Package: saferDev (via r-universe)

October 31, 2026

Type Package

Title Checking Tools for Safer R Function Development

Version 3.0.0

Date 2026-10-23

Encoding UTF-8

Maintainer Gael Millot <gael.millot@pasteur.fr>

Description Set of functions that check and correct the code of R functions according to the safer-r project recommendations (see <https://github.com/safer-r>): argument value checking, explicit package namespace qualification of the function calls, complete writing of the arguments of the called functions, curly brackets around the bodies of if, else, for, while, repeat and function, TRUE and FALSE written in full, controlled for() loop sequences and standard function() writing of lambda functions. The corrected function is returned or written into a file. Other functions check the R scope, catch messages, check the presence of packages and functions, and write log files.

URL https://github.com/safer-r/saferDev, https://safer-r.github.io/saferDev/

BugReports https://github.com/safer-r/saferDev/issues

License GPL-3

Depends R (>= 4.6.0)

Config/roxygen2/version 8.0.0

Imports ggplot2

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Repository https://safer-r.r-universe.dev

Date/Publication 2026-10-31 01:23:28 UTC

RemoteUrl https://github.com/safer-r/saferdev

RemoteRef HEAD

RemoteSha 938940c16187ec23a5232883effbbd7feefbae38

Contents

all_args_here 2
arg_check 7
colons_check 11
curly_bracket_check 15
env_check 17
for_loop_nb_check 19
get_message 22
is_function_here 24
is_package_here 26
lambda_check 27
report 30
super_assign_check 32
true_false_check 34

Index **37**

`all_args_here` *All Arguments Here*

Description

Verify that all the functions used inside a function are written with all their arguments. For instance: `base::paste0(letters[1:2], collapse = NULL, recycle0 = FALSE)` and not `paste0(letters[1:2])`.

Usage

```
all_args_here(
  x,
  table_export = FALSE,
  fun_export = FALSE,
  out_path = ".",
  df_name = "res.tsv",
  overwrite = FALSE,
  safer_check = TRUE,
  lib_path = NULL,
  error_text = ""
)
```

Arguments

`x`	Function name, written without quotes and brackets.
`table_export`	Single logical value. Export the result data frame into a `.tsv` file? If `TRUE`, the data frame is not returned by the function but only exported.

`fun_export` Single logical value. Export the modified function into a `.R` file? If `TRUE`, the corrected code (missing arguments added into the detected function calls, as proposed in the `$STATUS` column) is not returned by the function but only exported into a file made of the name of the submitted function with `_safer` added at the end (e.g., `test_safer.R`). The file contains an assignment of the modified function: `test_safer <- function(...) ...`, so that `base::source()`ing it recreates the modified function. An error is returned if the file already exists in `out_path` and `overwrite` is `FALSE`, and if the tested function is not designated by a function name (like `test` or `package::test`), i.e., written by an inline definition (like `function(x){x}`) or by a component writing (like `FUN_LIST$fun`). Ignored if no correction has been applied.

`out_path` Single character string indicating the absolute pathway of the folder where to export the data frame or the modified function. `out_path = "."` (default) means the R working directory set by the user. Ignored if `table_export` and `fun_export` are `FALSE`.

`df_name` Single character string indicating the name of the exported data frame file. Ignored if `table_export` is `FALSE`.

`overwrite` Single logical value. Overwrite potential files already existing in `out_path` (`df_name` for the table export, and the function name with `_safer` added at the end for the `fun_export`)? Ignored if `table_export` and `fun_export` are `FALSE`.

`safer_check` Single logical value. Perform the "safer" checks? If `TRUE`, checkings are performed before main code running (see the safer-r project): 1) required functions and related packages effectively present in local R libraries and 2) R classical operators (like `"<-"`) not overwritten by another package because of the R scope. Warning: must be set to `FALSE` if this function is used inside another "safer" function to avoid pointless multiple checkings.

`lib_path` Vector of characters specifying the absolute pathways of the directories containing the required packages for the function, if not in the default directories. Useful when R packages are not installed in the default directories because of lack of admin rights. More precisely, `lib_path` is passed through the `new` argument of `.libPaths()` so that the new library paths are `c(lib_path, .libPaths())`. Warning: `.libPaths()` is restored to the initial paths, after function execution. Ignored if `NULL` (default): only the pathways specified by the current `.libPaths()` are used for package calling.

`error_text` Single character string used to add information in error messages returned by the function, notably if the function is inside other functions, which is practical for debugging. Example: `error_text = " INSIDE <PACKAGE_1>::<FUNCTION_1> INSIDE <PACKAGE_2>::<FUNCTION_2>."`. If `NULL`, converted into `""`.

Details

More precisely, `all_args_here()` verifies that all the strings before an opening bracket `(` are written with all their arguments. Thus, it cannot check function names written without brackets, like in the `FUN` argument of some functions, e.g., `sapply(1:3, FUN = as.character)`.

The Perl regex used to detect a function name is: `"([a-zA-Z]|\.[a-zA-Z._])[a-zA-Z0-9._]*\s*\("`.

Currently, `all_args_here()` cannot detect functions written between quotes, like `"+"()` or `"rownames<-"(x, "a")`.

Function names preceded by `$` are not considered.

The following R functions are skipped: `function`, `if`, `for`, `while`, `repeat` and `else`.

Empty argument values (see the safer-r rules): 1) a default value that cannot be copied into the call is proposed with an empty value, meaning that the default value is used but cannot be fixed. This is the case when the default value depends on another argument (e.g., `envir = as.environment(pos)` in `utils::getFromNamespace()`, proposed as `utils::getFromNamespace(x, ns, pos = -1, envir = )`), on a variable of the internal environment of the function, or on the frame where it is evaluated (e.g., `parent.frame()`), because such a default value would be evaluated in the environment of the tested function and not in the one of the called function. 2) An argument with no default value and not written is also proposed with an empty value, because absence of value can mean something (e.g., `base::.libPaths(new = , include.site = TRUE)` returns the paths of the R libraries, whereas `base::.libPaths(new = base::.libPaths(), include.site = TRUE)` sets them). In both cases, the behavior of the code is not changed. An argument already written with an empty value (like `envir = `) is accepted as is when it has no default value or a default value that cannot be copied, and is not listed in the message above. If its default value can be copied (e.g., `safer_check = `), the default value is added (`safer_check = TRUE`). Of note, only the arguments before `...` can be written with a partial name, as in R. It is recommended to test (using `testthat`) that these default values are unchanged, for instance `testthat::expect_identical(object = base::formals(fun = utils::getFromNamespace)$envir, expected = base::quote(expr = as.environment(pos)))`.

Most of the time, `all_args_here()` does not check inside comments, but some unexpected writing could dupe `all_args_here()`. Please, report here if it is the case.

The returned line numbers are indicative, depending on which source is checked. For instance, `saferDev::report` (compiled) has not the same line numbers as its source file. Notably, compiled functions do not have comments anymore, compared to the same source function sourced into the working environment. In addition, the counting starts at the `"<- function"` line, i.e., without counting the `#'` header lines potentially present in source files.

The function works first by replacing in the code: 1) the three consecutive characters `'"'` or `'"'` by three spaces, 2) `"\"'"` and `'\'"'` by four spaces and 3) escape quotes like `\"` or `\'` by two spaces.

Pre-check: the submitted function is first checked by `lambda_check()`: if some lambda functions are written with the `\(x)` shorthand, an error is returned asking to run `saferDev::lambda_check()` first, because the `\` writings are not managed by the text-based detection of `all_args_here()`. As in `lambda_check()`, the `\(x)` writing is only detectable in a function carrying its source references (attribute `srcref`): a compiled or loaded package function prints `function(x)`, so that the pre-check silently passes in that case.

Known limitation: a string literal containing an escaped backslash-quote, like `"'\'"`, desynchronizes the in-quotes detection of `all_args_here()`, which can return an internal error (see the `dev/repro_all_args_here_internal_error_3.R` script of the package repository). Please, report here if it is the case.

Line breaks, comments and indentation of the modified function (`fun_export` argument and returned function): the `$STATUS` proposals are not pasted in place of the calls. Instead, the missing arguments are inserted into the printed code of the submitted function, at the positions given by the R parser: a missing argument whose value is written without its name (e.g., `round(v)`) gets its name inserted before this value (`round(x = v, digits = 0)`), and the other missing arguments are inserted after the last argument of the call. Thus, the line breaks, the indentation, the comments

and the strings of the submitted function are kept, including inside calls written on several lines, and nested calls (e.g., `paste0(mean(1:2))`) are all corrected. The comments and the original indentation require that the submitted function carries its source references (attribute `srcref`, i.e., `options(keep.source = TRUE)` when the function was created, the default in interactive sessions). Otherwise (e.g., a function of an installed package, or a function created in an `Rscript` session), the printed code is the deparsed code, without comments and with a standard indentation, but its line breaks are kept.

Corrections not applied: the call is left unchanged in the modified function and its proposal is listed in the `"CORRECTIONS NOT APPLIED"` message, with one of the following reasons: 1) the `$MISSING_ARGS` or `$DEF_ARGS` column of the call cannot be read; 2) the call cannot be located in the code of the function (e.g., call with an empty argument, like `c(1, )`); 3) the corrected call does not parse as R code; 4) an argument is written positionally and is also proposed with its name (e.g., `sum(x, na.rm)` proposed as `sum(x, na.rm, na.rm = FALSE)`), which would fail at call time; 5) the corrected call does not match the `$STATUS` proposal (compared whatever the order of the named arguments, the spaces, the line breaks and the comments). Of note, an empty argument value in a proposal (like `envir =` , see above) is never a reason: such a proposal is applied.

