Package ‘testthat’

May 4, 2023

Title Unit Testing for R
Version 3.1.8
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


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

Depends R (>= 3.1)
Imports brio, callr (>= 3.5.1), cli (>= 3.4.0), desc, digest, ellipsis (>= 0.2.0), evaluate, jsonlite, lifecycle, magrittr, methods, pkgload, praise, processx, ps (>= 1.3.4), R6 (>= 2.2.0), rlang (>= 1.1.0), utils, waldo (>= 0.5.0), withr (>= 2.4.3)
Suggests covr, curl (>= 0.9.5), diffviewer (>= 0.1.0), knitr, markdown, rstudioapi, shiny, usethis, vctrs (>= 0.1.0), xml2

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate
Config/testthat/edition 3
Config/testthat/start-first watcher, parallel*
Encoding UTF-8

RoxygenNote 7.2.3

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 2023-05-04 06:30:02 UTC
R topics documented:

auto_test .......................................................... 3
auto_test_package .................................................. 4
CheckReporter ......................................................... 4
comparison-expectations ........................................... 5
DebugReporter ........................................................ 6
describe ............................................................... 6
equality-expectations ................................................ 7
expect ................................................................. 9
expect_error ......................................................... 10
expect_invisible .................................................... 13
expect_length ....................................................... 14
expect_named ......................................................... 15
expect_no_error .................................................... 16
expect_output ....................................................... 17
expect_setequal ..................................................... 18
expect_silent ......................................................... 19
expect_snapshot ..................................................... 20
expect_snapshot_file ............................................... 21
expect_snapshot_value ............................................. 23
expect_vector ....................................................... 25
fail ................................................................. 25
FailReporter ........................................................ 26
inheritance-expectations .......................................... 26
is_testing ........................................................... 28
JunitReporter ......................................................... 28
ListReporter ........................................................ 29
local_mocked_bindings ............................................. 29
local_test_context .................................................. 31
LocationReporter ................................................... 33
logical-expectations ............................................... 33
MinimalReporter .................................................... 34
MultiReporter ........................................................ 35
ProgressReporter ................................................... 35
RStudioReporter ..................................................... 36
SilentReporter ......................................................... 36
skip ................................................................. 36
snapshot_accept .................................................... 38
StopReporter ........................................................ 39
SummaryReporter .................................................... 39
TapReporter .......................................................... 40
TeammcityReporter .................................................. 40
tear down_env ....................................................... 40
test_file ............................................................ 41
test_package ........................................................ 42
test_path ............................................................. 43
test_that ............................................................. 44
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
eenv 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

```r
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 designed to ensure that you see something useful there.

### See Also

Comparison-expectations

Does code return a number greater/less than the expected value?

Description

Does code return a number greater/less than the expected 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, 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**  
*Test reporter: start recovery.*

**Description**

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

**See Also**


---

**describe**  
*describe: a BDD testing language*

**Description**

A simple BDD DSL for writing tests. The language is similar 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**

```r
describe(description, code)

it(description, code = NULL)
```

**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 defined 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 intended 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 you 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.
equality-expectations

Examples

describe("matrix()", {  
  it("can be multiplied by a scalar", {  
    m1 <- matrix(1:4, 2, 2)  
    m2 <- m1 * 2  
    expect_equal(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_equal(1 + 1, addition(1, 1))  
    })  
  })  
  describe("division()", {  
    it("can divide two numbers", {  
      expect_equal(10 / 2, division(10, 2))  
    })  
    it("can handle division by 0") #not yet implemented  
  })  
})

equality-expectations  Does code return the expected 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

```r
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**  
*The building block of all expect functions*

Description

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

Usage

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

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.

- `trace_env`  
  If `is.null(trace)`, this is used to automatically generate a traceback running from `test_code()`/`test_file()` to `trace_env`. You'll generally only need to set this if you're wrapping an expectation inside another function.
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

**Does code throw 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**

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

expect_warning(
```
expect_error

```r
object,
regexp = NULL,
class = NULL,
..., 
inherit = TRUE,
all = FALSE,
info = NULL,
label = NULL
)
```

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

expect_condition(
```r
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)
# expect_invisible

Does code return a visible or invisible object?

