Package 'testthat'

May 8, 2026

Title Unit Testing for R

Version 3.3.2

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

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

Depends R (>= 4.1.0)

Imports brio (>= 1.1.5), callr (>= 3.7.6), cli (>= 3.6.5), desc (>= 1.4.3), evaluate (>= 1.0.4), jsonlite (>= 2.0.0), lifecycle (>= 1.0.4), magrittr (>= 2.0.3), methods, pkgload (>= 1.4.0), praise (>= 1.0.0), processx (>= 3.8.6), ps (>= 1.9.1), R6 (>= 2.6.1), rlang (>= 1.1.6), utils, waldo (>= 0.6.2), withr (>= 3.0.2)

Suggests covr, curl (>= 0.9.5), diffviewer (>= 0.1.0), digest (>= 0.6.33), gh, knitr, otel, otelsdk, rmarkdown, rstudioapi, S7, shiny, usethis, vctrs (>= 0.1.0), xml2

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Config/testthat/parallel true

Config/testthat/start-first watcher, parallel*

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation yes

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

Maintainer Hadley Wickham <hadley@posit.co>

Repository CRAN

Date/Publication 2026-01-11 09:10:02 UTC

Contents

`CheckReporter` *Report results for* `R CMD check`

Description

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

See Also

Other reporters: `DebugReporter`, `FailReporter`, `JunitReporter`, `ListReporter`, `LlmReporter`, `LocationReporter`, `MinimalReporter`, `MultiReporter`, `ProgressReporter`, `RStudioReporter`, `Reporter`, `SilentReporter`, `SlowReporter`, `StopReporter`, `SummaryReporter`, `TapReporter`, `TeamcityReporter`

`comparison-expectations`

Do you expect a value bigger or smaller than this?

Description

These functions compare values of comparable data types, such as numbers, dates, and times.

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`, `expected`
: A value to compare and its expected bound.

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

See Also

Other expectations: `equality-expectations`, `expect_error()`, `expect_length()`, `expect_match()`, `expect_named()`, `expect_null()`, `expect_output()`, `expect_reference()`, `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 *Interactively debug failing tests*

Description

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

See Also

Other reporters: CheckReporter, FailReporter, JunitReporter, ListReporter, LlmReporter, LocationReporter, MinimalReporter, MultiReporter, ProgressReporter, RStudioReporter, Reporter, SilentReporter, SlowReporter, StopReporter, SummaryReporter, TapReporter, TeamcityReporter

---

`equality-expectations` *Do you expect this value?*

Description

These functions provide two levels of strictness when comparing a computation to a reference value. `expect_identical()` is the baseline; `expect_equal()` relaxes the test to ignore small numeric differences.

In the 2nd edition, `expect_identical()` uses `identical()` and `expect_equal` uses `all.equal()`. In the 3rd edition, both functions use waldo. They differ only in that `expect_equal()` sets `tolerance = testthat_tolerance()` so that small floating point differences are ignored; this also implies that (e.g.) `1` and `1L` are treated as equal.

Usage

```
expect_equal(
  object,
  expected,
  ...,
  tolerance = if (edition_get() >= 3) testthat_tolerance(),
  info = NULL,
  label = NULL,
  expected.label = NULL
)

expect_identical(
  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.

`...` **3e**: passed on to `waldo::compare()`. See its docs to see other ways to control comparison.

2e: passed on to `compare()`/`identical()`.

`tolerance` **3e**: passed on to `waldo::compare()`. If non-`NULL`, will ignore small floating point differences. It uses same algorithm as `all.equal()` so the tolerance is usually relative (i.e. `mean(abs(x - y) / mean(abs(y)) < tolerance`), except when the differences are very small, when it becomes absolute (i.e. `mean(abs(x - y) < tolerance`). See waldo documentation for more details.

2e: passed on to `compare()`, if set. It's hard to reason about exactly what tolerance means because depending on the precise code path it could be either an absolute or relative tolerance.

`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()`/`expect_mapequal()` to test for set equality.
- `expect_reference()` to test if two names point to same memory address.

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

Examples

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

# Use expect_equal() when testing for numeric equality
## Not run:
expect_identical(sqrt(2) ^ 2, 2)

## End(Not run)
expect_equal(sqrt(2) ^ 2, 2)
```

---

expect_all_equal *Do you expect every value in a vector to have this value?*

Description

These expectations are similar to `expect_true(all(x == "x"))`, `expect_true(all(x))` and `expect_true(all(!x))` but give more informative failure messages if the expectations are not met.

Usage

```
expect_all_equal(object, expected)

expect_all_true(object)

expect_all_false(object)
```

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.

Examples

```
x1 <- c(1, 1, 1, 1, 1, 1)
expect_all_equal(x1, 1)

x2 <- c(1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 2)
show_failure(expect_all_equal(x2, 1))

# expect_all_true() and expect_all_false() are helpers for common cases
set.seed(1016)
show_failure(expect_all_true(rpois(100, 10) < 20))
show_failure(expect_all_false(rpois(100, 10) > 20))
```

---

expect_error *Do you expect an error, warning, message, or other condition?*

Description

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

In the 3rd edition, these functions match (at most) a single condition. All additional and non-matching (if `regexp` or `class` are used) conditions will bubble up outside the expectation. If these additional conditions are important you'll need to catch them with additional `expect_message()`/`expect_warning()` calls; if they're unimportant you can ignore with `suppressMessages()`/`suppressWarnings()`.

It can be tricky to test for a combination of different conditions, such as a message followed by an error. `expect_snapshot()` is often an easier alternative for these more complex cases.

Usage

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

expect_warning(
  object,
  regexp = NULL,
  class = NULL,
  ...,
  inherit = TRUE,
  all = FALSE,
  info = NULL,
  label = NULL
)

expect_message(
  object,
  regexp = NULL,
  class = NULL,
  ...,
  inherit = TRUE,
  all = FALSE,
  info = NULL,
  label = NULL
)

expect_condition(
  object,
  regexp = NULL,
  class = NULL,
  ...,
  inherit = 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.

- A character vector giving a regular expression that must match the error message.
- If `NULL`, the default, asserts that there should be an error, but doesn't test for a specific value.
- If `NA`, asserts that there should be no errors, but we now recommend using `expect_no_error()` and friends instead.

Note that you should only use `message` with errors/warnings/messages that you generate. Avoid tests that rely on the specific text generated by another package since this can easily change. If you do need to test text generated by another package, either protect the test with `skip_on_cran()` or use `expect_snapshot()`.

`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`

`fixed` If `TRUE`, treats `regexp` as a string to be matched exactly (not a regular expressions). Overrides `perl`.

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

`inherit` Whether to match `regexp` and `class` across the ancestry of chained errors.

`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.

`all` *DEPRECATED* If you need to test multiple warnings/messages you now need to use multiple calls to `expect_message()`/ `expect_warning()`

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`.

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 recommend using `expect_snapshot()` instead.

See Also

`expect_no_error()`, `expect_no_warning()`, `expect_no_message()`, and `expect_no_condition()` to assert that code runs without errors/warnings/messages/conditions.