See more examples here.

Warnings:

1. The following R functions are also skipped (as indicated by `"SKIPPED"` in the `DEF_ARGS` column of the returned data frame): `as.environment()` (see the R bug report 18849 in the R bug tracker).
2. Some functions, like `rownames()`, have different arguments depending on whether something is assigned to it (e.g., `rownames(x) <- "a"`) or not. The `all_args_here()` function always proposes the arguments defined in the help page without assignment, meaning that it cannot detect the assignment. A way to bypass this is to use the "exact" writing of the function. For instance, `base::"rownames<-"(x, "a")` instead of using `base::rownames(x) <- "a"`, and use `getAnywhere("rownames<-")` to see the arguments of the function.
3. The function could not properly work if any comma is present in default argument values. Please, report here if it is the case.
4. Proposals in the `STATUS` column are only suggestions, as it is difficult to anticipate all the exceptions with arguments writing.

Value

A message saying whether arguments are missing or everything seems fine, and a data frame indicating the missing arguments (see the column descriptions below). If `table_export` argument is `FALSE`, the data frame is returned as a message. If `TRUE`, it is exported as `df_name` in `out_path`. If `fun_export` argument is `FALSE`, the modified function (an R function object rebuilt from the corrected code, with its comments and indentation preserved if the submitted function carries its source references, see the details section) is returned, or `NULL` (invisible) if no modification has been applied. If `TRUE`, the modified function is exported as `<FUNCTION_NAME>_safer.R` in `out_path` and `NULL` is returned (invisible), or nothing is exported if no modification has been applied. A warning and `NULL` (invisible) are returned if the submitted function is empty of function, or only made of comments, or only made of `function`, `if`, `for`, `while`, `repeat`, `else`. Column descriptions:

- `$LINE_NB`: the line number in the function code (starting at the `"<- function"` line, i.e., without counting the `#'` header lines).

- $FUN_NAME: the function name.
- $FUN_ARGS: the written arguments of FUN_NAME. "NOT_CONSIDERED" means that the function is between quotes or after $.
- $FUN_POS: the position of the first character of the function name in the $LINE_NB line.
- $DEF_ARGS: the defaults arguments of FUN_NAME. "NO_ARGS" means that the function has no arguments. "INTERNAL_FUNCTION" means that the function has been created inside the checked function. "SKIPPED" means that the function is not analyzed for the reason indicated in the "details" section.
- $MISSING_ARG_NAMES: the missing argument names in FUN_ARGS.
- $MISSING_ARGS: the missing arguments with their values in FUN_ARGS.
- $STATUS: either "GOOD", meaning that all the arguments are already written, or a new proposal of arguments writing (arguments written with a partial name, like dig for digits, are written with their full name), or nothing.

An additional message "EVERYTHING SEEMS CLEAN" if the $STATUS column is only made of "" and "GOOD".

An additional message "ARGUMENTS WITH EMPTY VALUE" listing, for each line and function, the arguments proposed in the $STATUS column with an empty value (like envir =), with their default value, i.e., the potential flaws of the code (see the details section). Arguments already written with an empty value in the code are accepted as is and not listed. Use "= ," or "=)" to seek each case in the code.

An additional message "CORRECTIONS NOT APPLIED" listing the proposals of the $STATUS column that have not been inserted into the modified function, each followed by a REASON: line explaining why. These calls are left unchanged in the modified function and must be corrected manually (see the details section).

Author(s)

Gael Millot

Haiding Wang

Yushi Han

Examples

```
# Warning: these examples may not work well when using the "Run examples" link
# because of a particular environment. Please, copy-paste in a local environment.
# See also https://safer-r.github.io/saferDev/articles/all_args_here.html
fun1 <- function(x){
    base::paste0(x, "a") # collapse and recycle0 arguments not written
}
fun1_safer <- saferDev::all_args_here(x = fun1, safer_check = FALSE) # arguments added
fun1_safer # corrected function
# Write the table (res.tsv) and the corrected function (fun1_safer.R) in a new temporary folder:
tempo_dir <- base::tempfile(pattern = "dir", tmpdir = base::tempdir(), fileext = "")
base::dir.create(path = tempo_dir, showWarnings = TRUE, recursive = FALSE, mode = "0777")
saferDev::all_args_here(x = fun1, table_export = TRUE, fun_export = TRUE, out_path = tempo_dir,
    safer_check = FALSE)
```

arg_check *Argument Check*

Description

Check expected values of an argument of functions: class, type, mode, length, restricted values panel, kind of numeric values (in addition to the distinction between `'integer'` and `'double'`)? Proportion only? Inf values authorized? negative values authorized? Integers of type `'double'`?

Usage

```
arg_check(
  data,
  class = NULL,
  typeof = NULL,
  mode = NULL,
  length = NULL,
  prop = FALSE,
  options = NULL,
  double_as_integer_allowed = FALSE,
  all_options_in_data = FALSE,
  na_contain = FALSE,
  neg_values = TRUE,
  inf_values = TRUE,
  print = FALSE,
  data_name = NULL,
  data_arg = TRUE,
  safer_check = TRUE,
  lib_path = NULL,
  error_text = ""
)
```

Arguments

`data`	Object to test.
`class`	Character string or vector of character strings, tested exactly (same values, in the same order) against `class(data)`. Thus, `class = c("matrix", "array")` is required for matrices since R 4.0.0 and `class = c("ordered", "factor")` for ordered factors (see the details below). Write NULL to do not test the class.
`typeof`	Character string or vector of character strings, tested exactly (same values, in the same order) against `typeof(data)`. Since `typeof(data)` always returns a single string, a vector of more than one value in the `typeof` argument always returns a problem. Write NULL to do not test the type.
`mode`	Character string or vector of character strings, tested exactly (same values, in the same order) against `mode(data)`. Since `mode(data)` always returns a single string, a vector of more than one value in the `mode` argument always returns a problem. Write NULL to do not test the mode.

length — Single numeric value indicating the length of the object. Not considered if NULL.

prop — Single logical value. Are the numeric values between 0 and 1 (proportion)? If TRUE, the tested object must be of type "double" with all its values between -1 and 1 if neg_values is TRUE (default), or between 0 and 1 if neg_values is FALSE, and the class, typeof and mode arguments must be NULL. To check the class of a proportion object (a matrix for instance), use a second arg_check() call with prop = FALSE.

options — Vector of character strings or integers indicating all the possible option values for the data argument, or NULL. Numbers of type "double" are accepted if they have a 0 modulo (i.e., are integer like).

double_as_integer_allowed — Single logical value. If TRUE, no error is reported in the checking message if the typeof argument is set to "integer", while the reality is type "double" but with zero as modulo (reminder of a division). This means that i <- 1, which is typeof(i) == "double" is considered as an integer when setting double_as_integer_allowed = TRUE. The same rescue is performed when the class argument is set to "integer" and the tested object is of type "double" with a zero as modulo (because class() of such objects is "numeric", not "integer"). Warning: double_as_integer_allowed = TRUE uses data %% 1 == 0L but not isTRUE(all.equal(data %% 1, 0)) is used here because the argument checks for integers stored as double (does not check for decimal numbers that are approximate integers).

all_options_in_data — Single logical value. If FALSE, the tested object must be made of at least one of the options and nothing else. Example returning no error: data = c(1, 1, 2) and options = c(1, 2, 3). Example returning an error: data = c(1, 4) and options = c(1, 2, 3). If TRUE, the tested object must contain all of the options, at least once, and nothing else. Example returning no error: data = c(1, 1, 2, 3 and options = c(1, 2, 3). Example returning an error (missing 3): data = c(1, 1, 2 and options = c(1, 2, 3). Example returning an error (unwanted 4): data = c(1, 1, 4 and options = c(1, 2, 3). Ignored if the options argument is NULL.

na_contain — Single logical value. Can the data argument contain NA?

neg_values — Single logical value. Are negative numeric values authorized? Warning: the default setting is TRUE, meaning that, in that case, no check is performed for the presence of negative values. The checking is activated only when set to FALSE. In addition, FALSE can only be used when the typeof argument is set to "double" or "integer", or the mode argument to "numeric", or the prop argument to TRUE (values between 0 and 1 instead of -1 and 1). Otherwise an error message is returned. Finally, the presence of negative values is not checked with FALSE if the tested object is a factor and a message is returned: "OBJECT MUST BE MADE OF NON NEGATIVE VALUES BUT IS A FACTOR".

inf_values — Single logical value. Are infinite Inf or -Inf values authorized? Warning: the default setting is TRUE, meaning that, in that case, no check is performed for the presence of infinite values. The checking is activated only when set to FALSE. In addition, FALSE can only be used when the typeof argument is set to "double", or the mode argument to "numeric". Otherwise an error message is returned. Finally, the presence of infinite values is not checked with FALSE if

the tested object is a factor and a message is returned: "OBJECT MUST BE MADE OF NON NEGATIVE VALUES BUT IS A FACTOR".

print — Single logical value. Print the message if the $problem output is TRUE? Warning: set by default to FALSE, which facilitates the control of the checking message output when using arg_check() inside functions. See the example section.

data_name — Single character string indicating the name of the object to test. If NULL, use what is assigned to the data argument for the returned message.

data_arg — Single logical value. Is the tested object a function argument? If TRUE (default), "ARGUMENT" is written in output messages, otherwise "OBJECT".

safer_check — Single logical value. Perform the "safer" checks? If TRUE, checkings are performed before main code running (see the safer-r project): 1) correct lib_path argument value 2) required functions and related packages effectively present in local R libraries and 3) R classical operators (like "<-") not overwritten by another package because of the R scope. Warning: must be set to FALSE if this function is used inside another "safer" function to avoid pointless multiple checkings.

lib_path — Vector of characters specifying the absolute pathways of the directories containing the required packages for the function, if not in the default directories. Useful when R packages are not installed in the default directories because of lack of admin rights. More precisely, lib_path is passed through the new argument of .libPaths() so that the new library paths are unique(c(new, .Library.site, .Library)). Warning: .libPaths() is restored to the initial paths, after function execution. Ignored if NULL (default) or if the safer_check argument is FALSE: only the pathways specified by the current .libPaths() are used for package calling.

error_text — Single character string used to add information in error messages returned by the function, notably if the function is inside other functions, which is practical for debugging. Example: error_text = " INSIDE <PACKAGE_1>::<FUNCTION_1> INSIDE <PACKAGE_2>::<FUNCTION_2>.". If NULL, converted into "". Of note, in arg_check(), error_text is also used at the end of the string returned when no problem is detected.

