fuzzr coverage - 0.00%

- Files
- Source

File		Lines	Relevant		Covered	Missed	Hits / Line	Coverage
R/evaluators.R	275	111	0	111	0	0.00%		
R/inputs.R	199	71	0	71	0	0.00%		
R/outputs.R	108	38	0	38	0	0.00%		

```
1  # Exported functions ----
2
3  #' Summarize fuzz test results as a data frame
4  #'
5  #' @param x Object returned by \code{\link{fuzz_function}}.
6  #' @param ... Additional arguments to be passed to or from methods.
7  #' @param delim The delimiter to use for fields like \code{messages} or
8  #'    \code{warnings} in which there may be multiple results.
9  #'
10 #' @return A data frame with the following columns: \describe{
11 #'   \item{\code{fuzz_input}}{The name of the fuzz test performed.}
12 #'   \item{\code{output}}{Delimited outputs to the command line from the process, if applicable.}
13 #'   \item{\code{messages}}{Delimited messages, if applicable.}
14 #'   \item{\code{warnings}}{Delimited warnings, if applicable.}
15 #'   \item{\code{errors}}{Error returned, if applicable.}
16 #'   \item{\code{value_classes}}{Delimited classes of the object returned by the
17 #'    function, if applicable}
18 #'   \item{\code{results_index}}{Index of \code{x} from which the summary was
19 #'    produced.}
20 #'   }
21 #'
22 #' @export
```

```
as.data.frame.fuzz_results <- function(x, ..., delim = "; ") {
  ldf <- purrr::map(x, parse_fuzz_result_concat, delim = delim)
  df <- do.call("rbind", ldf)
  df[["results_index"]] <- seq_along(x)
  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, ...) {
  res <- search_results(fr, index, ...)
  res[["test_result"]][["value"]]
}

#' @describeIn fuzz_results Access the call used for the fuzz test
#' @export
fuzz_call <- function(fr, index = NULL, ...) {
  res <- search_results(fr, index, ...)
  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) {

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

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

  dfr[["output"]] <- elem_collapse(fr[["test_result"]][["output"]])
  dfr[["messages"]] <- elem_collapse(fr[["test_result"]][["messages"]])
  dfr[["warnings"]] <- elem_collapse(fr[["test_result"]][["warnings"]])
  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
  dfr[["result_classes"]] <- ifelse(
    is.null(fr[["test_result"]][["value"]]),
    NA_character_,
    paste(class(fr[["test_result"]][["value"]]), collapse = delim))

  dfr
}

# Find elements of the search results list
search_results <- function(fr, index, ...) {
```

```
!   assertthat::assert_that(inherits(fr, "fuzz_results"))

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

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

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

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

```

```
43 !   fuzz_asserts(fun, check_args, progress)
44 !   attr(fun, "fun_name") <- deparse(substitute(fun))
45 !   assertthat::assert_that(is_named_l(tests))
46
47     # Collect the unevaluated names of variables passed to the original call,
48     # keeping only those passed in as ... These will be used in the named list
49     # passed to p_fuzz_function
50 !   dots_call_names <- purrr::map_chr(as.list(match.call()), deparse)
51 !   .dots = list(...)
52 !   dots_call_names <- dots_call_names[names(.dots)]
53
54     # Check that arg_name is a string, and the tests passed is a named list
55 !   assertthat::assert_that(assertthat::is.string(arg_name), is_named_l(tests))
56
57     # Check that arguments passed to fun actually exist in fun
58 !   if (check_args)
59 !     assertthat::assert_that(
60 !       assertthat::has_args(fun, arg_name),
61 !       assertthat::has_args(fun, names(.dots)))
62
63     # Construct a list of arguments for p_fuzz_function, with tests assigned to
64     # arg_name, and the values passed via ... saved as lists named after their
65     # deparsed variable names.
66 !   test_args <- c(
67 !     purrr::set_names(list(tests), arg_name),
68 !     purrr::map2(.dots, dots_call_names, function(x, y) purrr::set_names(list(x), y)))
69
70 !   p_fuzz_function(fun, .l = test_args, check_args = check_args, progress = progress)
71   }
72
73   #' @rdname fuzz_function
74   #' @param .l A named list of tests.
75   #' @export
76   #' @examples
77   #'
78   #' # Pass tests to multiple arguments via a named list
79   #' test_args <- list(
80   #'    data = test_df(),
81   #'    subset = test_all(),
82   #'    # Specify custom tests with a new named list
83   #'    formula = list(all_vars = Sepal.Length ~ ., one_var = mpg ~ .))
84   #' fr <- p_fuzz_function(lm, test_args)
85   p_fuzz_function <- function(fun, .l, check_args = TRUE, progress = interactive()) {
86
87 !   fuzz_asserts(fun, check_args, progress)
88 !   if (is.null(attr(fun, "fun_name"))) {
89 !     fun_name <- deparse(substitute(fun))
90     } else {
91 !     fun_name <- attr(fun, "fun_name")
92     }
93
94 !   if (check_args)
95 !     assertthat::assert_that(assertthat::has_args(fun, names(.l)))
96
97     # Ensure .l is a named list of named lists
98 !   is_named_ll(.l)
99
100    # Replace any NULL test values with .null alias.
