
  check_arg(x, "mbt numeric matrix")
  check_arg(y, "mbt numeric matrix nrow(value)", .value = ncol(x))
  check_arg(z, "mbt numeric matrix nrow(value)", .value = ncol(y))

  # An alternative to the previous two lines:
  # check_arg(z, "mbt numeric matrix")
  # check_arg(y, "mbt numeric matrix nrow(value) ncol(value)",
  #           .value = list(nrow = ncol(x), ncol = nrow(z)))

  x %*% y %*% z
}

x = matrix(1, 2, 3)
y = matrix(2, 3, 5)
z = matrix(rnorm(10), 5,  2)

test_dynamic_restriction(x, y, z)

# Now error
try(test_dynamic_restriction(x, matrix(5, 1, 2), z))

# 'data'
# Computing maximum difference between two matrices
test_dynamic_bis = function(x, y){
  check_arg(x, "mbt numeric matrix")
  # we require y to be of the same dimension as x
  check_arg(y, "mbt numeric matrix nrow(data) ncol(data)", .data = x)

  max(abs(x - y))
}

test_dynamic_bis(x, x)

# Now error
try(test_dynamic_bis(x, y))


#
# V) Functions and lists
#

# You can restrict the number of arguments of a
# function with arg(a, b) [see Section IV) for details]

test_funlist = function(xfun, xlist){
  check_arg(xfun, "function arg(1,2)")
  check_arg(xlist, "list len(,3)")
  invisible(NULL)
```

```
}

# OK
test_funlist(xfun = sum, xlist = iris[c(1,2)])

# function checks:
try(test_funlist(xfun = function(x, y, z) x + y + z))

# list checks:
try(test_funlist(xlist = iris[1:4]))
try(test_funlist(xlist = list()))


#
# VI) Data.frame and custom class
#

test_df = function(xdf, xvdf, xcustom){
  # data.frame with at least 100 observations
  check_arg(xdf, "data.frame nrow(100,)")

  # data.frame or vector (vdata.frame)
  check_arg(xvdf, "vdata.frame")

  # Either: i) object of class glm or lm
  # ii) NA
  # iii) NULL
  check_arg(xcustom, "class(lm, glm)|NA|null")
  invisible(NULL)
}

# OK
m = lm(Sepal.Length~Species, iris)
test_df(xdf = iris, xcustom = m)
test_df(xvdf = iris$Sepal.Length)
test_df(xcustom = NULL)

# data.frame checks:
try(test_df(xdf = iris[1:50,]))
try(test_df(xdf = iris[integer(0)]))
try(test_df(xdf = iris$Sepal.Length))
# Note that the following works:
test_df(xvdf = iris$Sepal.Length)

# Custom class checks:
try(test_df(xcustom = iris))

#
# VIII) Formulas
#

# The keyword is 'formula'
# You can restrict the formula to be:
```

```
# - one sided with 'os'
# - two sided with 'ts'
#
# You can restrict that the variables of a forumula must be in
# a data set or in the environment with var(data, env)
# - var(data) => variables must be in the data set
# - var(env) => variables must be in the environment
# - var(data, env) => variables must be in the data set or in the environment
# Of course, if var(data), you must provide a data set
#
# Checking multipart formulas is included. You can use left(a, b)
#  and right(a, b) to put restrictions in the number of parts allowed
#  in the left and right-hand-sides
#

test_formulas = function(fml1, fml2, fml3, fml4, data = iris){
  # Regular formula, variables must be in the data set
  check_arg(fml1, "formula var(data)", .data = data)

  # One sided formula, variables in the environment
  check_arg(fml2, "os formula var(env)")

  # Two sided formula, variables in the data set or in the env.
  check_arg(fml3, "ts formula var(data, env)", .data = data)

  # One or two sided, at most two parts in the RHS, at most 1 in the LHS
  check_arg(fml4, "formula left(,1) right(,2)")

  invisible(NULL)
}

# We set x1 in the environment
x1 = 5

# Works
test_formulas(~Sepal.Length, ~x1, Sepal.Length~x1, a ~ b, data = iris)

# Now let's see errors
try(test_formulas(Sepal.Length~x1, data = iris))

try(test_formulas(fml2 = ~Sepal.Length, data = iris))
try(test_formulas(fml2 = Sepal.Length~x1, data = iris))

try(test_formulas(fml3 = ~x1, data = iris))
try(test_formulas(fml3 = x1~x555, data = iris))

try(test_formulas(fml4 = a ~ b | c | d))
try(test_formulas(fml4 = a | b ~ c | d))



#
# IX) Multiple types
```

```
#

# You can check multiple types using a pipe: '|'
# Note that global keywords (like NULL, eval, etc) need not be
# separated by pipes. They can be anywhere, the following are identical:
#  - "character scalar | data.frame NULL"
#  - "NULL character scalar | data.frame"
#  - "character scalar NULL | data.frame"
#  - "character scalar | data.frame | NULL"
#

test_mult = function(x){
  # x must be either:
  # i) a numeric vector of length at least 2
  # ii) a square character matrix
  # iii) an integer scalar (vector of length 1)
  check_arg(x, "numeric vector len(2,) | square character matrix | integer scalar")
  invisible(NULL)
}

# OK
test_mult(1)
test_mult(1:2)
test_mult(matrix("ok", 1, 1))

# Not OK, notice the very detailed error messages
try(test_mult(matrix("bonjour", 1, 2)))
try(test_mult(1.1))


#
# X) Multiple arguments
#

# You can check multiple arguments at once if they have the same type.
# You can add the type where you want but it must be a character literal.
# You can check up to 10 arguments with the same type.

test_multiarg = function(xlog1, xlog2, xnum1, xnum2, xnum3){
  # checking the logicals
  check_arg(xlog1, xlog2, "logical scalar")

