Package 'testthat'

March 2, 2020

Title Unit Testing for R

Version 2.3.2

Description Software testing is important, but, in part because it is frustrating and boring, many of us avoid it. 'testthat' is a testing framework for R that is easy to learn and use, and integrates with your existing 'workflow'.

License MIT + file LICENSE

URL http://testthat.r-lib.org, https://github.com/r-lib/testthat

BugReports https://github.com/r-lib/testthat/issues

Depends R (>= 3.1)

Imports cli, crayon (>= 1.3.4), digest, ellipsis, evaluate, magrittr, methods, pkgload, praise, R6 (>= 2.2.0), rlang (>= 0.4.1), withr (>= 2.0.0)

Suggests covr, curl (>= 0.9.5), devtools, knitr, rmarkdown, usethis, vctrs (>= 0.1.0), xml2

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.0.2

Collate 'auto-test.R' 'capture-condition.R' 'capture-output.R' 'colour-text.R' 'compare.R' 'compare-character.R' 'compare-numeric.R' 'compare-time.R' 'context.R' 'describe.R' 'evaluate-promise.R' 'example.R' 'expect-comparison.R' 'expect-condition.R' 'expect-equality.R' 'expect-inheritance.R' 'expect-invisible.R' 'expect-known.R' 'expect-length.R' 'expect-logical.R' 'expect-messages.R' 'expect-named.R' 'expect-null.R' 'expect-output.R' 'reporter.R' 'expect-self-test.R' 'expect-setequal.R' 'expect-silent.R' 'expect-that.R' 'expect-vector.R' 'expectation.R' 'expectations-matches.R' 'make-expectation.R' 'mock.R' 'old-school.R' 'praise.R' 'quasi-label.R' 'recover.R' 'reporter-check.R' 'reporter-debug.R' 'reporter-fail.R' 'reporter-junit.R' 'reporter-list.R' 'reporter-location.R' 'reporter-minimal.R' 'reporter-multi.R' 'stack.R'

'reporter-progress.R' 'reporter-rstudio.R' 'reporter-silent.R'
'reporter-stop.R' 'reporter-summary.R' 'reporter-tap.R'
'reporter-teamcity.R' 'reporter-zzz.R' 'skip.R' 'source.R'
'teardown.R' 'test-compiled-code.R' 'test-directory.R'
'test-example.R' 'test-files.R' 'test-path.R' 'test-that.R'
'try-again.R' 'utils-io.R' 'utils.R' 'verify-output.R'
'watcher.R'

NeedsCompilation yes

Author Hadley Wickham [aut, cre],
RStudio [cph, fnd],
R Core team [ctb] (Implementation of utils::recover())

Maintainer Hadley Wickham <hadley@rstudio.com>

Repository CRAN

Date/Publication 2020-03-02 15:40:02 UTC

R topics documented:

`auto_test` *Watches code and tests for changes, rerunning tests as appropriate.*

Description

The idea behind `auto_test()` is that you just leave it running while you develop your code. Everytime you save a file it will be automatically tested and you can easily see if your changes have caused any test failures.

Usage

```
auto_test(
  code_path,
  test_path,
  reporter = default_reporter(),
  env = test_env(),
  hash = TRUE
)
```

Arguments

`code_path`	path to directory containing code
`test_path`	path to directory containing tests
`reporter`	test reporter to use
`env`	environment in which to execute test suite.
`hash`	Passed on to `watch()`. When FALSE, uses less accurate modification time stamps, but those are faster for large files.

Details

The current strategy for rerunning tests is as follows:

- if any code has changed, then those files are reloaded and all tests rerun
- otherwise, each new or modified test is run

In the future, `auto_test()` might implement one of the following more intelligent alternatives:

- Use codetools to build up dependency tree and then rerun tests only when a dependency changes.
- Mimic ruby's autotest and rerun only failing tests until they pass, and then rerun all tests.

See Also

`auto_test_package()`

`auto_test_package` *Watches a package for changes, rerunning tests as appropriate.*

Description

Watches a package for changes, rerunning tests as appropriate.

Usage

```
auto_test_package(pkg = ".", reporter = default_reporter(), hash = TRUE)
```

Arguments

`pkg`	path to package
`reporter`	test reporter to use
`hash`	Passed on to `watch()`. When FALSE, uses less accurate modification time stamps, but those are faster for large files.

See Also

`auto_test()` for details on how method works

CheckReporter *Check reporter: 13 line summary of problems*

Description

R CMD check displays only the last 13 lines of the result, so this report is design to ensure that you see something useful there.

See Also

Other reporters: DebugReporter, FailReporter, ListReporter, LocationReporter, MinimalReporter, MultiReporter, ProgressReporter, Reporter, RstudioReporter, SilentReporter, StopReporter, SummaryReporter, TapReporter, TeamcityReporter

comparison-expectations *Expectation: is returned value less or greater than specified value?*

Description

Expectation: is returned value less or greater than specified value?

Usage

```
expect_lt(object, expected, label = NULL, expected.label = NULL)

expect_lte(object, expected, label = NULL, expected.label = NULL)

expect_gt(object, expected, label = NULL, expected.label = NULL)

expect_gte(object, expected, label = NULL, expected.label = NULL)
```

Arguments

object	Computation and value to compare it to. Both arguments supports limited unquoting to make it easier to generate readable failures within a function or for loop. See quasi_label for more details.
expected	Single numeric value to compare.
label	Used to customise failure messages. For expert use only.
expected.label	Used to customise failure messages. For expert use only.

See Also

Other expectations: equality-expectations, expect_error(), expect_length(), expect_match(), expect_message(), expect_named(), expect_null(), expect_output(), expect_silent(), inheritance-expectations, logical-expectations

Examples

```
a <- 9
expect_lt(a, 10)

## Not run:
expect_lt(11, 10)

## End(Not run)

a <- 11
expect_gt(a, 10)
## Not run:
expect_gt(9, 10)

## End(Not run)
```

DebugReporter *Test reporter: start recovery.*

Description

This reporter will call a modified version of `recover()` on all broken expectations.

See Also

Other reporters: `CheckReporter`, `FailReporter`, `ListReporter`, `LocationReporter`, `MinimalReporter`, `MultiReporter`, `ProgressReporter`, `Reporter`, `RstudioReporter`, `SilentReporter`, `StopReporter`, `SummaryReporter`, `TapReporter`, `TeamcityReporter`

---

describe *describe: a BDD testing language*

Description

A simple BDD DSL for writing tests. The language is similiar to RSpec for Ruby or Mocha for JavaScript. BDD tests read like sentences and it should thus be easier to understand what the specification of a function/component is.