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

Examples

```
# Errors -------------------------------------------------------------------
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)

# Note that `expect_error()` returns the error object so you can test
# its components if needed
err <- expect_error(rlang::abort("a", n = 10))
expect_equal(err$n, 10)

# 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)

# 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)
```

`expect_invisible` *Do you expect the result to be (in)visible?*

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 *Do you expect an object with this length or shape?*

Description

expect_length() inspects the length() of an object; expect_shape() inspects the "shape" (i.e. nrow(), ncol(), or dim()) of higher-dimensional objects like data.frames, matrices, and arrays.

Usage

```
expect_length(object, n)

expect_shape(object, ..., nrow, ncol, dim)
```

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.
...	Not used; used to force naming of other arguments.
nrow, ncol	Expected nrow()/ncol() of object.
dim	Expected dim() of object.

See Also

expect_vector() to make assertions about the "size" of a vector.

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

Examples

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

x <- matrix(1:9, nrow = 3)
expect_shape(x, nrow = 3)
show_failure(expect_shape(x, nrow = 4))
expect_shape(x, ncol = 3)
show_failure(expect_shape(x, ncol = 4))
expect_shape(x, dim = c(3, 3))
show_failure(expect_shape(x, dim = c(3, 4, 5)))
```

---

expect_match *Do you expect a string to match this pattern?*

Description

Do you expect a string to match this pattern?

Usage

```
expect_match(
  object,
  regexp,
  perl = FALSE,
  fixed = FALSE,
  ...,
  all = TRUE,
```

```
  info = NULL,
  label = NULL
)

expect_no_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`	If `TRUE`, treats `regexp` as a string to be matched exactly (not a regular expressions). Overrides `perl`.
`...`	Arguments passed on to `base::grepl` `ignore.case` logical. 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()` checks if a character vector matches a regular expression, powered by `grepl()`.

`expect_no_match()` provides the complementary case, checking that a character vector *does not* match a regular expression.

Functions

- `expect_no_match()`: Check that a string doesn't match a regular expression.

See Also

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

Examples

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

show_failure(expect_match("Testing is fun", "horrible"))
show_failure(expect_match("Testing is fun", "horrible", fixed = TRUE))

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

---

expect_named *Do you expect a vector with (these) names?*

Description

You can either check for the presence of names (leaving `expected` blank), specific names (by supplying 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_null(), expect_output(), expect_reference(), 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_no_error *Do you expect the absence of errors, warnings, messages, or other conditions?*

Description

These expectations are the opposite of expect_error(), expect_warning(), expect_message(), and expect_condition(). They assert the absence of an error, warning, or message, respectively.

Usage

```
expect_no_error(object, ..., message = NULL, class = NULL)

expect_no_warning(object, ..., message = NULL, class = NULL)

expect_no_message(object, ..., message = NULL, class = NULL)

expect_no_condition(object, ..., message = NULL, class = 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.
...	These dots are for future extensions and must be empty.
message, class	The default, message = NULL, class = NULL, will fail if there is any error/warning/message/condition.

In many cases, particularly when testing warnings and messages, you will want to be more specific about the condition you are hoping **not** to see, i.e. the condition that motivated you to write the test. Similar to `expect_error()` and friends, you can specify the `message` (a regular expression that the message of the condition must match) and/or the `class` (a class the condition must inherit from). This ensures that the message/warnings you don't want never recur, while allowing new messages/warnings to bubble up for you to deal with.

Note that you should only use `message` with errors/warnings/messages that you generate, or that base R generates (which tend to be stable). Avoid tests that rely on the specific text generated by another package since this can easily change. If you do need to test text generated by another package, either protect the test with `skip_on_cran()` or use `expect_snapshot()`.

Examples

```
expect_no_warning(1 + 1)

foo <- function(x) {
  warning("This is a problem!")
}

# warning doesn't match so bubbles up:
expect_no_warning(foo(), message = "bananas")

# warning does match so causes a failure:
try(expect_no_warning(foo(), message = "problem"))
```

---

expect_null *Do you expect* `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_named(), expect_output(), expect_reference(), expect_silent(), inheritance-expectations, logical-expectations

Examples

```
x <- NULL
y <- 10

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

---

expect_output *Do you expect printed output to match this pattern?*

Description

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

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).

`fixed` If `TRUE`, treats `regexp` as a string to be matched exactly (not a regular expressions). Overrides `perl`.

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

`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_named()`, `expect_null()`, `expect_reference()`, `expect_silent()`, `inheritance-expectations`, `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 *Do you expect a vector containing these 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_contains(x, y)` tests that `x` contains every element of `y` (i.e. `y` is a subset of `x`).
- `expect_in(x, y)` tests that every element of `x` is in `y` (i.e. `x` is a subset of `y`).
- `expect_disjoint(x, y)` tests that no element of `x` is in `y` (i.e. `x` is disjoint from `y`).
- `expect_mapequal(x, y)` treats lists as if they are mappings between names and values. Concretely, checks that `x` and `y` have the same names, then checks that `x[names(y)]` equals `y`.

Usage

```
expect_setequal(object, expected)

expect_mapequal(object, expected)

expect_contains(object, expected)

expect_in(object, expected)

expect_disjoint(object, expected)
```

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.

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 *Do you expect code to execute silently?*

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_named(), expect_null(), expect_output(), expect_reference(), inheritance-expectations, logical-expectations

Examples

```
expect_silent("123")

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

## End(Not run)
```

---

expect_snapshot *Do you expect this code to run the same way as last time?*

Description

Snapshot tests (aka golden tests) are similar to unit tests except that the expected result is stored in a separate file that is managed by testthat. Snapshot tests are useful for when the expected value is large, or when the intent of the code is something that can only be verified by a human (e.g. this is a useful error message). Learn more in `vignette("snapshotting")`.

`expect_snapshot()` runs code as if you had executed it at the console, and records the results, including output, messages, warnings, and errors. If you just want to compare the result, try `expect_snapshot_value()`.

Usage

```
expect_snapshot(
  x,
  cran = FALSE,
  error = FALSE,
  transform = NULL,
  variant = NULL,
  cnd_class = FALSE
)
```

Arguments

x	Code to evaluate.
cran	Should these expectations be verified on CRAN? By default, they are not, because snapshot tests tend to be fragile because they often rely on minor details of dependencies.
error	Do you expect the code to throw an error? The expectation will fail (even on CRAN) if an unexpected error is thrown or the expected error is not thrown.
transform	Optionally, a function to scrub sensitive or stochastic text from the output. Should take a character vector of lines as input and return a modified character vector as output.
variant	If non-NULL, results will be saved in _snaps/{variant}/{test.md}, so variant must be a single string suitable for use as a directory name. You can use variants to deal with cases where the snapshot output varies and you want to capture and test the variations. Common use cases include variations for operating system, R version, or version of key dependency. Variants are an advanced feature. When you use them, you'll need to carefully think about your testing strategy to ensure that all important variants are covered by automated tests, and ensure that you have a way to get snapshot changes out of your CI system and back into the repo. Note that there's no way to declare all possible variants up front which means that as soon as you start using variants, you are responsible for deleting snapshot variants that are no longer used. (testthat will still delete all variants if you delete the test.)
cnd_class	Whether to include the class of messages, warnings, and errors in the snapshot. Only the most specific class is included, i.e. the first element of class(cnd).