101 !  .l <- purrr::map(.l, function(li) {
102 !    purrr::map(li, function(lli) {
103 !      if (is.null(lli)) {
104 !        .null
105 !      } else {
106 !        lli
```

```
107 !       }
108 !     })
109 !   })
110
111     # Warn if combination of tests is potentially massive
112 !   num_tests <- purrr::reduce(purrr::map_int(.l, length), `*`)
113 !   if (num_tests >= 500000) {
114 !     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?"))
115 !     if (m != 1)
116 !       return(NULL)
117     }
118
119     # Generate the list of tests to be done
120 !   test_list <- named_cross_n(.l)
121
122     # After crossing, restore NULL test values
123 !   test_list <- purrr::modify_depth(test_list, 3, function(x) {
124 !       if (inherits(x, what = "fuzz-null")) {
125 !         NULL
126 !       } else {
127 !         x
128 !       }
129 !     })
130
131     # Create a progress bar, if called for
132 !   if (progress) {
133 !     pb <- progress::progress_bar$new(
134 !       format = " running tests [:bar] :percent eta: :eta",
135 !       total = length(test_list), clear = FALSE, width = 60)
136 !     pb$tick(0)
137     }
138
139     # For each test combination...
140 !   fr <- purrr::map(
141 !     test_list, function(x) {
142 !       if (exists("pb")) pb$tick()
143
144         # Extract values for testing
145 !       arglist <- purrr::map(x, getElement, name = "test_value")
146
147         # Extract names of tests
148 !       testnames <- purrr::map(x, getElement, name = "test_name")
149
150         # Create a result list with both the results of try_fuzz, as well as a
151         # named list pairing argument names with the test names supplied to them
152         # for this particular round
153 !       res <- list(test_result = try_fuzz(fun = fun, fun_name = fun_name,
154 !                                          all_args = arglist))
155 !       res[["test_name"]] <- testnames
156 !       res
157 !     })
158
159 !   structure(fr, class = "fuzz_results")
160   }
161
162   # Internal functions ----
163
164   # Pass NULL as a test value
165   #
166   # Because it is difficult to work with NULLs in lists as required by most of
167   # the fuzzr package, this function works as an alias to pass NULL values to
168   # function arguments for testing.
169   .null <- structure(list(), class = "fuzz-null")
170
```

```
171   # This set of assertions need to be checked for both functions
172   fuzz_asserts <- function(fun, check_args, progress) {
173 !   assertthat::assert_that(
174 !     is.function(fun), assertthat::is.flag(check_args),
175 !     assertthat::is.flag(progress))
176   }
177
178   # Is a list named, and is each of its elements also a named list?
179   is_named_ll <- function(l) {
180 !   assertthat::assert_that(is.list(l), is_named(l))
181 !   purrr::walk(l, function(x) assertthat::assert_that(is.list(x), is_named(x)))
182   }
183
184   # Is every element of a list named?
185   is_named_l <- function(l) {
186 !   is.list(l) & is_named(l)
187   }
188
189   assertthat::on_failure(is_named_l) <- function(call, env) {
190     "Not a named list."
191   }
192
193   # Check that object has no blank names
194   is_named <- function(x) {
195 !   nm <- names(x)
196 !   !is.null(nm) & all("" != nm)
197   }
198
199   assertthat::on_failure(is_named) <- function(call, env) {
200     "Not a completely-named object."
201   }
202
203   # Cross a list of named lists
204   named_cross_n <- function(ll) {
205
206     # Cross the values of the list...
207 !   crossed_values <- purrr::cross(ll)
208     # ... and then cross the names
209 !   crossed_names <- purrr::cross(purrr::map(ll, names))
210
211     # Then map through both values and names in order to
212 !   purrr::map2(crossed_values, crossed_names, function(x, y) {
213 !     purrr::map2(x, y, function(m, n) {
214 !       list(
215 !         test_name = n,
216 !         test_value = m
217 !       )
218 !     })
219 !   })
220   }
221
222   # Custom tryCatch/withCallingHandlers function to catch messages, warnings, and
223   # errors along with any values returned by the expression. Returns a list of
224   # value, messages, warnings, and errors.