Usage

```
describe(description, code)
```

Arguments

`description`	description of the feature
`code`	test code containing the specs

Details

Tests using the `describe` syntax not only verify the tested code, but also document its intended behaviour. Each `describe` block specifies a larger component or function and contains a set of specifications. A specification is definied by an `it` block. Each `it` block functions as a test and is evaluated in its own environment. You can also have nested `describe` blocks.

This test syntax helps to test the intented behaviour of your code. For example: you want to write a new function for your package. Try to describe the specification first using `describe`, before your write any code. After that, you start to implement the tests for each specification (i.e. the `it` block).

Use `describe` to verify that you implement the right things and use `test_that()` to ensure you do the things right.

Examples

```
describe("matrix()", {
  it("can be multiplied by a scalar", {
    m1 <- matrix(1:4, 2, 2)
    m2 <- m1 * 2
    expect_equivalent(matrix(1:4 * 2, 2, 2), m2)
  })
  it("can have not yet tested specs")
})

# Nested specs:
## code
addition <- function(a, b) a + b
division <- function(a, b) a / b

## specs
describe("math library", {
  describe("addition()", {
    it("can add two numbers", {
      expect_equivalent(1 + 1, addition(1, 1))
    })
  })
  describe("division()", {
    it("can divide two numbers", {
      expect_equivalent(10 / 2, division(10, 2))
    })
    it("can handle division by 0") #not yet implemented
  })
})
```

`equality-expectations` *Expectation: is the object equal to a value?*

Description

- `expect_identical()` compares values with `identical()`.
- `expect_equal()` compares values with `all.equal()`
- `expect_equivalent()` compares values with `all.equal()` and `check.attributes = FALSE`
- `expect_reference()` compares the underlying memory addresses.

Usage

```
expect_equal(
  object,
  expected,
  ...,
  info = NULL,
  label = NULL,
  expected.label = NULL
)

expect_equivalent(
  object,
  expected,
  ...,
  info = NULL,
  label = NULL,
  expected.label = NULL
)

expect_identical(
  object,
  expected,
  info = NULL,
  label = NULL,
  expected.label = NULL,
  ...
)

expect_reference(
  object,
  expected,
  info = NULL,
  label = NULL,
  expected.label = NULL
)
```

Arguments

`object, expected`
: Computation and value to compare it to.

Both arguments supports limited unquoting to make it easier to generate readable failures within a function or for loop. See quasi_label for more details.

`...` For `expect_equal()` and `expect_equivalent()`, passed on `compare()`, for `expect_identical()` passed on to `identical()`. Used to control the details of the comparison.

`info` Extra information to be included in the message. This argument is soft-deprecated and should not be used in new code. Instead see alternatives in quasi_label.

`label, expected.label`
Used to customise failure messages. For expert use only.

See Also

`expect_setequal()` to test for set equality.

Other expectations: `comparison-expectations`, `expect_error()`, `expect_length()`, `expect_match()`, `expect_message()`, `expect_named()`, `expect_null()`, `expect_output()`, `expect_silent()`, `inheritance-expectations`, `logical-expectations`

Examples

```
a <- 10
expect_equal(a, 10)

# Use expect_equal() when testing for numeric equality
sqrt(2) ^ 2 - 1
expect_equal(sqrt(2) ^ 2, 2)
# Neither of these forms take floating point representation errors into
# account
## Not run:
expect_true(sqrt(2) ^ 2 == 2)
expect_identical(sqrt(2) ^ 2, 2)

## End(Not run)

# You can pass on additional arguments to all.equal:
## Not run:
# Test the ABSOLUTE difference is within .002
expect_equal(10.01, 10, tolerance = .002, scale = 1)

## End(Not run)

# Test the RELATIVE difference is within .002
x <- 10
expect_equal(10.01, expected = x, tolerance = 0.002, scale = x)

# expect_equivalent ignores attributes
a <- b <- 1:3
names(b) <- letters[1:3]
expect_equivalent(a, b)
```

expect *The building block of all* expect_ *functions*

Description

Call expect() when writing your own expectations. See vignette("custom-expectation") for details.

Usage

```
expect(ok, failure_message, info = NULL, srcref = NULL, trace = NULL)
```

Arguments

ok	TRUE or FALSE indicating if the expectation was successful.
failure_message	Message to show if the expectation failed.
info	Character vector continuing additional information. Included for backward compatibility only and new expectations should not use it.
srcref	Location of the failure. Should only needed to be explicitly supplied when you need to forward a srcref captured elsewhere.
trace	An optional backtrace created by rlang::trace_back(). When supplied, the expectation is displayed with the backtrace.

Details

While expect() creates and signals an expectation in one go, exp_signal() separately signals an expectation that you have manually created with new_expectation(). Expectations are signalled with the following protocol:

- If the expectation is a failure or an error, it is signalled with base::stop(). Otherwise, it is signalled with base::signalCondition().
- The continue_test restart is registered. When invoked, failing expectations are ignored and normal control flow is resumed to run the other tests.

Value

An expectation object. Signals the expectation condition with a continue_test restart.

See Also

exp_signal()

expect_error *Expectation: does code throw error or other condition?*

Description

`expect_error()` and `expect_condition()` check that code throws an error or condition with a message that matches `regexp`, or a class that inherits from `class`. See below for more details.

Usage

```
expect_error(
  object,
  regexp = NULL,
  class = NULL,
  ...,
  info = NULL,
  label = NULL
)

expect_condition(
  object,
  regexp = NULL,
  class = NULL,
  ...,
  info = NULL,
  label = NULL
)
```

Arguments

`object`	Object to test. Supports limited unquoting to make it easier to generate readable failures within a function or for loop. See quasi_label for more details.
`regexp`	Regular expression to test against. • A character vector giving a regular expression that must match the error message. • If `NULL`, the default, asserts that there should be a error, but doesn't test for a specific value. • If `NA`, asserts that there should be no errors.
`class`	Instead of supplying a regular expression, you can also supply a class name. This is useful for "classed" conditions.
`...`	Arguments passed on to `expect_match` `all` Should all elements of actual value match `regexp` (TRUE), or does only one need to match (FALSE) `perl` logical. Should Perl-compatible regexps be used?

	`fixed` logical. If `TRUE`, `pattern` is a string to be matched as is. Overrides all conflicting arguments.
`info`	Extra information to be included in the message. This argument is soft-deprecated and should not be used in new code. Instead see alternatives in quasi_label.
`label`	Used to customise failure messages. For expert use only.

Value

If `regexp = NA`, the value of the first argument; otherwise the captured condition.

