Package 'precondition'

March 28, 2023

Title Lightweight Precondition, Postcondition, and Sanity Checks

Version 0.1.0

Description Implements fast, safe, and customizable assertions routines, which can be used in place of base::stopifnot().

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.2.0

Imports rlang (>= 1.0.6)

Suggests cli, spelling, testthat (>= 3.1.7), withr

Config/testthat/edition 3

Language en-US

NeedsCompilation yes

Author Taras Zakharko [aut, cre] (<https://orcid.org/0000-0001-7601-8424>)

Maintainer Taras Zakharko <taras.zakharko@gmail.com>

Repository CRAN

Date/Publication 2023-03-28 12:00:05 UTC

R topics documented:

`diagnose_assertion_failure` *Implement a custom assertion*

Description

`diagnose_assertion_failure()` displays customized failure message and diagnosis in assertions such as `precondition()`. This can be used to implement assertion helpers. This function does nothing if invoked outside an assertion (see details). The function `forwarded_arg_label()` looks up a forwarded argument and formats it as a string (used in custom diagnostic messages).

Usage

```
diagnose_assertion_failure(message, ..., .details)

forwarded_arg_label(arg)
```

Arguments

`message`	diagnostic message to show (see `rlang::format_error_bullets()`)
`...`	expressions to diagnose (forwarded to `diagnose_expressions()`)
`.details`	an optional data frame with diagnosis data
`arg`	a forwarded function argument

Details

If invoked as part of an assertion (e.g. `precondition()`), `diagnose_assertion_failure()` provides a custom failure message and diagnosis. If invoked in any other context, the function does nothing. This can be used to implement custom assertions helpers that behave like regular binary predicates (functions) under normal circumstances and generate a customized assertion failure report when used as part of an assertion (see examples).

The first argument to `diagnose_assertion_failure()` is a character vector with a custom failure message. This vector will be formatted as error bullets via`rlang::format_error_bullets()`. Any subsequent argument will be forwarded to `diagnose_assertion_failure()` for diagnosis. For custom diagnosis, the user can supply their own data frame with diagnosis details via optional argument `.details`. The format of this data frame must be identical to one returned by `diagnose_assertion_failure()`.

The function `forwarded_arg_label()` looks up a forwarded expression and formats it as a single string suitable for inclusion in diagnostic messages.

Value

`diagnose_assertion_failure()` always returns `FALSE`.

Examples

```
# returns TRUE if x is a positive, integer, FALSE otherwise
# if invoked as part of an assertion displays a custom failure diagnosis
is_positive_int <- function(x) {
  is.integer(x) && length(x) == 1L && (x > 0) || {
    diagnose_assertion_failure(
      sprintf("`%s` must be a positive integer", forwarded_arg_label(x)),
      {{x}}
    )
  }
}

# for all intends and purposes this is just a regular R function that returns
# TRUE or FALSE
is_positive_int(5L)
is_positive_int(-5L)

# guard to avoid throwing errors
if(FALSE) {

# ... but it will provide custom diagnosis if invoked inside an assertion
precondition(is_positive_int(-5L))

}
```

---

diagnose_expressions *Diagnose expressions and substitute debug markers*

Description

Assertions in the `precondition` package support debug markers to provide user-friendly assertion failure diagnosis. The low-level diagnostic machinery is implemented by `diagnose_expressions()`. Advanced users can make use of this function in their own code or when implementing custom assertion helpers (see `diagnose_assertion_failure()`).

Use single curly braces `{x}` to mark expressions of interest and make them appear as separate entries in the diagnostic output. Use double curly braces `{{x}}` to perform checks on behalf of a parent function and display diagnostics in the context of the parent.

Usage

```
diagnose_expressions(..., .env)
```

Arguments

`...`	expressions to diagnose
`.env`	(advanced) the environment where the diagnosis should be performed

Details

`diagnose_expressions()` supports two kinds of debug markers. Both rely on wrapping expressions in one or more curly braces `{}`.

- wrapping an expression in curly braces (e.g. `{x} > 0`) means that the this expression is of particular interest and should be diagnosed separately. The braces will be removed from the diagnostic output and the wrapped expression will be added as a separate entry in the diagnostic table (note: `diagnose_expressions({x} > 0)` is equivalent to `debug_expressions (x > 0, x)`).
- wrapping a function argument in two curly braces (e.g. ‘arg > 0) means that the argument is being been forwarded from a parent function. This concept of forwarding is borrowed from tidyverse's rlang::embrace-operator. A forwarded argument will be replaced by the original caller expression in the diagnostic output.

`diagnose_expressions()` returns a data frame with one row per diagnosed expression(either supplied as an argument or marked via `{}`) and three columns. The column `expr` is a list of diagnosed expressions, with debug markers processed and substituted. The column `eval_result` is a list of evaluated results for each diagnosed expressions. The column `is_error` is a logical vector where value of `TRUE` indicates that an error occurred when evaluating the respective expression. In this case the corresponding value of `eval_result` will capture the error condition.