**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**

```r
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

```r
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

Does code return a vector with the specified length?

Description

Does code return a vector with the specified length?

Usage

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

Arguments

object Object to test.

n Expected length.

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`
**expect_named**

**Examples**

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

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

```r
## End(Not run)
```

---

**expect_named** *Does code return a vector with (given) 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**

```r
eexpect_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

```r
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**

Does code run without error, warning, message, or other condition?

---

**Description**

[Experimental]

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**

```r
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.
  - If many cases, particularly when testing warnings and message, 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

```r
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"))
```

### 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

```r
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.
expect_setequal

• 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.

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 Does code return a vector containing the expected 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 y.

Usage

expect_setequal(object, expected)

expect_mapequal(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)))

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

```r
expect_silent("123")

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

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

```r
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.

**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.

---

**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/{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**

```r
effect_snapshot_file(
  path,
  name = basename(path),
  binary = lifecycle::deprecated(),
  cran = FALSE,
  compare = NULL,
  transform = NULL,
  variant = NULL
)
```

```r
announce_snapshot_file(path, name = basename(path))
```

```r
compare_file_binary(old, new)
```

```r
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`.)
- `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}.{ext}. This allows you to create different snapshots for different scenarios, like different operating systems or different R versions.)
- `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. However we want to preserve
snapshot files when the R code wasn’t executed because of an unexpected error or because of a `skip()`. Let testthat know about these files by calling `announce_snapshot_file()` before `expect_snapshot_file()`.

**Examples**

```r
# 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) {
  # Other packages might affect results
  skip_if_not_installed("ggplot2", "2.0.0")
  # Or maybe the output is different on some operation systems
  skip_on_os("windows")
  # You'll need to carefully think about and experiment with these skips
  name <- paste0(name, ".png")

  # Announce the file before touching `code`. This way, if `code`
  # unexpectedly fails or skips, testthat will not auto-delete the
  # corresponding snapshot file.
  announce_snapshot_file(name = name)

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

**expect_snapshot_value**

Snapshot testing for values
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

```r
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.
expect_vector

Does code return a vector with the expected 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

```r
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

```r
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

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

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

Arguments

message a string to display.
info Character vector continuing additional information. Included for backward compatibility only and new expectations should not use it.
trace_env If is.null(trace), this is used to automatically generate a traceback running from test_code() / test_file() to trace_env. You’ll generally only need to set this if you’re wrapping an expectation inside another function.

Examples

```r
## 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


inheritance-expectations Does code return an object inheriting from the expected base type, S3 class, or S4 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.

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)

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**: Either a character vector of class names, or for `expect_s3_class()` and `expect_s4_class()`, an NA to assert that object isn’t an S3 or S4 object.
- **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_named(), expect_null(), expect_output(), expect_reference(), expect_silent(), logical-expectations

Examples

```r
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")
```
is_testing  

Determine testing status

Description

- is_testing() determine if code is being run as part of a test
- is_parallel() if the test is being run in parallel.
- testing_package() gives name of the package being tested.

These are thin wrappers that retrieve the values of environment variables. You will usually want to avoid a runtime dependency on testthat, i.e. you won’t want to use testthat in the code below R/. In that case, you should inline the source of these functions into your package, in whatever location you are using for any other test helper functions.

Usage

is_testing()

is_parallel()

testing_package()

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/

See Also

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


local_mocked_bindings  Mocking tools

Description

[Experimental]

with_mocked_bindings() and local_mockedBindings() 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).

These functions represent a second attempt at bringing mocking to testthat, incorporating what we’ve learned from the mockr, mockery, and mockthat packages.

Usage

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

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

Arguments

... Name-value pairs providing functions to mock.

.package The name of the package where mocked functions should be inserted. Generally, you should not need to 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()).