Testing `message` vs `class`

When checking that code generates an error, it's important to check that the error is the one you expect. There are two ways to do this. The first way is the simplest: you just provide a `regexp` that match some fragment of the error message. This is easy, but fragile, because the test will fail if the error message changes (even if its the same error).

A more robust way is to test for the class of the error, if it has one. You can learn more about custom conditions at https://adv-r.hadley.nz/conditions.html#custom-conditions, but in short, errors are S3 classes and you can generate a custom class and check for it using `class` instead of `regexp`. Because this is a more reliable check, you `expect_error()` will warn if the error has a custom class but you are testing the message. Eliminate the warning by using `class` instead of `regexp`. Alternatively, if you think the warning is a false positive, use `class = "error"` to suppress it for any input.

If you are using `expect_error()` to check that an error message is formatted in such a way that it makes sense to a human, we now recommend using `verify_output()` instead.

See Also

Other expectations: comparison-expectations, equality-expectations, expect_length(), expect_match(), expect_message(), expect_named(), expect_null(), expect_output(), expect_silent(), inheritance-expectations, logical-expectations

Examples

```
f <- function() stop("My error!")
expect_error(f())
expect_error(f(), "My error!")

# You can use the arguments of grepl to control the matching
expect_error(f(), "my error!", ignore.case = TRUE)

# If you are working with classed conditions, it's better to test for
# the class name, rather than the error message (which may change over time)
custom_err <- function(var) {
  rlang::abort("A special error", var = var, .subclass = "testthat_special")
}
expect_error(custom_err("a"), class = "testthat_special")

# Note that `expect_error()` returns the error object so you can test
```

```
# its components if needed
err <- expect_error(custom_err("a"), class = "testthat_special")
expect_equal(err$var, "a")
```

`expect_invisible` *Expectation: does expression return visibily or invisibly?*

Description

Use this to test whether a function returns a visible or invisible output. Typically you'll use this to check that functions called primarily for their side-effects return their data argument invisibly.

Usage

```
expect_invisible(call, label = NULL)

expect_visible(call, label = NULL)
```

Arguments

`call`	A function call.
`label`	Used to customise failure messages. For expert use only.

Value

The evaluated `call`, invisibly.

Examples

```
expect_invisible(x <- 10)
expect_visible(x)

# Typically you'll assign the result of the expectation so you can
# also check that the value is as you expect.
greet <- function(name) {
  message("Hi ", name)
  invisible(name)
}
out <- expect_invisible(greet("Hadley"))
expect_equal(out, "Hadley")
```

expect_length *Expectation: does a vector have the specified length?*

Description

Expectation: does a vector have the specified length?

Usage

```
expect_length(object, n)
```

Arguments

object	Object to test. Supports limited unquoting to make it easier to generate readable failures within a function or for loop. See quasi_label for more details.
n	Expected length.

See Also

Other expectations: comparison-expectations, equality-expectations, expect_error(), expect_match(), expect_message(), expect_named(), expect_null(), expect_output(), expect_silent(), inheritance-expectations, logical-expectations

Examples

```
expect_length(1, 1)
expect_length(1:10, 10)

## Not run:
expect_length(1:10, 1)

## End(Not run)
```

expect_match *Expectation: does string match a regular expression?*

Description

Expectation: does string match a regular expression?

Usage

```
expect_match(
  object,
  regexp,
  perl = FALSE,
  fixed = FALSE,
  ...,
  all = TRUE,
  info = NULL,
  label = NULL
)
```

Arguments

object	Object to test. Supports limited unquoting to make it easier to generate readable failures within a function or for loop. See quasi_label for more details.
regexp	Regular expression to test against.
perl	logical. Should Perl-compatible regexps be used?
fixed	logical. If TRUE, pattern is a string to be matched as is. Overrides all conflicting arguments.
...	Arguments passed on to base::grepl ignore.case if FALSE, the pattern matching is *case sensitive* and if TRUE, case is ignored during matching. useBytes logical. If TRUE the matching is done byte-by-byte rather than character-by-character. See 'Details'.
all	Should all elements of actual value match regexp (TRUE), or does only one need to match (FALSE)
info	Extra information to be included in the message. This argument is soft-deprecated and should not be used in new code. Instead see alternatives in quasi_label.
label	Used to customise failure messages. For expert use only.

Details

expect_match() is a wrapper around grepl(). See its documentation for more detail about the individual arguments.

See Also

Other expectations: comparison-expectations, equality-expectations, expect_error(), expect_length(), expect_message(), expect_named(), expect_null(), expect_output(), expect_silent(), inheritance-expectations, logical-expectations

Examples

```
expect_match("Testing is fun", "fun")
expect_match("Testing is fun", "f.n")

## Not run:
expect_match("Testing is fun", "horrible")

# Zero-length inputs always fail
expect_match(character(), ".")

## End(Not run)
```

`expect_message` *Expectation: does code produce warnings or messages?*

Description

Use `expect_message()` and `expect_warning()` to check if the messages or warnings match the given regular expression.

Usage

```
expect_message(
  object,
  regexp = NULL,
  ...,
  all = FALSE,
  info = NULL,
  label = NULL
)

expect_warning(
  object,
  regexp = NULL,
  ...,
  all = FALSE,
  info = NULL,
  label = NULL
)
```

Arguments

`object`	Object to test. Supports limited unquoting to make it easier to generate readable failures within a function or for loop. See quasi_label for more details.
`regexp`	Regular expression to test against.

- A character vector giving a regular expression that must match the message/warning
- If NULL, the default, asserts that there should be a messsage/warning, but doesn't test for a specific value.
- If NA, asserts that there shouldn't be any messages or warnings.

`...` Arguments passed on to expect_match

`perl` logical. Should Perl-compatible regexps be used?

`fixed` logical. If TRUE, `pattern` is a string to be matched as is. Overrides all conflicting arguments.

`all` Do messages/warnings need to match the `regexp` (TRUE), or does only one need to match (FALSE)?

`info` Extra information to be included in the message. This argument is soft-deprecated and should not be used in new code. Instead see alternatives in quasi_label.

`label` Used to customise failure messages. For expert use only.

Value

The first argument, invisibly.