Details

At least one of the class, type, mode, length, prop or options must be set by the user.

If options == NULL, then at least class or type or mode or length argument must be non-null.

If options is non-null, then class, type and mode must be NULL, and length can be NULL or specified.

If prop is TRUE, then class, type and mode must be NULL, and length can be NULL or specified. The values must be between -1 and 1 with neg_values = TRUE (default), and between 0 and 1 with neg_values = FALSE. Consequently, inf_values must stay TRUE (pointless anyway, since proportions are not infinite).

The function tests what is written in its arguments, even if what is written is incoherent. For instance with arg_check(data = factor(1), class = "factor", mode = "character"), the function tests that data is class "factor" and mode "character", even if no factor object can be

class "factor" and mode "character" (factors are class "factor" and mode "numeric"). Thus, arg_check(data = factor(1), class = "factor", mode = "character") will return a problem.

If the tested object is NULL, then the function will always return a checking problem.

The class, typeof and mode arguments are compared to class(data), typeof(data) and mode(data) exactly: the written values must be identical, element per element, in the same order. Since typeof(data) and mode(data) return a single string, passing several values in the typeof or mode arguments always returns a problem. In addition, since class(data) of objects of class "matrix" is c("matrix", "array") since R 4.0.0, class = "matrix" alone always returns a problem for matrices (use class = c("matrix", "array")). Similarly, class = "factor" alone always returns a problem for ordered factors (use class = c("ordered", "factor")), and class = "vector" is no longer a shortcut for the classes "numeric", "integer", "character", "logical", "complex" or "expression": write the exact class instead, or nothing.

Example of code that overcomes the problem of R versions, with a matrix object:

```
tempo1 <- saferDev::arg_check(data = data, class = "matrix") # before R 4.0.0
tempo2 <- saferDev::arg_check(data = data, class = c("matrix", "array")) # since R 4.0.0
if(tempo1$problem == TRUE & tempo2$problem == TRUE){
    print("data ARGUMENT MUST BE A MATRIX")
}
```

To test several possibilities for the tested object:

```
tempo1 <- saferDev::arg_check(data = data, mode = "character")
tempo2 <- saferDev::arg_check(data = data, class = "factor")
tempo3 <- saferDev::arg_check(data = data, class = "integer")
if(tempo1$problem == TRUE & tempo2$problem == TRUE & tempo3$problem == TRUE){
   print("data ARGUMENT MUST BE A VECTOR OF (1) STRINGS, (2) FACTORS OR (3) INTEGERS")
}
```

Of note, the length of object of class "environment" is always 0.

Value

A list containing:

- problem: logical. Is there any problem detected?
- text: message indicating the details of the problem, or the absence of problem.
- object_name: value of the data_name argument (i.e., name of the checked object if provided, NULL otherwise).

Author(s)

Gael Millot

Haiding Wang

Yushi Han

See Also

class.

Examples

```
# Warning: these examples may not work well when using the "Run examples" link
# because of a particular environment. Please, copy-paste in a local environment.
# See also https://safer-r.github.io/saferDev/articles/arg_check.html
test <- matrix(1:3)
## Not run:
# Example that returns an error (since R 4.0.0, class(matrix(1:3)) is c("matrix", "array"))
saferDev::arg_check(data = test, print = TRUE, class = "vector", mode = "numeric")
# Example that returns an error as well, because class = "matrix" must match class(data) exactly
saferDev::arg_check(data = test, print = TRUE, class = "matrix", mode = "numeric")

## End(Not run)
saferDev::arg_check(data = test, print = TRUE, class = c("matrix", "array"), mode = "numeric")
saferDev::arg_check(
    data = test,
    print = TRUE,
    class = c("matrix", "array"),
    mode = "numeric",
    error_text = " using saferDev::arg_check()"
)
```

---

colons_check *Colons Check*

Description

Verify that all the functions used inside a function are preceded by a package attribution. For instance: `base::mean()` and not `mean()`, or `saferDev:::.base_op_check()` and not `.base_op_check()`. Warning: does not check that the package is the good one. Use `all_args_here()` for that.

Usage

```
colons_check(
  x,
  export = FALSE,
  out_path = ".",
  safer_check = TRUE,
  lib_path = NULL,
  error_text = ""
)
```

Arguments

x	a function name, written without quotes and brackets.
export	Single logical value. Export the modified code into a .R file? If TRUE, the modified function is not returned but only exported. The name of the file is the name of the tested function, with _safer added at the end (e.g., test_safer.R). The written code is the code of the submitted function, with its comments and indentation, only modified by the added package attributions (see the Details section). An error is returned if the file already exists in out_path, and if the tested function is not designated by a function name (like test or package::test), i.e., written by an inline definition (like function(x){x}) or by a component writing (like FUN_LIST$fun). If FALSE (default), the modified function is not written but returned.
out_path	Single character string indicating the pathway of the folder where to export the modified code. out_path = "." (default) means the R working directory set by the user. A non-existing directory, NULL or an empty string ("") returns an error. Ignored if export is FALSE.
safer_check	Single logical value. Perform the "safer" checks? If TRUE, checkings are performed before main code running (see the safer-r project): 1) required functions and related packages effectively present in local R libraries and 2) R classical operators (like "<-") not overwritten by another package because of the R scope. Warning: must be set to FALSE if this function is used inside another "safer" function to avoid pointless multiple checkings.
lib_path	Vector of characters specifying the absolute pathways of the directories containing the required packages for the function, if not in the default directories. Useful when R packages are not installed in the default directories because of lack of admin rights. More precisely, lib_path is passed through the new argument of .libPaths() so that the new library paths are unique(c(new, .Library.site, .Library)). Warning: .libPaths() is restored to the initial paths, after function execution. Ignored if NULL (default): only the pathways specified by the current .libPaths() are used for package calling.
error_text	Single character string used to add information in error messages returned by the function, notably if the function is inside other functions, which is practical for debugging. Example: error_text = " INSIDE <PACKAGE_1>::<FUNCTION_1> INSIDE <PACKAGE_2>::<FUNCTION_2>.". If NULL, converted into "".

Details

The result is used to modify the code inside the tested function like this: <PACKAGE>::<FUNCTION> (or <PACKAGE>:::<FUNCTION> for function names starting by a dot).

Since saferDev v3.0.0, the modified function is returned: the package attributions of the basic R functions, as indicated in the "INSIDE test(), SOME :: OR ::: ARE MISSING BEFORE BASIC FUNCTIONS" section of the returned message, are added to the submitted function code, and the modified function is returned using return(). The package attributions of the other functions, as indicated in the "INSIDE test(), SOME :: OR ::: ARE MISSING BEFORE OTHER FUNCTIONS" section of the returned message, are NOT added automatically, because the correct package cannot be determined reliably: a base::warning() is issued instead (in addition to the printed message). NULL is returned if no basic function required a package attribution.

The modification is only performed on the code lines indicated in the "INSIDE test(), SOME :: OR ::: ARE MISSING BEFORE BASIC FUNCTIONS" section of the message, which is the same analysis as the one performed by colons_check() itself: functions written between quotes or preceded by $ are not modified, and internal functions (created inside the tested function) are not modified either. The analysis is performed on the code with comments removed and multiple spaces collapsed, but the package attributions are added to the code of the submitted function as printed: the returned (and exported) function therefore keeps the comments and the indentation of the submitted function. Warning: comments are only present when the submitted function carries its source references (attribute srcref, e.g., a function defined in the console or in a file loaded with keep.source = TRUE). Otherwise, as for compiled package functions, the code printed by R (without comments and reindented) is used. The modified function is returned with the environment of the submitted function, so that the functions used inside it still resolve the same way. If the modified code does not parse (a rare situation, e.g., a function name split across lines in a way the analyzer could not handle correctly), this is an internal bug: an internal error is returned when export = TRUE, and the submitted function is returned unchanged, with no error, when export = FALSE.