Workflow

The first time that you run a snapshot expectation it will run x, capture the results, and record them in tests/testthat/_snaps/{test}.md. Each test file gets its own snapshot file, e.g. test-foo.R will get _snaps/foo.md.

It's important to review the Markdown files and commit them to git. They are designed to be human readable, and you should always review new additions to ensure that the salient information has been captured. They should also be carefully reviewed in pull requests, to make sure that snapshots have updated in the expected way.

On subsequent runs, the result of `x` will be compared to the value stored on disk. If it's different, the expectation will fail, and a new file `_snaps/{test}.new.md` will be created. If the change was deliberate, you can approve the change with `snapshot_accept()` and then the tests will pass the next time you run them.

Note that snapshotting can only work when executing a complete test file (with `test_file()`, `test_dir()`, or friends) because there's otherwise no way to figure out the snapshot path. If you run snapshot tests interactively, they'll just display the current value.

`expect_snapshot_file` *Do you expect this code to create the same file as last time?*

Description

Whole file snapshot testing is designed for testing objects that don't have a convenient textual representation, with initial support for images (`.png`, `.jpg`, `.svg`), data frames (`.csv`), and text files (`.R`, `.txt`, `.json`, ...).

The first time `expect_snapshot_file()` is run, it will create `_snaps/{test}/{name}.{ext}` containing reference output. Future runs will be compared to this reference: if different, the test will fail and the new results will be saved in `_snaps/{test}/{name}.new.{ext}`. To review failures, call `snapshot_review()`.

We generally expect this function to be used via a wrapper that takes care of ensuring that output is as reproducible as possible, e.g. automatically skipping tests where it's known that images can't be reproduced exactly.

Usage

```
expect_snapshot_file(
  path,
  name = basename(path),
  binary = deprecated(),
  cran = FALSE,
  compare = NULL,
  transform = NULL,
  variant = NULL
)

announce_snapshot_file(path, name = basename(path))

compare_file_binary(old, new)

compare_file_text(old, new)
```

Arguments

`path`	Path to file to snapshot. Optional for `announce_snapshot_file()` if `name` is supplied.

name	Snapshot name, taken from path by default.
binary	**[Deprecated]** Please use the compare argument instead.
cran	Should these expectations be verified on CRAN? By default, they are not, because snapshot tests tend to be fragile because they often rely on minor details of dependencies.
compare	A function used to compare the snapshot files. It should take two inputs, the paths to the old and new snapshot, and return either TRUE or FALSE. This defaults to compare_file_text if name has extension .r, .R, .Rmd, .md, or .txt, and otherwise uses compare_file_binary. compare_file_binary() compares byte-by-byte and compare_file_text() compares lines-by-line, ignoring the difference between Windows and Mac/Linux line endings.
transform	Optionally, a function to scrub sensitive or stochastic text from the output. Should take a character vector of lines as input and return a modified character vector as output.
variant	If not-NULL, results will be saved in _snaps/{variant}/{test}/{name}. This allows you to create different snapshots for different scenarios, like different operating systems or different R versions. Note that there's no way to declare all possible variants up front which means that as soon as you start using variants, you are responsible for deleting snapshot variants that are no longer used. (testthat will still delete all variants if you delete the test.)
old, new	Paths to old and new snapshot files.

Announcing snapshots

testthat automatically detects dangling snapshots that have been written to the _snaps directory but which no longer have corresponding R code to generate them. These dangling files are automatically deleted so they don't clutter the snapshot directory.

This can cause problems if your test is conditionally executed, either because of an if statement or a skip(). To avoid files being deleted in this case, you can call announce_snapshot_file() before the conditional code.

```
test_that("can save a file", {
  if (!can_save()) {
    announce_snapshot_file(name = "data.txt")
    skip("Can't save file")
  }
  path <- withr::local_tempfile()
  expect_snapshot_file(save_file(path, mydata()), "data.txt")
})
```

Examples

```
# To use expect_snapshot_file() you'll typically need to start by writing
# a helper function that creates a file from your code, returning a path
save_png <- function(code, width = 400, height = 400) {
```

```
  path <- tempfile(fileext = ".png")
  png(path, width = width, height = height)
  on.exit(dev.off())
  code

  path
}
path <- save_png(plot(1:5))
path

## Not run:
expect_snapshot_file(save_png(hist(mtcars$mpg)), "plot.png")

## End(Not run)

# You'd then also provide a helper that skips tests where you can't
# be sure of producing exactly the same output.
expect_snapshot_plot <- function(name, code) {
  # Announce the file before touching skips or running `code`. This way,
  # if the skips are active, testthat will not auto-delete the corresponding
  # snapshot file.
  name <- paste0(name, ".png")
  announce_snapshot_file(name = name)

  # Other packages might affect results
  skip_if_not_installed("ggplot2", "2.0.0")
  # Or maybe the output is different on some operating systems
  skip_on_os("windows")
  # You'll need to carefully think about and experiment with these skips

  path <- save_png(code)
  expect_snapshot_file(path, name)
}
```

`expect_snapshot_value` *Do you expect this code to return the same value as last time?*

Description

Captures the result of function, flexibly serializing it into a text representation that's stored in a snapshot file. See `expect_snapshot()` for more details on snapshot testing.

Usage

```
expect_snapshot_value(
  x,
  style = c("json", "json2", "deparse", "serialize"),
  cran = FALSE,
  tolerance = testthat_tolerance(),
  ...,
```

```
  variant = NULL
)
```

Arguments

x	Code to evaluate.
style	Serialization style to use: • json uses jsonlite::fromJSON() and jsonlite::toJSON(). This produces the simplest output but only works for relatively simple objects. • json2 uses jsonlite::serializeJSON() and jsonlite::unserializeJSON() which are more verbose but work for a wider range of type. • deparse uses deparse(), which generates a depiction of the object using R code. • serialize() produces a binary serialization of the object using serialize(). This is all but guaranteed to work for any R object, but produces a completely opaque serialization.
cran	Should these expectations be verified on CRAN? By default, they are not, because snapshot tests tend to be fragile because they often rely on minor details of dependencies.
tolerance	Numerical tolerance: any differences (in the sense of base::all.equal()) smaller than this value will be ignored. The default tolerance is sqrt(.Machine$double.eps), unless long doubles are not available, in which case the test is skipped.
...	Passed on to waldo::compare() so you can control the details of the comparison.
variant	If non-NULL, results will be saved in _snaps/{variant}/{test.md}, so variant must be a single string suitable for use as a directory name. You can use variants to deal with cases where the snapshot output varies and you want to capture and test the variations. Common use cases include variations for operating system, R version, or version of key dependency. Variants are an advanced feature. When you use them, you'll need to carefully think about your testing strategy to ensure that all important variants are covered by automated tests, and ensure that you have a way to get snapshot changes out of your CI system and back into the repo. Note that there's no way to declare all possible variants up front which means that as soon as you start using variants, you are responsible for deleting snapshot variants that are no longer used. (testthat will still delete all variants if you delete the test.)