See Also

Other expectations: comparison-expectations, equality-expectations, expect_error(), expect_length(), expect_match(), expect_named(), expect_null(), expect_output(), expect_silent(), inheritance-expectations, logical-expectations

Examples

```
# Messages ------------------------------------------------------------------

f <- function(x) {
  if (x < 0) {
    message("*x* is already negative")
    return(x)
  }

  -x
}
expect_message(f(-1))
expect_message(f(-1), "already negative")
expect_message(f(1), NA)

# To test message and output, store results to a variable
expect_message(out <- f(-1), "already negative")
expect_equal(out, -1)

# You can use the arguments of grepl to control the matching
expect_message(f(-1), "*x*", fixed = TRUE)
expect_message(f(-1), "NEGATIVE", ignore.case = TRUE)
```

```
# Warnings -------------------------------------------------------------------
f <- function(x) {
  if (x < 0) {
    warning("*x* is already negative")
    return(x)
  }
  -x
}
expect_warning(f(-1))
expect_warning(f(-1), "already negative")
expect_warning(f(1), NA)

# To test message and output, store results to a variable
expect_warning(out <- f(-1), "already negative")
expect_equal(out, -1)

# You can use the arguments of grepl to control the matching
expect_warning(f(-1), "*x*", fixed = TRUE)
expect_warning(f(-1), "NEGATIVE", ignore.case = TRUE)
```

`expect_named` *Expectation: does object have names?*

Description

You can either check for the presence of names (leaving `expected` blank), specific names (by suppling a vector of names), or absence of names (with `NULL`).

Usage

```
expect_named(
  object,
  expected,
  ignore.order = FALSE,
  ignore.case = FALSE,
  info = NULL,
  label = NULL
)
```

Arguments

`object`	Object to test. Supports limited unquoting to make it easier to generate readable failures within a function or for loop. See quasi_label for more details.
`expected`	Character vector of expected names. Leave missing to match any names. Use `NULL` to check for absence of names.
`ignore.order`	If `TRUE`, sorts names before comparing to ignore the effect of order.

ignore.case	If TRUE, lowercases all names to ignore the effect of case.
info	Extra information to be included in the message. This argument is soft-deprecated and should not be used in new code. Instead see alternatives in quasi_label.
label	Used to customise failure messages. For expert use only.

See Also

Other expectations: comparison-expectations, equality-expectations, expect_error(), expect_length(), expect_match(), expect_message(), expect_null(), expect_output(), expect_silent(), inheritance-expectations, logical-expectations

Examples

```
x <- c(a = 1, b = 2, c = 3)
expect_named(x)
expect_named(x, c("a", "b", "c"))

# Use options to control sensitivity
expect_named(x, c("B", "C", "A"), ignore.order = TRUE, ignore.case = TRUE)

# Can also check for the absence of names with NULL
z <- 1:4
expect_named(z, NULL)
```

---

expect_null *Expectation: is an object* NULL*?*

Description

This is a special case because NULL is a singleton so it's possible check for it either with expect_equal(x,NULL) or expect_type(x,"NULL").

Usage

```
expect_null(object, info = NULL, label = NULL)
```

Arguments

object	Object to test. Supports limited unquoting to make it easier to generate readable failures within a function or for loop. See quasi_label for more details.
info	Extra information to be included in the message. This argument is soft-deprecated and should not be used in new code. Instead see alternatives in quasi_label.
label	Used to customise failure messages. For expert use only.

See Also

Other expectations: comparison-expectations, equality-expectations, expect_error(), expect_length(), expect_match(), expect_message(), expect_named(), expect_output(), expect_silent(), inheritance-expectations, logical-expectations

Examples

```
x <- NULL
y <- 10

expect_null(x)
show_failure(expect_null(y))
```

---

expect_output *Expectation: does code print output to the console?*

Description

Test for output produced by `print()` or `cat()`. This is best used for very simple output; for more complex cases use `verify_output()`.

Usage

```
expect_output(
  object,
  regexp = NULL,
  ...,
  info = NULL,
  label = NULL,
  width = 80
)
```

Arguments

`object`	Object to test. Supports limited unquoting to make it easier to generate readable failures within a function or for loop. See quasi_label for more details.
`regexp`	Regular expression to test against. • A character vector giving a regular expression that must match the output. • If `NULL`, the default, asserts that there should output, but doesn't check for a specific value. • If `NA`, asserts that there should be no output.
`...`	Arguments passed on to `expect_match` `all` Should all elements of actual value match `regexp` (TRUE), or does only one need to match (FALSE)

perl logical. Should Perl-compatible regexps be used?

fixed logical. If TRUE, pattern is a string to be matched as is. Overrides all conflicting arguments.

info — Extra information to be included in the message. This argument is soft-deprecated and should not be used in new code. Instead see alternatives in quasi_label.

label — Used to customise failure messages. For expert use only.

width — Number of characters per line of output. This does not inherit from getOption("width") so that tests always use the same output width, minimising spurious differences.

Value

The first argument, invisibly.

See Also

Other expectations: comparison-expectations, equality-expectations, expect_error(), expect_length(), expect_match(), expect_message(), expect_named(), expect_null(), expect_silent(), inheritance-expectation logical-expectations

Examples

```
str(mtcars)
expect_output(str(mtcars), "32 obs")
expect_output(str(mtcars), "11 variables")

# You can use the arguments of grepl to control the matching
expect_output(str(mtcars), "11 VARIABLES", ignore.case = TRUE)
expect_output(str(mtcars), "$ mpg", fixed = TRUE)
```

---

expect_setequal *Expectation: do two vectors contain the same values?*

Description

- expect_setequal(x,y) tests that every element of x occurs in y, and that every element of y occurs in x.
- expect_mapequal(x,y) tests that x and y have the same names, and that x[names(y)] equals x.

Usage

```
expect_setequal(object, expected)

expect_mapequal(object, expected)
```

Arguments

`object`	Computation and value to compare it to. Both arguments supports limited unquoting to make it easier to generate readable failures within a function or for loop. See quasi_label for more details.
`expected`	Computation and value to compare it to. Both arguments supports limited unquoting to make it easier to generate readable failures within a function or for loop. See quasi_label for more details.

Details

Note that `expect_setequal()` ignores names, and you will be warned if both `object` and `expected` have them.

Examples

```
expect_setequal(letters, rev(letters))
show_failure(expect_setequal(letters[-1], rev(letters)))

x <- list(b = 2, a = 1)
expect_mapequal(x, list(a = 1, b = 2))
show_failure(expect_mapequal(x, list(a = 1)))
show_failure(expect_mapequal(x, list(a = 1, b = "x")))
show_failure(expect_mapequal(x, list(a = 1, b = 2, c = 3)))
```

`expect_silent` *Expectation: is the code silent?*

Description

Checks that the code produces no output, messages, or warnings.

Usage

```
expect_silent(object)
```

Arguments

`object`	Object to test. Supports limited unquoting to make it easier to generate readable failures within a function or for loop. See quasi_label for more details.

Value

The first argument, invisibly.