The name of the exported file (when export = TRUE) is the name of the submitted function, as written by the user, with _safer added at the end, e.g., test_safer.R for colons_check(x = test). If the submitted function was written as package::function, like colons_check(x = saferDev::test), the name is test_safer.R (the package name is not part of the file name). The exported file contains an assignment of the modified function: test_safer <- function(...) ..., so that base::source()ing it recreates the modified function. Warning: the tested function must be designated by a function name (like test or package::test) when export = TRUE: an inline definition (like function(x){x}) or a component writing (like FUN_LIST$fun) returns an error, because the name of the exported file could not be determined reliably.

Pre-check: the submitted function is first checked by lambda_check(): if some lambda functions are written with the \(x) shorthand, an error is returned asking to run saferDev::lambda_check() first, because the \ writings are not managed by the text-based detection of colons_check(). As in lambda_check(), the \(x) writing is only detectable in a function carrying its source references (attribute srcref): a compiled or loaded package function prints function(x), so that the pre-check silently passes in that case.

More precisely, colons_check() verifies that all the strings before an opening bracket (are preceded by ::.

::: are not checked per se, because incorrect writing, like saferDev:::.colons_check_message() returns an error when executing the tested function, and because base:::sum() is as ok as base::sum(). In the same manner, more than three colons are not checked because it returns an error when executing the tested function.

Warning: colons_check() cannot check function names written without brackets, like in the FUN argument of some functions, e.g., sapply(1:3, FUN = as.character).

The Perl regex used to detect a function name is: "([a-zA-Z]|\.[a-zA-Z._])[a-zA-Z0-9._]*\s*\(".

Currently, colons_check() cannot detect functions written between quotes, like "+"() or "rownames<-"(x, "a").

Function names preceded by $ are not considered.

The following R functions are skipped: "function", "if", "for", "while", "repeat" and "else".

Most of the time, colons_check() does not check inside comments, but some unexpected writing could dupe colons_check(). Please, report here if it is the case.

The returned line numbers are indicative, depending on which source is checked. For instance, `saferDev::report` (compiled) has not the same line numbers as its source file (source file). Notably, compiled functions do not have comments anymore, compared to the same source function sourced into the working environment. In addition, the counting starts at the "`<- function`" line, i.e., without counting the `#'` header lines potentially present in source files.

Of note, during package creation, the `devtools::check()` command tells which functions where wrongly attributed to package. Example:

```
checking dependencies in R code ... WARNING
  '::' or ':::' import not declared from: 'sbase'
  Missing or unexported objects:
   'base::dev.off' 'base::graphics.off' 'base::hcl' 'base::par' 'base::read.table' 'saferGG::report'
```

Value

A table-like text message indicating the missing `::` or `:::` or a message saying that everything seems fine.

The message is always printed (see the `Details` section).

A warning and `NULL` are returned if the submitted function is empty of function, or only made of comments, or only made of `function`, `if`, `for`, `while`, `repeat`, `else`.

In addition, if the `export` argument is `FALSE` (default), the function returns `NULL` if no basic function required a package attribution, or the modified function if basic functions were modified (see the `Details` section).

If the `export` argument is `TRUE`, the modified code is exported in the `.R` file specified by the `out_path` argument, instead of being returned, and a message indicating where the modified code has been exported is printed.

Table-like: column 1, the line number in the function code (starting at the "`<- function`" line, i.e., without counting the `#'` header lines); column 2, the function name; column 3, the code preceding the function name.

With missing `::` or `:::`, the message also indicates if internal functions are created inside the checked function code, since these functions cannot have `::` or `:::`.

Author(s)

Gael Millot

Haiding Wang

Yushi Han

Examples

```
# Warning: these examples may not work well when using the "Run examples" link
# because of a particular environment. Please, copy-paste in a local environment.
# See also https://safer-r.github.io/saferDev/articles/colons_check.html
fun1 <- function(x){
    y <- mean(x) # mean() written without base::
    base::return(y)
}
```

```
fun1_safer <- saferDev::colons_check(fun1) # base:: added before mean()
fun1_safer # corrected function
# Write the corrected function in the fun1_safer.R file of a new temporary folder:
tempo_dir <- base::tempfile(pattern = "dir", tmpdir = base::tempdir(), fileext = "")
base::dir.create(path = tempo_dir, showWarnings = TRUE, recursive = FALSE, mode = "0777")
saferDev::colons_check(x = fun1, export = TRUE, out_path = tempo_dir)
```

`curly_bracket_check` *Curly Brackets Check*

Description

Verify that all the bodies of the `if`, `else`, `for`, `while`, `repeat` and `function` are delimited by curly brackets `{ }`. For instance: `if(x > 1){ x }` and not `if(x > 1) x`. Return table message of the line with problems and a corrected version of the submitted function.

Usage

```
curly_bracket_check(
  x,
  export = FALSE,
  out_path = ".",
  safer_check = TRUE,
  lib_path = NULL,
  error_text = ""
)
```

Arguments

`x`	A function name, written without quotes and brackets.
`export`	Single logical value. Write the corrected function into a file instead of returning it? If `TRUE`, the file `<function name>_safer.R` is written in `out_path` (e.g., `fun_safer.R` for `x = fun` or `x = pkg::fun`). It contains `fun_safer <- function(...) ...`, so that `base::source()` recreates the corrected function. Nothing is written if nothing has to be corrected. An error is returned if the file already exists, or if `x` is not a function name (e.g., `function(x){x}` or `FUN_LIST$fun`).
`out_path`	Single character string indicating the path of the folder where to write the file when `export = TRUE`. `"."` (default) means the R working directory. Ignored if `export = FALSE`.
`safer_check`	Single logical value. Perform the "safer" checks? If `TRUE`, checkings are performed before main code running (see the safer-r project): 1) required functions and related packages effectively present in local R libraries and 2) R classical operators (like `"<-"`) not overwritten by another package because of the R scope. Warning: must be set to `FALSE` if this function is used inside another "safer" function to avoid pointless multiple checkings.

lib_path — Vector of characters specifying the absolute pathways of the directories containing the required packages for the function, if not in the default directories. Useful when R packages are not installed in the default directories because of lack of admin rights. More precisely, lib_path is passed through the new argument of .libPaths() so that the new library paths are unique(c(new, .Library.site, .Library)). Warning: .libPaths() is restored to the initial paths, after function execution. Ignored if NULL (default): only the pathways specified by the current .libPaths() are used for package calling.

error_text — Single character string used to add information in error messages returned by the function, notably if the function is inside other functions, which is practical for debugging. Example: error_text = " INSIDE <PACKAGE_1>::<FUNCTION_1> INSIDE <PACKAGE_2>::<FUNCTION_2>.". If NULL, converted into "".

Details

Comments are ignored. In else if chains, the else is not reported: the chained if is checked on its own.

Bodies reduced to .Primitive(...), like in base::`+`, are skipped, because they cannot be enclosed in curly brackets. Bodies made of UseMethod(...), like in base::mean, are reported.

Correction: a { is added before each reported body and a } after it, separated from the body by a space. The rest of the code is unchanged. The corrected code is checked by parsing it. Lines can become longer than 80 characters. The returned function keeps the environment of the submitted function.

The comments and the indentation of the submitted function are kept, in the returned function and in the exported file. Warning: comments are only available when the submitted function carries its source references (attribute srcref). Otherwise (e.g., compiled package functions, or functions created with options(keep.source = FALSE), default of Rscript), the code is the deparsed one: no comments and standard indentation.

Constructs written with backquotes, like `if`(x > 1, x, 0), may not be checked: use the standard writing.

Line numbers are indicative. They start at the <- function line (#' header lines not counted). Compiled functions, like saferDev::colons_check, have no comments: their line numbers differ from their source file.

Value

A table-like message is always printed, with one row per missing { } (column 1: line number; column 2: construct; column 3: code before the body), or a message saying that everything seems fine.

If export = FALSE: the corrected function, or NULL if nothing has to be corrected.

If export = TRUE: NULL, returned invisibly. The corrected function is written in a file (see the export argument).

Author(s)

Gael Millot

Haiding Wang

Yushi Han

Examples

```
# Warning: these examples may not work well when using the "Run examples" link
# because of a particular environment. Please, copy-paste in a local environment.
# See also https://safer-r.github.io/saferDev/articles/curly_bracket_check.html
fun1 <- function(x){ if(x > 1){ base::return(x) } }
saferDev::curly_bracket_check(fun1) # everything fine: NULL returned
fun2 <- function(x){ if(x > 1) base::return(x) }
fun2_safer <- saferDev::curly_bracket_check(fun2) # missing { } reported
fun2_safer # corrected function
# Write the corrected function in the fun2_safer.R file of a new temporary folder:
tempo_dir <- base::tempfile(pattern = "dir", tmpdir = base::tempdir(), fileext = "")
base::dir.create(path = tempo_dir, showWarnings = TRUE, recursive = FALSE, mode = "0777")
saferDev::curly_bracket_check(x = fun2, export = TRUE, out_path = tempo_dir)
```

---

env_check *Environment Check*

Description

Verify that object names in the environment defined by the pos parameter are identical or not to object names in the above environments (following R Scope). This can be used to verify that names used for objects inside a function or in the working environment do not override names of objects already present in the above R environments, following the R scope.