---

expect_success *Test your custom expectations*

Description

`expect_success()` checks that there's exactly one success and no failures; `expect_failure()` checks that there's exactly one failure and no successes. `expect_snapshot_failure()` records the failure message so that you can manually check that it is informative.

Use `show_failure()` in examples to print the failure message without throwing an error.

Usage

```
expect_success(expr)

expect_failure(expr, message = NULL, ...)

expect_snapshot_failure(expr)

show_failure(expr)
```

Arguments

`expr`	Code to evaluate
`message`	Check that the failure message matches this regexp.
`...`	Other arguments passed on to `expect_match()`.

---

expect_vector *Do you expect a vector with this size and/or prototype?*

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

```
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))
```

---

extract_test *Extract a reprex from a failed expectation*

Description

`extract_test()` creates a minimal reprex for a failed expectation. It extracts all non-test code before the failed expectation as well as all code inside the test up to and including the failed expectation.

This is particularly useful when you're debugging test failures in someone else's package.

Usage

```
extract_test(location, path = stdout(), package = Sys.getenv("TESTTHAT_PKG"))
```

Arguments

`location`	A string giving the location in the form `FILE:LINE[:COLUMN]`.
`path`	Path to write the reprex to. Defaults to `stdout()`.
`package`	If supplied, will be used to construct a test environment for the extracted code.

Value

This function is called for its side effect of rendering a reprex to `path`. This function will never error: if extraction fails, the error message will be written to `path`.

Examples

```
# If you see a test failure like this:
# -- Failure (test-extract.R:46:3): errors if can't find test -------------
# Expected FALSE to be TRUE.
# Differences:
# `actual`:   FALSE
# `expected`: TRUE

# You can run this:
## Not run: extract_test("test-extract.R:46:3")
# to see just the code needed to reproduce the failure
```

`fail` *Declare that an expectation either passes or fails*

Description

These are the primitives that you can use to implement your own expectations. Every path through an expectation should either call `pass()`, `fail()`, or throw an error (e.g. if the arguments are invalid). Expectations should always return `invisible(act$val)`.

Learn more about creating your own expectations in `vignette("custom-expectation")`.

Usage

```
fail(
  message = "Failure has been forced",
  info = NULL,
  srcref = NULL,
  trace_env = caller_env(),
  trace = NULL
)

pass()
```

Arguments

`message`	A character vector describing the failure. The first element should describe the expected value, and the second (and optionally subsequence) elements should describe what was actually seen.
`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_env`	If `trace` is not specified, this is used to generate an informative traceback for failures. You should only need to set this if you're calling `fail()` from a helper function; see `vignette("custom-expectation")` for details.
`trace`	An optional backtrace created by `rlang::trace_back()`. When supplied, the expectation is displayed with the backtrace. Expert use only.

Examples

```
expect_length <- function(object, n) {
  act <- quasi_label(rlang::enquo(object), arg = "object")

  act_n <- length(act$val)
  if (act_n != n) {
    fail(sprintf("%s has length %i, not length %i.", act$lab, act_n, n))
  } else {
```

```
    pass()
  }

  invisible(act$val)
}
```

`FailReporter` *Fail if any tests fail*

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`, `JunitReporter`, `ListReporter`, `LlmReporter`, `LocationReporter`, `MinimalReporter`, `MultiReporter`, `ProgressReporter`, `RStudioReporter`, `Reporter`, `SilentReporter`, `SlowReporter`, `StopReporter`, `SummaryReporter`, `TapReporter`, `TeamcityReporter`

`inheritance-expectations`
Do you expect an S3/S4/R6/S7 object that inherits from this class?

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_s3_class(x, NA)` checks that `x` isn't an S3 object.
- `expect_s4_class(x, class)` checks that `x` is an S4 object that `is()` `class`.
- `expect_s4_class(x, NA)` checks that `x` isn't an S4 object.
- `expect_r6_class(x, class)` checks that `x` an R6 object that inherits from `class`.
- `expect_s7_class(x, Class)` checks that `x` is an S7 object that `S7::S7_inherits()` from `Class`

See `expect_vector()` for testing properties of objects created by vctrs.

Usage

```
expect_type(object, type)

expect_s3_class(object, class, exact = FALSE)

expect_s4_class(object, class)

expect_r6_class(object, class)

expect_s7_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	The required type varies depending on the function: • expect_type(): a string. • expect_s3_class(): a string or character vector. The behaviour of multiple values (i.e. a character vector) is controlled by the exact argument. • expect_s4_class(): a string. • expect_r6_class(): a string. • expect_s7_class(): an S7::S7_class() object. For historical reasons, expect_s3_class() and expect_s4_class() also take NA to assert that the object is not an S3 or S4 object.
exact	If FALSE, the default, checks that object inherits from any element of class. If TRUE, checks that object has a class that exactly matches class.

See Also

Other expectations: comparison-expectations, equality-expectations, expect_error(), expect_length(), expect_match(), expect_named(), expect_null(), expect_output(), expect_reference(), 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")

f <- factor(c("a", "b", "c"))
o <- ordered(f)
```

```
# Using multiple class names tests if the object inherits from any of them
expect_s3_class(f, c("ordered", "factor"))
# Use exact = TRUE to test for exact match
show_failure(expect_s3_class(f, c("ordered", "factor"), exact = TRUE))
expect_s3_class(o, c("ordered", "factor"), exact = TRUE)

# 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")
```

`is_testing` *Determine testing status*

Description

These functions help you determine if you code is running in a particular testing context:

- `is_testing()` is `TRUE` inside a test.
- `is_snapshot()` is `TRUE` inside a snapshot test
- `is_checking()` is `TRUE` inside of `R CMD check` (i.e. by `test_check()`).
- `is_parallel()` is `TRUE` if the tests are run in parallel.
- `testing_package()` gives name of the package being tested.

A common use of these functions is to compute a default value for a `quiet` argument with `is_testing() && !is_snapshot()`. In this case, you'll want to avoid an run-time dependency on testthat, in which case you should just copy the implementation of these functions into a `utils.R` or similar.

Usage

```
is_testing()

is_parallel()

is_checking()

is_snapshot()

testing_package()
```

JunitReporter *Report results 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.

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: https://github.com/testmoapp/junitxml

See Also

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

ListReporter *Capture test results and metadata*

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, JunitReporter, LlmReporter, LocationReporter, MinimalReporter, MultiReporter, ProgressReporter, RStudioReporter, Reporter, SilentReporter, SlowReporter, StopReporter, SummaryReporter, TapReporter, TeamcityReporter

LlmReporter *Report test progress for LLMs*

Description

`LlmReporter` is designed for use with Large Language Models (LLMs). It reports problems (warnings, skips, errors, and failures) as they occur and the total number of successes at the end.

`LlmReporter` is used by default when tests are run by a coding agent. Currently we detect Claude Code, Cursor, and Gemini CLI. If using another tool, configure it to set env var `AGENT=1`.

