fuzzr coverage - 96.82%

- Files
- Source

File	Lines	Relevant	Covered	Missed	Hits / Line	Coverage
R/evaluators.R	275	111	104	7	2855	93.69%
R/inputs.R	199	71	71	0	18	100.00%
R/outputs.R	108	38	38	0	2726	100.00%

```
# Exported functions ----

#' Summarize fuzz test results as a data frame
#'
#' @param x Object returned by \code{\link{fuzz_function}}.
#' @param ... Additional arguments to be passed to or from methods.
#' @param delim The delimiter to use for fields like \code{messages} or
#'    \code{warnings} in which there may be multiple results.
#'
#' @return A data frame with the following columns: \describe{
#'   \item{\code{fuzz_input}}{The name of the fuzz test performed.}
#'   \item{\code{output}}{Delimited outputs to the command line from the process, if applicable.}
#'   \item{\code{messages}}{Delimited messages, if applicable.}
#'   \item{\code{warnings}}{Delimited warnings, if applicable.}
#'   \item{\code{errors}}{Error returned, if applicable.}
#'   \item{\code{value_classes}}{Delimited classes of the object returned by the
#'    function, if applicable}
#'   \item{\code{results_index}}{Index of \code{x} from which the summary was
#'    produced.}
#'   }
#'
#' @export
as.data.frame.fuzz_results <- function(x, ..., delim = "; ") {
```

```
8x       ldf <- purrr::map(x, parse_fuzz_result_concat, delim = delim)
8x       df <- do.call("rbind", ldf)
8x       df[["results_index"]] <- seq_along(x)
8x       df
       }

       #' Access individual fuzz test results
       #'
       #' @param fr \code{fuzz_results} object
       #' @param index The test index (by position) to access. Same as the
       #'   \code{results_index} in the data frame returned by
       #'   \code{\link{as.data.frame.fuzz_results}}.
       #' @param ... Additional arguments must be named regex patterns that will be used to match against test names. The names of the patterns must match the function argument name(s) whose test names you wish to match.
       #' @name fuzz_results
       NULL

       #' @describeIn fuzz_results Access the object returned by the fuzz test
       #' @export
       fuzz_value <- function(fr, index = NULL, ...) {
10x      res <- search_results(fr, index, ...)
9x       res[["test_result"]][["value"]]
       }

       #' @describeIn fuzz_results Access the call used for the fuzz test
       #' @export
       fuzz_call <- function(fr, index = NULL, ...) {
9x       res <- search_results(fr, index, ...)
8x       res[["test_result"]][["call"]]
       }

       # Internal functions ----

       # For each result, create a one-row data frame of test names, outputs, messages,
       # warnings, errors, and result classes.
       parse_fuzz_result_concat <- function(fr, delim) {

4689x    dfr <- as.data.frame(fr[["test_name"]], stringsAsFactors = FALSE)

4689x    elem_collapse <- function(elem) {
18756x     if (is.null(elem)) {
13919x       return(NA_character_)
4689x      } else {
4837x        paste(elem, collapse = delim)
4689x      }
4689x    }

4689x    dfr[["output"]] <- elem_collapse(fr[["test_result"]][["output"]])
4689x    dfr[["messages"]] <- elem_collapse(fr[["test_result"]][["messages"]])
4689x    dfr[["warnings"]] <- elem_collapse(fr[["test_result"]][["warnings"]])
4689x    dfr[["errors"]] <- elem_collapse(fr[["test_result"]][["errors"]])

         # If no object was returned by the function under given test conditions,
         # record value as NA in the data frame
4689x    dfr[["result_classes"]] <- ifelse(
4689x      is.null(fr[["test_result"]][["value"]]),
4689x      NA_character_,
4689x      paste(class(fr[["test_result"]][["value"]]), collapse = delim))

4689x    dfr
       }

       # Find elements of the search results list
       search_results <- function(fr, index, ...) {
19x      assertthat::assert_that(inherits(fr, "fuzz_results"))

```

```
        # value supplied to index takes priority
  19x   if (!is.null(index)) {
   5x     assertthat::assert_that(assertthat::is.count(index) && index <= length(fr))
   5x     res <- fr[[index]]
        } else {

          # if no index, then check based on test name
  14x     .dots <- list(...)
