fuzzr coverage - 77.27%

- Files
- Source

File		Lines	Relevant	Covered	Missed	Hits / Line	Coverage
R/outputs.R	108	38	0	38	0		0.00%
R/evaluators.R	275	111	99	12	303		89.19%
R/inputs.R	199	71	71	0	3		100.00%

```
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, ...) {
```

```
87 !   assertthat::assert_that(inherits(fr, "fuzz_results"))
88
89     # value supplied to index takes priority
90 !   if (!is.null(index)) {
91 !     assertthat::assert_that(assertthat::is.count(index) && index <= length(fr))
92 !     res <- fr[[index]]
93     } else {
94
95       # if no index, then check based on test name
96 !     .dots <- list(...)
97 !     purrr::walk(.dots, function(p) assertthat::assert_that(assertthat::is.string(p)))
98
99 !     assertthat::assert_that(all(names(.dots) %in% names(fr[[1]][["test_name"]])))
100
101 !    res <- purrr::detect(fr, function(el) {
102 !      all(purrr::map2_lgl(.dots, names(.dots), function(p, n) grepl(p, x = el[["test_name"]][[n]])))
103 !    })
104 !    if (length(res) == 0)
105 !      warning("Zero matches found.")
106    }
107 !  res
108  }
```

```
1      # Exported functions ----
2
3      #' Fuzz-test a function
4      #'
5      #' Evaluate how a function responds to unexpected or non-standard inputs.
6      #'
7      #' \code{fuzz_function} provides a simple interface to fuzz test a single
8      #' argument of a function by passing the function, name of the argument, static
9      #' values of other required arguments, and a named list of test values.
10     #'
11     #' \code{p_fuzz_function} takes a nested list of arguments paired with lists of
12     #' tests to run on each argument, and will evaluate every combination of
13     #' argument and provided test.
14     #'
15     #' @note The user will be asked to confirm before proceeding if the combinations
16     #'   of potential tests exceeds 500,000.
17     #'
18     #' @param fun A function.
19     #' @param arg_name Quoted name of the argument to fuzz test.
20     #' @param ... Other non-dynamic arguments to pass to \code{fun}. These will be
21     #'   repeated for every one of the \code{tests}.
22     #' @param tests Which fuzz tests to run. Accepts a named list of inputs,
23     #'   defaulting to \code{\link{test_all}}.
24     #' @param check_args Check if \code{arg_name} and any arguments passed as
25     #'   \code{...} are accepted by \code{fun}. Set to \code{FALSE} if you need to
26     #'   pass arguments to a function that accepts arguments via \code{...}.
27     #' @param progress Show a progress bar while running tests?
28     #'
29     #' @return A \code{fuzz_results} object.
30     #'
31     #' @seealso \code{\link{fuzz_results}} and
32     #'   \code{\link{as.data.frame.fuzz_results}} to access fuzz test results.
33     #'
34     #' @export
35     #' @examples
36     #' # Evaluate the 'formula' argument of lm, passing additional required variables
37     #' fr <- fuzz_function(lm, "formula", data = iris)
38     #'
39     #' # When evaluating a function that takes ..., set check_args to FALSE
40     #' fr <- fuzz_function(paste, "x", check_args = FALSE)
41     fuzz_function <- function(fun, arg_name, ..., tests = test_all(), check_args = TRUE, progress = interactive()) {
42
```

```
43  2x    fuzz_asserts(fun, check_args, progress)
44  2x    attr(fun, "fun_name") <- deparse(substitute(fun))
45  2x    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  2x    dots_call_names <- purrr::map_chr(as.list(match.call()), deparse)
51  2x    .dots = list(...)
52  2x    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  2x    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  2x    if (check_args)
59  1x      assertthat::assert_that(
60  1x        assertthat::has_args(fun, arg_name),
61  1x        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  2x    test_args <- c(
67  2x      purrr::set_names(list(tests), arg_name),
68  2x      purrr::map2(.dots, dots_call_names, function(x, y) purrr::set_names(list(x), y)))
69
70  2x    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  3x    fuzz_asserts(fun, check_args, progress)
88  3x    if (is.null(attr(fun, "fun_name"))) {
89  1x      fun_name <- deparse(substitute(fun))
90          } else {
91  2x      fun_name <- attr(fun, "fun_name")
92          }
93
94  3x    if (check_args)
95  2x      assertthat::assert_that(assertthat::has_args(fun, names(.l)))
96
97          # Ensure .l is a named list of named lists
98  3x    is_named_ll(.l)
99
100         # Replace any NULL test values with .null alias.