  # checking the numerics
  # => Alternatively, you can add the type first
  check_arg("numeric vector", xnum1, xnum2, xnum3)

  invisible(NULL)
}

# Let's throw some errors
try(test_multiarg(xlog2 = 4))
try(test_multiarg(xnum3 = "test"))
```

```
#
# XI) Multiple sub-stypes
#

# For atomic arguments (like vector or matrices),
# you can check the type of underlying data: is it integer, numeric, etc?
# There are five simple sub-types:
# - integer
# - numeric
# - factor
# - logical
# - loose logical: either TRUE/FALSE, either 0/1
#
# If you require that the data is of one sub-type only:
# - a) if it's one of the simple sub-types: add the keyword directly in the type
# - b) otherwise: add the sub-type in parentheses
#
# Note that the parentheses MUST follow the main class directly.
#
# Example:
# - a) "integer scalar"
# - b) "scalar(Date)"
#
# If you want to check multiple sub-types: you must add them in parentheses.
# Again, the parentheses MUST follow the main class directly.
# Examples:
# "vector(character, factor)"
# "scalar(integer, logical)"
# "matrix(Date, integer, logical)"
#
# In check_set_arg, you can use the keyword "conv" to convert to the
# desired type
#

test_multi_subtypes = function(x, y){
  check_arg(x, "scalar(integer, logical)")
  check_arg(y, "vector(character, factor, Date)")
  invisible(NULL)
}

# What follows doesn't work
try(test_multi_subtypes(x = 5.5))

# Note that it works if x = 5
#  (for check_arg 5 is integer although is.integer(5) returns FALSE)
test_multi_subtypes(x = 5)

try(test_multi_subtypes(y = 5.5))

# Testing the "conv" keyword:

test_conv = function(x, type){
```

```
  check_set_arg(x, .type = type)
  x
}

class(test_conv(5L, "numeric scalar conv"))
class(test_conv(5, "integer scalar conv"))
class(test_conv(5, "integer scalar"))

# You can use the "conv" keyword in multi-types
# Remember that types are checked in ORDER! (see the behavior)
test_conv(5:1, "vector(logical, character conv)")
test_conv(c(TRUE, FALSE), "vector(logical, character conv)")


#
# XII) Nested checking: using .up
#

# Say you have two user level functions
# But you do all the computation in an internal function.
# The error message should be at the level of the user-level function
# You can use the argument .up to do that
#

sum_fun = function(x, y){
  my_internal(x, y, sum = TRUE)
}

diff_fun = function(x, y){
  my_internal(x, y, sum = FALSE)
}

my_internal = function(x, y, sum){
  # The error messages will be at the level of the user-level functions
  # which are 1 up the stack
  check_arg(x, y, "numeric scalar mbt", .up = 1)

  if(sum) return(x + y)
  return(x - y)
}

# we check it works
sum_fun(5, 6)
diff_fun(5, 6)

# Let's throw some errors
try(sum_fun(5))
try(diff_fun(5, 1:5))

# The errors are at the level of sum_fun/diff_fun although
# the arguments have been checked in my_internal.
# => much easier for the user to understand the problem
```

```
#
# XIII) Using check_set_arg to check and set list defaults
#

# Sometimes it is useful to have arguments that are themselves
# list of arguments.
# Witch check_set_arg you can check the arguments nested in lists
# and easily set default values at the same time.
#
# When you check a list element, you MUST use the syntax argument$element
#

# Function that performs a regression then plots it
plot_cor = function(x, y, lm.opts = list(), plot.opts = list(), line.opts = list()){

  check_arg(x, y, "numeric vector")

  # First we ensure the arguments are lists
  check_arg(lm.opts, plot.opts, line.opts, "named list")

  # The linear regression
  lm.opts$formula = y ~ x
  reg = do.call("lm", lm.opts)

  # plotting the correlation, with defaults
  check_set_arg(plot.opts$main, "character scalar NULL{'Correlation between x and y'}")

  # you can use variables created in the function when setting the default
  x_name = deparse(substitute(x))
  check_set_arg(plot.opts$xlab, "character scalar NULL{x_name}")
  check_set_arg(plot.opts$ylab, "character scalar NULL{'y'}")

  # we restrict to only two plotting types: p or h
  check_set_arg(plot.opts$type, "NULL{'p'} match(p, h)")

  plot.opts$x = x
  plot.opts$y = y
  do.call("plot", plot.opts)

  # with the fit
  check_set_arg(line.opts$col, "NULL{'firebrick'}") # no checking but default setting
  check_set_arg(line.opts$lwd, "integer scalar GE{0} NULL{2}") # check + default
  line.opts$a = reg
  do.call("abline", line.opts)
}

sepal_length = iris$Sepal.Length ; y = iris$Sepal.Width
plot_cor(sepal_length, y)

plot_cor(sepal_length, y, plot.opts = list(col = iris$Species, main = "Another title"))

# Now throwing errors
```

```
try(plot_cor(sepal_length, y, plot.opts = list(type = "l")))
try(plot_cor(sepal_length, y, line.opts = list(lwd = -50)))


#
# XIV) Checking '...' (dot-dot-dot)
#

# You can also check the '...' argument if you expect all objects
# to be of the same type.
#
# To do so, you MUST place the ... in the first argument of check_arg
#

sum_check = function(...){
  # we want each element of ... to be numeric vectors without NAs
  # we want at least one element to be there (mbt)
  check_arg(..., "numeric vector mbt")

  # once the check is done, we apply sum
  sum(...)
}

sum_check(1:5, 5:20)

# Now let's compare the behavior of sum_check() with that of sum()
# in the presence of errors
x = 1:5 ; y = pt
try(sum_check(x, y))
try(sum(x, y))

# As you can see, in the first call, it's very easy to spot and debug the problem
# while in the second call it's almost impossible



#
# XV) Developer mode
#

# If you're new to check_arg, given the many types available,
# it's very common to make mistakes when creating check_arg calls.
# The developer mode ensures that any problematic call is spotted
# and the problem is clearly stated
#
# Note that since this mode ensures a detailed cheking of the call
# it is thus a strain on performance and should be always turned off
# otherwise needed.
#