  14x     purrr::walk(.dots, function(p) assertthat::assert_that(assertthat::is.string(p)))

  14x     assertthat::assert_that(all(names(.dots) %in% names(fr[[1]][["test_name"]])))

  12x     res <- purrr::detect(fr, function(el) {
 236x       all(purrr::map2_lgl(.dots, names(.dots), function(p, n) grepl(p, x = el[["test_name"]][[n]])))
  12x     })
  12x     if (length(res) == 0)
   2x       warning("Zero matches found.")
        }
  17x   res
      }
```

```
      # Exported functions ----

      #' Fuzz-test a function
      #'
      #' Evaluate how a function responds to unexpected or non-standard inputs.
      #'
      #' \code{fuzz_function} provides a simple interface to fuzz test a single
      #' argument of a function by passing the function, name of the argument, static
      #' values of other required arguments, and a named list of test values.
      #'
      #' \code{p_fuzz_function} takes a nested list of arguments paired with lists of
      #' tests to run on each argument, and will evaluate every combination of
      #' argument and provided test.
      #'
      #' @note The user will be asked to confirm before proceeding if the combinations
      #'   of potential tests exceeds 500,000.
      #'
      #' @param fun A function.
      #' @param arg_name Quoted name of the argument to fuzz test.
      #' @param ... Other non-dynamic arguments to pass to \code{fun}. These will be
      #'   repeated for every one of the \code{tests}.
      #' @param tests Which fuzz tests to run. Accepts a named list of inputs,
      #'   defaulting to \code{\link{test_all}}.
      #' @param check_args Check if \code{arg_name} and any arguments passed as
      #'   \code{...} are accepted by \code{fun}. Set to \code{FALSE} if you need to
      #'   pass arguments to a function that accepts arguments via \code{...}.
      #' @param progress Show a progress bar while running tests?
      #'
      #' @return A \code{fuzz_results} object.
      #'
      #' @seealso \code{\link{fuzz_results}} and
      #'   \code{\link{as.data.frame.fuzz_results}} to access fuzz test results.
      #'
      #' @export
      #' @examples
      #' # Evaluate the 'formula' argument of lm, passing additional required variables
      #' fr <- fuzz_function(lm, "formula", data = iris)
      #'
      #' # When evaluating a function that takes ..., set check_args to FALSE
      #' fr <- fuzz_function(paste, "x", check_args = FALSE)
      fuzz_function <- function(fun, arg_name, ..., tests = test_all(), check_args = TRUE, progress = interactive()) {

  15x   fuzz_asserts(fun, check_args, progress)
  12x   attr(fun, "fun_name") <- deparse(substitute(fun))
  12x   assertthat::assert_that(is_named_l(tests))
```

```

       # Collect the unevaluated names of variables passed to the original call,
       # keeping only those passed in as ... These will be used in the named list
       # passed to p_fuzz_function
  9x   dots_call_names <- purrr::map_chr(as.list(match.call()), deparse)
  9x   .dots = list(...)
  9x   dots_call_names <- dots_call_names[names(.dots)]

       # Check that arg_name is a string, and the tests passed is a named list
  9x   assertthat::assert_that(assertthat::is.string(arg_name), is_named_l(tests))

       # Check that arguments passed to fun actually exist in fun
  9x   if (check_args)
  8x     assertthat::assert_that(
  8x       assertthat::has_args(fun, arg_name),
  8x       assertthat::has_args(fun, names(.dots)))

       # Construct a list of arguments for p_fuzz_function, with tests assigned to
       # arg_name, and the values passed via ... saved as lists named after their
       # deparsed variable names.
  7x   test_args <- c(
  7x     purrr::set_names(list(tests), arg_name),
  7x     purrr::map2(.dots, dots_call_names, function(x, y) purrr::set_names(list(x), y)))

  7x   p_fuzz_function(fun, .l = test_args, check_args = check_args, progress = progress)
     }

     #' @rdname fuzz_function
     #' @param .l A named list of tests.