Usage

```
env_check(
  pos = 1,
  name = NULL,
  safer_check = TRUE,
  lib_path = NULL,
  error_text = ""
)
```

Arguments

pos	Single non null positive integer indicating the position of the environment checked (argument n of the parent.frame() function). Value 1 means one step above the env_check() local environment (by default). This means that when env_check(pos = 1) is used inside a function A, it checks if the name of any object in the local environment of this function A is also present in above environments, following R Scope, starting by the just above environment. When env_check(pos = 1) is used in the working (Global) environment (named .GlobalEnv), it checks the

object names of this .GlobalEnv environment, in the above environments. See the examples below.

name — Single character string of the name of the checked environment (used in the output string only, where the name of the ckecked env is replaced by the name value). Can be used when env_check() is used inside a function "A" to specify that the environment of "A" is checked.

safer_check — Single logical value. Perform the "safer" checks? If TRUE, checkings are performed before main code running (see the safer-r project): 1) required functions and related packages effectively present in local R libraries and 2) R classical operators (like "<-") not overwritten by another package because of the R scope. Warning: must be set to FALSE if this function is used inside another "safer" function to avoid pointless multiple checkings.

lib_path — Vector of characters specifying the absolute pathways of the directories containing the required packages for the function, if not in the default directories. Useful when R packages are not installed in the default directories because of lack of admin rights. More precisely, lib_path is passed through the new argument of .libPaths() so that the new library paths are unique(c(new, .Library.site, .Library)). Warning: .libPaths() is restored to the initial paths, after function execution. Ignored if NULL (default): only the pathways specified by the current .libPaths() are used for package calling.

error_text — Single character string used to add information in error messages returned by the function, notably if the function is inside other functions, which is practical for debugging. Example: error_text = " INSIDE <PACKAGE_1>::<FUNCTION_1> INSIDE <PACKAGE_2>::<FUNCTION_2>.". If NULL, converted into "".

Value

A character string indicating the object names of the tested environment that match object names in the above environments, following the R scope, or NULL if no match.

Author(s)

Gael Millot

Haiding Wang

Yushi Han

See Also

exists.

Examples

```
# Warning: these examples may not work well when using the "Run examples" link
# because of a particular environment. Please, copy-paste in a local environment.
# See also https://safer-r.github.io/saferDev/articles/env_check.html
# Inside a function: the names t.test and mean, created inside fun1(),
# also exist in the stats and base environments located above:
fun1 <- function(){
```

```
    t.test <- 0
    mean <- 5
    saferDev::env_check(pos = 1, name = "fun1")
}
a <- fun1()
base::cat(a)
# NULL returned when no name is also present in the environments above:
fun2 <- function(){
    tempo_name_used_nowhere_else <- 0
    saferDev::env_check(pos = 1, name = "fun2")
}
fun2()
# pos = 2 checks the environment one step above the function (here, the calling environment).
# See the env_check() article of the website for more examples.
```

`for_loop_nb_check` `for` *Loop Number Check*

Description

Check the sequence expression of every `for` loop of a function. If the number of iterations is not fixed by the writing itself (e.g., `1:length(x)`), a checking line is added before the loop, issuing a warning at execution if the sequence is empty, contains `NA` or infinite values, has a single value, or is decreasing. Return a table-like message of the problematic lines and a corrected version of the submitted function.

Usage

```
for_loop_nb_check(
  x,
  export = FALSE,
  out_path = ".",
  safer_check = TRUE,
  lib_path = NULL,
  error_text = ""
)
```

Arguments

`x`	A function name, written without quotes and brackets.
`export`	Single logical value. Write the corrected function into a file instead of returning it? If `TRUE`, the file `<function name>_safer.R` is written in `out_path` (e.g., `fun_safer.R` for `x = fun` or `x = pkg::fun`). It contains `fun_safer <- function(...) ...`, so that `base::source()` recreates the corrected function. Nothing is written if nothing has to be corrected. An error is returned if the file already exists, or if `x` is not a function name (e.g., `function(x){x}` or `FUN_LIST$fun`).

`out_path` Single character string indicating the path of the folder where to write the file when `export = TRUE`. `"."` (default) means the R working directory. Ignored if `export = FALSE`.

`safer_check` Single logical value. Perform the "safer" checks? If `TRUE`, checkings are performed before main code running (see the safer-r project): 1) required functions and related packages effectively present in local R libraries and 2) R classical operators (like `"<-"`) not overwritten by another package because of the R scope. Warning: must be set to `FALSE` if this function is used inside another "safer" function to avoid pointless multiple checkings.

`lib_path` Vector of characters specifying the absolute pathways of the directories containing the required packages for the function, if not in the default directories. Useful when R packages are not installed in the default directories because of lack of admin rights. More precisely, `lib_path` is passed through the `new` argument of `.libPaths()` so that the new library paths are `unique(c(new, .Library.site, .Library))`. Warning: `.libPaths()` is restored to the initial paths, after function execution. Ignored if `NULL` (default): only the pathways specified by the current `.libPaths()` are used for package calling.

`error_text` Single character string used to add information in error messages returned by the function, notably if the function is inside other functions, which is practical for debugging. Example: `error_text = " INSIDE <PACKAGE_1>::<FUNCTION_1> INSIDE <PACKAGE_2>::<FUNCTION_2>."`. If `NULL`, converted into `""`.

Details

Not reported: hard numbers, i.e., an integer-valued number, possibly negative, or two such numbers joined by `:` (e.g., `3`, `1:3`, `1e3`, `1L:3L`, `0:0`, `-3:-1`, `1:0`), and a single `seq_len(...)` or `seq_along(...)` call. These writings fully fix the iterations, and their control is left to the writer.

Reported: everything else, e.g., `for(i in x)`, `1:n`, `1:length(x)`, `1:nrow(df)`, `1:(n - 1)`, `seq_along(x) - 1`, `1:2.5`, `1:NA` or `1:Inf`. Comments and strings are ignored.

Why: with `x <- character()`, `for(i in 1:length(x))` runs twice, because `1:length(x)` is `c(1, 0)`. Use `base::seq_along(along.with = x)` instead.

Correction: each reported loop becomes `tempo_for_seqN <- <SEQUENCE> ; <CHECKS> ; for(i in tempo_for_seqN)<BODY>`, N being the rank of the report. The checks warn if the sequence is empty, contains `NA` or infinite values, has a single value, has only negative values, or is decreasing (these last two checks only for numeric sequences, i.e., not for factors, dates, characters or complex numbers). The loop thus runs on the checked sequence. Errors of the sequence itself (like `1:NA`) are not caught. If the loop is not a standalone statement (e.g., unbraced body of an `if`, or argument value), the whole is enclosed in curly brackets. The rest of the code is unchanged. The comments and the indentation of the submitted function are kept, in the returned function and in the exported file. Warning: comments are only available when the submitted function carries its source references (attribute srcref). Otherwise (e.g., compiled package functions, or functions created with `options(keep.source = FALSE)`, default of Rscript), the code is the deparsed one: no comments and standard indentation. The returned function is evaluated in a new environment whose parent is the environment of the submitted function: check it before replacing the original function.

Warning: a reported loop is left unchanged, and listed in the message, when:

- the name `tempo_for_seqN` is already used in the function code (including comments and strings);

- the loop is inside the sequence expression of another reported loop;
- the loop must be enclosed in curly brackets and contains comments;
- the loop must be enclosed in curly brackets and ends with another loop that must also be (e.g., `for(a in 1:n) for(b in 1:length(x)){...}`).

Warning: only the writing is analyzed. In `n <- length(x) ; for(i in 1:n)`, `1:n` is reported as a variable, whatever `n`.

Line numbers are indicative. They start at the `<- function` line (`#'` header lines not counted). Compiled functions, like `saferDev::colons_check`, have no comments: their line numbers differ from their source file.

Value

A table-like message is always printed, with one row per reported loop (column 1: line number; column 2: construct; column 3: code before the body), or a message saying that everything seems fine.

If `export = FALSE`: the corrected function, or `NULL` if nothing has been corrected.

If `export = TRUE`: `NULL`, returned invisibly. The corrected function is written in a file (see the `export` argument).

Author(s)

Gael Millot

Haiding Wang

Yushi Han

Examples

```
# Warning: these examples may not work well when using the "Run examples" link
# because of a particular environment. Please, copy-paste in a local environment.
# See also https://safer-r.github.io/saferDev/articles/for_loop_nb_check.html
fun1 <- function(x){ for(i in base::seq_along(along.with = x)){ base::print(x = i) } }
saferDev::for_loop_nb_check(fun1) # everything fine: NULL returned
fun2 <- function(x){ for(i in 1:length(x)){ base::print(x = i) } }
fun2_safer <- saferDev::for_loop_nb_check(fun2) # 1:length(x) reported
fun2_safer # corrected function
fun2_safer(x = base::character()) # warning: 1:length(x) is c(1, 0), a decreasing sequence
# Write the corrected function in the fun2_safer.R file of a new temporary folder:
tempo_dir <- base::tempfile(pattern = "dir", tmpdir = base::tempdir(), fileext = "")
base::dir.create(path = tempo_dir, showWarnings = TRUE, recursive = FALSE, mode = "0777")
saferDev::for_loop_nb_check(x = fun2, export = TRUE, out_path = tempo_dir)
```

get_message *Get Message*

Description

Evaluate an instruction written between "" and return the first of the error message, or the last of the warning or standard (non error non warning) messages if ever exist.

Using argument print_no = FALSE, return NULL if nothing reported, which is convenient in some cases.