# Setting the developer mode on:
setDreamerr_dev.mode(TRUE)
```

```
# Creating some 'wrong' calls => the problem is pinpointed

test_err1 = function(x) check_arg(x, "integer scalar", "numeric vector")
try(test_err1())

test_err2 = function(...) check_arg("numeric vector", ...)
try(test_err2())

test_err3 = function(x) check_arg(x$a, "numeric vector")
try(test_err3())

test_err4 = function(x) check_arg(x, "numeric vector integer")
try(test_err4())

# Setting the developer mode off:
setDreamerr_dev.mode(FALSE)


#
# XVI) Using check_value
#

# The main function for checking arguments is check_arg.
# But sometimes you only know if an argument is valid after
#  having perfomed some modifications on it.
# => that's when check_value kicks in.
#
# It's better with an example.
#
# In this example we'll construct a plotting function
# using a formula, with a rock-solid argument checking.
#

# Plotting function, but using a formula
# You want to plot only numeric values
plot_fml = function(fml, data, ...){
  # We first check the arguments
  check_arg(data, "data.frame mbt")
  check_arg(fml, "ts formula mbt var(data)", .data = data)

  # We extract the values of the formula
  y = fml[[2]]
  x = fml[[3]]

  # Now we check that x and y are valid => with check_value
  # We also use the possibility to assign the value of y and x directly
  # We add a custom message because y/x are NOT arguments
  check_set_value(y, "evalset numeric vector", .data = data,
                  .message = "In the argument 'fml', the LHS must be numeric.")
  check_set_value(x, "evalset numeric vector", .data = data,
                  .message = "In the argument 'fml', the RHS must be numeric.")

  # The dots => only arguments to plot are valid
```

```
  args_ok = c(formalArgs(plot.default), names(par()))
  validate_dots(valid_args = args_ok, stop = TRUE)

  # We also set the xlab/ylab
  dots = list(...) # dots has a special meaning in check_value (no need to pass .message)
  check_set_value(dots$ylab, "NULL{deparse(fml[[2]])} character vector conv len(,3)")
  check_set_value(dots$xlab, "NULL{deparse(fml[[3]])} character vector conv len(,3)")

  dots$y = y
  dots$x = x

  do.call("plot", dots)

}

# Let's check it works
plot_fml(Sepal.Length ~ Petal.Length + Sepal.Width, iris)
plot_fml(Sepal.Length ~ Petal.Length + Sepal.Width, iris, xlab = "Not the default xlab")

# Now let's throw some errors
try(plot_fml(Sepal.Length ~ Species, iris))
try(plot_fml(Sepal.Length ~ Petal.Length, iris, xlab = iris))
try(plot_fml(Sepal.Length ~ Petal.Length, iris, xlab = iris$Species))
```

`check_expr` *Checks the evaluation of an expression*

Description

This functions checks the evaluation of an expression and, if an error is thrown, captures it and integrates the captured message after a custom error message.

Usage

```
check_expr(expr, ..., clean, up = 0, arg_name, verbatim = FALSE)

check_expr_hook(expr, ..., clean, arg_name, verbatim = FALSE)

generate_check_expr_hook(namespace)
```

Arguments

`expr`	An expression to be evaluated.
`...`	Character scalars. The values of `...` will be coerced with the function string_magic. This means that string interpolation is allowed. Ex: `"Arg. {arg} should be positive"` leads to `"Arg. power should be positive"` if `arg` is equal to "power". If argument `verbatim` is `TRUE`, the values are instead coereced with `paste0`.

clean	Character vector, default is missing. If provided, the function string_clean is applied to the *captured error message* to clean it when necessary. Each element of the vector should be of the form "pat => rep" with pat a regular expression to be replace and rep the replacement.
up	Integer, default is 0. It is used to construct the call in the error message. By default the call reported is the function containing check_expr. If you want to report a function higher in the stack, use up = 1, or higher.
arg_name	Character scalar, default is missing. Used when the expression in expr leads to an error and the custom message is missing (i.e. no element is provided in ...). In that case, the default message is: "The argument {arg_name} could not be evaluated.". The default value for arg_name is deparse(substitute(expr)), if this guess is wrong, use arg_name.
verbatim	Logical scalar, default is FALSE. By default the elements of ... allow string interpolation with "{}" using stringmagic. If TRUE, no interpolation is performed.
namespace	Character scalar giving the namespace for which the hooks are valid. Only useful when hook functions are used in a package.

Details

The purpose of this functions is to provide useful error messages to the user.

Functions

- check_expr_hook(): As check_expr but sets the error call at the level of the hooked function
- generate_check_expr_hook(): Generates a package specific check_expr_hook function

Author(s)

Laurent Berge

See Also

For general argument checking, see check_arg() and check_set_arg().

Examples

```
test = function(x, y){
  check_expr(mean(x, y), "Computing the mean didn't work:")
}
```

`enumerate_items` *Enumerates the elements of a vector*

Description

Transforms a vector into a single character string enumerating the values of the vector. Many options exist to customize the result. The main purpose of this function is to ease the creation of user-level messages.

Usage