     #' @export
     #' @examples
     #'
     #' # Pass tests to multiple arguments via a named list
     #' test_args <- list(
     #'    data = test_df(),
     #'    subset = test_all(),
     #'    # Specify custom tests with a new named list
     #'    formula = list(all_vars = Sepal.Length ~ ., one_var = mpg ~ .))
     #' fr <- p_fuzz_function(lm, test_args)
     p_fuzz_function <- function(fun, .l, check_args = TRUE, progress = interactive()) {

 21x   fuzz_asserts(fun, check_args, progress)
 18x   if (is.null(attr(fun, "fun_name"))) {
 11x     fun_name <- deparse(substitute(fun))
       } else {
  7x     fun_name <- attr(fun, "fun_name")
       }

 18x   if (check_args)
 16x     assertthat::assert_that(assertthat::has_args(fun, names(.l)))

       # Ensure .l is a named list of named lists
 18x   is_named_ll(.l)

       # Replace any NULL test values with .null alias.
 13x   .l <- purrr::map(.l, function(li) {
 27x     purrr::map(li, function(lli) {
616x       if (is.null(lli)) {
 12x         .null
 27x       } else {
604x         lli
 27x       }
 27x     })
 13x   })

```

```
         # Warn if combination of tests is potentially massive
   13x   num_tests <- purrr::reduce(purrr::map_int(.l, length), `*`)
   13x   if (num_tests >= 500000) {
    1x     m <- utils::menu(choices = c("Yes", "No"), title = paste("The supplied tests have", num_tests, "combinations, which may be prohibitively large to calculate. Attempt to proceed?"))
     !     if (m != 1)
     !       return(NULL)
         }

         # Generate the list of tests to be done
   12x   test_list <- named_cross_n(.l)

         # After crossing, restore NULL test values
   12x   test_list <- purrr::modify_depth(test_list, 3, function(x) {
30510x       if (inherits(x, what = "fuzz-null")) {
  203x         NULL
   12x       } else {
30307x         x
   12x       }
   12x     })

         # Create a progress bar, if called for
   12x   if (progress) {
     !     pb <- progress::progress_bar$new(
     !       format = " running tests [:bar] :percent eta: :eta",
     !       total = length(test_list), clear = FALSE, width = 60)
     !     pb$tick(0)
         }

         # For each test combination...
   12x   fr <- purrr::map(
   12x     test_list, function(x) {
     !       if (exists("pb")) pb$tick()

             # Extract values for testing
 5208x       arglist <- purrr::map(x, getElement, name = "test_value")

             # Extract names of tests
 5208x       testnames <- purrr::map(x, getElement, name = "test_name")

             # Create a result list with both the results of try_fuzz, as well as a
             # named list pairing argument names with the test names supplied to them
             # for this particular round
 5208x       res <- list(test_result = try_fuzz(fun = fun, fun_name = fun_name,
 5208x                                          all_args = arglist))
 5208x       res[["test_name"]] <- testnames
 5208x       res
   12x     })

   12x   structure(fr, class = "fuzz_results")
       }

       # Internal functions ----

       # Pass NULL as a test value
       #
       # Because it is difficult to work with NULLs in lists as required by most of
       # the fuzzr package, this function works as an alias to pass NULL values to
       # function arguments for testing.
       .null <- structure(list(), class = "fuzz-null")

       # This set of assertions need to be checked for both functions
       fuzz_asserts <- function(fun, check_args, progress) {
   36x   assertthat::assert_that(
   36x     is.function(fun), assertthat::is.flag(check_args),
   36x     assertthat::is.flag(progress))
```

```
       }

       # Is a list named, and is each of its elements also a named list?
       is_named_ll <- function(l) {
18x      assertthat::assert_that(is.list(l), is_named(l))
16x      purrr::walk(l, function(x) assertthat::assert_that(is.list(x), is_named(x)))
       }

       # Is every element of a list named?
       is_named_l <- function(l) {
21x      is.list(l) & is_named(l)
       }

       assertthat::on_failure(is_named_l) <- function(call, env) {
         "Not a named list."