See Also

Other expectations: comparison-expectations, equality-expectations, expect_error(), expect_length(), expect_match(), expect_message(), expect_named(), expect_null(), expect_output(), inheritance-expectation logical-expectations

Examples

```
expect_silent("123")

f <- function() {
  message("Hi!")
  warning("Hey!!")
  print("OY!!!")
}
## Not run:
expect_silent(f())

## End(Not run)
```

---

expect_vector *Expectation: does the object have vctr properties?*

Description

`expect_vector()` is a thin wrapper around `vctrs::vec_assert()`, converting the results of that function in to the expectations used by testthat. This means that it used the vctrs of `ptype` (prototype) and `size`. See details in https://vctrs.r-lib.org/articles/type-size.html

Usage

```
expect_vector(object, ptype = NULL, size = NULL)
```

Arguments

`object`	Object to test. Supports limited unquoting to make it easier to generate readable failures within a function or for loop. See quasi_label for more details.
`ptype`	(Optional) Vector prototype to test against. Should be a size-0 (empty) generalised vector.
`size`	(Optional) Size to check for.

Examples

```
if (requireNamespace("vctrs") && packageVersion("vctrs") > "0.1.0.9002") {
expect_vector(1:10, ptype = integer(), size = 10)
show_failure(expect_vector(1:10, ptype = integer(), size = 5))
show_failure(expect_vector(1:10, ptype = character(), size = 5))
}
```

fail *Default expectations that always succeed or fail.*

Description

These allow you to manually trigger success or failure. Failure is particularly useful to a precondition or mark a test as not yet implemented.

Usage

```
fail(message = "Failure has been forced", info = NULL)

succeed(message = "Success has been forced", info = NULL)
```

Arguments

message	a string to display.
info	Character vector continuing additional information. Included for backward compatibility only and new expectations should not use it.

Examples

```
## Not run:
test_that("this test fails", fail())
test_that("this test succeeds", succeed())

## End(Not run)
```

FailReporter *Test reporter: fail at end.*

Description

This reporter will simply throw an error if any of the tests failed. It is best combined with another reporter, such as the SummaryReporter.

See Also

Other reporters: CheckReporter, DebugReporter, ListReporter, LocationReporter, MinimalReporter, MultiReporter, ProgressReporter, Reporter, RstudioReporter, SilentReporter, StopReporter, SummaryReporter, TapReporter, TeamcityReporter

inheritance-expectations *Expectation: does the object inherit from a S3 or S4 class, or is it a base type?*

Description

See https://adv-r.hadley.nz/oo.html for an overview of R's OO systems, and the vocabulary used here.

- `expect_type(x,type)` checks that `typeof(x)` is `type`.
- `expect_s3_class(x,class)` checks that `x` is an S3 object that `inherits()` from `class`
- `expect_s4_class(x,class)` checks that `x` is an S4 object that `is()` `class`.

Usage

```
expect_type(object, type)

expect_s3_class(object, class, exact = FALSE)

expect_s4_class(object, class)
```

Arguments

`object`	Object to test. Supports limited unquoting to make it easier to generate readable failures within a function or for loop. See quasi_label for more details.
`type`	String giving base type (as returned by `typeof()`).
`class`	character vector of class names
`exact`	If `FALSE`, the default, checks that `object` inherits from `class`. If `TRUE`, checks that object has a class that's identical to `class`.

See Also

Other expectations: `comparison-expectations`, `equality-expectations`, `expect_error()`, `expect_length()`, `expect_match()`, `expect_message()`, `expect_named()`, `expect_null()`, `expect_output()`, `expect_silent()`, `logical-expectations`

Examples

```
x <- data.frame(x = 1:10, y = "x", stringsAsFactors = TRUE)
# A data frame is an S3 object with class data.frame
expect_s3_class(x, "data.frame")
show_failure(expect_s4_class(x, "data.frame"))
# A data frame is built from a list:
expect_type(x, "list")
```

```
# An integer vector is an atomic vector of type "integer"
expect_type(x$x, "integer")
# It is not an S3 object
show_failure(expect_s3_class(x$x, "integer"))

# Above, we requested data.frame() converts strings to factors:
show_failure(expect_type(x$y, "character"))
expect_s3_class(x$y, "factor")
expect_type(x$y, "integer")
```

`JunitReporter` *Test reporter: summary of errors in jUnit XML format.*

Description

This reporter includes detailed results about each test and summaries, written to a file (or stdout) in jUnit XML format. This can be read by the Jenkins Continuous Integration System to report on a dashboard etc. Requires the *xml2* package.

Details

To fit into the jUnit structure, context() becomes the <testsuite> name as well as the base of the <testcase> classname. The test_that() name becomes the rest of the <testcase> classname. The deparsed expect_that() call becomes the <testcase> name. On failure, the message goes into the <failure> node message argument (first line only) and into its text content (full message).

Execution time and some other details are also recorded.

References for the jUnit XML format: http://llg.cubic.org/docs/junit/

`ListReporter` *List reporter: gather all test results along with elapsed time and file information.*

Description

This reporter gathers all results, adding additional information such as test elapsed time, and test filename if available. Very useful for reporting.

See Also

Other reporters: `CheckReporter`, `DebugReporter`, `FailReporter`, `LocationReporter`, `MinimalReporter`, `MultiReporter`, `ProgressReporter`, `Reporter`, `RstudioReporter`, `SilentReporter`, `StopReporter`, `SummaryReporter`, `TapReporter`, `TeamcityReporter`

LocationReporter *Test reporter: location*

Description

This reporter simply prints the location of every expectation and error. This is useful if you're trying to figure out the source of a segfault, or you want to figure out which code triggers a C/C++ breakpoint

See Also

Other reporters: CheckReporter, DebugReporter, FailReporter, ListReporter, MinimalReporter, MultiReporter, ProgressReporter, Reporter, RstudioReporter, SilentReporter, StopReporter, SummaryReporter, TapReporter, TeamcityReporter

logical-expectations *Expectation: is the object true/false?*

Description

These are fall-back expectations that you can use when none of the other more specific expectations apply. The disadvantage is that you may get a less informative error message.

Usage

```
expect_true(object, info = NULL, label = NULL)

expect_false(object, info = NULL, label = NULL)
```

Arguments

object	Object to test. Supports limited unquoting to make it easier to generate readable failures within a function or for loop. See quasi_label for more details.
info	Extra information to be included in the message. This argument is soft-deprecated and should not be used in new code. Instead see alternatives in quasi_label.
label	Used to customise failure messages. For expert use only.

Details

Attributes are ignored.

