Package 'testwhat'

March 17, 2016

Type Package

Title Easily write submission correctness tests for R exercises

Version 3.0.2

Date 2014-07-31

Description This package makes it easy for teachers to test students'
code submissions for R exercises, e.g., for interactive R courses on
\{ }url{www.DataCamp.com}.

URL www.DataCamp.com

Depends R (>= 3.0.0), testthat, knitr, rmarkdown

Imports methods, markdown, stringdist

License MIT + file LICENSE

LazyData true

Author Filip Schouwenaars <`filip@datacamp.com`>

Maintainer Filip Schouwenaars <`filip@datacamp.com`>

RoxygenNote 5.0.1

R topics documented:

`build_doc_structure` *build R markdown document structure, using knitr functions*

Description

build R markdown document structure, using knitr functions

Usage

```
build_doc_structure(text)
```

Arguments

`text`	text representing an R Markdown document

`DataCampReporter-class` *DataCamp reporter: gather test results along with elapsed time and feedback messages.*

Description

This reporter gathers all results, adding additional information such as test elapsed time and feedback messages.

`get_clean_lines` *convert student/solution code to vector of clean strings with the pipe operator removed.*

Description

convert student/solution code to vector of clean strings with the pipe operator removed.

Usage

```
get_clean_lines(code)
```

Arguments

`code`	the code to convert to a vector of unpiped clean strings

`get_solution_code` *Get solution environment (backwards comp)*

Description

Get solution environment (backwards comp)

Usage

```
get_solution_code()
```

`get_solution_env` *Get solution environment (backwards comp)*

Description

Get solution environment (backwards comp)

Usage

```
get_solution_env()
```

get_student_code *Get solution environment (backwards comp)*

Description

Get solution environment (backwards comp)

Usage

```
get_student_code()
```

get_student_output *Get solution environment (backwards comp)*

Description

Get solution environment (backwards comp)

Usage

```
get_student_output()
```

parse_docs *Parse both the student and solution document*

Description

Parse both the student and solution document

Usage

```
parse_docs()
```

`success_msg` *Define the success message*

Description

If all tests in an SCT pass, the students gets a congratulatory message. You can specify this message with /codesuccess_msg(). It does not matter where in the SCT you specify this message, but at the end makes most sense.

Usage

```
success_msg(msg)
```

Arguments

`msg`	The congratulatory message as a character string.

Details

For multiple choice exercises, the success message is specified inside `test_mc`, so an additional call of `success_msg` is not necessary.

`testwhat` *Easily write submission correctness tests for DataCamp*

Description

This package makes it easy for teachers to test students' code submissions for R exercises, e.g., for interactive R courses on `www.DataCamp.com`.

Details

This package contains a bunch of high-level functions to test objects, function calls, output, function definition, control structures and many more.

For more information, see `docs.datacamp.com/teach/sct-design-r.html`.

For a bunch of SCT examples, see `www.github.com/data-camp/introduction_to_R`.

Note

testwhat is a wrapper around **testthat** such that teachers can write their tests using a familiar framework.

References

`www.DataCamp.com`

test_an_object *Check if the student defined an object, independent of the name*

Description

This function is an adaption of test_object. The function will check if a specific object, defined in the solution, exists. The object the student defined doesn't have to have the same name. In other words, this function will check if any defined variable by the user corresponds to a specific variable in the solution code.

Usage

```
test_an_object(name, undefined_msg = NULL, eq_condition = "equivalent")
```

Arguments

name	name of object in solution to test.
undefined_msg	feedback message in case the student did not define an object that corresponds to the solution object. This argument should always be specified.
eq_condition	character string indicating how to compare. Possible values are "equivalent" (the default), "equal" and "identical". See expect_equivalent, expect_equal, and expect_identical, respectively.

Examples

```
## Not run:
# Example 1 solution code:
# x <- 5

# sct command to test whether student defined _an_ object with same value
test_an_object("x")

# All of the following student submissions are accepted
# x <- 5
# y <- 5
# z <- 4 + 1 + 1e-8

## End(Not run)
```

test_axis *Test whether the student correctly defined axis properties (ggvis exercises)*

Description

Test whether the student correctly assigned axis properties. The student's and solution code is automatically compared to one another for specific properties of the axes.