       }

       # Check that object has no blank names
       is_named <- function(x) {
77x      nm <- names(x)
77x      !is.null(nm) & all("" != nm)
       }

       assertthat::on_failure(is_named) <- function(call, env) {
         "Not a completely-named object."
       }

       # Cross a list of named lists
       named_cross_n <- function(ll) {

         # Cross the values of the list...
12x      crossed_values <- purrr::cross(ll)
         # ... and then cross the names
12x      crossed_names <- purrr::cross(purrr::map(ll, names))

         # Then map through both values and names in order to
12x      purrr::map2(crossed_values, crossed_names, function(x, y) {
5208x      purrr::map2(x, y, function(m, n) {
15255x       list(
15255x         test_name = n,
15255x         test_value = m
15255x       )
5208x      })
12x      })
       }

       # Custom tryCatch/withCallingHandlers function to catch messages, warnings, and
       # errors along with any values returned by the expression. Returns a list of
       # value, messages, warnings, and errors.
       try_fuzz <- function(fun, fun_name, all_args) {

5208x    call <- list(fun = fun_name, args = all_args)
5208x    messages <- NULL
5208x    output <- NULL
5208x    warnings <- NULL
5208x    errors <- NULL

5208x    message_handler <- function(c) {
92x        messages <<- c(messages, conditionMessage(c))
92x        invokeRestart("muffleMessage")
5208x    }

5208x    warning_handler <- function(c) {
132x       warnings <<- c(warnings, conditionMessage(c))
132x       invokeRestart("muffleWarning")
5208x    }

5208x    error_handler <- function(c) {
5018x      errors <<- c(errors, conditionMessage(c))
5018x      return(NULL)
5208x    }

         # Little trick: that first tryCatch() will return values from the expression
         # to the "value" index in this list, but will pass errors to error_handler
         # (which returns NULL "value", incidentally.) In the event of messages or
         # warnings, handling is passed up to withCallingHandlers, which passes them
         # down again to message_handler or warning_handler, respectively. Once the
         # expression is done evaluating, messages, warnings, and errors are assigned
         # to the list, which is returned as the final result of try_fuzz

5208x    output <- utils::capture.output({
5208x      value <- withCallingHandlers(
5208x        tryCatch(do.call(fun, args = all_args), error = error_handler),
5208x        message = message_handler,
5208x        warning = warning_handler
5208x      )}, type = "output")

5208x    if (length(output) == 0) {
5162x      output <- NULL
         }

5208x    list(
5208x      call = call,
5208x      value = value,
5208x      output = output,
5208x      messages = messages,
5208x      warnings = warnings,
5208x      errors = errors
5208x    )
       }
```

```
       # Data types ----

       #' Fuzz test inputs
       #'
       #' Each \code{test_all} returns a named list that concatenates all the available
       #' tests specified below.