See Also

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

local_edition *Temporarily change the active testthat edition*

Description

`local_edition()` allows you to temporarily (within a single test or a single test file) change the active edition of testthat. `edition_get()` allows you to retrieve the currently active edition.

Usage

```
local_edition(x, .env = parent.frame())

edition_get()
```

Arguments

`x`	Edition Should be a single integer.
`.env`	Environment that controls scope of changes. For expert use only.

`local_mocked_bindings` *Temporarily redefine function definitions*

Description

`with_mocked_bindings()` and `local_mocked_bindings()` provide tools for "mocking", temporarily redefining a function so that it behaves differently during tests. This is helpful for testing functions that depend on external state (i.e. reading a value from a file or a website, or pretending a package is or isn't installed).

Learn more in `vignette("mocking")`.

Usage

```
local_mocked_bindings(..., .package = NULL, .env = caller_env())

with_mocked_bindings(code, ..., .package = NULL)
```

Arguments

`...`	Name-value pairs providing new values (typically functions) to temporarily replace the named bindings.
`.package`	The name of the package where mocked functions should be inserted. Generally, you should not supply this as it will be automatically detected when whole package tests are run or when there's one package under active development (i.e. loaded with `pkgload::load_all()`). We don't recommend using this to mock functions in other packages, as you should not modify namespaces that you don't own.
`.env`	Environment that defines effect scope. For expert use only.
`code`	Code to execute with specified bindings.

Use

There are four places that the function you are trying to mock might come from:

- Internal to your package.
- Imported from an external package via the `NAMESPACE`.
- The base environment.
- Called from an external package with `::`.

They are described in turn below.

(To mock S3 & S4 methods and R6 classes see `local_mocked_s3_method()`, `local_mocked_s4_method()`, and `local_mocked_r6_class()`.)

Internal & imported functions:

You mock internal and imported functions the same way. For example, take this code:

```
some_function <- function() {
  another_function()
}
```

It doesn't matter whether `another_function()` is defined by your package or you've imported it from a dependency with `@import` or `@importFrom`, you mock it the same way:

```
local_mocked_bindings(
  another_function = function(...) "new_value"
)
```

Base functions:
To mock a function in the base package, you need to make sure that you have a binding for this function in your package. It's easiest to do this by binding the value to `NULL`. For example, if you wanted to mock `interactive()` in your package, you'd need to include this code somewhere in your package:

```
interactive <- NULL
```

Why is this necessary? `with_mocked_bindings()` and `local_mocked_bindings()` work by temporarily modifying the bindings within your package's namespace. When these tests are running inside of `R CMD check` the namespace is locked which means it's not possible to create new bindings so you need to make sure that the binding exists already.

Namespaced calls:
It's trickier to mock functions in other packages that you call with `::`. For example, take this minor variation:

```
some_function <- function() {
  anotherpackage::another_function()
}
```

To mock this function, you'd need to modify `another_function()` inside the `anotherpackage` package. You *can* do this by supplying the `.package` argument to `local_mocked_bindings()` but we don't recommend it because it will affect all calls to `anotherpackage::another_function()`, not just the calls originating in your package. Instead, it's safer to either import the function into your package, or make a wrapper that you can mock:

```
some_function <- function() {
  my_wrapper()
}
my_wrapper <- function(...) {
  anotherpackage::another_function(...)
}

local_mocked_bindings(
  my_wrapper = function(...) "new_value"
)
```

Multiple return values / sequence of outputs:
To mock a function that returns different values in sequence, for instance an API call whose status would be 502 then 200, or an user input to `readline()`, you can use `mock_output_sequence()`

```
local_mocked_bindings(readline = mock_output_sequence("3", "This is a note", "n"))
```

See Also

Other mocking: `mock_output_sequence()`

`local_mocked_r6_class` *Mock an R6 class*

Description

This function allows you to temporarily override an R6 class definition. It works by creating a subclass then using `local_mocked_bindings()` to temporarily replace the original definition. This means that it will not affect subclasses of the original class; please file an issue if you need this.

Learn more about mocking in `vignette("mocking")`.

Usage

```
local_mocked_r6_class(
  class,
  public = list(),
  private = list(),
  frame = caller_env()
)
```

Arguments

`class`	An R6 class definition.
`public, private`	A named list of public and private methods/data.
`frame`	Calling frame which determines the scope of the mock. Only needed when wrapping in another local helper.

`local_mocked_s3_method`
Mock S3 and S4 methods

Description

These functions temporarily override S3 or S4 methods. They can mock methods that don't already exist, or temporarily remove a method by setting `definition = NULL`.

Learn more about mocking in `vignette("mocking")`.

Usage

```
local_mocked_s3_method(generic, signature, definition, frame = caller_env())

local_mocked_s4_method(generic, signature, definition, frame = caller_env())
```

Arguments

generic	A string giving the name of the generic.
signature	A character vector giving the signature of the method.
definition	A function providing the method definition, or NULL to temporarily remove the method.
frame	Calling frame which determines the scope of the mock. Only needed when wrapping in another local helper.

Examples

```
x <- as.POSIXlt(Sys.time())
local({
  local_mocked_s3_method("length", "POSIXlt", function(x) 42)
  length(x)
})

length(x)
```

---

local_test_context *Temporarily set options for maximum reproducibility*

Description

`local_test_context()` is run automatically by `test_that()` but you may want to run it yourself if you want to replicate test results interactively. If run inside a function, the effects are automatically reversed when the function exits; if running in the global environment, use `withr::deferred_run()` to undo.

`local_reproducible_output()` is run automatically by `test_that()` in the 3rd edition. You might want to call it to override the the default settings inside a test, if you want to test Unicode, coloured output, or a non-standard width.

Usage

```
local_test_context(.env = parent.frame())

local_reproducible_output(
  width = 80,
  crayon = FALSE,
  unicode = FALSE,
  rstudio = FALSE,
  hyperlinks = FALSE,
  lang = "C",
  .env = parent.frame()
)
```

Arguments

.env	Environment to use for scoping; expert use only.
width	Value of the "width" option.
crayon	Determines whether or not crayon (now cli) colour should be applied.
unicode	Value of the "cli.unicode" option. The test is skipped if l10n_info()$`UTF-8` is FALSE.
rstudio	Should we pretend that we're inside of RStudio?
hyperlinks	Should we use ANSI hyperlinks.
lang	Optionally, supply a BCP47 language code to set the language used for translating error messages. This is a lower case two letter ISO 639 country code, optionally followed by "_" or "-" and an upper case two letter ISO 3166 region code.

Details

local_test_context() sets TESTTHAT = "true", which ensures that is_testing() returns TRUE and allows code to tell if it is run by testthat.