Usage

```
test_axis(index = 1, type = NULL, props = NULL, not_called_msg = NULL,
  incorrect_msg = NULL, incorrect_number_of_calls_msg = NULL)
```

Arguments

`index`	exercise to be checked (solution and student code should have same number of calls!) properties inside the first mentioned function by the teacher.
`type`	which axis to check (x or y). Only one axis is tested at the same time.
`props`	set of axis properties to be checked. If not specified, all properties found in the solution or checked on.
`not_called_msg`	feedback message in case the specified axis type (x or y) was not found
`incorrect_msg`	feedback message in case the axes properties defined by the student did not correspond with the one of the solution.
`incorrect_number_of_calls_msg`	feedback message in case the student did enter the same amount of commands as the solution did.

Details

This test is implemented using `test_that`.

`test_chunk_options` *Test whether the student defined the correct chunk options (R Markdown exercises)*

Description

Test whether the student defined the correct chunk options in an R Markdown exercise

Usage

```
test_chunk_options(options = NULL, allow_extra = TRUE,
  not_called_msg = NULL, incorrect_msg = NULL)
```

Arguments

`options`	Set of options
`allow_extra`	whether or not the definition of additional options is accepted (default TRUE)
`not_called_msg`	feedback message if option was not specified
`incorrect_msg`	feedback message if option was incorrectly set

Details

This test is implemented using `test_what`. This test can only be called inside a test_rmd_group() call!

test_correct — *Test things. If it fails, test additional things.*

Description

Test if a set of tests passes, and do additional, more precise tests if there were failures. The teacher should specify two code chunks;

- `check_code`: specifies the code that checks on the (typically, final results of the) student's code. These tests are executed silently, without the reporter generating information for these.
- `diagnose_code`: Set of tests that gets executed if the tests in `check_code` fail. These tests contain more detailed tests, to pinpoint the problem. To make sure there is a fail in the end, the tests in `check_code` are run afterwards, this time 'loudly'.

Usage

```
test_correct(check_code, diagnose_code, env = parent.frame())
```

Arguments

`check_code`	High-level tests. Also provide feedback messages here, as this code is run loudly after executing the `diagnose_code` code, in the case of failing tests.
`diagnose_code`	Low-level tests that are run if tests in `check_code` fail.
`env`	environment in which to execute tests.

Details

`test_correct` reduces computation time (if it's ok, the additional battery of tests is not run) and increases the flexibility for the student (if the final result is ok, different paths towards this result are allowed).

Examples

```
## Not run:
# Example 1 solution code:
# x <- mean(1:3, na.rm = TRUE)

# Example SCT
test_correct({
 test_object("x")
}, {
 # this code only is run if test_object("x") fails
 test_function("mean", "x")
 # test_object("x") is automatically run again to generate a fail if test_function passed.
})

## End(Not run)
```

`test_data_frame` *Test list elements (or data frame columns)*

Description

Test whether a student defined a list, and if this is the case, whether the elements of the list correspond to the ones in the solution. A data frame is also a list, so you can use this function to test the correspondence of data frame columns.

Usage

```
test_data_frame(name, columns = NULL, eq_condition = "equivalent",
  undefined_msg = NULL, undefined_cols_msg = NULL, incorrect_msg = NULL)
```

Arguments

`name`	name of the list or data frame to test.
`columns`	character vector or integer vector of list elements or indices to test.
`eq_condition`	character string indicating how to compare. Possible values are `"equivalent"` (the default), `"equal"` and `"identical"`. See `expect_equivalent`, `expect_equal`, and `expect_identical`, respectively.
`undefined_msg`	optional feedback message if list is not defined.
`undefined_cols_msg`	optional feedback message if not all specified elements of the solution list were found in the student's list.
`incorrect_msg`	optional feedback message if not all specified elements of the solution list match those in the student list.

Examples

```
## Not run:
# Example 1 solution code:
# df <- data.frame(a = 1:3, b = LETTERS[1:3])

# sct command to test column a
test_data_frame("df", columns = "a")

# sct command to test column b
test_data_frame("df", columns = "b")

## End(Not run)
```

`test_error` *Check whether the student's submission threw an error.*

Description

With information gathered from the R Backend, `test_error` detects whether the student's submission generated an error. Automatically, a feedback is generated, which can be appended with an additional incorrect_msg.

Usage

```
test_error(incorrect_msg = NULL)
```

Arguments

`incorrect_msg`	feeback message that is appended to the error message generated by R.

Examples

```
## Not run:
# Example student code: x <- 4 + "a"

# R error message as feedback:
test_error()

# R error message as feedback, with additional info:
test_error("Don't sum numerics and characters!")

## End(Not run)
```

`test_exercise` *Run all tests for an exercise*

Description

Run all tests for an exercise and report the results (including feedback). This function is run by R Backend and should not be used by course creators.