See Also

is_false() for complement

Other expectations: comparison-expectations, equality-expectations, expect_error(), expect_length(), expect_match(), expect_message(), expect_named(), expect_null(), expect_output(), expect_silent(), inheritance-expectations

Examples

```
expect_true(2 == 2)
# Failed expectations will throw an error
## Not run:
expect_true(2 != 2)

## End(Not run)
expect_true(!(2 != 2))
# or better:
expect_false(2 != 2)

a <- 1:3
expect_true(length(a) == 3)
# but better to use more specific expectation, if available
expect_equal(length(a), 3)
```

MinimalReporter *Test reporter: minimal.*

Description

The minimal test reporter provides the absolutely minimum amount of information: whether each expectation has succeeded, failed or experienced an error. If you want to find out what the failures and errors actually were, you'll need to run a more informative test reporter.

See Also

Other reporters: CheckReporter, DebugReporter, FailReporter, ListReporter, LocationReporter, MultiReporter, ProgressReporter, Reporter, RstudioReporter, SilentReporter, StopReporter, SummaryReporter, TapReporter, TeamcityReporter

MultiReporter *Multi reporter: combine several reporters in one.*

Description

This reporter is useful to use several reporters at the same time, e.g. adding a custom reporter without removing the current one.

See Also

Other reporters: CheckReporter, DebugReporter, FailReporter, ListReporter, LocationReporter, MinimalReporter, ProgressReporter, Reporter, RstudioReporter, SilentReporter, StopReporter, SummaryReporter, TapReporter, TeamcityReporter

ProgressReporter *Test reporter: interactive progress bar of errors.*

Description

This reporter is a reimagining of SummaryReporter desgined to make the most information available up front, while taking up less space overall. It is the default reporting reporter used by test_dir() and test_file().

Details

As an additional benefit, this reporter will praise you from time-to-time if all your tests pass.

See Also

Other reporters: CheckReporter, DebugReporter, FailReporter, ListReporter, LocationReporter, MinimalReporter, MultiReporter, Reporter, RstudioReporter, SilentReporter, StopReporter, SummaryReporter, TapReporter, TeamcityReporter

RstudioReporter *Test reporter: RStudio*

Description

This reporter is designed for output to RStudio. It produces results in any easily parsed form.

See Also

Other reporters: CheckReporter, DebugReporter, FailReporter, ListReporter, LocationReporter, MinimalReporter, MultiReporter, ProgressReporter, Reporter, SilentReporter, StopReporter, SummaryReporter, TapReporter, TeamcityReporter

SilentReporter *Test reporter: gather all errors silently.*

Description

This reporter quietly runs all tests, simply gathering all expectations. This is helpful for programmatically inspecting errors after a test run. You can retrieve the results with the `expectations()` method.

See Also

Other reporters: `CheckReporter`, `DebugReporter`, `FailReporter`, `ListReporter`, `LocationReporter`, `MinimalReporter`, `MultiReporter`, `ProgressReporter`, `Reporter`, `RstudioReporter`, `StopReporter`, `SummaryReporter`, `TapReporter`, `TeamcityReporter`

skip *Skip a test.*

Description

This function allows you to skip a test if it's not currently available. This will produce an informative message, but will not cause the test suite to fail.

Usage

```
skip(message)

skip_if_not(condition, message = deparse(substitute(condition)))

skip_if(condition, message = NULL)

skip_if_not_installed(pkg, minimum_version = NULL)

skip_if_offline(host = "r-project.org")

skip_on_cran()

skip_on_os(os)

skip_on_travis()

skip_on_appveyor()

skip_on_ci()
```

```
skip_on_covr()

skip_on_bioc()

skip_if_translated(msgid = "'%s' not found")
```

Arguments

message	A message describing why the test was skipped.
condition	Boolean condition to check. skip_if_not() will skip if FALSE, skip_if() will skip if TRUE.
pkg	Name of package to check for
minimum_version	Minimum required version for the package
host	A string with a hostname to lookup
os	Character vector of system names. Supported values are "windows", "mac", "linux" and "solaris".
msgid	R message identifier used to check for translation: the default uses a message included in most translation packs. See the complete list in R-base.pot.

Details

skip* functions are intended for use within test_that() blocks. All expectations following the skip* statement within the same test_that block will be skipped. Test summaries that report skip counts are reporting how many test_that blocks triggered a skip* statement, not how many expectations were skipped.

Helpers

skip_if_not() works like stopifnot(), generating a message automatically based on the first argument.

skip_if_offline() skips tests if an internet connection is not available using curl::nslookup().

skip_on_cran() skips tests on CRAN, using the NOT_CRAN environment variable set by devtools.

skip_on_travis() skips tests on Travis CI by inspecting the TRAVIS environment variable.

skip_on_appveyor() skips tests on AppVeyor by inspecting the APPVEYOR environment variable.

skip_on_ci() skips tests on continuous integration systems by inspecting the CI environment variable.

skip_on_covr() skips tests when covr is running by inspecting the R_COVR environment variable

skip_on_bioc() skips tests on Bioconductor by inspecting the BBS_HOME environment variable.

skip_if_not_installed() skips a tests if a package is not installed or cannot be loaded (useful for suggested packages). It loads the package as a side effect, because the package is likely to be used anyway.

Examples

```
if (FALSE) skip("No internet connection")

## The following are only meaningful when put in test files and
## run with `test_file`, `test_dir`, `test_check`, etc.

test_that("skip example", {
  expect_equal(1, 1L)    # this expectation runs
  skip('skip')
  expect_equal(1, 2)     # this one skipped
  expect_equal(1, 3)     # this one is also skipped
})
```

StopReporter *Test reporter: stop on error.*

Description

The default reporter, executed when `expect_that` is run interactively. It responds by `stop()`ping on failures and doing nothing otherwise. This will ensure that a failing test will raise an error.

Details

This should be used when doing a quick and dirty test, or during the final automated testing of R CMD check. Otherwise, use a reporter that runs all tests and gives you more context about the problem.

See Also

Other reporters: `CheckReporter`, `DebugReporter`, `FailReporter`, `ListReporter`, `LocationReporter`, `MinimalReporter`, `MultiReporter`, `ProgressReporter`, `Reporter`, `RstudioReporter`, `SilentReporter`, `SummaryReporter`, `TapReporter`, `TeamcityReporter`

SummaryReporter *Test reporter: summary of errors.*

Description

This is a reporter designed for interactive usage: it lets you know which tests have run successfully and as well as fully reporting information about failures and errors.

Details

You can use the `max_reports` field to control the maximum number of detailed reports produced by this reporter. This is useful when running with `auto_test()`

As an additional benefit, this reporter will praise you from time-to-time if all your tests pass.