101 3x    .l <- purrr::map(.l, function(li) {
102 6x      purrr::map(li, function(lli) {
103 149x     if (is.null(lli)) {
104 3x        .null
105 6x      } else {
106 146x       lli
```

```
107 6x      }
108 6x    })
109 3x  })
110
111     # Warn if combination of tests is potentially massive
112 3x  num_tests <- purrr::reduce(purrr::map_int(.l, length), `*`)
113 3x  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 3x  test_list <- named_cross_n(.l)
121
122     # After crossing, restore NULL test values
123 3x  test_list <- purrr::modify_depth(test_list, 3, function(x) {
124 3102x     if (inherits(x, what = "fuzz-null")) {
125 12x        NULL
126 3x      } else {
127 3090x       x
128 3x      }
129 3x    })
130
131     # Create a progress bar, if called for
132 3x  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 3x  fr <- purrr::map(
141 3x    test_list, function(x) {
142 !       if (exists("pb")) pb$tick()
143
144         # Extract values for testing
145 564x    arglist <- purrr::map(x, getElement, name = "test_value")
146
147         # Extract names of tests
148 564x    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 564x    res <- list(test_result = try_fuzz(fun = fun, fun_name = fun_name,
154 564x                                       all_args = arglist))
155 564x    res[["test_name"]] <- testnames
156 564x    res
157 3x    })
158
159 3x  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    5x    assertthat::assert_that(
174    5x      is.function(fun), assertthat::is.flag(check_args),
175    5x      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    3x    assertthat::assert_that(is.list(l), is_named(l))
181    3x    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    4x    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   13x    nm <- names(x)
196   13x    !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    3x    crossed_values <- purrr::cross(ll)
208          # ... and then cross the names
209    3x    crossed_names <- purrr::cross(purrr::map(ll, names))
210
211          # Then map through both values and names in order to
212    3x    purrr::map2(crossed_values, crossed_names, function(x, y) {
213  564x      purrr::map2(x, y, function(m, n) {
214 1551x        list(
215 1551x          test_name = n,
216 1551x          test_value = m
217 1551x        )
218  564x      })
219    3x    })
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  564x    call <- list(fun = fun_name, args = all_args)
228  564x    messages <- NULL
229  564x    output <- NULL
230  564x    warnings <- NULL
231  564x    errors <- NULL
232
233  564x    message_handler <- function(c) {
234    !       messages <<- c(messages, conditionMessage(c))
```

```
235  !    invokeRestart("muffleMessage")
236  564x  }
237
238  564x  warning_handler <- function(c) {
239  !    warnings <<- c(warnings, conditionMessage(c))
240  !    invokeRestart("muffleWarning")
241  564x  }
242
243  564x  error_handler <- function(c) {
244  506x    errors <<- c(errors, conditionMessage(c))
245  506x    return(NULL)
246  564x  }
247
248        # Little trick: that first tryCatch() will return values from the expression
249        # to the "value" index in this list, but will pass errors to error_handler
250        # (which returns NULL "value", incidentally.) In the event of messages or
251        # warnings, handling is passed up to withCallingHandlers, which passes them
252        # down again to message_handler or warning_handler, respectively. Once the
253        # expression is done evaluating, messages, warnings, and errors are assigned
254        # to the list, which is returned as the final result of try_fuzz
255
256  564x  output <- utils::capture.output({
257  564x    value <- withCallingHandlers(
258  564x      tryCatch(do.call(fun, args = all_args), error = error_handler),
259  564x      message = message_handler,
260  564x      warning = warning_handler
261  564x    )}, type = "output")
262
263  564x  if (length(output) == 0) {
264  564x    output <- NULL
265        }
266
267  564x  list(
268  564x    call = call,
269  564x    value = value,
270  564x    output = output,
271  564x    messages = messages,
272  564x    warnings = warnings,
273  564x    errors = errors
274  564x  )
275      }
```

```
1   # Data types ----
2
3   #' Fuzz test inputs
4   #'
5   #' Each \code{test_all} returns a named list that concatenates all the available
6   #' tests specified below.
7   #'
8   #' @export
9   test_all <- function() {
10 3x   c(test_char(), test_int(), test_dbl(), test_fctr(), test_lgl(), test_date(),
11 3x     test_raw(), test_df(), test_null())
12  }
13
14  #' @describeIn test_all Character vectors \itemize{
15  #'  \item \code{char_empty}: \code{character(0)}
16  #'  \item \code{char_single}: \code{"a"}
17  #'  \item \code{char_single_blank}: \code{""}
18  #'  \item \code{char_multiple}: \code{c("a", "b", "c")}
19  #'  \item \code{char_multiple_blank}: \code{c("a", "b", "c", "")}
20  #'  \item \code{char_with_na}: \code{c("a", "b", NA)}
21  #'  \item \code{char_single_na}: \code{NA_character_}
22  #'  \item \code{char_all_na}: \code{c(NA_character_, NA_character_, NA_character_)}
23  #' }
```

```
#' @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 4x   iris_na <- datasets::iris
162 4x   iris_na[c(1, 10, 100), 1] <- NA
163 4x   iris_na[c(5, 15, 150), 3] <- NA
164 4x   iris_na[c(7, 27, 75), 5] <- NA
165
166 4x   list(
167 4x     df_complete = datasets::iris,
168 4x     df_empty = data.frame(NULL),
169 4x     df_one_row = datasets::iris[1, ],
170 4x     df_one_col = datasets::iris[ ,1],
171 4x     df_with_na = iris_na
172 4x   )
173 }
174
175 #' @describeIn test_all Null value \itemize{
176 #'  \item \code{null_value}: \code{NULL}
177 #' }
178 #' @export
179 test_null <- function() {
180 3x   list(
181 3x     null_value = NULL
182 3x   )
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 # }
```