Usage

```
test_exercise(sct, ex_type, pec, student_code, solution_code, solution_env,
  output_list, env = test_env())
```

Arguments

`sct`	Submission correctness tests as a character string.
`ex_type`	Type of the exercise
`pec`	pre-exercise-code
`student_code`	character string representing the student code
`solution_code`	character string representing the solution code
`solution_env`	environment containing the objects defined by solution code
`output_list`	the output structure that is generated by RBackend
`env`	environment in which to execute tests.

Value

A list with components `passed` that indicates whether all tests were sucessful, and `feedback` that contains a feedback message.

`test_expression_output` *Test output of expression*

Description

Test whether the given expression gives the same output in the student and the solution environment.

Usage

```
test_expression_output(expr, incorrect_msg = NULL)
```

Arguments

`expr`	The expression that is executed in both environments.
`incorrect_msg`	Optional feedback message in case the evaluation is not the same in both environments. Automatically generated if not specified.

Examples

```
## Not run:
# Example 1 solution code:
# my_fun <- function(a, b) { a + b }

# Test whether my_fun(1,2) and my_fun(1,2)
# give same _output_
test_function_definition({
 test_expression_output(my_fun(1,2))
 test_expression_output(my_fun(-1,-2))
})

## End(Not run)
```

test_expression_result — *Test result of expression*

Description

Test whether the given expression gives the same result in the student and the solution environment.

Usage

```
test_expression_result(expr, eq_condition = "equivalent",
  incorrect_msg = NULL)
```

Arguments

expr	The expression that is executed in both environments.
eq_condition	character string indicating how to compare. Possible values are "equivalent" (the default), "equal" and "identical". See expect_equivalent, expect_equal, and expect_identical, respectively.
incorrect_msg	Optional feedback message in case the evaluation is not the same in both environments. Automatically generated if not specified.

Examples

```
## Not run:
# Example 1 solution code:
# my_fun <- function(a, b) { a + b }

# Test whether my_fun(1,2) and my_fun(1,2)
# give same _result_
test_function_definition({
 test_expression_result(my_fun(1,2))
 test_expression_result(my_fun(-1,-2))
})

## End(Not run)
```

test_file_exists — *Test whether a file exists*

Description

Test whether a file exists

Usage

```
test_file_exists(path, incorrect_msg = NULL)
```

Arguments

`path`	Path to the file you want to check
`incorrect_msg`	Optional feedback message in case the file does not exist

Examples

```
## Not run:
# Example 1 solution code:
# write("hello", file = "test.txt")

# SCT to test if file exists
test_file_exists("test.txt")

## End(Not run)
```

`test_for_loop` *Test a for loop*

Description

Test whether a student coded a for loop correctly. The function parses the student and solution code and selects the first, second ... for loop in the code depending on the `index` argument, and then runs two chunks of tests:

- `cond_test`: `testwhat` tests specifically for the iteration part of the for loop, inside the parentheses of `for`.
- `expr_test`: `testwhat` for the code inside the for loop itself.

The tests for the iteration part and the expression part of the for loop can only be text-based. You cannot use functions such as `test_object` that also depend on the student and solution environment.

Usage

```
test_for_loop(index = 1, cond_test = NULL, expr_test = NULL,
  not_found_msg = NULL, env = parent.frame())
```

Arguments

`index`	The index of the for loop to check.
`cond_test`	testwhat tests for the condition part of the for loop
`expr_test`	testwhat tests for the expression part of the for loop
`not_found_msg`	optional feedback message in case the for loop (at given index) is not found.
`env`	Environment in which to run the additional testwhat tests.

Examples

```
## Not run:
# Example 1 solution code:
for(i in 1:5) {
 print("hurray!")
}

# SCT to test this loop:
test_for_loop({
 test_student_typed("in")
 test_student_typed("1")
 test_student_typed("5")
}, {
 test_function("print")
})

## End(Not run)
```

---

test_function *Test whether a student correctly called a function*

Description

Test whether a student called a function, possibly with certain arguments, correctly.