```
enumerate_items(
  x,
  type,
  verb = FALSE,
  s = FALSE,
  past = FALSE,
  or = FALSE,
  start_verb = FALSE,
  quote = FALSE,
  enum = FALSE,
  other = "",
  nmax = 7
)
```

Arguments

`x`	A vector.
`type`	A single character string, optional. If this argument is used, it supersedes all other arguments. It compactly provides the arguments of the function: it must be like `"arg1.arg2.arg3"`, i.e. a list of arguments separated by a point. The arguments are: "s" (to add a starting s if `length(x)>1`), "or" (to have "or" instead of "and"), "start" (to place the verb at the start instead of in the end), "quote" (to quote the elements of the vector), "enum" (to make an enumeration), "past" (to put the verb in past tense), a verb (i.e. anything different from the previous codes is a verb). Use `other(XX)` to set the argument `other` to `XX`. See details and examples.
`verb`	Default is `FALSE`. If provided, a verb is added at the end of the string, at the appropriate form. You add the verb at the start of the string using the argument `start_verb`. Valid verbs are: "be", "is", "has", "have", and any other verb with a regular form.
`s`	Logical, default is `FALSE`. If `TRUE` a `s` is added at the beginning of the string if the length of `x` is greater than one.
`past`	Logical, default is `FALSE`. If `TRUE` the verb is put at the past tense.
`or`	Logical, default is `FALSE`. If `TRUE` the two last items of the vector are separated by "or" instead of "and".

start_verb	Logical, default is FALSE. If TRUE the verb is placed at the beginning of the string instead of the end.
quote	Logical, default is FALSE. If TRUE all items are put in between single quotes.
enum	Logical, default is FALSE. If provided, an enumeration of the items of x is created. The possible values are "i", "I", "1", "a" and "A". Example: x = c(5, 3, 12), enum = "i" will lead to "i) 5, ii) 3, and iii) 12".
other	Character scalar, defaults to the empty string: "". If there are more than nmax elements, then the character string will end with "and XX others" with XX the number of remaining items. Use this argument to change what is between the and and the XX. E.g. if other = "any of", then you would get "... and any of 15 others" instead of "... and 15 others".
nmax	Integer, default is 7. If x contains more than nmax items, then these items are grouped into an "other" group.

Value

It returns a character string of lentgh one.

The argument type

The argument type is a "super argument". When provided, it supersedes all other arguments. It offers a compact way to give the arguments to the function.

Its sytax is as follows: "arg1.arg2.arg2", where argX is an argument code. The codes are "s", "past", "or", "start", "quote", "enum" – they refer to the function arguments. If you want to add a verb, since it can have a free-form, it is deduced as the argument not equal to the previous codes. For example, if you have type = "s.contain", this is identical to calling the function with s = TRUE and verb = "contain".

A note on enum. The argument enum can be equal to "i", "I", "a", "A" or "1". When you include it in type, by default "i" is used. If you want another one, add it in the code. For example type = "is.enum a.past" is identical to calling the function with verb = "is", past = TRUE and enum = "a".

Author(s)

Laurent Berge

Examples

```
# Let's say you write an error/information message to the user
# I just use the "type" argument but you can obtain the
#  same results by using regular arguments

x = c("x1", "height", "width")
message("The variable", enumerate_items(x, "s.is"), " not in the data set.")
# Now just the first item
message("The variable", enumerate_items(x[1], "s.is"), " not in the data set.")

# Past
```

```
message("The variable", enumerate_items(x, "s.is.past"), " not found.")
message("The variable", enumerate_items(x[1], "s.is.past"), " not found.")

# Verb first
message("The problematic variable", enumerate_items(x, "s.is.start.quote"), ".")
message("The problematic variable", enumerate_items(x[1], "s.is.start.quote"), ".")

# covid times
todo = c("wash your hands", "stay home", "code")
message("You should: ", enumerate_items(todo[c(1, 1, 2, 3)], "enum 1"), "!")
message("You should: ", enumerate_items(todo, "enum.or"), "?")
```

`fit_screen` *Nicely fits a message in the current R console*

Description

Utility to display long messages with nice formatting. This function cuts the message to fit the current screen width of the R console. Words are never cut in the middle.

Usage

```
fit_screen(msg, width = NULL, leading_ws = TRUE, leader = "")
```

Arguments

`msg`	Text message: character vector.
`width`	A number between 0 and 1, or an integer. The maximum width of the screen the message should take. Numbers between 0 and 1 represent a fraction of the screen. You can also refer to the screen width with the special variable `.sw`. Integers represent the number of characters and cannot be lower than 15. Default is `min(120, 0.95*.sw)` (the min between 120 characters and 90% of the screen width).
`leading_ws`	Logical, default is `TRUE`. Whether to keep the leading white spaces when the line is cut.
`leader`	Character scalar, default is the empty string. If provided, this value will be placed in front of every line.

Details

This function does not handle tabulations.

Value

It returns a single character vector with line breaks at the appropriate width.

Examples

```
# A long message of two lines with a few leading spaces
msg = enumerate_items(state.name, nmax = Inf)
msg = paste0("     ", gsub("Michigan, ", "\n", msg))

# by default the message takes 95% of the screen
cat(fit_screen(msg))

# Now we reduce it to 50%
cat(fit_screen(msg, 0.5))

# we add leading_ws = FALSE to avoid the continuation of leading WS
cat(fit_screen(msg, 0.5, FALSE))

# We add "#> " in front of each line
cat(fit_screen(msg, 0.5, leader = "#> "))
```

`fsignif` *Formatting numbers with display of significant digits*

Description

Formatting of numbers, when they are to appear in messages. Displays only significant digits in a "nice way" and adds commas to separate thousands. It does much less than the `format` function, but also a bit more though.

Usage

```
fsignif(x, s = 2, r = 0, commas = TRUE)

signif_plus
```

Arguments

`x`	A numeric vector.
`s`	The number of significant digits to be displayed. Defaults to 2. All digits not in the decimal are always shown.
`r`	For large values, the number of digits after the decimals to be displayed (beyond the number of significant digits). Defaults to 0. It is useful to suggest that a number is not an integer.
`commas`	Whether or not to add commas to separate thousands. Defaults to `TRUE`.

Format

An object of class `function` of length 1.

Value

It returns a character vector of the same length as the input.

Examples

```
x = rnorm(1e5)
x[sample(1e5, 1e4, TRUE)] = NA

# Dumb function telling the number of NA values
tell_na = function(x) message("x contains ", fsignif(sum(is.na(x))), " NA values.")

tell_na(x)

# Some differences with signif:
show_diff = function(x, d = 2) cat("signif(x, ", d, ") -> ", signif(x, d),
                                   " vs fsignif(x, ", d, ") -> ",
                                   fsignif(x, d), "\n", sep = "")

