

fuzzr coverage - 96.82%

- [Files](#)
- [Source](#)

File	Lines	Relevant	Covered	Missed	Hits / Line	Coverage
R/evaluators.R	275	111	104	7	92	93.69%
R/inputs.R	199	71	71	0	12	100.00%
R/outputs.R	108	38	38	0	131	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
23     as.data.frame.fuzz_results <- function(x, ..., delim = "; ") {
```

```

24 5x ldf <- purrr::map(x, parse_fuzz_result_concat, delim = delim)
25 5x df <- do.call("rbind", ldf)
26 5x df[["results_index"]] <- seq_along(x)
27 5x df
28 }
29
30 #' Access individual fuzz test results
31 #'
32 #' @param fr \code{fuzz_results} object
33 #' @param index The test index (by position) to access. Same as the
34 #' \code{results_index} in the data frame returned by
35 #' \code{\link{as.data.frame.fuzz_results}}.
36 #' @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.
37 #' @name fuzz_results
38 NULL
39
40 #' @describeIn fuzz_results Access the object returned by the fuzz test
41 #' @export
42 fuzz_value <- function(fr, index = NULL, ...) {
43 8x res <- search_results(fr, index, ...)
44 7x res[["test_result"]][["value"]]
45 }
46
47 #' @describeIn fuzz_results Access the call used for the fuzz test
48 #' @export
49 fuzz_call <- function(fr, index = NULL, ...) {
50 7x res <- search_results(fr, index, ...)
51 6x res[["test_result"]][["call"]]
52 }
53
54 # Internal functions ----
55
56 # For each result, create a one-row data frame of test names, outputs, messages,
57 # warnings, errors, and result classes.
58 parse_fuzz_result_concat <- function(fr, delim) {
59
60 213x dfr <- as.data.frame(fr[["test_name"]], stringsAsFactors = FALSE)
61
62 213x elem_collapse <- function(elem) {
63 852x if (is.null(elem)) {
64 472x return(NA_character_)
65 213x } else {
66 380x paste(elem, collapse = delim)
67 213x }
68 213x }
69
70 213x dfr[["output"]] <- elem_collapse(fr[["test_result"]][["output"]])
71 213x dfr[["messages"]] <- elem_collapse(fr[["test_result"]][["messages"]])
72 213x dfr[["warnings"]] <- elem_collapse(fr[["test_result"]][["warnings"]])
73 213x dfr[["errors"]] <- elem_collapse(fr[["test_result"]][["errors"]])
74
75 # If no object was returned by the function under given test conditions,
76 # record value as NA in the data frame
77 213x dfr[["result_classes"]] <- ifelse(
78 213x is.null(fr[["test_result"]][["value"]]),
79 213x NA_character_,
80 213x paste(class(fr[["test_result"]][["value"]]), collapse = delim))
81
82 213x dfr
83 }
84
85 # Find elements of the search results list
86 search_results <- function(fr, index, ...) {
87 15x assertthat::assert_that(inherits(fr, "fuzz_results"))
88

```

```

89 # value supplied to index takes priority
90 15x if (!is.null(index)) {
91 5x   assertthat::assert_that(assertthat::is.count(index) && index <= length(fr))
92 5x   res <- fr[[index]]
93   } else {
94     # if no index, then check based on test name
95     .dots <- list(...)
96 10x   purrr::walk(.dots, function(p) assertthat::assert_that(assertthat::is.string(p)))
97 10x   assertthat::assert_that(all(names(.dots) %in% names(fr[[1]][["test_name"]])))
98
99 10x   res <- purrr::detect(fr, function(el) {
100     all(purrr::map2_lgl(.dots, names(.dots), function(p, n) grepl(p, x = el[["test_name"]][n])))
101   })
102 8x   if (length(res) == 0)
103     warning("Zero matches found.")
104   }
105 2x   res
106 }
107 13x }
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 11x fuzz_asserts(fun, check_args, progress)
44 8x attr(fun, "fun_name") <- deparse(substitute(fun))

```

```

45 8x 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 5x dots_call_names <- purrr::map_chr(as.list(match.call()), deparse)
51 5x .dots = list(...)
52 5x 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 5x 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 5x if (check_args)
59 5x   assertthat::assert_that(
60 5x     assertthat::has_args(fun, arg_name),
61 5x     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 3x test_args <- c(
67 3x   purrr::set_names(list(tests), arg_name),
68 3x   purrr::map2(.dots, dots_call_names, function(x, y) purrr::set_names(list(x), y)))
69
70 3x 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 15x fuzz_asserts(fun, check_args, progress)
88 12x if (is.null(attr(fun, "fun_name"))) {
89 9x   fun_name <- deparse(substitute(fun))
90   } else {
91 3x   fun_name <- attr(fun, "fun_name")
92   }
93
94 12x if (check_args)
95 11x   assertthat::assert_that(assertthat::has_args(fun, names(.l)))
96
97 # Ensure .l is a named list of named lists
98 12x is_named_ll(.l)
99
100 # Replace any NULL test values with .null alias.
101 7x .l <- purrr::map(.l, function(li) {
102 14x   purrr::map(li, function(lli) {
103 311x     if (is.null(lli)) {
104 6x       .null
105 14x     } else {
106 305x       lli
107 14x     }
108 14x   })
109 7x })

```