Usage

```
test_function(name, args = NULL, index = 1, ignore = NULL,
  allow_extra = TRUE, eval = TRUE, eq_condition = "equivalent",
  not_called_msg = NULL, args_not_specified_msg = NULL,
  incorrect_msg = NULL)

test_function_v2(name, args = NULL, index = 1, ignore = NULL,
  allow_extra = TRUE, eval = TRUE, eq_condition = "equivalent",
  not_called_msg = NULL, args_not_specified_msg = NULL,
  incorrect_msg = NULL)
```

Arguments

`name`	name of the function to test.
`args`	character vector of argument names that the student should have supplied in the function calls.
`index`	integer that specifies which call of `name` in the solution code will be checked.
`ignore`	character vector of argument names that should not be tested (useful in combination with `allow_extra = FALSE` to allow certain arguments to be ignored, but not others).
`allow_extra`	indicates whether extra arguments not specified by `args` or `ignore` are allowed in the student's function calls.

`eval`	logical vector indicating whether the corresponding argument should be evaluated before testing. Setting this to `FALSE` can be useful, e.g., to test whether the student supplied a large predefined object, as only the corresponding `name` is compared in this case (use with care!).
`eq_condition`	character vector indicating how to perform the comparison for each argument. See `test_object`
`not_called_msg`	feedback message in case the student did not call the function often enough.
`args_not_specified_msg`	feedback message in case the student did call the function with the arguments listed in `args`
`incorrect_msg`	feedback message in case the student did not call the function with the same argument values as in the sample solution. If there are multiple function calls in the sample solution, a vector of feedback messages can be supplied.

Examples

```
## Not run:
# Suppose the solution contains: mean(1:3, na.rm = TRUE)
# To test this submission, provide the following in the sct
test_function("mean", c("x", "na.rm"))

## End(Not run)
```

`test_function_definition` *Check whether the student defined a function correctly*

Description

Check whether the student defined a function correctly

Usage

```
test_function_definition(name, function_test = NULL, body_test = NULL,
  undefined_msg = NULL, incorrect_number_arguments_msg = NULL,
  env = parent.frame())
```

Arguments

`name`	The name of the function to test
`function_test`	tests to perform on the function (use `test_expression_output` and `test_expression_result`).
`body_test`	Additional tests to perform on the body of the function if the tests in `function_test` fail. Only able to test on strings here!
`undefined_msg`	Optional feedback message in case the specified function was not defined
`incorrect_number_arguments_msg`	Optional feedback message in case the function does not have the correct number of arguments.
`env`	Environment in which to perform the tests

Examples

```
## Not run:
# Example 1 solution code:
# my_fun <- function(a, b) { a + b }

# SCT testing both result and printouts:
test_function_definition({
 test_expression_result(my_fun(1,2))
 test_expression_output(my_fun(1,2))
}, {
 test_student_typed("+")
})

## End(Not run)
```

`test_function_result` *Check the result of a function call*

Description

TODO More information here.

Usage

```
test_function_result(name = NULL, index = 1, eq_condition = "equivalent",
  not_called_msg = NULL, incorrect_msg = NULL)
```

Arguments

`name`	name of the function whose output you would like to check.
`index`	Ordinal number of the solution call you want to check
`eq_condition`	character vector indicating how to perform the comparison for each argument. See `test_object`
`not_called_msg`	feedback message in case the function is not retrieved.
`incorrect_msg`	feedback message in case the evaluation was not the same as in the solution

---

`test_ggplot` *Test ggplot call*

Description

Test ggplot call

Usage

```
test_ggplot(index = 1, all_fail_msg = NULL, check_data = TRUE,
  data_fail_msg = NULL, check_aes = TRUE, aes_fail_msg = NULL,
  exact_aes = FALSE, check_geom = TRUE, geom_fail_msg = NULL,
  exact_geom = FALSE, check_geom_params = NULL, check_facet = TRUE,
  facet_fail_msg = NULL, check_scale = TRUE, scale_fail_msg = NULL,
  exact_scale = FALSE, check_coord = TRUE, coord_fail_msg = NULL,
  exact_coord = FALSE, check_stat = TRUE, stat_fail_msg = NULL,
  exact_stat = FALSE, check_extra = NULL, extra_fail_msg = NULL,
  exact_extra = NULL, check = NULL)
```