# Main difference is for large numbers
show_diff(95123.125)
show_diff(95123.125, 7)

# Identical for small numbers
show_diff(pi / 500)
```

---

generate_set_hook *Error displaying a call located at a hook location*

Description

When devising complex functions, errors or warnings can be deeply nested in internal function calls while the user-relevant call is way up the stack. In such cases, these "hook" functions facilitate the creation of error/warnings informative for the user.

Usage

```
generate_set_hook(namespace)

generate_stop_hook(namespace)

generate_warn_hook(namespace)

set_hook()

generate_get_hook(namespace)
```

```
stop_hook(..., msg = NULL, envir = parent.frame(), verbatim = FALSE)

warn_hook(..., envir = parent.frame(), immediate. = FALSE, verbatim = FALSE)
```

Arguments

namespace	Character scalar giving the namespace for which the hooks are valid. Only useful when hook functions are used in a package.
...	Objects that will be coerced to character and will compose the error message.
msg	A character vector, default is NULL. If provided, this message will be displayed right under the error message. This is mostly useful when the text contains formatting because the function stop used to send the error message erases any formatting.
envir	An environment, default is parent.frame(). Only relevant if the error/warning message contains interpolation (interpolation is performed with stringmagic). It tells where the variables to be interpolated should be found. In general you should not worry about this argument.
verbatim	Logical scalar, default is FALSE. By default the error/warning message allows variable interpolation with stringmagic. To disable interpolation, use verbatim = TRUE.
immediate.	Whether the warning message should be prompted directly. Defaults to FALSE.

Details

These functions are useful when developing complex functions relying on nested internal functions. It is important for the user to know where the errors/warnings come from for quick debugging. This "_hook" family of functions write the call of the user-level function even if the errors happen at the level of the internal functions.

If you need these functions within a package, you need to generate the set_hook, stop_hook and warn_hook functions so that they set, and look up for, hooks speficic to your function. This ensures that if other functions outside your package also use hooks, there will be no conflict. The only thing to do is to write this somewhere in the package files:

```
set_hook = generate_set_hook("pkg_name")
stop_hook = generate_stop_hook("pkg_name")
warn_hook = generate_warn_hook("pkg_name")
```

Functions

- generate_set_hook(): Generates a package specific set_hook function
- generate_stop_hook(): Generates a package specific stop_hook function
- generate_warn_hook(): Generates a package specific warn_hook function
- set_hook(): Marks the function as the hook
- generate_get_hook(): Generates the function giving the number of frames we need to go up the call stack to find the hooked function
- warn_hook(): Warning with a call located at a hook location

Author(s)

Laurent Berge

See Also

Regular stop functions with interpolation: `stop_up()`. Regular argument checking with `check_arg()` and `check_set_arg()`.

Examples

```
# The example needs to be complex since it's about nested functions, sorry
# Let's say you have an internal function that is dispatched into several
# user-level functions

my_mean = function(x, drop_na = FALSE){
  set_hook()
  my_mean_internal(x = x, drop_na = drop_na)
}

my_mean_skip_na = function(x){
  set_hook()
  my_mean_internal(x = x, drop_na = TRUE)
}

my_mean_internal = function(x, drop_na){
  # simple check
  if(!is.numeric(x)){
    # note that we use string interpolation with stringmagic.
   stop_hook("The argument `x` must be numeric. PROBLEM: it is of class {enum.bq ? class(x)}.")
  }

  if(drop_na){
    return(mean(x, na.rm = TRUE))
  } else {
    return(mean(x, na.rm = FALSE))
  }
}

# Let's run the function with a wrong argument
x = "five"
try(my_mean(x))

# => the error message reports that the error comes from my_mean
#    and *not* my_mean_internal
```

`ifsingle` *Conditional element selection*

Description

Tiny functions shorter, and hopefully more explicit, than `ifelse`.

Usage

```
ifsingle(x, yes, no)

ifunit(x, yes, no)
```

Arguments

`x`	A vector (`ifsingle`) or a numeric of length 1 (`ifunit`).
`yes`	Something of length 1. Result if the condition is fulfilled.
`no`	Something of length 1. Result if the condition is not fulfilled.

Details

Yes, `ifunit` is identical to `ifelse(test == 1, yes, no)`. And regarding `ifsingle`, it is identical to `ifelse(length(test) == 1, yes, no)`.

Why writing these functions then? Actually, I've found that they make the code more explicit, and this helps!

Value

Returns something of length 1.

Functions

- `ifunit()`: Conditional element selection depending on whether `x` is equal to unity or not.

Author(s)

Laurent Berge

Examples

```
# Let's create an error message when NAs are present
my_crossprod = function(mat){
 if(anyNA(mat)){
   row_na = which(rowSums(is.na(mat)) > 0)
   n_na = length(row_na)
   stop("In argument 'mat': ", n_letter(n_na), " row", plural(n_na, "s.contain"),
        " NA values (", ifelse(n_na<=3, "", "e.g. "), "row",
        enumerate_items(head(row_na, 3), "s"), ").
```

```
        Please remove ", ifunit(n_na, "it", "them"), " first.")
 }
 crossprod(mat)
}

mat = matrix(rnorm(30), 10, 3)
mat4 = mat1 = mat
mat4[c(1, 7, 13, 28)] = NA
mat1[7] = NA

# Error raised because of NA: informative (and nice) messages
try(my_crossprod(mat4))
try(my_crossprod(mat1))
```

`n_times` *Numbers in letters*

Description

Set of (tiny) functions that convert integers into words.

Usage

```
n_times(n)

n_th(n)

n_letter(n)
```

Arguments

`n`	An integer vector.

Value

It returns a character vector of length one.

Functions

- `n_th()`: Transforms the integer `n` to `nth` appropiately.
- `n_letter()`: Transforms small integers to words.

Author(s)

Laurent Berge

Examples

```
find = function(v, x){
  if(x %in% v){
    message("The number ", n_letter(x), " appears ", n_times(sum(v == x)),
            ", the first occurrence is the ", n_th(which(v==x)[1]), " element.")
  } else message("The number ", n_letter(x), " was not found.")
}

v = sample(100, 500, TRUE)
find(v, 6)
```

---

package_stats *Provides package statistics*

Description

Summary statistics of a packages: number of lines, number of functions, etc...

Usage