In the third edition, local_test_context() also calls local_reproducible_output() which temporary sets the following options:

- cli.dynamic = FALSE so that tests assume that they are not run in a dynamic console (i.e. one where you can move the cursor around).
- cli.unicode (default: FALSE) so that the cli package never generates unicode output (normally cli uses unicode on Linux/Mac but not Windows). Windows can't easily save unicode output to disk, so it must be set to false for consistency.
- cli.condition_width = Inf so that new lines introduced while width-wrapping condition messages don't interfere with message matching.
- crayon.enabled (default: FALSE) suppresses ANSI colours generated by the cli and crayon packages (normally colours are used if cli detects that you're in a terminal that supports colour).
- cli.num_colors (default: 1L) Same as the crayon option.
- lifecycle_verbosity = "warning" so that every lifecycle problem always generates a warning (otherwise deprecated functions don't generate a warning every time).
- max.print = 99999 so the same number of values are printed.
- OutDec = "." so numbers always uses . as the decimal point (European users sometimes set OutDec = ",").
- rlang_interactive = FALSE so that rlang::is_interactive() returns FALSE, and code that uses it pretends you're in a non-interactive environment.
- useFancyQuotes = FALSE so base R functions always use regular (straight) quotes (otherwise the default is locale dependent, see sQuote() for details).
- width (default: 80) to control the width of printed output (usually this varies with the size of your console).

And modifies the following env vars:

- Unsets RSTUDIO, which ensures that RStudio is never detected as running.
- Sets LANGUAGE = "en", which ensures that no message translation occurs.

Finally, it sets the collation locale to "C", which ensures that character sorting the same regardless of system locale.

Examples

```
local({
  local_test_context()
  cat(cli::col_blue("Text will not be colored"))
  cat(cli::symbol$ellipsis)
  cat("\n")
})
test_that("test ellipsis", {
  local_reproducible_output(unicode = FALSE)
  expect_equal(cli::symbol$ellipsis, "...")

  local_reproducible_output(unicode = TRUE)
  expect_equal(cli::symbol$ellipsis, "\u2026")
})
```

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, JunitReporter, ListReporter, LlmReporter, MinimalReporter, MultiReporter, ProgressReporter, RStudioReporter, Reporter, SilentReporter, SlowReporter, StopReporter, SummaryReporter, TapReporter, TeamcityReporter

---

logical-expectations *Do you expect* TRUE *or* 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.

Attributes are ignored.

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.

See Also

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

Examples

```
expect_true(2 == 2)
# Failed expectations will throw an error
show_failure(expect_true(2 != 2))

# where possible, use more specific expectations, to get more informative
# error messages
a <- 1:4
show_failure(expect_true(length(a) == 3))
show_failure(expect_equal(length(a), 3))

x <- c(TRUE, TRUE, FALSE, TRUE)
show_failure(expect_true(all(x)))
show_failure(expect_all_true(x))
```

MinimalReporter *Report minimal results as compactly as possible*

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, JunitReporter, ListReporter, LlmReporter, LocationReporter, MultiReporter, ProgressReporter, RStudioReporter, Reporter, SilentReporter, SlowReporter, StopReporter, SummaryReporter, TapReporter, TeamcityReporter

---

`mock_output_sequence` *Mock a sequence of output from a function*

Description

Specify multiple return values for mocking

Usage

```
mock_output_sequence(..., recycle = FALSE)
```

Arguments

`...`	<dynamic-dots> Values to return in sequence.
`recycle`	whether to recycle. If `TRUE`, once all values have been returned, they will be returned again in sequence.

Value

A function that you can use within `local_mocked_bindings()` and `with_mocked_bindings()`

See Also

Other mocking: `local_mocked_bindings()`

Examples

```
# inside local_mocked_bindings()
## Not run:
local_mocked_bindings(readline = mock_output_sequence("3", "This is a note", "n"))

## End(Not run)
# for understanding
mocked_sequence <- mock_output_sequence("3", "This is a note", "n")
mocked_sequence()
mocked_sequence()
mocked_sequence()
try(mocked_sequence())
recycled_mocked_sequence <- mock_output_sequence(
  "3", "This is a note", "n",
  recycle = TRUE
)
recycled_mocked_sequence()
```

```
recycled_mocked_sequence()
recycled_mocked_sequence()
recycled_mocked_sequence()
```

`MultiReporter` *Run multiple reporters at the same time*

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`, `JunitReporter`, `ListReporter`, `LlmReporter`, `LocationReporter`, `MinimalReporter`, `ProgressReporter`, `RStudioReporter`, `Reporter`, `SilentReporter`, `SlowReporter`, `StopReporter`, `SummaryReporter`, `TapReporter`, `TeamcityReporter`

`ProgressReporter` *Report progress interactively*

Description

`ProgressReporter` is designed for interactive use. Its goal is to give you actionable insights to help you understand the status of your code. This reporter also praises you from time-to-time if all your tests pass. It's the default reporter for `test_dir()`.

`ParallelProgressReporter` is very similar to `ProgressReporter`, but works better for packages that want parallel tests.

`CompactProgressReporter` is a minimal version of `ProgressReporter` designed for use with single files. It's the default reporter for `test_file()`.

See Also

Other reporters: `CheckReporter`, `DebugReporter`, `FailReporter`, `JunitReporter`, `ListReporter`, `LlmReporter`, `LocationReporter`, `MinimalReporter`, `MultiReporter`, `RStudioReporter`, `Reporter`, `SilentReporter`, `SlowReporter`, `StopReporter`, `SummaryReporter`, `TapReporter`, `TeamcityReporter`

RStudioReporter *Report results to 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, JunitReporter, ListReporter, LlmReporter, LocationReporter, MinimalReporter, MultiReporter, ProgressReporter, Reporter, SilentReporter, SlowReporter, StopReporter, SummaryReporter, TapReporter, TeamcityReporter

set_state_inspector *Check for global state changes*

Description

One of the most pernicious challenges to debug is when a test runs fine in your test suite, but fails when you run it interactively (or similarly, it fails randomly when running your tests in parallel). One of the most common causes of this problem is accidentally changing global state in a previous test (e.g. changing an option, an environment variable, or the working directory). This is hard to debug, because it's very hard to figure out which test made the change.

Luckily testthat provides a tool to figure out if tests are changing global state. You can register a state inspector with `set_state_inspector()` and testthat will run it before and after each test, store the results, then report if there are any differences. For example, if you wanted to see if any of your tests were changing options or environment variables, you could put this code in `tests/testthat/helper-state.R`:

```
set_state_inspector(function() {
  list(
    options = options(),
    envvars = Sys.getenv()
  )
})
```

(You might discover other packages outside your control are changing the global state, in which case you might want to modify this function to ignore those values.)

Other problems that can be troublesome to resolve are CRAN check notes that report things like connections being left open. You can easily debug that problem with:

```
set_state_inspector(function() {
  getAllConnections()
})
```

Usage

```
set_state_inspector(callback, tolerance = testthat_tolerance())
```

Arguments

`callback`	Either a zero-argument function that returns an object capturing global state that you're interested in, or `NULL`.
`tolerance`	If non-`NULL`, used as threshold for ignoring small floating point difference when comparing numeric vectors. Using any non-`NULL` value will cause integer and double vectors to be compared based on their values, not their types, and will ignore the difference between `NaN` and `NA_real_`. It uses the same algorithm as `all.equal()`, i.e., first we generate `x_diff` and `y_diff` by subsetting `x` and `y` to look only locations with differences. Then we check that `mean(abs(x_diff - y_diff)) / mean(abs(y_diff))` (or just `mean(abs(x_diff - y_diff))` if `y_diff` is small) is less than `tolerance`.