Arguments

`index`	which call to check
`all_fail_msg`	Message if all fails
`check_data`	Whether or not to check data latyer
`data_fail_msg`	Message in case data layer fails
`check_aes`	Whether or not to check aes latyer
`aes_fail_msg`	Message in case aes layer fails
`exact_aes`	Should the aesthetics be exact?
`check_geom`	Whether or not to check geom layer
`geom_fail_msg`	Message in case geom layer fails
`exact_geom`	Should the geoms be exact?
`check_geom_params`	Should the geom parameters be checked?
`check_facet`	Whether or not to check facet latyer
`facet_fail_msg`	Message in case facet layer fails
`check_scale`	Whether or not to check scale latyer
`scale_fail_msg`	Message in case scale layer fails
`exact_scale`	Whether or not scales should be defined exactly
`check_coord`	Whether or not to check coord latyer
`coord_fail_msg`	Message in case coord layer fails
`exact_coord`	Whether or not coords should be defined exactly
`check_stat`	Whether or not to check stat latyer
`stat_fail_msg`	Message in case stat layer fails
`exact_stat`	Whether or not stats should be defined exactly
`check_extra`	Whether to check extra stuff
`extra_fail_msg`	Message in case extra stuff fails
`exact_extra`	Whether or not extra info should be exactly specified.
`check`	Which layers to check

`test_if_else` *Test a conditional statement*

Description

Check whether the student correctly coded a conditional statement. The function parses all `if-else` constructs and then runs tests for all composing parts of this constructions.

Usage

```
test_if_else(index = 1, if_cond_test = NULL, if_expr_test = NULL,
  else_expr_test = NULL, not_found_msg = NULL, missing_else_msg = NULL,
  env = parent.frame())
```

Arguments

`index`	The index of the control structure to check.
`if_cond_test`	tests to perform in the if condition part of the control structure
`if_expr_test`	tests to perform in the if expression part of the control structure
`else_expr_test`	tests to perform in the else expression part of the control structure
`not_found_msg`	Message in case the control structure (at given index) is not found.
`missing_else_msg`	Messing in case the else part of the control structure should be there but is missing
`env`	Environment in which to perform all these SCTs

Details

If there's an `else if` in there, this counts as a 'sub-conditional' statement (see example).

Examples

```
## Not run:
# Example solution code
vec <- c("a", "b", "c")
if("a" %in% vec) {
 print("a in here")
} else if(any("b" > vec)) {
 cat("b not smallest")
} else {
 str(vec)
}

# SCT to test this loop
test_if_else({
 test_student_typed("%in%")
}, {
 # test if expr part
 test_function("print")
}, {
 # test else expr part
```

```
  test_if_else({
    # test cond part of else if
    test_student_typed(">")
  }, {
    # test else if expr part
    test_function("cat")
  }, {
    # test else part
    test_function("str")
  })
})

## End(Not run)
```

`test_instruction` *Test a single instruction of the challenges interface*

Description

Test a single instruction of the challenges interface

Usage

```
test_instruction(index, code, env = parent.frame())
```

Arguments

`index`	the instruction index
`code`	the test code for that instruction
`env`	environment in which to execute tests.

`test_library_function` *Test whether the library function was called correctly*

Description

Convenience function to test in a very hacky way whether the library function was called correctly in its most simple form. There is support for the different ways to call the library function

Usage

```
test_library_function(package, not_called_msg = NULL, incorrect_msg = NULL)
```

Arguments

`package`	package name for which the library() function should've been called
`not_called_msg`	optional feedback message in case the library function wasn't called a single time
`incorrect_msg`	optional feedback message in case the library function wasn't called for the specified package.

Examples

```
## Not run:
# Example solution code
library(ggvis)
library(dplyr)

# SCT to test both library calls:
test_library_function("ggvis")
test_library_function("dplyr")

## End(Not run)
```

---

test_mc *Test a multiple choice exercise*

Description

Test a multiple choice exercise using `test_what`. This code expects the DM.result variable to be defined by the DataCamp frontend. There is need to define the success_msg seperately, since it is defined inside the function.

Usage

```
test_mc(correct = NULL, no_selection_msg = NULL, feedback_msgs = NULL)
```

Arguments

`correct`	number of the correct answer
`no_selection_msg`	feedback message in case the student did not select an answer.
`feedback_msgs`	vector of feedback messages for both the incorrect exercises as the correct exercise. Order the messages according to how they are listed in the instructions. For example, if there are four options, the second of which is correct, a vector of four feedback messages should be provided. The first message corresponds to feedback on the incorrect selection of the first option, the second message corresponds to the feedback message for the correct collection. The third and fourth messages correspond to feedback on the incorrect selection of the third and fourth option.

Examples

```
## Not run:
# Example solution: second instruction correct.

# Corresponding SCT:
msg1 <- "Not good, try again!"
msg2 <- "Nice one!"
msg3 <- "Not quite, give it another shot."
msg4 <- "Don't be silly..."
test_mc(2, feedback_msgs = c(msg1, msg2, msg3, msg4))

## End(Not run)
```

test_object — *Test R object existence and value*

Description

Test whether a student defined a certain object. If this is the case, and if eval is TRUE, also check whether the value of the object matches that of the solution.