```
package_stats()
```

Details

This function looks for files in the `R/` and `src/` folders and gives some stats. If there is no `R/` folder directly accessible from the working directory, there will be no stats displayed.

Why this function? Well, it's just some goodies for package developers trying to be user-friendly!

The number of documentation lines (and number of words) corresponds to the number of non-empty roxygen documentation lines. So if you don't document your code with roxygen, well, this stat won't prompt.

Code lines correspond to non-commented, non-empty lines (by non empty: at least one letter must appear).

Comment lines are non-empty comments.

Value

Doesn't return anything, just a prompt in the console.

Examples

```
package_stats()
```

plural *Adds an s and/or a singular/plural verb depending on the argument's length*

Description

Utilities to write user-level messages. These functions add an 's' or a verb at the appropriate form depending on whether the argument is equal to unity (`plural`) or of length one (`plural_len`).

Usage

```
plural(x, type, s, verb = FALSE, past = FALSE)

plural_len(x, type, s, verb = FALSE, past = FALSE)
```

Arguments

`x`	An integer of length one (`plural`) or a vector `plural_len`.
`type`	Character string, default is missing. If `type = "s.is.past"` it means that an "s" will be added if `x` is greater than 1 (or of length greater than one for `plural_len`); it will be followed by the verb "to be" in past tense in singular or plural form depending on `x`. This argument must be made of keywords separated by points without space, the keywords are "s", "past" and a verb (i.e. any thing different than "s" and "past"). Missing keywords mean their value is equal to `FALSE`.
`s`	Logical, used only if the argument type is missing. Whether to add an "s" if the form of `x` is plural. Default is missing: equals to `TRUE` if no other argument is provided, `FALSE` otherwise.
`verb`	Character string or `FALSE`, used only if the argument type is missing. The verb to be inserted in singular or plural depending on the value of `x`. default is `FALSE`.
`past`	Logical, used only if the argument type is missing. Whether the verb should be in past tense. Default is `FALSE`.

Value

Returns a character string of length one.

Functions

- `plural_len()`: Adds an s and conjugate a verb depending on the length of `x`

Author(s)

Laurent Berge

Examples

```
# Let's create an error message when NAs are present
my_crossprod = function(mat){
 if(anyNA(mat)){
   row_na = which(rowSums(is.na(mat)) > 0)
   n_na = length(row_na)
   stop("In argument 'mat': ", n_letter(n_na), " row", plural(n_na, "s.contain"),
        " NA values (", ifelse(n_na<=3, "", "e.g. "), "row",
        enumerate_items(head(row_na, 3), "s"),
        "). Please remove ", ifunit(n_na, "it", "them"), " first.")
 }
 crossprod(mat)
}

mat = matrix(rnorm(30), 10, 3)
mat4 = mat1 = mat
mat4[c(1, 7, 13, 28)] = NA
mat1[7] = NA

# Error raised because of NA: informative (and nice) messages
try(my_crossprod(mat4))
try(my_crossprod(mat1))
```

`setDreamerr_check` *Sets dreamerr argument checking functions on or off*

Description

This function allows to disable, or re-enable, all calls to `check_arg` within any function. Useful only when running (very) large loops (>100K iter.) over small functions that use dreamerr's `check_arg`.

Usage

```
setDreamerr_check(check = TRUE)
```

Arguments

`check`	Strict logical: either `TRUE` of `FALSE`. Default is `TRUE`.

Author(s)

Laurent Berge

Examples

```
# Let's create a small function that returns the argument
#  if it is a single character string, and throws an error
#  otherwise:

test = function(x){
  check_arg(x, "scalar character")
  x
}

# works:
test("hey")
# error:
try(test(55))

# Now we disable argument checking
setDreamerr_check(FALSE)
# works (although it shouldn't!):
test(55)

# re-setting argument checking on:
setDreamerr_check(TRUE)
```

`setDreamerr_dev.mode` *Sets the developer mode to help form check_arg calls*

Description

Turns on/off a full fledged checking of calls to `check_arg`. If on, it enables the developer mode which checks extensively calls to check_arg, allowing to find any problem. If a problem is found, it is pinpointed and the associated help is referred to.

Usage

```
setDreamerr_dev.mode(dev.mode = FALSE)
```

Arguments

`dev.mode`	A logical, default is `FALSE`.

Details

Since this mode ensures a detailed cheking of all `check_arg` calls, it is thus a strain on performance and should be always turned off otherwise needed.

Author(s)

Laurent Berge

See Also

check_arg

Examples

```
# If you're new to check_arg, given the many types available,
# it's very common to make mistakes when creating check_arg calls.
# The developer mode ensures that any problematic call is spotted
# and the problem is clearly stated
#
# Note that since this mode ensures a detailed cheking of the call
# it is thus a strain on performance and should be always turned off
# otherwise needed.
#

# Setting the developer mode on:
setDreamerr_dev.mode(TRUE)

# Creating some 'wrong' calls => the problem is pinpointed

test = function(x) check_arg(x, "integer scalar", "numeric vector")
try(test())

test = function(...) check_arg("numeric vector", ...)
try(test())

test = function(x) check_arg(x$a, "numeric vector")
try(test())

test = function(x) check_arg(x, "numeric vector integer")
try(test())

test = function(x) check_arg(x, "vector len(,)")
try(test())

# etc...

# Setting the developer mode off:
setDreamerr_dev.mode(FALSE)
```

---

setDreamerr_show_stack

Settings telling whether or not to display the full call stack on errors

---

Description

Errors generated with dreamerr functions only shows the call to which the error should point to. If setDreamerr_show_stack is set to TRUE, error will display the full call stack instead.

Usage