SilentReporter *Silently collect and all expectations*

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 `$expectations()`.

See Also

Other reporters: `CheckReporter`, `DebugReporter`, `FailReporter`, `JunitReporter`, `ListReporter`, `LlmReporter`, `LocationReporter`, `MinimalReporter`, `MultiReporter`, `ProgressReporter`, `RStudioReporter`, `Reporter`, `SlowReporter`, `StopReporter`, `SummaryReporter`, `TapReporter`, `TeamcityReporter`

---

skip *Skip a test for various reasons*

Description

`skip_if()` and `skip_if_not()` allow you to skip tests, immediately concluding a `test_that()` block without executing any further expectations. This allows you to skip a test without failure, if for some reason it can't be run (e.g. it depends on the feature of a specific operating system, or it requires a specific version of a package).

See `vignette("skipping")` for more details.

Usage

```
skip(message = "Skipping")

skip_if_not(condition, message = NULL)

skip_if(condition, message = NULL)

skip_if_not_installed(pkg, minimum_version = NULL)

skip_unless_r(spec)

skip_if_offline(host = "captive.apple.com")

skip_on_cran()

local_on_cran(on_cran = TRUE, frame = caller_env())

skip_on_os(os, arch = NULL)

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
spec	A version specification like '>= 4.1.0' denoting that this test should only be run on R versions 4.1.0 and later.
host	A string with a hostname to lookup
on_cran	Pretend we're on CRAN (TRUE) or not (FALSE).
frame	Calling frame to tie change to; expect use only.
os	Character vector of one or more operating systems to skip on. Supported values are "windows", "mac", "linux", "solaris", and "emscripten".
arch	Character vector of one or more architectures to skip on. Common values include "i386" (32 bit), "x86_64" (64 bit), and "aarch64" (M1 mac). Supplying arch makes the test stricter; i.e. both os and arch must match in order for the test to be skipped.

`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`.

Helpers

- `skip_if_not_installed("pkg")` skips tests if package "pkg" is not installed or cannot be loaded (using `requireNamespace()`). Generally, you can assume that suggested packages are installed, and you do not need to check for them specifically, unless they are particularly difficult to install.
- `skip_if_offline()` skips if an internet connection is not available (using `curl::nslookup()`) or if the test is run on CRAN. Requires {curl} to be installed and included in the dependencies of your package.
- `skip_if_translated("msg")` skips tests if the "msg" is translated.
- `skip_on_bioc()` skips on Bioconductor (using the `IS_BIOC_BUILD_MACHINE` env var).
- `skip_on_cran()` skips on CRAN (using the `NOT_CRAN` env var set by devtools and friends). `local_on_cran()` gives you the ability to easily simulate what will happen on CRAN.
- `skip_on_covr()` skips when covr is running (using the `R_COVR` env var).
- `skip_on_ci()` skips on continuous integration systems like GitHub Actions, travis, and appveyor (using the `CI` env var).
- `skip_on_os()` skips on the specified operating system(s) ("windows", "mac", "linux", or "solaris").

Examples

```
if (FALSE) skip("Some Important Requirement is not available")

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
})
```

SlowReporter *Find slow tests*

Description

`SlowReporter` is designed to identify slow tests. It reports the execution time for each test and can optionally filter out tests that run faster than a specified threshold (default: 1 second). This reporter is useful for performance optimization and identifying tests that may benefit from optimization or parallelization.

`SlowReporter` is designed to identify slow tests. It reports the execution time for each test, ignoring tests faster than a specified threshold (default: 0.5s).

The easiest way to run it over your package is with `devtools::test(reporter = "slow")`.

See Also

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

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

---

`snapshot_accept` *Accept or reject modified snapshots*

Description

- `snapshot_accept()` accepts all modified snapshots.
- `snapshot_reject()` rejects all modified snapshots by deleting the `.new` variants.
- `snapshot_review()` opens a Shiny app that shows a visual diff of each modified snapshot. This is particularly useful for whole file snapshots created by `expect_snapshot_file()`.

Usage

```
snapshot_accept(files = NULL, path = "tests/testthat")

snapshot_reject(files = NULL, path = "tests/testthat")

snapshot_review(files = NULL, path = "tests/testthat", ...)
```

Arguments

`files`	Optionally, filter effects to snapshots from specified files. This can be a snapshot name (e.g. `foo` or `foo.md`), a snapshot file name (e.g. `testfile/foo.txt`), or a snapshot file directory (e.g. `testfile/`).
`path`	Path to tests.
`...`	Additional arguments passed on to `shiny::runApp()`.

snapshot_download_gh *Download snapshots from GitHub*

Description

If your snapshots fail on GitHub, it can be a pain to figure out exactly why, or to incorporate them into your local package. This function makes it easy, only requiring you to interactively select which job you want to take the artifacts from.

Note that you should not generally need to use this function manually; instead copy and paste from the hint emitted on GitHub. This hint is only emitted when running in a job named "R-CMD-check", since that's where the testthat artifact is typically uploaded.

Usage

```
snapshot_download_gh(repository, run_id, dest_dir = ".")
```

Arguments

repository	Repository owner/name, e.g. `"r-lib/testthat"`.
run_id	Run ID, e.g. `"47905180716"`. You can find this in the action url.
dest_dir	Package root directory. Defaults to the current directory.

StopReporter *Error if any test fails*

Description

The default reporter used when `expect_that()` is run interactively. It responds by displaying a summary of the number of successes and failures and `stop()`ping on if there are any failures.

See Also

Other reporters: `CheckReporter`, `DebugReporter`, `FailReporter`, `JunitReporter`, `ListReporter`, `LlmReporter`, `LocationReporter`, `MinimalReporter`, `MultiReporter`, `ProgressReporter`, `RStudioReporter`, `Reporter`, `SilentReporter`, `SlowReporter`, `SummaryReporter`, `TapReporter`, `TeamcityReporter`

SummaryReporter *Report a summary of failures*

Description

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

You can use the `max_reports` field to control the maximum number of detailed reports produced by this reporter.

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, JunitReporter, ListReporter, LlmReporter, LocationReporter, MinimalReporter, MultiReporter, ProgressReporter, RStudioReporter, Reporter, SilentReporter, SlowReporter, StopReporter, TapReporter, TeamcityReporter

TapReporter *Report results in 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, JunitReporter, ListReporter, LlmReporter, LocationReporter, MinimalReporter, MultiReporter, ProgressReporter, RStudioReporter, Reporter, SilentReporter, SlowReporter, StopReporter, SummaryReporter, TeamcityReporter

TeamcityReporter *Report results in 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, JunitReporter, ListReporter, LlmReporter, LocationReporter, MinimalReporter, MultiReporter, ProgressReporter, RStudioReporter, Reporter, SilentReporter, SlowReporter, StopReporter, SummaryReporter, TapReporter

teardown_env *Run code after all test files*

Description

This environment has no purpose other than as a handle for `withr::defer()`: use it when you want to run code after all tests have been run. Typically, you'll use `withr::defer(cleanup(), teardown_env())` immediately after you've made a mess in a `setup-*.R` file.

Usage