Usage

```
get_message(
  data,
  kind = "error",
  header = TRUE,
  print_no = FALSE,
  text = NULL,
  env = NULL,
  safer_check = TRUE,
  lib_path = NULL,
  error_text = ""
)
```

Arguments

data	Single character string of an instruction to evaluate
kind	Single character string. Either "error" to get a potential error message, or "warning" to get a potential warning message, or "message" to get a potential standard (non error and non warning) message.
header	Single logical value. Add a header in the returned message?
print_no	Single logical value. Print a message saying that nothing (NULL output) has to be reported?
text	Single character string added to the header of the output message, even if nothing reported (print_no is TRUE). Ignored if the header argument is FALSE. Write NULL if not required.
env	An object corresponding to an existing environment. Then the data argument value is evaluated only in the indicated environment. Write NULL if not required (R scope is used). Example env = .GlobalEnv. Example env = asNamespace("stats").
safer_check	Single logical value. Perform the "safer" checks? If TRUE, checkings are performed before main code running (see the safer-r project): 1) required functions and related packages effectively present in local R libraries and 2) R classical operators (like "<-") not overwritten by another package because of the R scope. Warning: must be set to FALSE if this function is used inside another "safer" function to avoid pointless multiple checkings.

`lib_path` Vector of characters specifying the absolute pathways of the directories containing the required packages for the function, if not in the default directories. Useful when R packages are not installed in the default directories because of lack of admin rights. More precisely, `lib_path` is passed through the `new` argument of `.libPaths()` so that the new library paths are `unique(c(new, .Library.site, .Library))`. Warning: `.libPaths()` is restored to the initial paths, after function execution. Ignored if `NULL` (default): only the pathways specified by the current `.libPaths()` are used for package calling.

`error_text` Single character string used to add information in error messages returned by the function, notably if the function is inside other functions, which is practical for debugging. Example: `error_text = " INSIDE <PACKAGE_1>::<FUNCTION_1> INSIDE <PACKAGE_2>::<FUNCTION_2>."`. If `NULL`, converted into `""`.

Details

Warnings:

- Only the first error or last warning/standard message is returned.
- Always use the `env` argument when `get_message()` is used inside functions.
- The function does not prevent printing, for instance if `print()` or `show()` is used inside the instruction tested. To prevent that, use `tempo <- utils::capture.output(error <- get_message(data = "arg_check(data = 'a', class = mean, neg_values = FALSE, print = TRUE)"))`. The return of `get_message()` is assigned into `error` and the printed messages are captured by `utils::capture.output()` and assigned into `tempo`. See the examples.

The `env` argument is used as value for the `envir` argument of the `base::eval()` function.

Value

The function returns the error, warning, or standard message, as a single character string, depending on what has been selected using the `kind` argument and if such message exists.

`NULL` if no selected message returned and the `print_no` argument is `FALSE`.

The following message if no selected message returned and the `print_no` argument is `TRUE`: `"NO (ERROR|WARNING|STANDARD) MESSAGE REPORTED"`.

The following message if 1) the `kind` argument is `"warning"` or `"message"` while an error message is returned and 2) the `print_no` argument is `TRUE`: `"NO POTENTIAL (WARNING|STANDARD) MESSAGE BECAUSE OF ERROR MESSAGE REPORTED"`.

Author(s)

Gael Millot

Haiding Wang

Yushi Han

See Also

`try`.

Examples

```
# Warning: these examples may not work well when using the "Run examples" link
# because of a particular environment. Please, copy-paste in a local environment.
# See also https://safer-r.github.io/saferDev/articles/get_message.html

# Report error message by default. Here no error to report:
saferDev::get_message(data = "wilcox.test(c(1,1,3), c(1, 2, 4), paired = TRUE)")

# Warning message to report:
saferDev::get_message(data = "wilcox.test(c(1,1,3), c(1, 2, 4), paired = TRUE)", kind = "warning")

# No standard message to report:
saferDev::get_message(data = "wilcox.test(c(1,1,3), c(1, 2, 4), paired = TRUE)", kind = "message",
print_no = TRUE, text = "IN A")

# Standard message cannot be caught because wilcox.test() returns an error message:
saferDev::get_message(data = "wilcox.test()", kind = "message", print_no = TRUE, text = "IN A")

# Text added in warning message to report:
saferDev::get_message(data = "wilcox.test()", kind = "error", print_no = TRUE, text = "IN A")

# No error message to report with text added:
saferDev::get_message(data = "sum(1)", kind = "error", print_no = TRUE, text = "IN A")

# No error message to report with text added:
saferDev::get_message(data = "message('ahah')", kind = "error", print_no = TRUE, text = "IN A")

# Standard message reported with text added:
saferDev::get_message(data = "message('ahah')", kind = "message", print_no = TRUE, text = "IN A")

# Messages from ggplot2 functions caugth:
saferDev::get_message(data = "ggplot2::ggplot(data = data.frame(X = 1:10, stringsAsFactors = TRUE),
mapping = ggplot2::aes(x = X)) + ggplot2::geom_histogram()", kind = "message",
print_no = TRUE, text = "IN INSTRUCTION 1")
```

---

is_function_here *Is Function Here*

Description

Check if required functions are present in installed packages. This controls the potential modifications of package contents.

Usage

```
is_function_here(fun, safer_check = TRUE, lib_path = NULL, error_text = "")
```

Arguments

fun	Character vector of the names of the required functions, preceded by the name of the package they belong to and a double or triple colon. Example: `c("ggplot2::geom_point", "grid::gpar")`. Warning: do not write `"()"` at the end of each tested function.
safer_check	Single logical value. Perform the "safer" checks? If `TRUE`, checkings are performed before main code running (see the safer-r project): 1) required functions and related packages effectively present in local R libraries and 2) R classical operators (like `"<-"`) not overwritten by another package because of the R scope. Warning: must be set to `FALSE` if this function is used inside another "safer" function to avoid pointless multiple checkings.
lib_path	Vector of characters specifying the absolute pathways of the directories containing the required packages for the function, if not in the default directories. Useful when R packages are not installed in the default directories because of lack of admin rights. More precisely, `lib_path` is passed through the `new` argument of `.libPaths()` so that the new library paths are `unique(c(new, .Library.site, .Library))`. Warning: `.libPaths()` is restored to the initial paths, after function execution. Ignored if `NULL` (default): only the pathways specified by the current `.libPaths()` are used for package calling.
error_text	Single character string used to add information in error messages returned by the function, notably if the function is inside other functions, which is practical for debugging. Example: `error_text = " INSIDE <PACKAGE_1>::<FUNCTION_1> INSIDE <PACKAGE_2>::<FUNCTION_2>."`. If `NULL`, converted into `""`.

Value

An error message if at least one of the checked packages is missing in `lib_path`, or if at least one of the checked functions is missing in the required package, nothing otherwise.

Author(s)

Gael Millot

Haiding Wang

Yushi Han

See Also

`exists` and `findFunction`

Examples

```
# Warning: these examples may not work well when using the "Run examples" link
# because of a particular environment. Please, copy-paste in a local environment.
# See also https://safer-r.github.io/saferDev/articles/is_function_here.html
## Not run:
# Example that returns an error
saferDev::is_function_here(fun = "ggplot2::notgood", error_text = " INSIDE P1::F1")
saferDev::is_function_here(fun = c("ggplot2::geom_point", "grid::gpar"))
saferDev::is_function_here(fun = "c")
```

```
## End(Not run)
saferDev::is_function_here(fun = "base::c", error_text = " INSIDE P1::F1")
```

`is_package_here` *Is Package Here*

Description

Check if required packages are installed locally.

Usage

```
is_package_here(
  req_package,
  safer_check = TRUE,
  lib_path = NULL,
  error_text = ""
)
```

Arguments

`req_package`	Character vector of package names to check.
`safer_check`	Single logical value. Perform the "safer" checks? If `TRUE`, checkings are performed before main code running (see the safer-r project): 1) required functions and related packages effectively present in local R libraries and 2) R classical operators (like `"<-"`) not overwritten by another package because of the R scope. Warning: must be set to `FALSE` if this function is used inside another "safer" function to avoid pointless multiple checkings.
`lib_path`	Vector of characters specifying the absolute pathways of the directories containing the required packages for the function, if not in the default directories. Useful when R packages are not installed in the default directories because of lack of admin rights. More precisely, `lib_path` is passed through the `new` argument of `.libPaths()` so that the new library paths are `unique(c(lib_path, .libPaths()))`. Warning: `.libPaths()` is restored to the initial paths, after function execution. Ignored if `NULL` (default): only the pathways specified by the current `.libPaths()` are used for package calling.
`error_text`	Single character string used to add information in error messages returned by the function, notably if the function is inside other functions, which is practical for debugging. Example: `error_text = " INSIDE <PACKAGE_1>::<FUNCTION_1> INSIDE <PACKAGE_2>::<FUNCTION_2>."`. If `NULL`, converted into `""`.

Value

An error message if at least one of the checked packages is missing in `lib_path`, nothing otherwise.

Author(s)

Gael Millot

Haiding Wang

Yushi Han

See Also

require.

Examples

```
# Warning: these examples may not work well when using the "Run examples" link
# because of a particular environment. Please, copy-paste in a local environment.
# See also https://safer-r.github.io/saferDev/articles/is_package_here.html
## Not run:
# Example that returns an error
saferDev::is_package_here(req_package = "nopackage", error_text = " INSIDE P1::F1")
# Example that returns an error
saferDev::is_package_here(req_package = "ggplot2", lib_path = "NOTGOOD")

## End(Not run)
saferDev::is_package_here(req_package = "ggplot2")
```

---

lambda_check *Lambda Check*

Description

Verify if lambda functions are written with the shorthand \(x) (like \(x) x * 2). Return table message of the line with problems and a corrected version of the submitted function.