Note that expressions or their parts might be evaluated more then once during diagnosis. Side effects in diagnosed expressions can lead to unexpected behavior.

Value

a data frame with diagnostic information

Examples

```
x <- 10
diagnose_expressions({x} > 0, {x} > 15)

helper <- function(arg) {
   cat(sprintf("`arg` is forwarded `%s`\n", forwarded_arg_label(arg)))
   diagnose_expressions({{{arg}}} > 0)
}
fun <- function(x) {
  helper(x)
}
fun(10)
```

`fatal_error` *Signal a fatal error (and optionally terminate the program execution)*

Description

`fatal_error()` is equivalent to the base function `base::stop()`, except it is intended to signal critical errors where recovery is impossible or unfeasible.

Fatal errors are signaled via `rlang::abort()` with the class `precondition/fatal_error`. The option `fatal_error_action` controls the behavior of the fatal errors.

- `option(fatal_error_action = "inform")` will display a warning if a fatal error has been prevented from bubbling up to the # user(either via 'tryCatch() or some other error handling mechanism). This is the default setting and will draw user's attention to a fatal error occurring.
- `option(fatal_error_action = "none")` will make fatal errors behave like regular R error conditions. Use this if your code contains custom logic for handling fatal errors.
- `option(fatal_error_action = 'terminate')` will immediately the program execution without saving the workspace or running finalizers when a fatal error occurs.

Usage

```
fatal_error(bullets, ...)
```

Arguments

`bullets`	a character vector containing the error message, can be formatted in the style of `rlang::format_error_bullets()`
`...`	reserved for future use

Details

`fatal_error()` is used in `sanity_check()` to report critical assertion failures.

`precondition` *Pre- and postcondition checking (assertions)*

Description

The assertions described here are similar in functionality to the base R function `base::stopifnot()`, but focusing on better diagnostics, safer behavior, and customizability.

- `precondition()` fails with diagnosis if its arguments do not evaluate as `TRUE`. Use this assertion function to check function arguments or data inputs against code invariants.
- `postcondition()` is as above, but the assertion is performed when the calling function successfully returns. Use this assertion to check that the function has produced a well-formed result (via base::returnValue ()) or behavior.
- `sanity_check()` is as above, but the program execution will immediately terminate via `fatal_error()`, bypassing R's error-checking mechanisms. Use this predicate to validate critical internal assumptions your code relies upon. Failing a sanity check means that your program contains an unrecoverable logical error and cannot reasonably continue execution.

To facilitate debugging, the assertions used with these functions can be enhanced with debug markers. This enables informative error messages and makes it easier to understand why the assertion has failed. First, assertions can include custom informative messages, supplied via literal string arguments to the assertion function. Second, key parts of the assertion expression can be wrapped in curly braces(e.g. `{x} > 0`). If the assertion fails, the values marked in such way will be diagnosed and displayed as a separate entry in the error message. See the examples on how to use these features and the details section how to implement even more custom functionality.

Under certain circumstances these predicates might evaluate the assertion expression multiple times. Beware of combining them with side effects.

Usage

```
precondition(...)

postcondition(...)

sanity_check(...)
```

Arguments

...	one or more expressions to check (with optional assertion messages)

Details

A precondition is an assertion that specifies a set of conditions that must be true in order for the execution to proceed in a meaningful way. This is usually conditioned on the user input or environment in some way. A postcondition is an assertion that must be true if a function has executed in a meaningful way. Pre- and postconditions explicitly state the contract of a function and make it easier to debug correct function usage. Note: `postcondition(check)` is similar to `on.exit(stopifnot(check))`, except that the postcondition will not be checked if an error occurred during function execution.

A sanity check is an assertion that specifies a set of conditions that the program implicitly assumes to be true. A sanity check failure means that the core logic of the program is broken and error recovery is either impossible or not meaningful. Sanity checks are used to test the internal logic of your code and will result in an immediate program termination if failed (via fatal_error).

The arguments to these assertion functions are either expressions that should evaluate to `TRUE` or literal string constants containing informative messages (e.g. `sanity_check("x is not NULL", !is.null(x))`). Should the assertion fail, the provided message will be displayed. Note that this message *must* be a string literal, you cannot compute it or use a variable. The following will not work correctly: 'sanity_check(paste0("x is not", "NULL"), !is.null(x)).

Assertion expression support debug-markers. See `diagnose_assertion_failure()` on how to implement custom assertion helpers.

Value

`TRUE` on assertion success, raises an error of class `precondition/assertion_error` on assertion failure

Examples

```
# These examples are guarded to avoid throwing errors
if (FALSE) {

# function contract is accepting a positive value and returning up to 20
fun <- function(x) {
  precondition("`x` should be positive", {x} > 0)
  postcondition(returnValue() <= 20)

  out <- x*2
  sanity_check("twice `x` is larger than `x`", {out} > {x})

  out
}

fun(5)
fun(0)
fun(10)
}
```

Index