```
setDreamerr_show_stack(show_full_stack = FALSE)
```

Arguments

show_full_stack
: Logical scalar, default is FALSE. If TRUE, then errors generated by dreamerr functions (like stop_up()/stopi()) will display the full call stack.

Author(s)

Laurent Berge

Examples

```
# Let's create a toy example of a function relying on an internal function
# for the heavy lifting (although here it's pretty light!)
make_sum = function(a, b){
  make_sum_internal(a, b)
}

make_sum_internal = function(a, b){
  if(!is.numeric(a)) stop_up("arg. 'a' must be numeric!")
  a + b
}

# By default if you feed stg non numeric, the call shown is
# make_sum, and not make_sum_internal, since the user could not
# care less of the internal structure of your functions

try(make_sum("five", 55))

# Now with setDreamerr_show_stack(TRUE), you would get the full call stack
setDreamerr_show_stack(TRUE)
try(make_sum("five", 55))
```

---

set_check *Sets argument checking on/off "semi-globally"*

Description

You can allow your users to turn off argument checking within your function by using set_check. Only the functions check_arg nd check_value can be turned off that way.

Usage

```
set_check(x)
```

Arguments

x	A logical scalar, no default.

Details

This function can be useful if you develop a function that may be used in large range loops (>100K). In such situations, it may be good to still check all arguments, but to offer the user to turn this checking off with an extra argument (named `arg.check` for instance). Doing so you would achieve the feat of i) having a user-friendly function thanks to argument checking and, ii) still achieve high performance in large loops (although the computational footprint of argument checking is quite low (around 30 micro seconds for missing arguments to 80 micro seconds for non-missing arguments of simple type)).

Examples

```
# Let's give an example
test_check = function(x, y, arg.check = TRUE){
  set_check(arg.check)
  check_arg(x, y, "numeric scalar")
  x + y
}

# Works: argument checking on
test_check(1, 2)

# If mistake, nice error msg
try(test_check(1, "a"))

# Now argument checking turned off
test_check(1, 2, FALSE)
# But if mistake: "not nice" error message
try(test_check(1, "a", FALSE))
```

---

set_up *Sets "semi-globally" the 'up' argument of dreamerr's functions*

Description

When `check_arg` (or `stop_up`) is used in non user-level functions, the argument `.up` is used to provide an appropriate error message referencing the right function.

Usage

```
set_up(.up = 1)
```

Arguments

.up	An integer greater or equal to 0.

Details

To avoid repeating the argument .up in each check_arg call, you can set it (kind of) "globally" with set_up.

The function set_up does not set the argument up globally, but only for all calls to check_arg and check_value within the same function.

Examples

```
# Example with computation being made within a non user-level function

sum_fun = function(x, y){
  my_internal(x, y, sum = TRUE)
}

diff_fun = function(x, y){
  my_internal(x, y, sum = FALSE)
}

my_internal = function(x, y, sum){
  set_up(1) # => errors will be at the user-level function
  check_arg(x, y, "numeric scalar mbt")

  # Identical to calling
  # check_arg(x, y, "numeric scalar mbt", .up = 1)

  if(sum) return(x + y)
  return(x - y)
}

# we check it works
sum_fun(5, 6)
diff_fun(5, 6)

# Let's throw some errors
try(sum_fun(5))
try(sum_fun(5, 1:5))
```

---

sfill *Fills a string vector with a symbol*

Description

Fills a string vector with a user-provided symbol, up to the required length.

Usage

```
sfill(x = "", n = NULL, symbol = " ", right = FALSE, anchor, na = "NA")
```

Arguments

x	A character vector.
n	A positive integer giving the total expected length of each character string. Can be NULL (default). If NULL, then n is set to the maximum number of characters in x (i.e. max(nchar(x))).
symbol	Character scalar, default to " ". The symbol used to fill.
right	Logical, default is FALSE. Whether the character vector should be filled on the left(default) or on the right.
anchor	Character scalar, can be missing. If provided, the filling is done up to this anchor. See examples.
na	Character that will replace any NA value in input. Default is "NA".

Value

Returns a character vector of the same length as x.

Examples

```
# Some self-explaining examples
x = c("hello", "I", "am", "No-one")
cat(sep = "\n", sfill(x))
cat(sep = "\n", sfill(x, symbol = "."))
cat(sep = "\n", sfill(x, symbol = ".", n = 15))
cat(sep = "\n", sfill(x, symbol = ".", right = TRUE))

cat(sep = "\n", paste(sfill(x, symbol = ".", right = TRUE), ":", 1:4))

# Argument 'anchor' can be useful when using numeric vectors
x = c(-15.5, 1253, 32.52, 665.542)
cat(sep = "\n", sfill(x))
cat(sep = "\n", sfill(x, anchor = "."))
```

---

stop_up *Stops (or warns in) sub-function execution*

Description

Useful if you employ non-user level sub-functions within user-level functions or if you want string interpolation in error messages. When an error is thrown in the sub function, the error message will integrate the call of the user-level function, which is more informative and appropriate for the user. It offers a similar functionality for warning.

Usage

```
stop_up(..., up = 1, msg = NULL, envir = parent.frame(), verbatim = FALSE)

stopi(..., envir = parent.frame())

warni(..., envir = parent.frame(), immediate. = FALSE)

warn_up(
  ...,
  up = 1,
  immediate. = FALSE,
  envir = parent.frame(),
  verbatim = FALSE
)
```

Arguments

`...`	Objects that will be coerced to character and will compose the error message.
`up`	The number of frames up, default is 1. The call in the error message will be based on the function up frames up the stack. See examples. If you have many calls to `stop_up`/`warn_up` with a value of `up` different than one, you can use `set_up` to change the default value of `up` within the function.
`msg`	A character vector, default is `NULL`. If provided, this message will be displayed right under the error message. This is mostly useful when the text contains formatting because the function `stop` used to send the error message erases any formatting.
`envir`	An environment, default is `parent.frame()`. Only relevant if the error/warning message contains interpolation (interpolation is performed with stringmagic). It tells where the variables to be interpolated should be found. In general you should not worry about this argument.
`verbatim`	Logical scalar, default is `FALSE`. By default the error/warning message allows variable interpolation with stringmagic. To disable interpolation, use `verbatim = TRUE`.
`immediate.`	Whether the warning message should be prompted directly. Defaults to `FALSE`.

Details

These functions are really made for package developers to facilitate the good practice of providing informative user-level error/warning messages.

The error/warning messages allow variable interpolation by making use of stringmagic's interpolation.

Functions

- `stopi()`: Error messages with string interpolation
- `warni()`: Warnings with string interpolation
- `warn_up()`: Warnings at the level of user-level functions

Author(s)

Laurent Berge

See Also

For general argument checking, see check_arg() and check_set_arg().

Examples