Usage

```
lambda_check(
  x,
  export = FALSE,
  out_path = ".",
  safer_check = TRUE,
  lib_path = NULL,
  error_text = ""
)
```

Arguments

x	A function name, written without quotes and brackets.
export	Single logical value. Write the corrected function into a file instead of returning it? If TRUE, the file <function name>_safer.R is written in out_path (e.g., fun_safer.R for x = fun). Nothing is returned or exported if nothing has to be corrected. An error is returned if exported and the file already exists.
out_path	Single character string indicating the path of the folder where to write the file when export = TRUE. "." (default) means the R working directory. Ignored if export = FALSE.
safer_check	Single logical value. Perform the "safer" checks? If TRUE, checkings are performed before main code running (see the safer-r project): 1) required functions and related packages effectively present in local R libraries and 2) R classical operators (like "<-") not overwritten by another package because of the R scope. Warning: must be set to FALSE if this function is used inside another "safer" function to avoid pointless multiple checkings.
lib_path	Vector of characters specifying the absolute pathways of the directories containing the required packages for the function, if not in the default directories. Useful when R packages are not installed in the default directories because of lack of admin rights. More precisely, lib_path is passed through the new argument of .libPaths() so that the new library paths are unique(c(new, .Library.site, .Library)). Warning: .libPaths() is restored to the initial paths, after function execution. Ignored if NULL (default): only the pathways specified by the current .libPaths() are used for package calling.
error_text	Single character string used to add information in error messages returned by the function, notably if the function is inside other functions, which is practical for debugging. Example: error_text = " INSIDE <PACKAGE_1>::<FUNCTION_1> INSIDE <PACKAGE_2>::<FUNCTION_2>.". If NULL, converted into "".

Details

Warning: the \(x) writing of a lambda function is only kept in the code of the submitted function when the function carries its source references (attribute srcref, e.g., after a base::parse(keep.source = TRUE) or a function defined in a file loaded with keep.source = TRUE). Otherwise, R prints the lambda as function(x), so that nothing can be reported: run lambda_check() on the source code itself, not on a compiled or on a loaded package function. The base::getOption("keep.source") default is TRUE in interactive sessions and FALSE in Rscript.

Reported: lambdas written \(x), like \(x) x * 2, \(x){ x } or \() base::mean(1:3), wherever they are written in the submitted function (right side of an assignment, argument value of a function call, etc.). In the message table, the SYMBOL column shows the lambda writing with its formals only when they are written on a single line: for a lambda whose formals span several lines, the fallback writing \() is shown (the whole writing is still replaced correctly in the code, because only the \ token is replaced).

Not reported: the function(x) standard writing, and lambda writings of the past (e.g., function(x) x) reported by other functions. Strings, comments and backquoted names (like a backquoted backslash, the deprecated function() writing of R versions before 4.1.0) are ignored: a backslash followed by (inside a string, a raw string, a comment or a backquoted name is never reported

nor modified, whatever the rest of the written text (the detection is token-based, not text-based). A lambda body itself is not modified: run `curly_bracket_check()` and `true_false_check()` afterwards to check the corrected function further.

Correction: each reported `\` lambda writing is replaced by `function` (e.g., `\(x) x * 2` becomes `function(x) x * 2`). The rest of the code is unchanged. The corrected code is checked by parsing it. Lines can become longer than 80 characters. The returned function keeps the environment of the submitted function.

The comments and the indentation of the submitted function are kept, in the returned function and in the exported file, which contain the recovered code verbatim, only modified by the `\` -> `function` replacements. Warning: comments are only available when the submitted function carries its source references (attribute srcref). Otherwise (e.g., compiled package functions, or functions created with `options(keep.source = FALSE)`, default of Rscript), the code is the deparsed one: no comments and standard indentation.

Line numbers are indicative. They start at the `<- function` line (`#'` header lines not counted). Compiled functions, like `saferDev::colons_check`, have no comments: their line numbers differ from their source file.

Value

A table-like message is always printed, with one row per reported lambda (column 1: line number; column 2: symbol; column 3: code before the lambda), or a message saying that everything seems fine.

If `export = FALSE`: the corrected function, or `NULL` if nothing has to be corrected.

If `export = TRUE`: `NULL`, returned invisibly. The corrected function is written in a file (see the `export` argument).

Author(s)

Gael Millot

Examples

```
# Warning: these examples may not work well when using the "Run examples" link
# because of a particular environment. Please, copy-paste in a local environment.
# See also https://safer-r.github.io/saferDev/articles/lambda_check.html
# The \(x) writing is only kept in a function carrying its source references,
# hence the parse(keep.source = TRUE) writing used here:
code1 <- "function(x){ base::sapply(X = 1:3, FUN = function(y){ y * 2 }) }"
fun1 <- eval(parse(text = code1, keep.source = TRUE))
saferDev::lambda_check(fun1) # everything fine: NULL returned
code2 <- "function(x){ base::sapply(X = 1:3, FUN = \\(y){ y * 2 }) }"
fun2 <- eval(parse(text = code2, keep.source = TRUE))
fun2_safer <- saferDev::lambda_check(fun2) # lambda writing reported
fun2_safer # corrected function
# A backslash followed by ( in a string, a raw string, a comment or a backquoted name
# is not a lambda writing (token-based detection): not reported, not modified.
code3 <- paste0(
    "function(x){ y <- \"the text \\\\(x) is inside a string\" ; ",
    "z <- r\"(raw \\(x))\" ; # a \\(x) in a comment\n    base::`\\\\`(x) ; x }"
```

```
)
fun3 <- eval(parse(text = code3, keep.source = TRUE))
saferDev::lambda_check(fun3) # everything fine: NULL returned
# Write the corrected function in the fun2_safer.R file of a new temporary folder:
tempo_dir <- base::tempfile(pattern = "dir", tmpdir = base::tempdir(), fileext = "")
base::dir.create(path = tempo_dir, showWarnings = TRUE, recursive = FALSE, mode = "0777")
saferDev::lambda_check(x = fun2, export = TRUE, out_path = tempo_dir)
```

`report` *Report*

Description

Print a character string or a data object into a same output file. Convenient for log file generation.

Usage

```
report(
  data,
  output = "log.txt",
  path,
  append = FALSE,
  rownames_kept = FALSE,
  vector_cat = FALSE,
  noquote = TRUE,
  sep = 2,
  safer_check = TRUE,
  lib_path = NULL,
  error_text = ""
)
```

Arguments

`data`	Object to print in the output file. If `NULL`, nothing is done, with no warning.
`output`	Single character string. Name of the output file.
`path`	Single character string indicating the path where to write the output file.
`append`	Single logical value. If the output file already exists and `append` is `FALSE`, an error message is returned (no overwrite of existing file). Otherwise, the printing is appended (and the output file is created if it does not exist yet).
`rownames_kept`	Single logical value. Defines whether row names have to be removed or not in 2D objects. Warning: in 1D tables, names over the values are taken as row names, and are thus removed if `rownames_kept` is `FALSE`.
`vector_cat`	Single logical value. If `TRUE`, print a vector of length > 1 using `cat()` instead of `capture.output()`. Otherwise (default `FALSE`) print a vector of length > 1 using `capture.output()`. Names of values are not printed when `TRUE`.

noquote	Single logical value. If TRUE no quote are added to the returned character strings.
sep	Single non null and positive integer representing the number of empty lines added in the output file after the printed data.
safer_check	Single logical value. Perform the "safer" checks? If TRUE, checkings are performed before main code running (see the safer-r project): 1) required functions and related packages effectively present in local R libraries and 2) R classical operators (like "<-") not overwritten by another package because of the R scope. Warning: must be set to FALSE if this function is used inside another "safer" function to avoid pointless multiple checkings.
lib_path	Vector of characters specifying the absolute pathways of the directories containing the required packages for the function, if not in the default directories. Useful when R packages are not installed in the default directories because of lack of admin rights. More precisely, lib_path is passed through the new argument of .libPaths() so that the new library paths are unique(c(new, .Library.site, .Library)). Warning: .libPaths() is restored to the initial paths, after function execution. Ignored if NULL (default): only the pathways specified by the current .libPaths() are used for package calling.
error_text	Single character string used to add information in error messages returned by the function, notably if the function is inside other functions, which is practical for debugging. Example: error_text = " INSIDE <PACKAGE_1>::<FUNCTION_1> INSIDE <PACKAGE_2>::<FUNCTION_2>.". If NULL, converted into "".

Value

Nothing (implementation of an existing file) or a text file.

Author(s)

Gael Millot

Haiding Wang

Yushi Han

See Also

capture.output.