```

110
111 # Warn if combination of tests is potentially massive
112 7x num_tests <- purrr::reduce(purrr::map_int(.l, length), `*`)
113 7x if (num_tests >= 500000) {
114 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?"))
115 ! if (m != 1)
116 ! return(NULL)
117 }
118
119 # Generate the list of tests to be done
120 6x test_list <- named_cross_n(.l)
121
122 # After crossing, restore NULL test values
123 6x test_list <- purrr::modify_depth(test_list, 3, function(x) {
124 668x if (inherits(x, what = "fuzz-null")) {
125 2x NULL
126 6x } else {
127 666x x
128 6x }
129 6x })
130
131 # Create a progress bar, if called for
132 6x 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 6x fr <- purrr::map(
141 6x test_list, function(x) {
142 ! if (exists("pb")) pb$tick()
143
144 # Extract values for testing
145 168x arglist <- purrr::map(x, getElement, name = "test_value")
146
147 # Extract names of tests
148 168x 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 168x res <- list(test_result = try_fuzz(fun = fun, fun_name = fun_name,
154 168x all_args = arglist))
155 168x res[["test_name"]] <- testnames
156 168x res
157 6x })
158
159 6x 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 26x assertthat::assert_that(
174 26x is.function(fun), assertthat::is.flag(check_args),

```

```

175 26x 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 12x assertthat::assert_that(is.list(l), is_named(l))
181 10x 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 13x 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 50x nm <- names(x)
196 50x !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 6x crossed_values <- purrr::cross(ll)
208   # ... and then cross the names
209 6x crossed_names <- purrr::cross(purrr::map(ll, names))
210
211   # Then map through both values and names in order to
212 6x purrr::map2(crossed_values, crossed_names, function(x, y) {
213 168x purrr::map2(x, y, function(m, n) {
214 334x   list(
215 334x     test_name = n,
216 334x     test_value = m
217 334x   )
218 168x })
219 6x })
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 168x call <- list(fun = fun_name, args = all_args)
228 168x messages <- NULL
229 168x output <- NULL
230 168x warnings <- NULL
231 168x errors <- NULL
232
233 168x message_handler <- function(c) {
234 92x   messages <- c(messages, conditionMessage(c))
235 92x   invokeRestart("muffleMessage")
236 168x }
237
238 168x warning_handler <- function(c) {
239 128x   warnings <- c(warnings, conditionMessage(c))

```

```

240 128x invokeRestart("muffleWarning")
241 168x }
242
243 168x error_handler <- function(c) {
244 59x   errors <- c(errors, conditionMessage(c))
245 59x   return(NULL)
246 168x }
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 168x output <- utils::capture.output({
257 168x   value <- withCallingHandlers(
258 168x     tryCatch(do.call(fun, args = all_args), error = error_handler),
259 168x     message = message_handler,
260 168x     warning = warning_handler
261 168x   ), type = "output")
262
263 168x if (length(output) == 0) {
264 122x   output <- NULL
265   }
266
267 168x list(
268 168x   call = call,
269 168x   value = value,
270 168x   output = output,
271 168x   messages = messages,
272 168x   warnings = warnings,
273 168x   errors = errors
274 168x )
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  9x   c(test_char(), test_int(), test_dbl(), test_fctr(), test_lgl(), test_date(),
11  9x     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   #' }
24   #' @export
25   test_char <- function() {
26 15x   list(
27 15x     char_empty = character(),
28 15x     char_single = letters[1],
29 15x

```