```
# We create a main user-level function
# The computation is done by an internal function
# Here we compare stop_up with a regular stop

main_function = function(x = 1, y = 2){
  my_internal_function(x, y)
}

my_internal_function = function(x, y){
  if(!is.numeric(x)){
    stop_up("Argument 'x' must be numeric but currently isn't.")
  }

  # Now regular stop
  if(!is.numeric(y)){
    stop("Argument 'y' must be numeric but currently isn't.")
  }

  nx = length(x)
  ny = length(y)
  if(nx != ny){
    # Note that we use string interpolation with {}
    warn_up("The lengths of x and y don't match: {nx} vs {ny}.")
  }

  x + y
}

# Let's compare the two error messages
# stop_up:
try(main_function(x = "a"))
# => the user understands that the problem is with x

# Now compare with the regular stop:
try(main_function(y = "a"))
# Since the user has no clue of what my_internal_function is,
#  s/he will be puzzled of what to do to sort this out

# Same with the warning => much clearer with warn_up
main_function(1, 1:2)
```

suggest_item *Suggest the the closest elements from a string vector*

Description

Compares a character scalar to the elements from a character vector and returns the elements that are the closest to the input.

Usage

```
suggest_item(
  x,
  items,
  msg.write = FALSE,
  msg.newline = TRUE,
  msg.item = "variable"
)
```

Arguments

x	Character scalar, must be provided. This reference will be compared to the elements of the string vector in the argument items.
items	Character vector, must be provided. Elements to which the value in argument x will be compared.
msg.write	Logical scalar, default is FALSE. If TRUE, a message is returned, equal to "Maybe you meant {enum.bq.or ? matches}?" (see stringmagic for information on the interpolation) if there were matches. If no matches were found, the message is "FYI the {msg.item}{$s, are, enum.bq ? items}.".
msg.newline	Logical scalar, default is TRUE. Only used if msg.write = TRUE. Whether to add a new line just before the message.
msg.item	Character scalar, default is "variable". Only used if msg.write = TRUE. What does the items represent?

Details

This function is useful when used internally to guide the user to relevant choices.

The choices to which the user is guided are in decreasing quality. First light mispells are checked. Then more important mispells. Finally very important mispells. Completely off potential matches are not reported.

If the argument msg.write is TRUE, then a character scalar is returned containing a message suggesting the matches.

Value

It returns a vector of matches. If no matches were found

Author(s)

Laurent Berge

Examples

```
# function reporting the sum of a variable
sum_var = function(data, var){
  # var: a variable name, should be part of data
  if(!var %in% names(data)){
    suggestion = suggest_item(var, names(data), msg.write = TRUE)
    stopi("The variable `{var}` is not in the data set. {suggestion}")
  }

  sum(data[[var]])
}

# The error message guides us to a suggestion
try(sum_var(iris, "Petal.Le"))
```

---

validate_dots *Checks the arguments in dots from methods*

Description

This function informs the user of arguments passed to a method but which are not used by the method.

Usage

```
validate_dots(
  valid_args = c(),
  suggest_args = c(),
  message,
  warn,
  stop,
  call. = FALSE,
  immediate. = TRUE
)
```

Arguments

valid_args	A character vector, default is missing. Arguments that are not in the definition of the function but which are considered as valid. Typically internal arguments that should not be directly accessed by the user.
suggest_args	A character vector, default is missing. If the user provides invalid arguments, he might not be aware of the main arguments of the function. Use this argument to inform the user of these main arguments.

message	Logical, default is FALSE. If TRUE, a standard message is prompted to the user (instead of a warning).
warn	Logical, default is TRUE. If TRUE, when the user provides invalid arguments, the function will call warning (default). If FALSE (and so are the other arguments stop and message), then no message is prompted to the user, rather it is the only output of the function.
stop	Logical, default is FALSE. If TRUE, when the user provides invalid arguments, the function will call stop instead of prompting a warning (default).
call.	Logical, default is FALSE. If TRUE, when the user provides invalid arguments, then the message will also contain the call to the initial function (by default, only the function name is shown).
immediate.	Logical, default is FALSE. Can be only used with the argument warn = TRUE: whether the warning is immediately displayed or not.

Value

This function returns the message to be displayed. If no message is to be displayed because all the arguments are valid, then NULL is returned.

Examples

```
# The typical use of this function is within methods

# Let's create a 'my_class' object and a summary method
my_obj = list()
class(my_obj) = "my_class"

# In the summary method, we add validate_dots
# to inform the user of invalid arguments

summary.my_class = function(object, arg_one, arg_two, ...){

  validate_dots()
  # CODE of summary.my_class
  invisible(NULL)
}

# Now let's test it, we add invalid arguments
summary(my_obj, wrong = 3)
summary(my_obj, wrong = 3, info = 5)

# Now let's :
#   i) inform the user that argument arg_one is the main argument
#  ii) consider 'info' as a valid argument (but not shown to the user)
# iii) show a message instead of a warning

summary.my_class = function(object, arg_one, arg_two, ...){

  validate_dots(valid_args = "info", suggest_args = "arg_one", message = TRUE)
  # CODE of summary.my_class
```

```
  invisible(NULL)
}

# Let's retest it
summary(my_obj, wrong = 3) # not OK => suggestions
summary(my_obj, info = 5)  # OK
```

Index