```
teardown_env()
```

test_dir *Run all tests in a directory*

Description

This function is the low-level workhorse that powers `test_local()` and `test_package()`. Generally, you should not call this function directly. In particular, you are responsible for ensuring that the functions to test are available in the test env (e.g. via `load_package`).

See `vignette("special-files")` to learn more about the conventions for test, helper, and setup files that testthat uses, and what you might use each for.

Usage

```
test_dir(
  path,
  filter = NULL,
  reporter = NULL,
  env = NULL,
  ...,
  load_helpers = TRUE,
  stop_on_failure = TRUE,
  stop_on_warning = FALSE,
  package = NULL,
  load_package = c("none", "installed", "source"),
  shuffle = FALSE
)
```

Arguments

path	Path to directory containing tests.
filter	If not NULL, only tests with file names matching this regular expression will be executed. Matching is performed on the file name after it's 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.
load_helpers	Source helper files before running the tests?
stop_on_failure	If TRUE, throw an error if any tests fail.
stop_on_warning	If TRUE, throw an error if any tests generate warnings.
package	If these tests belong to a package, the name of the package.
load_package	Strategy to use for load package code: • "none", the default, doesn't load the package. • "installed", uses library() to load an installed package. • "source", uses pkgload::load_all() to a source package. To configure the arguments passed to load_all(), add this field in your DESCRIPTION file: Config/testthat/load-all: list(export_all = FALSE, helpers = FALSE)
shuffle	If TRUE, randomly reorder the top-level expressions in the file.

Value

A list (invisibly) containing data about the test results.

Environments

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

---

test_file *Run tests in a single file*

Description

Helper, setup, and teardown files located in the same directory as the test will also be run. See vignette("special-files") for details.

Usage

```
test_file(
  path,
  reporter = default_compact_reporter(),
  desc = NULL,
  package = NULL,
  shuffle = FALSE,
  ...
)
```

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.
desc	Optionally, supply a string here to run only a single test (test_that() or describe()) with this description.
package	If these tests belong to a package, the name of the package.
shuffle	If TRUE, randomly reorder the top-level expressions in the file.
...	Additional parameters passed on to test_dir()

Value

A list (invisibly) containing data about the test results.

Environments

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

Examples

```
path <- testthat_example("success")
test_file(path)
test_file(path, desc = "some tests have warnings")
test_file(path, reporter = "minimal")
```

test_package *Run all tests in a package*

Description

- `test_local()` tests a local source package.
- `test_package()` tests an installed package.
- `test_check()` checks a package during `R CMD check`.

See `vignette("special-files")` to learn about the various files that testthat works with.

Usage

```
test_package(package, reporter = check_reporter(), ...)

test_check(package, reporter = check_reporter(), ...)

test_local(
  path = ".",
  reporter = NULL,
  ...,
  load_package = "source",
  shuffle = FALSE
)
```

Arguments

package	If these tests belong to a package, the name of the package.
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.
...	Additional arguments passed to `test_dir()`
path	Path to directory containing tests.
load_package	Strategy to use for load package code: • "none", the default, doesn't load the package. • "installed", uses `library()` to load an installed package. • "source", uses `pkgload::load_all()` to a source package. To configure the arguments passed to `load_all()`, add this field in your DESCRIPTION file: `Config/testthat/load-all: list(export_all = FALSE, helpers = FALSE)`
shuffle	If `TRUE`, randomly reorder the top-level expressions in the file.

Value

A list (invisibly) containing data about the test results.

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")
```

Environments

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

---

test_path *Locate a file in the testing directory*

Description

Many tests require some external file (e.g. a .csv if you're testing a data import function) but the working directory varies depending on the way that you're running the test (e.g. interactively, with devtools::test(), or with R CMD check). test_path() understands these variations and automatically generates a path relative to tests/testthat, regardless of where that directory might reside relative to the current working directory.

Usage

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

Arguments

...	Character vectors giving path components.

Value

A character vector giving the path.

Examples

```
## Not run:
test_path("foo.csv")
test_path("data", "foo.csv")

## End(Not run)
```

`test_that` *Run a test*

Description

A test encapsulates a series of expectations about a small, self-contained unit of functionality. Each test contains one or more expectations, such as `expect_equal()` or `expect_error()`, and lives in a `test/testthat/test*` file, often together with other tests that relate to the same function or set of functions.

Each test has its own execution environment, so an object created in a test also dies with the test. Note that this cleanup does not happen automatically for other aspects of global state, such as session options or filesystem changes. Avoid changing global state, when possible, and reverse any changes that you do make.

Usage

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

Arguments

`desc`	Test name. Names should be brief, but evocative. It's common to write the description so that it reads like a natural sentence, e.g. `test_that("multiplication works", { ... })`.
`code`	Test code containing expectations. Braces (`{}`) should always be used in order to get accurate location data for test failures.

Value

When run interactively, returns `invisible(TRUE)` if all tests pass, otherwise throws 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)
})

## Not run:
test_that("trigonometric functions match identities", {
  expect_equal(sin(pi / 4), 1)
})

## End(Not run)
```

try_again *Evaluate an expectation multiple times until it succeeds*

Description

If you have a flaky test, you can use `try_again()` to run it a few times until it succeeds. In most cases, you are better fixing the underlying cause of the flakeyness, but sometimes that's not possible.

Usage

```
try_again(times, code)
```

Arguments

`times`	Number of times to retry.
`code`	Code to evaluate.

Examples

```
usually_return_1 <- function(i) {
  if (runif(1) < 0.1) 0 else 1
}

## Not run:
# 10% chance of failure:
expect_equal(usually_return_1(), 1)

# 1% chance of failure:
try_again(1, expect_equal(usually_return_1(), 1))

# 0.1% chance of failure:
try_again(2, expect_equal(usually_return_1(), 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.

You will also need to:

- Add xml2 to Suggests, with e.g. usethis::use_package("xml2", "Suggests")
- Add testthat to LinkingTo, with e.g. usethis::use_package("testthat", "LinkingTo")

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 evaluates 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
```

Assuming you have `useDynLib(<pkg>, .registration = TRUE)` in your package's `NAMESPACE` file, this implies having routine registration code of the form:

```
// The definition for this function comes from the file 'src/test-runner.cpp',
// which is generated via `testthat::use_catch()`.
extern SEXP run_testthat_tests();

static const R_CallMethodDef callMethods[] = {
  // other .Call method definitions,
  {"run_testthat_tests", (DL_FUNC) &run_testthat_tests, 0},
  {NULL, NULL, 0}
};

void R_init_<pkg>(DllInfo* dllInfo) {
  R_registerRoutines(dllInfo, NULL, callMethods, NULL, NULL);
  R_useDynamicSymbols(dllInfo, FALSE);
}
```

replacing <pkg> above with the name of your package, as appropriate.

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.

Index