```

    char_single_blank = "",
30 15x char_multiple = letters[1:3],
31 15x char_multiple_blank = c(letters[1:3], ""),
32 15x char_with_na = c(letters[1:2], NA),
33 15x char_single_na = NA_character_,
34 15x char_all_na = rep(NA_character_, 3)
35 15x )
36 }
37
38 #' @describeIn test_all Integer vectors \itemize{
39 #' \item \code{int_empty}: \code{integer(0)}
40 #' \item \code{int_single}: \code{1L}
41 #' \item \code{int_multiple}: \code{1:3}
42 #' \item \code{int_with_na}: \code{c(1L, 2L, NA)}
43 #' \item \code{int_single_na}: \code{NA_integer_}
44 #' \item \code{int_all_na}: \code{c(NA_integer_, NA_integer_, NA_integer_)}
45 #' }
46 #' @export
47 test_int <- function() {
48 11x list(
49 11x int_empty = integer(),
50 11x int_single = 1L,
51 11x int_multiple = 1L:3L,
52 11x int_with_na = c(1L:2L, NA),
53 11x int_single_na = NA_integer_,
54 11x int_all_na = rep(NA_integer_, 3)
55 11x )
56 }
57
58 #' @describeIn test_all Double vectors \itemize{
59 #' \item \code{dbl_empty}: \code{numeric(0)}
60 #' \item \code{dbl_single}: \code{1.5}
61 #' \item \code{dbl_multiple}: \code{c(1.5, 2.5, 3.5)}
62 #' \item \code{dbl_with_na}: \code{c(1.5, 2.5, NA)}
63 #' \item \code{dbl_single_na}: \code{NA_real_}
64 #' \item \code{dbl_all_na}: \code{c(NA_real_, NA_real_, NA_real_)}
65 #' }
66 #' @export
67 test_dbl <- function() {
68 11x list(
69 11x dbl_empty = double(),
70 11x dbl_single = 1.5,
71 11x dbl_multiple = 1:3 + 0.5,
72 11x dbl_with_na = c(1:2 + 0.5, NA),
73 11x dbl_single_na = NA_real_,
74 11x dbl_all_na = rep(NA_real_, 3)
75 11x )
76 }
77
78 #' @describeIn test_all Logical vectors \itemize{
79 #' \item \code{lgl_empty}: \code{logical(0)}
80 #' \item \code{lgl_single}: \code{TRUE}
81 #' \item \code{lgl_multiple}: \code{c(TRUE, FALSE, FALSE)}
82 #' \item \code{lgl_with_na}: \code{c(TRUE, NA, FALSE)}
83 #' \item \code{lgl_single_na}: \code{NA}
84 #' \item \code{lgl_all_na}: \code{c(NA, NA, NA)}
85 #' }
86 #' @export
87 test_lgl <- function() {
88 11x list(
89 11x lgl_empty = logical(),
90 11x lgl_single = TRUE,
91 11x lgl_multiple = c(TRUE, FALSE, FALSE),
92 11x lgl_with_na = c(TRUE, NA, FALSE),

```



```

93 11x   lgl_single_na = NA,
94 11x   lgl_all_na = rep(NA, 3)
95 11x   )
96     }
97
98     #' @describeIn test_all Factor vectors \itemize{
99     #'   \item \code{fctr_empty}: \code{structure(integer(0), .Label = character(0), class = "factor")}
100    #'   \item \code{fctr_single}: \code{structure(1L, .Label = "a", class = "factor")}
101    #'   \item \code{fctr_multiple}: \code{structure(1:3, .Label = c("a", "b", "c"), class = "factor")}
102    #'   \item \code{fctr_with_na}: \code{structure(c(1L, 2L, NA), .Label = c("a", "b"), class = "factor")}
103    #'   \item \code{fctr_missing_levels}: \code{structure(1:3, .Label = c("a", "b", "c", "d"), class = "factor")}
104    #'   \item \code{fctr_single_na}: \code{structure(NA_integer_, .Label = character(0), class = "factor")}
105    #'   \item \code{fctr_all_na}: \code{structure(c(NA_integer_, NA_integer_, NA_integer_), .Label = character(0), class = "factor")}
106    #' }
107    #' @export
108    test_fctr <- function() {
109 11x   list(
110 11x     fctr_empty = factor(),
111 11x     fctr_single = as.factor("a"),
112 11x     fctr_multiple = as.factor(c("a", "b", "c")),
113 11x     fctr_with_na = as.factor(c("a", "b", NA)),
114 11x     fctr_missing_levels = factor(c("a", "b", "c"), levels = letters[1:4]),
115 11x     fctr_single_na = factor(NA),
116 11x     fctr_all_na = factor(rep(NA, 3))
117 11x   )
118   }
119
120   #' @describeIn test_all Date vectors \itemize{
121   #'   \item \code{date_single}: \code{as.Date("2001-01-01")}
122   #'   \item \code{date_multiple}: \code{as.Date(c("2001-01-01", "1950-05-05"))}
123   #'   \item \code{date_with_na}: \code{as.Date(c("2001-01-01", NA, "1950-05-05"))}
124   #'   \item \code{date_single_na}: \code{as.Date(NA_integer_, origin = "1971-01-01")}
125   #'   \item \code{date_all_na}: \code{as.Date(rep(NA_integer_, 3), origin = "1971-01-01")}
126   #' }
127   #' @export
128   test_date <- function() {
129 11x   list(
130 11x     date_single = as.Date("2001-01-01"),
131 11x     date_multiple = as.Date(c("2001-01-01", "1950-05-05")),
132 11x     date_with_na = as.Date(c("2001-01-01", NA, "1950-05-05")),
133 11x     date_single_na = as.Date(NA_integer_, origin = "1971-01-01"),
134 11x     date_all_na = as.Date(rep(NA_integer_, 3), origin = "1971-01-01")
135 11x   )
136   }
137
138   #' @describeIn test_all Raw vectors \itemize{
139   #'   \item \code{raw_empty}: \code{raw(0)}
140   #'   \item \code{raw_char}: \code{as.raw(0x62)},
141   #'   \item \code{raw_na}: \code{charToRaw(NA_character_)}
142   #' }
143   #' @export
144   test_raw <- function() {
145 11x   list(
146 11x     raw_empty = raw(),
147 11x     raw_char = charToRaw("b"),
148 11x     raw_na = charToRaw(NA_character_)
149 11x   )
150   }
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 # }

```