       #'
       #' @export
       test_all <- function() {
15x      c(test_char(), test_int(), test_dbl(), test_fctr(), test_lgl(), test_date(),
15x        test_raw(), test_df(), test_null())
       }

       #' @describeIn test_all Character vectors \itemize{
       #'   \item \code{char_empty}: \code{character(0)}
       #'   \item \code{char_single}: \code{"a"}
       #'   \item \code{char_single_blank}: \code{""}
       #'   \item \code{char_multiple}: \code{c("a", "b", "c")}
       #'   \item \code{char_multiple_blank}: \code{c("a", "b", "c", "")}
       #'   \item \code{char_with_na}: \code{c("a", "b", NA)}
       #'   \item \code{char_single_na}: \code{NA_character_}
       #'   \item \code{char_all_na}: \code{c(NA_character_, NA_character_, NA_character_)}
       #' }
       #' @export
       test_char <- function() {
21x      list(
21x        char_empty = character(),
21x        char_single = letters[1],
21x        char_single_blank = "",
21x        char_multiple = letters[1:3],
  21x      char_multiple_blank = c(letters[1:3], ""),
  21x      char_with_na = c(letters[1:2], NA),
  21x      char_single_na = NA_character_,
  21x      char_all_na = rep(NA_character_, 3)
  21x    )
       }

       #' @describeIn test_all Integer vectors \itemize{
       #'  \item \code{int_empty}: \code{integer(0)}
       #'  \item \code{int_single}: \code{1L}
       #'  \item \code{int_multiple}: \code{1:3}
       #'  \item \code{int_with_na}: \code{c(1L, 2L, NA)}
       #'  \item \code{int_single_na}: \code{NA_integer_}
       #'  \item \code{int_all_na}: \code{c(NA_integer_, NA_integer_, NA_integer_)}
       #' }
       #' @export
       test_int <- function() {
  17x    list(
  17x      int_empty = integer(),
  17x      int_single = 1L,
  17x      int_multiple = 1L:3L,
  17x      int_with_na = c(1L:2L, NA),
  17x      int_single_na = NA_integer_,
  17x      int_all_na = rep(NA_integer_, 3)
  17x    )
       }

       #' @describeIn test_all Double vectors \itemize{
       #'  \item \code{dbl_empty}: \code{numeric(0)}
       #'  \item \code{dbl_single}: \code{1.5}
       #'  \item \code{dbl_mutliple}: \code{c(1.5, 2.5, 3.5)}
       #'  \item \code{dbl_with_na}: \code{c(1.5, 2.5, NA)}
       #'  \item \code{dbl_single_na}: \code{NA_real_}
       #'  \item \code{dbl_all_na}: \code{c(NA_real_, NA_real_, NA_real_)}
       #' }
       #' @export
       test_dbl <- function() {
  18x    list(
  18x      dbl_empty = double(),
  18x      dbl_single = 1.5,
  18x      dbl_mutliple = 1:3 + 0.5,
  18x      dbl_with_na = c(1:2 + 0.5, NA),
  18x      dbl_single_na = NA_real_,
  18x      dbl_all_na = rep(NA_real_, 3)
  18x    )
       }

       #' @describeIn test_all Logical vectors \itemize{
       #'  \item \code{lgl_empty}: \code{logical(0)}
       #'  \item \code{lgl_single}: \code{TRUE}
       #'  \item \code{lgl_mutliple}: \code{c(TRUE, FALSE, FALSE)}
       #'  \item \code{lgl_with_na}: \code{c(TRUE, NA, FALSE)}
       #'  \item \code{lgl_single_na}: \code{NA}
       #'  \item \code{lgl_all_na}: \code{c(NA, NA, NA)}
       #' }
       #' @export
       test_lgl <- function() {
  17x    list(
  17x      lgl_empty = logical(),
  17x      lgl_single = TRUE,
  17x      lgl_mutliple = c(TRUE, FALSE, FALSE),
  17x      lgl_with_na = c(TRUE, NA, FALSE),
  17x      lgl_single_na = NA,
  17x      lgl_all_na = rep(NA, 3)
  17x    )
```

```
     }

     #' @describeIn test_all Factor vectors \itemize{
     #'  \item \code{fctr_empty}: \code{structure(integer(0), .Label = character(0), class = "factor")}
     #'  \item \code{fctr_single}: \code{structure(1L, .Label = "a", class = "factor")}
     #'  \item \code{fctr_multiple}: \code{structure(1:3, .Label = c("a", "b", "c"), class = "factor")}