See Also

Other reporters: CheckReporter, DebugReporter, FailReporter, ListReporter, LocationReporter, MinimalReporter, MultiReporter, ProgressReporter, Reporter, RstudioReporter, SilentReporter, StopReporter, TapReporter, TeamcityReporter

---

TapReporter *Test reporter: TAP format.*

Description

This reporter will output results in the Test Anything Protocol (TAP), a simple text-based interface between testing modules in a test harness. For more information about TAP, see http://testanything.org

See Also

Other reporters: CheckReporter, DebugReporter, FailReporter, ListReporter, LocationReporter, MinimalReporter, MultiReporter, ProgressReporter, Reporter, RstudioReporter, SilentReporter, StopReporter, SummaryReporter, TeamcityReporter

---

TeamcityReporter *Test reporter: Teamcity format.*

Description

This reporter will output results in the Teamcity message format. For more information about Teamcity messages, see http://confluence.jetbrains.com/display/TCD7/Build+Script+Interaction+with+TeamCity

See Also

Other reporters: CheckReporter, DebugReporter, FailReporter, ListReporter, LocationReporter, MinimalReporter, MultiReporter, ProgressReporter, Reporter, RstudioReporter, SilentReporter, StopReporter, SummaryReporter, TapReporter

teardown *Run code on setup/teardown*

Description

Code in a `setup()` block is run immediately in a clean environment. Code in a `teardown()` block is run upon completion of a test file, even if it exits with an error. Multiple calls to `teardown()` will be executed in the order they were created.

Usage

```
teardown(code, env = parent.frame())

setup(code, env = parent.frame())
```

Arguments

code	Code to evaluate
env	Environment in which code will be evaluted. For expert use only.

Examples

```
## Not run:

tmp <- tempfile()
setup(writeLines(tmp, "some test data"))
teardown(unlink(tmp))


## End(Not run)
```

test_dir *Run all tests in directory or package*

Description

Use `test_dir()` for a collection of tests in a directory; use `test_package()` interactively at the console, and `test_check()` inside of R CMD check.

In your own code, you can use `is_testing()` to determine if code is being run as part of a test and `testing_package()` to retrieve the name of the package being tested. You can also check the underlying env var directly `identical(Sys.getenv("TESTTHAT"),"true")` to avoid creating a run-time dependency on testthat.

Usage

```
test_dir(
  path,
  filter = NULL,
  reporter = default_reporter(),
  env = test_env(),
  ...,
  encoding = "unknown",
  load_helpers = TRUE,
  stop_on_failure = FALSE,
  stop_on_warning = FALSE,
  wrap = TRUE
)

test_package(
  package,
  filter = NULL,
  reporter = check_reporter(),
  ...,
  stop_on_failure = TRUE,
  stop_on_warning = FALSE
)

test_check(
  package,
  filter = NULL,
  reporter = check_reporter(),
  ...,
  stop_on_failure = TRUE,
  stop_on_warning = FALSE,
  wrap = TRUE
)

is_testing()

testing_package()
```

Arguments

path	Path to directory containing tests.
filter	If not NULL, only tests with file names matching this regular expression will be executed. Matching be performed on the file name after it has been stripped of "test-" and ".R".
reporter	Reporter to use to summarise output. Can be supplied as a string (e.g. "summary") or as an R6 object (e.g. SummaryReporter$new()). See Reporter for more details and a list of built-in reporters.
env	Environment in which to execute the tests. Expert use only.

...	Additional arguments passed to grepl() to control filtering.
encoding	Deprecated. All files now assumed to be UTF-8.
load_helpers	Source helper files before running the tests? See source_test_helpers() for more details.
stop_on_failure	If TRUE, throw an error if any tests fail. For historical reasons, the default value of stop_on_failure is TRUE for test_package() and test_check() but FALSE for test_dir(), so if you're calling test_dir() you may want to consider explicitly setting stop_on_failure = TRUE.
stop_on_warning	If TRUE, throw an error if any tests generate warnings.
wrap	Automatically wrap all code within test_that()? This ensures that all expectations are reported, even if outside a test block.
package	Name of installed package.

Value

A list of test results.

Test files

For package code, tests should live in tests/testthat.

There are four classes of .R files that have special behaviour:

- Test files start with test and are executed in alphabetical order.
- Helper files start with helper and are executed before tests are run and from devtools::load_all().
- Setup files start with setup and are executed before tests, but not during devtools::load_all().
- Teardown files start with teardown and are executed after the tests are run.

Environments

Each test is run in a clean environment to keep tests as isolated as possible. For package tests, that environment that inherits from the package's namespace environment, so that tests can access internal functions and objects.

R CMD check

To run testthat automatically from R CMD check, make sure you have a tests/testthat.R that contains:

```
library(testthat)
library(yourpackage)

test_check("yourpackage")
```

Examples

```
test_dir(testthat_examples(), reporter = "summary")
test_dir(testthat_examples(), reporter = "minimal")
```

test_file *Run all tests in specified file*

Description

Execute code in the specified file, displaying results using a `reporter`. Use this function when you want to run a single file's worth of tests. You are responsible for ensuring that the functions to test are available in the global environment.

Usage

```
test_file(
  path,
  reporter = default_reporter(),
  env = test_env(),
  start_end_reporter = TRUE,
  load_helpers = TRUE,
  encoding = "unknown",
  wrap = TRUE
)
```

Arguments

`path`	Path to file.
`reporter`	Reporter to use to summarise output. Can be supplied as a string (e.g. "summary") or as an R6 object (e.g. `SummaryReporter$new()`). See Reporter for more details and a list of built-in reporters.
`env`	Environment in which to execute the tests. Expert use only.
`start_end_reporter`	Should the reporters `start_reporter()` and `end_reporter()` methods be called? For expert use only.
`load_helpers`	Source helper files before running the tests? See `source_test_helpers()` for more details.
`encoding`	Deprecated. All files now assumed to be UTF-8.
`wrap`	Automatically wrap all code within `test_that()`? This ensures that all expectations are reported, even if outside a test block.

Details

Any errors that occur in code run outside of `test_that()` will generate a test failure and terminate execution of that test file.

Value

Invisibily, a list with one element for each test.

Examples

```
path <- testthat_example("success")
test_file(path, reporter = "minimal")

# test_file() invisibly returns a list, with one element for each test.
# This can be useful if you want to compute on your test results.
out <- test_file(path, reporter = "minimal")
str(out[[1]])
```

---

test_path *Locate file in testing directory.*

Description

This function is designed to work both interatively and during tests, locating files in the `tests/testthat` directory

Usage

```
test_path(...)
```

Arguments

`...`	Character vectors giving path component.

Value

A character vector giving the path.