Usage

```
test_object(name, eq_condition = "equivalent", eval = TRUE,
  undefined_msg = NULL, incorrect_msg = NULL)
```

Arguments

name	name of the object to test.
eq_condition	character string indicating how to compare. Possible values are "equivalent" (the default), "equal" and "identical". See expect_equivalent, expect_equal, and expect_identical, respectively.
eval	Next to existence, check if the value of the object corresponds between student en solution environment.
undefined_msg	Optional feedback message in case the student did not define the object. A meaningful message is automatically generated if not supplied.
incorrect_msg	optional feedback message in case the student's object is not the same as in the sample solution. Only used if eval is TRUE. A meaningful message is automatically generated if not supplied.

Examples

```
## Not run:
# Example 1 solution code:
# x <- mean(1:3, na.rm = TRUE)

# sct command to test existence and value of x:
test_object("x")

# sct command to test only existence of x:
test_object("x", eval = FALSE)

# Example 2 solution code:
# y <- list(a = 2, b = 3, c = 4)

# Small numerical difference allowed + no check on attributes
test_object(y)

# Small numerical difference allowed + check attributes
test_object(y, eq_condition = "equals")

# No numerical difference allowed + check attributes
test_object(y, eq_condtion = "identical")

## End(Not run)
```

test_or *Test if one of the given sct parts are correct.*

Description

Test if one of the given SCT code batteries are evaluated as being correct. If not, the feedback message of the first fail is standardly given. Can be used nested.

Usage

```
test_or(..., incorrect_msg = NULL, choose_feedback = 1, subs = TRUE,
  env = parent.frame())
```

Arguments

`...`	one of these code blocks with tests should succeed
`incorrect_msg`	msg displayed when none succeeds
`choose_feedback`	choose feedback of test with this index
`subs`	substitute content of ...
`env`	environment in which to execute tests.

Details

- `...`: an arbritrary amount of code blocks containing SCT code. `test_or` will check if one of the code blocks results in a successful SCT evaluation.

Examples

```
## Not run:
  # test if either the object a or the object b is correct
  test_or(test_object("a"), test_object("b"))

## End(Not run)
```

test_output_contains *Check whether the student printed something to the console*

Description

Function checks whether the student's console contains the output one gets by evaluating the character string provided in `expr` provided to expr. This function needs refactoring, as all new lines etc are removed.

Usage

```
test_output_contains(expr, times = 1, incorrect_msg = NULL,
  env = globalenv())
```

Arguments

`expr`	The expression (as string) for which the output should be in the student's console output.
`times`	How often the expression's output should occur in the student's console
`incorrect_msg`	feeback message in case the output did not contain the expression
`env`	environment where the code in expr exectued.

Examples

```
## Not run:
# SCT to test whether student printed numbers 1 to 10
test_output_contains("for(i in 1:10) print(i)")

## End(Not run)
```

`test_pipe` *Test whether a student used the pipe operator sufficiently (ggvis and dplyr exercises)*

Description

Test whether a student used the pipe sufficiently. By default, the function only checks if the pipe was used at least once. The user can also select the minimal number of occurrences of the pipe.

Usage

```
test_pipe(num = 1, absent_msg = NULL, insuf_msg = NULL)
```

Arguments

`num`	minimal number of times the pipe operator has to appear (default = 1)
`absent_msg`	feedback message in case the student did not use a single pipe.
`insuf_msg`	feeback message in case the student did not use the pipe operator sufficiently.

Details

This test is implemented using `test_that`.

test_props — *Test whether the student used the correct properties (ggvis exercises)*

Description

Test whether the student used at least as many and the correct properties as the solution inside a specific command and inside a specific function. By default, this function will compare the ggvis functions of both student and solution. However, the teacher can also state that the definition of data can be done in other functions.