.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 \texttt{NAMESPACE}.
- The base environment.
- Called from an external package with \texttt{::}.

They are described in turn below.

\textbf{Internal \& imported functions:}

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

\begin{verbatim}
some_function <- function() {
  another_function()
}
\end{verbatim}

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

\begin{verbatim}
local_mocked_bindings(
  another_function = function(...) "new_value"
)
\end{verbatim}

\textbf{Base functions:}

Note that it's not possible to mock functions in the base namespace (i.e. functions that you can use without explicitly importing them) since currently we don't know of a way to to mock them without potentially affecting all running code. If you need to mock a base function, you'll need to create a wrapper, as described below.

\textbf{Namespaced calls:}

It's trickier to mock functions in other packages that you call with \texttt{::}. For example, take this minor variation:

\begin{verbatim}
some_function <- function() {
  anotherpackage::another_function()
}
\end{verbatim}

To mock here, you'll need to modify \texttt{another_function()} inside the \texttt{anotherpackage} package. You can do this by supplying the \texttt{.package} argument:

\begin{verbatim}
local_mocked_bindings(
  another_function = function(...) "new_value",
  .package = "anotherpackage"
)
\end{verbatim}

But it's not a great idea to mock a namespace that you don't own because it affects all code in that package, not just code 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"
)

---

**local_test_context**  
Locally set options for maximal test 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 default settings inside a test, if you want to test Unicode, coloured output, or a non-standard width.

**Usage**

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

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

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.env</code></td>
<td>Environment to use for scoping; expert use only.</td>
</tr>
<tr>
<td><code>width</code></td>
<td>Value of the &quot;width&quot; option.</td>
</tr>
<tr>
<td><code>crayon</code></td>
<td>Determines whether or not crayon (now cli) colour should be applied.</td>
</tr>
<tr>
<td><code>unicode</code></td>
<td>Value of the &quot;cli.unicode&quot; option. The test is skipped if <code>l10n_info()``:UTF-8</code> is FALSE.</td>
</tr>
<tr>
<td><code>rstudio</code></td>
<td>Should we pretend that we’re inside of RStudio?</td>
</tr>
</tbody>
</table>

---
Should we use ANSI hyperlinks.

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.
LocationReporter

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


logical-expectations  Does code return 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.

Usage

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

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>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.</td>
</tr>
<tr>
<td>info</td>
<td>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.</td>
</tr>
<tr>
<td>label</td>
<td>Used to customise failure messages. For expert use only.</td>
</tr>
</tbody>
</table>

**Details**

Attributes are ignored.

**See Also**

- is_false() for complement
- 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**

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

## End(Not run)
eexpect_true(!(2 != 2))
# or better:
eexpect_false(2 != 2)

ea <- 1:3
eexpect_true(length(a) == 3)
# but better to use more specific expectation, if available
eexpect_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.
**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**


---

**ProgressReporter**

*Test reporter: interactive progress bar of errors.*

**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**

RStudioReporter  
*Test reporter: RStudio*

**Description**

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

**See Also**


SilentReporter  
*Test reporter: gather all errors silently.*

**Description**

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

**See Also**


skip  
*Skip a test*

**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.
skip

Usage

skip(message = "Skipping")

skip_if_not(condition, message = NULL)

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, 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

host A string with a hostname to lookup

os Character vector of one or more operating systems to skip on. Supported values are "windows", "mac", "linux", and "solaris".

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.

• **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).

• **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

```r
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
})
```

---

**snapshot_accept**

**Snapshot management**

### Description

• **snapshot_accept()** accepts all modified snapshots.