---

test_that *Create a test.*

Description

A test encapsulates a series of expectations about small, self-contained set of functionality. Each test is contained in a context and contains multiple expectations.

Usage

```
test_that(desc, code)
```

Arguments

`desc`	test name. Names should be kept as brief as possible, as they are often used as line prefixes.
`code`	test code containing expectations

Details

Tests are evaluated in their own environments, and should not affect global state.

When run from the command line, tests return `NULL` if all expectations are met, otherwise it raises an error.

Examples

```
test_that("trigonometric functions match identities", {
  expect_equal(sin(pi / 4), 1 / sqrt(2))
  expect_equal(cos(pi / 4), 1 / sqrt(2))
  expect_equal(tan(pi / 4), 1)
})
# Failing test:
## Not run:
test_that("trigonometric functions match identities", {
  expect_equal(sin(pi / 4), 1)
})

## End(Not run)
```

---

use_catch *Use Catch for C++ Unit Testing*

Description

Add the necessary infrastructure to enable C++ unit testing in R packages with Catch and `testthat`.

Usage

```
use_catch(dir = getwd())
```

Arguments

`dir`	The directory containing an R package.

Details

Calling `use_catch()` will:

1. Create a file `src/test-runner.cpp`, which ensures that the `testthat` package will understand how to run your package's unit tests,
2. Create an example test file `src/test-example.cpp`, which showcases how you might use Catch to write a unit test,
3. Add a test file `tests/testthat/test-cpp.R`, which ensures that `testthat` will run your compiled tests during invocations of `devtools::test()` or R CMD check, and
4. Create a file `R/catch-routine-registration.R`, which ensures that R will automatically register this routine when `tools::package_native_routine_registration_skeleton()` is invoked.

C++ unit tests can be added to C++ source files within the `src` directory of your package, with a format similar to R code tested with `testthat`. Here's a simple example of a unit test written with `testthat` + Catch:

```
context("C++ Unit Test") {
  test_that("two plus two is four") {
    int result = 2 + 2;
    expect_true(result == 4);
  }
}
```

When your package is compiled, unit tests alongside a harness for running these tests will be compiled into your R package, with the C entry point `run_testthat_tests()`. `testthat` will use that entry point to run your unit tests when detected.

Functions

All of the functions provided by Catch are available with the `CATCH_` prefix – see here for a full list. `testthat` provides the following wrappers, to conform with `testthat`'s R interface:

Function	Catch	Description
`context`	`CATCH_TEST_CASE`	The context of a set of tests.
`test_that`	`CATCH_SECTION`	A test section.
`expect_true`	`CATCH_CHECK`	Test that an expression evaluates to `true`.
`expect_false`	`CATCH_CHECK_FALSE`	Test that an expression evalutes to `false`.
`expect_error`	`CATCH_CHECK_THROWS`	Test that evaluation of an expression throws an exception.
`expect_error_as`	`CATCH_CHECK_THROWS_AS`	Test that evaluation of an expression throws an exception of a specific class

In general, you should prefer using the `testthat` wrappers, as `testthat` also does some work to ensure that any unit tests within will not be compiled or run when using the Solaris Studio compilers (as these are currently unsupported by Catch). This should make it easier to submit packages to CRAN that use Catch.

Symbol Registration

If you've opted to disable dynamic symbol lookup in your package, then you'll need to explicitly export a symbol in your package that `testthat` can use to run your unit tests. `testthat` will look for a routine with one of the names:

```
C_run_testthat_tests
c_run_testthat_tests
run_testthat_tests
```

See Controlling Visibility and Registering Symbols in the **Writing R Extensions** manual for more information.

Advanced Usage

If you'd like to write your own Catch test runner, you can instead use the `testthat::catchSession()` object in a file with the form:

```
#define TESTTHAT_TEST_RUNNER
#include <testthat.h>

void run()
{
    Catch::Session& session = testthat::catchSession();
    // interact with the session object as desired
}
```

This can be useful if you'd like to run your unit tests with custom arguments passed to the Catch session.

Standalone Usage

If you'd like to use the C++ unit testing facilities provided by Catch, but would prefer not to use the regular `testthat` R testing infrastructure, you can manually run the unit tests by inserting a call to:

```
.Call("run_testthat_tests", PACKAGE = <pkgName>)
```

as necessary within your unit test suite.

See Also

Catch, the library used to enable C++ unit testing.

---

verify_output *Verify output*

Description

This is a regression test that records interwoven code and output into a file, in a similar way to kniting an `.Rmd` (but see caveats below).

`verify_output()` designed particularly for testing print methods and error messages, where the primary goal is to ensure that the output is helpful to a human. Obviously, you can't test that with code, so the best you can do is make the results explicit by saving them to text file. This makes the output easy to see in code reviews, and ensures that you don't change the output accidentally.

`verify_output()` is designed to be used with git: to see what has changed from the previous run, you'll need to use git diff or similar.

Usage

```
verify_output(
  path,
  code,
  width = 80,
  crayon = FALSE,
  unicode = FALSE,
  env = caller_env()
)
```

Arguments

path	Path to record results. This should usually be a call to test_path() to ensures that same path is used when run interactively (when the working directory is typically the project root), and when run as an autmated test (when the working directory will be tests/testthat).
code	Code to execute. This will usually be a multiline expression contained within {} (similarly to test_that() calls).
width	Width of console output
crayon	Enable crayon package colouring?
unicode	Enable cli package UTF-8 symbols? If you set this to TRUE, call skip_if(!cli::is_utf8_output()) to disable the test on your CI platforms that don't support UTF-8 (e.g. Windows).
env	The environment to evaluate code in.

Syntax

verify_output() can only capture the abstract syntax tree, losing all whitespace and comments. To mildy offset this limitation:

- Strings are converted to R comments in the output.
- Strings starting with # are converted to headers in the output.

CRAN

On CRAN, verify_output() will never fail, even if the output changes. This avoids false positives because tests of print methods and error messages are often fragile due to implicit dependencies on other packages, and failure does not imply incorrect computation, just a change in presentation.

Examples

```
# The first argument would usually be `test_path("informative-name.txt")`
# but that is not permitted in examples
path <- tempfile()
verify_output(path, {
   head(mtcars)
   log(-10)
```

```
   "a" * 3
})
writeLines(readLines(path))

# Use strings to create comments in the output
verify_output(tempfile(), {
   "Print method"
   head(mtcars)

   "Warning"
   log(-10)

   "Error"
   "a" * 3
})

# Use strings starting with # to create headings
verify_output(tempfile(), {
   "# Base functions"
   head(mtcars)
   log(-10)
   "a" * 3
})
```

Index