     #'  \item \code{fctr_with_na}: \code{structure(c(1L, 2L, NA), .Label = c("a", "b"), class = "factor")}
     #'  \item \code{fctr_missing_levels}: \code{structure(1:3, .Label = c("a", "b", "c", "d"), class = "factor")}
     #'  \item \code{fctr_single_na}: \code{structure(NA_integer_, .Label = character(0), class = "factor")}
     #'  \item \code{fctr_all_na}: \code{structure(c(NA_integer_, NA_integer_, NA_integer_), .Label = character(0), class = "factor")}
     #' }
     #' @export
     test_fctr <- function() {
17x    list(
17x      fctr_empty = factor(),
17x      fctr_single = as.factor("a"),
17x      fctr_multiple = as.factor(c("a", "b", "c")),
17x      fctr_with_na = as.factor(c("a", "b", NA)),
17x      fctr_missing_levels = factor(c("a", "b", "c"), levels = letters[1:4]),
17x      fctr_single_na = factor(NA),
17x      fctr_all_na = factor(rep(NA, 3))
17x    )
     }

     #' @describeIn test_all Date vectors \itemize{
     #'  \item \code{date_single}: \code{as.Date("2001-01-01")}
     #'  \item \code{date_multiple}: \code{as.Date(c("2001-01-01", "1950-05-05"))}
     #'  \item \code{date_with_na}: \code{as.Date(c("2001-01-01", NA, "1950-05-05"))}
     #'  \item \code{date_single_na}: \code{as.Date(NA_integer_, origin = "1971-01-01")}
     #'  \item \code{date_all_na}: \code{as.Date(rep(NA_integer_, 3), origin = "1971-01-01")}
     #' }
     #' @export
     test_date <- function() {
18x    list(
18x      date_single = as.Date("2001-01-01"),
18x      date_multiple = as.Date(c("2001-01-01", "1950-05-05")),
18x      date_with_na = as.Date(c("2001-01-01", NA, "1950-05-05")),
18x      date_single_na = as.Date(NA_integer_, origin = "1971-01-01"),
18x      date_all_na = as.Date(rep(NA_integer_, 3), origin = "1971-01-01")
18x    )
     }

     #' @describeIn test_all Raw vectors \itemize{
     #'  \item \code{raw_empty}: \code{raw(0)}
     #'  \item \code{raw_char}: \code{as.raw(0x62)},
     #'  \item \code{raw_na}: \code{charToRaw(NA_character_)}
     #' }
     #' @export
     test_raw <- function() {
17x    list(
17x      raw_empty = raw(),
17x      raw_char = charToRaw("b"),
17x      raw_na = charToRaw(NA_character_)
17x    )
     }

     #' @describeIn test_all Data frames \itemize{
     #'   \item \code{df_complete}: \code{datasets::iris}
     #'   \item \code{df_empty}: \code{data.frame(NULL)}
     #'   \item \code{df_one_row}: \code{datasets::iris[1, ]}
     #'   \item \code{df_one_col}: \code{datasets::iris[ ,1]}
     #'   \item \code{df_with_na}: \code{iris} with several NAs added to each column.
     #' }
     #' @export
     test_df <- function() {
```

```
18x    iris_na <- datasets::iris
18x    iris_na[c(1, 10, 100), 1] <- NA
18x    iris_na[c(5, 15, 150), 3] <- NA
18x    iris_na[c(7, 27, 75), 5] <- NA

18x    list(
18x      df_complete = datasets::iris,
18x      df_empty = data.frame(NULL),
18x      df_one_row = datasets::iris[1, ],
18x      df_one_col = datasets::iris[ ,1],
18x      df_with_na = iris_na
18x    )
     }

     #' @describeIn test_all Null value \itemize{
     #'  \item \code{null_value}: \code{NULL}
     #' }
     #' @export
     test_null <- function() {
17x    list(
17x      null_value = NULL
17x    )
     }

     # Development utility function ----

     # This is a non-exported, non-checked function (hence it's being commented out)
     # to be used to quickly generate the \itemize{...} sections of documentation for
     # vector-based tests. NOTE do not use the verbatim results if they are too
     # lengthy.

     # doc_test <- function(test) {
     #   tnames <- names(test)
     #   tval <- purrr::map_chr(test, deparse)
     #   clipr::write_clip(
     #     c("\\itemize{",
     #     paste0("#'  \\item \\code{", tnames, "}: \\code{", tval, "}", collapse = "\n"),
     #     "#' }"))
     # }
```