• **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

```r
snapshot_accept(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.
StopReporter

Description

The default reporter used 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


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

**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**


**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**


**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-* file.

**Usage**

`teardown_env()`
test_file

Run all 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

```r
test_file(path, reporter = default_compact_reporter(), package = NULL, ...)
```

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.
- **package**: If these tests belong to a package, the name of the package.
- **...**: 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 that inherits from the package’s namespace environment, so that tests can access internal functions and objects.

Examples

```r
path <- testthat_example("success")
test_file(path)
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")

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)

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 that 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

```r
## 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/testhat/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)
**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**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dir</code></td>
<td>The directory containing an R package.</td>
</tr>
</tbody>
</table>

**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:

```cpp
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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Catch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>context</td>
<td>CATCH_TEST_CASE</td>
<td>The context of a set of tests.</td>
</tr>
<tr>
<td>test_that</td>
<td>CATCH_SECTION</td>
<td>A test section.</td>
</tr>
<tr>
<td>expect_true</td>
<td>CATCH_CHECK</td>
<td>Test that an expression evaluates to true.</td>
</tr>
<tr>
<td>expect_false</td>
<td>CATCH_CHECK_FALSE</td>
<td>Test that an expression evaluates to false.</td>
</tr>
<tr>
<td>expect_error</td>
<td>CATCH_CHECK_THROWS</td>
<td>Test that evaluation of an expression throws an exception.</td>
</tr>
<tr>
<td>expect_error_as</td>
<td>CATCH_CHECK_THROWS_AS</td>
<td>Test that evaluation of an expression throws an exception of a specific class</td>
</tr>
</tbody>
</table>

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:

```cpp
#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:

```r
.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

* **debugging**
  - auto_test, 3
  - auto_test_package, 4

* **expectations**
  - comparison-expectations, 5
  - equality-expectations, 7
  - expect_error, 10
  - expect_length, 14
  - expect_named, 15
  - expect_output, 17
  - expect_silent, 19
  - inheritance-expectations, 26
  - logical-expectations, 33

* **reporters**
  - CheckReporter, 4, 6, 26, 28, 29, 33, 35, 36, 39, 40
  - DebugReporter, 4, 6, 26, 28, 29, 33, 35, 36, 39, 40
  - FailReporter, 26
  - JunitReporter, 28
  - ListReporter, 29
  - LocationReporter, 33
  - MinimalReporter, 34
  - MultiReporter, 35
  - ProgressReporter, 35
  - RStudioReporter, 36
  - SilentReporter, 36
  - StopReporter, 39
  - SummaryReporter, 39
  - TapReporter, 40
  - TeamcityReporter, 40

all.equal(), 7, 8
announce_snapshot_file
  - (expect_snapshot_file), 21
auto_test, 3
auto_test(), 4, 39
auto_test_package, 4
auto_test_package(), 3

base::all.equal(), 24
base::signalCondition(), 10

base::stop(), 10
CheckReporter, 4, 6, 26, 28, 29, 33, 35, 36, 39, 40
CompactProgressReporter
  - (ProgressReporter), 35
compare(), 8
compare_file_binary
  - (expect_snapshot_file), 21
compare_file_text
  - (expect_snapshot_file), 21
comparison-expectations, 5
curl::nslookup(), 38
DebugReporter, 4, 6, 26, 28, 29, 33, 35, 36, 39, 40
deparse(), 24
describe, 6
equality-expectations, 7
exp_signal(), 10
expect, 9
expect_condition(expect_error), 10
expect_equal(equality-expectations), 7
expect_equal(), 44
expect_error, 5, 8, 10, 14, 15, 18, 19, 27, 34
expect_error(), 16, 44
expect_false(logical-expectations), 33
expect_gt(comparison-expectations), 5
expect_gte(comparison-expectations), 5
expect_identical
  - (equality-expectations), 7
expect_invisible, 13
expect_length, 5, 8, 12, 14, 15, 18, 19, 27, 34
expect_lt(comparison-expectations), 5
expect_lte(comparison-expectations), 5
expect_mapequal(expect_setequal), 18
expect_mapequal(), 8
expect_match, 5, 8, 12, 14, 15, 18, 19, 27, 34
expect_message(expect_error), 10
expect_named, 5, 8, 12, 14, 15, 18, 19, 27, 34
expect_no_condition (expect_no_error), 16
expect_no_error, 16
expect_no_error(), 11, 12
expect_no_message (expect_no_error), 16
expect_no_warning (expect_no_error), 16
expect_null, 5, 8, 12, 14, 15, 18, 19, 27, 34
expect_output, 5, 8, 12, 14, 15, 17, 19, 27, 34
expect_reference, 5, 8, 12, 14, 15, 18, 19, 27, 34
expect_reference(), 8
expect_s3_class
(inheritance-expectations), 26
expect_s4_class
(inheritance-expectations), 26
expect_setequal, 18
expect_setequal(), 8
expect_silent, 5, 8, 12, 14, 15, 18, 19, 27, 34
expect_snapshot, 20
expect_snapshot(), 10, 12, 17, 24
expect_snapshot_file, 21
expect_snapshot_value, 23
expect_snapshot_value(), 20
expect_that(), 39
expect_true (logical-expectations), 33
expect_type (inheritance-expectations), 26
expect_vector, 25
expect_vector(), 14, 26
expect_visible (expect_invisible), 13
expect_warning (expect_error), 10
fail, 25
FailReporter, 4, 6, 26, 28, 29, 33, 35, 36, 39, 40