Usage

```
test_props(index = 1, funs = "ggvis", props = NULL, allow_extra = TRUE,
  not_called_msg = NULL, incorrect_msg = NULL,
  incorrect_number_of_calls_msg = NULL)
```

Arguments

`index`	exercise to be checked (solution and student code should have same number of calls!)
`funs`	the function in which to look for the x and y data. If the same info is found in one function, the test passes. All the functions that the teacher specifies, must be present in the students' solution! The function only looks for properties inside the first mentioned function by the teacher.
`props`	set of properties to be checked. If not specified, all properties found in the solution or checked on. If specified as an empty charactor vector (c()), only the calling of the functions will be checked on.
`allow_extra`	whether or not the definition of additional properties is accepted (default TRUE)
`not_called_msg`	feedback message in case the specified function(s) was/were not found.
`incorrect_msg`	feedback message in case the student specified properties do not correspond with the ones in the solution.
`incorrect_number_of_calls_msg`	feedback message in case the student did enter the same amount of commands as the solution did.

test_rmd_file — *Test R Markdown file*

Description

Test a single R Markdown file

Usage

```
test_rmd_file(code, student_file = NULL, solution_file = NULL,
  env = parent.frame())
```

Arguments

`code`	the SCT code for the file
`student_file`	the name of the student file to be tested
`solution_file`	the name of the solution file to be tested
`env`	The environment in which the code should be tested.

Details

This test should be called when there are multiple files in the submission.

`test_rmd_group` *Test a single R Markdown file group (R Markdown exercises)*

Description

Test a single R Markdown file group (R Markdown exercises) with arbitrary testwhat functions. This test is implemented using `test_what`.

Usage

```
test_rmd_group(group_number, code, env = parent.frame())
```

Arguments

`group_number`	Number of the group.
`code`	SCT code to test the group (in curly braces)
`env`	The environment in which the code should be tested.

`test_student_typed` *Test student's submission as text*

Description

Test whether a student typed something in his submission. Some basic string formatting is performed to allow for different ways of saying the same things (removing spaces, changing single quotes to double quotes, changing TRUE to T ...).

Usage

```
test_student_typed(strings, fixed = TRUE, not_typed_msg = NULL)
```

Arguments

`strings`	A set of strings, at least one of which must be in the student_code
`fixed`	exact string matching (TRUE) or use regex (FALSE)?
`not_typed_msg`	Feedback message in case the student did not type the string.

Details

Using this function should be a last resort, as there are myriad ways of solving the same problems in R!

Examples

```
## Not run:
# Example solution code: TRUE & FALSE

# SCT to test this as a string (both T & F and F & T should be accepted)
test_student_typed(c("TRUE & FALSE", "FALSE & TRUE"))

## End(Not run)
```

`test_subexpr_eval` *Test whether a student called a subexpression correctly. (dplyr and ggvis exercises)*

Description

Test whether a student called a(n) (sub)expression. If yes, test for this function call if the result corresponds to the subexpression called in the solution.

Usage

```
test_subexpr_eval(index = 1, fun = NULL, not_called_msg = NULL,
  incorrect_msg = NULL, incorrect_number_of_calls_msg = NULL)
```

Arguments

`index`	exercise to be checked (solution and student code should have same number of calls!)
`fun`	name of the function to be checked. if fun = NULL, check the entire command.
`not_called_msg`	feedback message in case the function is not retrieved.
`incorrect_msg`	feedback message in case the evaluation was not the same as in the solution
`incorrect_number_of_calls_msg`	feedback message in case the student did enter the same amount of commands as the solution did.

Details

This test is implemented using `test_that`. When testing whether the resut is the same, small numeric differences or differences in attributes are allowed.

`test_text` *Test inline text and formatting (Markdown)*

Description

Test inline text and its formatting for R Markdown exercises. This test can only be called inside a test_rmd_group() call!

Usage

```
test_text(text, format = "any", freq = 1, not_called_msg = NULL,
  incorrect_msg = NULL)
```

Arguments

`text`	Text to match (can be a regular expression!)
`format`	the format of the text that the text should be in ("any", "italics", "bold", "code", "inline_code", "brackets", "parentheses", "list"). If none of the above, the format string is appended to text in front and in the back and used as a regexp.
`freq`	How often the text should appear with this formatting
`not_called_msg`	feedback message if the text was not there
`incorrect_msg`	feedback message if the text was not properly formatted

`test_tree_contains` *Test whether a student's function call contains a certain character (dplyr and ggvis exercises)*

Description

For a specified command in the student's code, check whether a particular function contains a set of queries. Also the number of times these queries have to appear can be specified.