Examples

```
# Warning: these examples may not work well when using the "Run examples" link
# because of a particular environment. Please, copy-paste in a local environment.
# See also https://safer-r.github.io/saferDev/articles/report.html
# Write in the results.txt file of a new temporary folder:
tempo_dir <- base::tempfile(pattern = "dir", tmpdir = base::tempdir(), fileext = "")
base::dir.create(path = tempo_dir, showWarnings = TRUE, recursive = FALSE, mode = "0777")
saferDev::report(data = "THE FOLLOWING VECTOR IS:\n", output = "results.txt", path = tempo_dir,
    append = TRUE, sep = 1)
saferDev::report(data = 1:3, output = "results.txt", path = tempo_dir, append = TRUE,
    rownames_kept = FALSE, vector_cat = FALSE, noquote = FALSE, sep = 2)
```

```
saferDev::report(data = "THE FOLLOWING DATA FRAME IS:\n", output = "results.txt",
    path = tempo_dir, append = TRUE, sep = 1)
saferDev::report(data = base::data.frame(A = 1:3, B = letters[1:3]), output = "results.txt",
    path = tempo_dir, append = TRUE, rownames_kept = FALSE, vector_cat = FALSE,
    noquote = FALSE, sep = 1)
base::cat(base::readLines(con = base::file.path(tempo_dir, "results.txt")), sep = "\n")
```

`super_assign_check` *Superassignment Check*

Description

Verify that the superassignment operators `<<-` and `->>` are not used in a function. Return an error with a table of the lines with problems if any, since these operators are not allowed in safer-r functions.

Usage

```
super_assign_check(x, safer_check = TRUE, lib_path = NULL, error_text = "")
```

Arguments

`x`	A function name, written without quotes and brackets.
`safer_check`	Single logical value. Perform the "safer" checks? If `TRUE`, checkings are performed before main code running (see the safer-r project): 1) required functions and related packages effectively present in local R libraries and 2) R classical operators (like `"<-"`) not overwritten by another package because of the R scope. Warning: must be set to `FALSE` if this function is used inside another "safer" function to avoid pointless multiple checkings.
`lib_path`	Vector of characters specifying the absolute pathways of the directories containing the required packages for the function, if not in the default directories. Useful when R packages are not installed in the default directories because of lack of admin rights. More precisely, `lib_path` is passed through the `new` argument of `.libPaths()` so that the new library paths are `unique(c(new, .Library.site, .Library))`. Warning: `.libPaths()` is restored to the initial paths, after function execution. Ignored if `NULL` (default): only the pathways specified by the current `.libPaths()` are used for package calling.
`error_text`	Single character string used to add information in error messages returned by the function, notably if the function is inside other functions, which is practical for debugging. Example: `error_text = " INSIDE <PACKAGE_1>::<FUNCTION_1> INSIDE <PACKAGE_2>::<FUNCTION_2>."`. If `NULL`, converted into `""`.

Details

Why an error: the superassignment operators `<<-` and `->>` assign in the first environment, among the parent environments of the evaluation environment, where the name already exists, and in the global environment if the name exists nowhere. What is modified thus depends on the environments where the function is defined and called, not on the code of the function itself: it is difficult to guess what the writer had in mind, and nothing can be corrected automatically. The code must be rewritten, for instance by returning the value, or by using `base::assign()` with an explicit `envir` argument.

Reported: every `<<-` and `->>` operator written in the submitted function, including inside the functions defined in the submitted function (e.g., a `FUN` argument of `base::lapply()` or the handlers of `base::tryCatch()`).

Not reported: the `<-`, `->` and `=` assignment operators, and the `<<-` or `->>` writings inside strings, comments or backquoted names (the detection is token-based, not text-based).

Line numbers are indicative. They start at the `<- function` line (`#'` header lines not counted). Compiled functions, like `saferDev::colons_check`, have no comments: their line numbers differ from their source file. A `->>` operator is printed `<<-` in the code of a function without source references (attribute `srcref`).

Value

If no superassignment operator is detected: a message saying that everything seems fine, and `NULL` returned invisibly.

Otherwise: an error, with a table-like message made of one row per reported operator (column 1: line number; column 2: symbol; column 3: code of the line).

Author(s)

Gael Millot

Examples

```
# Warning: these examples may not work well when using the "Run examples" link
# because of a particular environment. Please, copy-paste in a local environment.
# See also https://safer-r.github.io/saferDev/articles/super_assign_check.html
fun1 <- function(x){ y <- x ; base::return(y) }
saferDev::super_assign_check(fun1) # everything fine: NULL returned
fun2 <- function(x){ y <<- x ; base::return(y) }
try(saferDev::super_assign_check(fun2)) # error: <<- reported
# A <<- written in a string or in a comment is not an operator: not reported.
fun3 <- function(x){ y <- "the <<- writing is just a string" ; x } # a <<- in a comment
saferDev::super_assign_check(fun3) # everything fine: NULL returned
```

`true_false_check` *True False Check*

Description

Verify that the `TRUE` and `FALSE` logical constants are always written in full when used as values (argument values, conditions, comparisons, indexing) and never abbreviated as `T` or `F`, which can be overwritten by the user. Return table message of the line with problems and a corrected version of the submitted function.

Usage

```
true_false_check(
  x,
  export = FALSE,
  out_path = ".",
  safer_check = TRUE,
  lib_path = NULL,
  error_text = ""
)
```

Arguments

`x`	A function name, written without quotes and brackets.
`export`	Single logical value. Write the corrected function into a file instead of returning it? If `TRUE`, the file `<function name>_safer.R` is written in `out_path` (e.g., `fun_safer.R` for `x = fun` or `x = pkg::fun`). It contains `fun_safer <- function(...) ...`, so that `base::source()` recreates the corrected function. Nothing is written if nothing has to be corrected. An error is returned if the file already exists, or if `x` is not a function name (e.g., `function(x){x}` or `FUN_LIST$fun`).
`out_path`	Single character string indicating the path of the folder where to write the file when `export = TRUE`. `"."` (default) means the R working directory. Ignored if `export = FALSE`.
`safer_check`	Single logical value. Perform the "safer" checks? If `TRUE`, checkings are performed before main code running (see the safer-r project): 1) required functions and related packages effectively present in local R libraries and 2) R classical operators (like `"<-"`) not overwritten by another package because of the R scope. Warning: must be set to `FALSE` if this function is used inside another "safer" function to avoid pointless multiple checkings.
`lib_path`	Vector of characters specifying the absolute pathways of the directories containing the required packages for the function, if not in the default directories. Useful when R packages are not installed in the default directories because of lack of admin rights. More precisely, `lib_path` is passed through the `new` argument of `.libPaths()` so that the new library paths are `unique(c(new, .Library.site, .Library))`. Warning: `.libPaths()` is restored to the initial

paths, after function execution. Ignored if NULL (default): only the pathways specified by the current .libPaths() are used for package calling.

error_text — Single character string used to add information in error messages returned by the function, notably if the function is inside other functions, which is practical for debugging. Example: error_text = " INSIDE <PACKAGE_1>::<FUNCTION_1> INSIDE <PACKAGE_2>::<FUNCTION_2>.". If NULL, converted into "".

Details

Warning: true_false_check() only checks the T and F symbols used as values: the package association of the function calls is not checked (run saferDev::colons_check() separately for that). Function names written without brackets (e.g., FUN = as.character) or used as function names (e.g., T()) are not reported.

Reported: T or F used as a value, e.g., f(log = T), f(T), if(T), while(T), x == T or x[T].

Not reported: T or F used as a name or assigned, e.g., T <- 1, x <- T, function(T), f(T = 1), df$T, for(T in 1:3), T(), ?T or y ~ T. Strings, comments and backquoted names (`T`) are ignored. If T or F is defined anywhere in the function (argument, loop variable or assignment), none of its uses is reported.

Correction: each reported T is replaced by TRUE and each F by FALSE (base::T also becomes TRUE). The rest of the code is unchanged. The comments and the indentation of the submitted function are kept, in the returned function and in the exported file. Warning: comments are only available when the submitted function carries its source references (attribute srcref). Otherwise (e.g., compiled package functions, or functions created with options(keep.source = FALSE), default of Rscript), the code is the deparsed one: no comments and standard indentation. The returned function is evaluated in a new environment whose parent is the environment of the submitted function: check it before replacing the original function.

Line numbers are indicative. They start at the <- function line (#' header lines not counted). Compiled functions, like saferDev::colons_check, have no comments: their line numbers differ from their source file.

Value

A table-like message is always printed, with one row per reported symbol (column 1: line number; column 2: symbol; column 3: code before the symbol), or a message saying that everything seems fine.

If export = FALSE: the corrected function, or NULL if nothing has to be corrected.

If export = TRUE: NULL, returned invisibly. The corrected function is written in a file (see the export argument).

Author(s)

Gael Millot

Haiding Wang

Yushi Han

Examples

```
# Warning: these examples may not work well when using the "Run examples" link
# because of a particular environment. Please, copy-paste in a local environment.
# See also https://safer-r.github.io/saferDev/articles/true_false_check.html
fun1 <- function(x){ if(x == TRUE){ base::return(x) } }
saferDev::true_false_check(fun1) # everything fine: NULL returned
fun2 <- function(x){ if(x == T){ base::return(x) } }
fun2_safer <- saferDev::true_false_check(fun2) # T reported
fun2_safer # corrected function
# Write the corrected function in the fun2_safer.R file of a new temporary folder:
tempo_dir <- base::tempfile(pattern = "dir", tmpdir = base::tempdir(), fileext = "")
base::dir.create(path = tempo_dir, showWarnings = TRUE, recursive = FALSE, mode = "0777")
saferDev::true_false_check(x = fun2, export = TRUE, out_path = tempo_dir)
```

Index

all_args_here, 2
arg_check, 7

capture.output, *31*
class, *11*
colons_check, 11
curly_bracket_check, 15

env_check, 17
exists, *18*, *25*

findFunction, *25*
for_loop_nb_check, 19

get_message, 22

is_function_here, 24
is_package_here, 26

lambda_check, 27

report, 30
require, *27*

super_assign_check, 32

true_false_check, 34
try, *23*