225   try_fuzz <- function(fun, fun_name, all_args) {
226
227 !   call <- list(fun = fun_name, args = all_args)
228 !   messages <- NULL
229 !   output <- NULL
230 !   warnings <- NULL
231 !   errors <- NULL
232
233 !   message_handler <- function(c) {
234 !     messages <<- c(messages, conditionMessage(c))
```

```
!     invokeRestart("muffleMessage")
!   }

!   warning_handler <- function(c) {
!     warnings <<- c(warnings, conditionMessage(c))
!     invokeRestart("muffleWarning")
!   }

!   error_handler <- function(c) {
!     errors <<- c(errors, conditionMessage(c))
!     return(NULL)
!   }

    # 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

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

!   if (length(output) == 0) {
!     output <- NULL
    }

!   list(
!     call = call,
!     value = value,
!     output = output,
!     messages = messages,
!     warnings = warnings,
!     errors = errors
!   )
  }
# 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() {
!   c(test_char(), test_int(), test_dbl(), test_fctr(), test_lgl(), test_date(),
!     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() {
  list(
    char_empty = character(),
    char_single = letters[1],
    char_single_blank = "",
    char_multiple = letters[1:3],
    char_multiple_blank = c(letters[1:3], ""),
    char_with_na = c(letters[1:2], NA),
    char_single_na = NA_character_,
    char_all_na = rep(NA_character_, 3)
  )
}

#' @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() {
  list(
    int_empty = integer(),
    int_single = 1L,
    int_multiple = 1L:3L,
    int_with_na = c(1L:2L, NA),
    int_single_na = NA_integer_,
    int_all_na = rep(NA_integer_, 3)
  )
}

#' @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() {
  list(
    dbl_empty = double(),
    dbl_single = 1.5,
    dbl_mutliple = 1:3 + 0.5,
    dbl_with_na = c(1:2 + 0.5, NA),
    dbl_single_na = NA_real_,
    dbl_all_na = rep(NA_real_, 3)
  )
}

#' @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() {
```

```
!   list(
!     lgl_empty = logical(),
!     lgl_single = TRUE,
!     lgl_mutliple = c(TRUE, FALSE, FALSE),
!     lgl_with_na = c(TRUE, NA, FALSE),
!     lgl_single_na = NA,
!     lgl_all_na = rep(NA, 3)
!   )
  }

  #' @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() {
!   list(
!     fctr_empty = factor(),
!     fctr_single = as.factor("a"),
!     fctr_multiple = as.factor(c("a", "b", "c")),
!     fctr_with_na = as.factor(c("a", "b", NA)),
!     fctr_missing_levels = factor(c("a", "b", "c"), levels = letters[1:4]),
!     fctr_single_na = factor(NA),
!     fctr_all_na = factor(rep(NA, 3))
!   )
  }

  #' @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() {
!   list(
!     date_single = as.Date("2001-01-01"),
!     date_multiple = as.Date(c("2001-01-01", "1950-05-05")),
!     date_with_na = as.Date(c("2001-01-01", NA, "1950-05-05")),
!     date_single_na = as.Date(NA_integer_, origin = "1971-01-01"),
!     date_all_na = as.Date(rep(NA_integer_, 3), origin = "1971-01-01")
!   )
  }

  #' @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() {
!   list(
!     raw_empty = raw(),
!     raw_char = charToRaw("b"),
!     raw_na = charToRaw(NA_character_)
!   )
  }
```

```
151
152 #' @describeIn test_all Data frames \itemize{
153 #'   \item \code{df_complete}: \code{datasets::iris}
154 #'   \item \code{df_empty}: \code{data.frame(NULL)}
155 #'   \item \code{df_one_row}: \code{datasets::iris[1, ]}
156 #'   \item \code{df_one_col}: \code{datasets::iris[ ,1]}
157 #'   \item \code{df_with_na}: \code{iris} with several NAs added to each column.
158 #' }
159 #' @export
160 test_df <- function() {
161 !   iris_na <- datasets::iris
162 !   iris_na[c(1, 10, 100), 1] <- NA
163 !   iris_na[c(5, 15, 150), 3] <- NA
164 !   iris_na[c(7, 27, 75), 5] <- NA
165
166 !   list(
167 !     df_complete = datasets::iris,
168 !     df_empty = data.frame(NULL),
169 !     df_one_row = datasets::iris[1, ],
170 !     df_one_col = datasets::iris[ ,1],
171 !     df_with_na = iris_na
172 !   )
173 }
174
175 #' @describeIn test_all Null value \itemize{
176 #'  \item \code{null_value}: \code{NULL}
177 #' }
178 #' @export
179 test_null <- function() {
180 !   list(
181 !     null_value = NULL
182 !   )
183 }
184
185 # Development utility function ----
186
187 # This is a non-exported, non-checked function (hence it's being commented out)
188 # to be used to quickly generate the \itemize{...} sections of documentation for
189 # vector-based tests. NOTE do not use the verbatim results if they are too
190 # lengthy.
191
192 # doc_test <- function(test) {
193 #   tnames <- names(test)
194 #   tval <- purrr::map_chr(test, deparse)
195 #   clipr::write_clip(
196 #     c("\\itemize{",
197 #     paste0("#'  \\item \\code{", tnames, "}: \\code{", tval, "}", collapse = "\n"),
198 #     "#' }"))
199 # }
```