Usage

```
test_tree_contains(index = 1, fun = NULL, queries = NULL, times = NULL,
  contain_all = TRUE, fixed_order = FALSE, not_called_msg = NULL,
  absent_msg = NULL, incorrect_number_of_calls_msg = NULL)
```

Arguments

`index`	exercise to be checked (solution and student code should have same number of calls!)
`fun`	name of the function to be checked. if fun = NULL, check the entire command.
`queries`	single character or vector of character that have to be present in the function
`times`	number of times each of the entered queries have to be available. (ones on default)

`contain_all`	AND versus OR. if contain_all = TRUE (by default), all strings in the queries vector must exist in the tree. If contain_all = FALSE, the test passes if one of the strings in the queries vector exists in the tree the specified number of times.
`fixed_order`	whether or not the queries have to appear in the function call the same order as specified in the queries vector (default FALSE). This functionality can not be used when contain_all is FALSE. This option also only works when each query has to be present only once. The order is determined on the LAST occurrences of the queries (in embedded notation!!)
`not_called_msg`	feedback message in case the function is not retrieved
`absent_msg`	feedback message in case one of the queries was not available
`incorrect_number_of_calls_msg`	feedback message in case the student did enter the same amount of commands as the solution did.

Details

This test is implemented using `test_that`. Only exact string matching is performed for the moment.

`test_tree_order` *Test whether a student's collection of function calls follows that of the solution (dplyr exercises)*

Description

For a specified command in the student's code, check whether the order of function calls follows the one specified by the teacher. The teacher can set several orders that are accepted.

Usage

```
test_tree_order(index = 1, custom_orders = NULL, allow_extra = TRUE,
  incorrect_msg = NULL, incorrect_number_of_calls_msg = NULL)
```

Arguments

`index`	exercise to be checked (solution and student code should have same number of calls!)
`custom_orders`	list of character vectors. Every vector represents an order that is acceptable. Example of syntax to define orders: list(c("select","filter","arrange"),c("filter","arrange","select")). This specific example would mean that a function call where a select inside filter inside arrange, or a filter inside arrange or inside select is allowed.
`allow_extra`	if TRUE, allows the student to define additional function, as long as the functions in the solution are followed in the same order.
`incorrect_msg`	feedback message in case the student order did not match any of the accepted orders.
`incorrect_number_of_calls_msg`	feedback message in case the student did enter the same amount of commands as the solution did.

Details

This test is implemented using `test_that`. Only exact string matching is performed for the moment. No support for summarize(), only summarise()! (unless teacher codes it explicitly)

`test_what` *Expectation wrapper*

Description

This function wraps around an expect_... function. When the expectation fails to be met, the feedback message is sent to the reporter.

Usage

```
test_what(code, feedback, feedback_msg)
```

Arguments

`code`	The expectation that should be wrapped
`feedback`	A character string with feedback when the expection is not met OR a list object, containing multiple pieces of information.
`feedback_msg`	deprecated argument, for backwards compatibility

`test_while_loop` *Test a while loop*

Description

Test whether a student correctly coded a while loop. The function parses the student and solution code and selects the first, second ... while loop in the code depending on the `index` argument, and then runs two chunks of tests:

- `cond_test`: `testwhat` tests specifically for the condition part of the while loop.
- `expr_test`: `testwhat` tests for the code inside the while loop

The tests for the conditional part and the expression part of the while loop can only be text-based. You cannot use functions such as `test_object` that also depend on the student and solution environment.

Usage

```
test_while_loop(index = 1, cond_test = NULL, expr_test = NULL,
  not_found_msg = NULL, env = parent.frame())
```

Arguments

`index`	The index of the while loop to check.
`cond_test`	SCT to perform on the condition part of the while loop
`expr_test`	SCT to perform on the expression part of the while loop
`not_found_msg`	Message in case the while loop (at given index) is not found.
`env`	Environment in which to perform all these SCTs

Examples

```
## Not run:
# Example solution code:
while(x < 18) {
 x <- x + 5
 print(x)
}

# SCT to test this loop:
test_while_loop({
 test_student_typed(c("< 18", "18 >"))
}, {
 test_student_Typed(c("x + 5", "5 = x"))
 test_function("print", eval = FALSE) # no actual value matching possible!!
})

## End(Not run)
```

`test_yaml_header` *Test yaml header (Markdown)*

Description

Test whether the student specified the correct options in the yaml header (for R Markdown exercises). This test should be called outside an test_rmd_group call.

Usage

```
test_yaml_header(options = NULL, check_equality = TRUE,
  allow_extra = TRUE, not_called_msg = NULL, incorrect_msg = NULL)
```

Arguments

`options`	Set of options. Embedded options have to be specified using the dot notation.
`check_equality`	whether or not to actually check the value assigned to the option (default TRUE)
`allow_extra`	whether or not the definition of additional options is accepted (default TRUE)
`not_called_msg`	feedback message if option was not specified (optional but recommended)
`incorrect_msg`	feedback message if option was incorrectly set (optional but recommended)

Index