identical(), 7, 8
inheritance-expectations, 26
inherits(), 26
is(), 26
is_false(), 34
is_parallel (is_testing), 28
is_testing, 28
is_testing(), 32
it (describe), 6

jsonlite::toJSON(), 24
jsonlite::unserializeJSON(), 24
JUnitReporter, 4, 6, 26, 28, 29, 33, 35, 36, 39, 40

library(), 42
ListReporter, 4, 6, 26, 28, 29, 33, 35, 36, 39, 40
local_mocked_bindings, 29
local_reproducible_output
(local_test_context), 31
local_test_context, 31
LocationReporter, 4, 6, 26, 28, 29, 33, 35, 36, 39, 40
logical-expectations, 33

MinimalReporter, 4, 6, 26, 28, 29, 33, 34, 35, 36, 39, 40
MultiReporter, 4, 6, 26, 28, 29, 33, 35, 36, 39, 40

new_expectation(), 10

ParallelProgressReporter
(ProgressReporter), 35
pkgload::load_all(), 29, 42
ProgressReporter, 4, 6, 26, 28, 29, 33, 35, 35, 36, 39, 40

quasi_label, 8, 11, 12, 14–19, 25, 27, 34

recover(), 6
Reporter, 4, 6, 26, 28, 29, 33, 35, 36–39, 42
rlang::is_interactive(), 32
rlang::trace_back(), 9
RStudioReporter, 4, 6, 26, 28, 29, 33, 35, 36, 36, 39, 40

serialize(), 24
SilentReporter, 4, 6, 26, 28, 29, 33, 35, 36, 39, 40

skip, 36
skip(), 23
skip if (skip), 36
skip if_not (skip), 36
skip if_not_installed (skip), 36
skip if_offline (skip), 36
skip if_translated (skip), 36
skip on bioc (skip), 36
skip on ci (skip), 36

jsonlite::serializeJSON(), 24
skip_on_covr (skip), 36
skip_on_cran (skip), 36
skip_on_os (skip), 36
snapshot_accept, 38
snapshot_accept(), 21
snapshot_review (snapshot_accept), 38
snapshot_review(), 21
sQuote(), 32
stop(), 39
StopReporter, 4, 6, 26, 28, 29, 33, 35, 36, 39, 39, 40
succeed (Fail), 25
SummaryReporter, 4, 6, 26, 28, 29, 33, 35, 36, 39, 39, 40
suppressMessages(), 10
suppressWarnings(), 10
TapReporter, 4, 6, 26, 28, 29, 33, 35, 36, 39, 40, 40
TeamcityReporter, 4, 6, 26, 28, 29, 33, 35, 36, 39, 39, 40, 40
teardown_env, 40
test_check (test_package), 42
test_dir(), 21, 35, 42
test_file, 41
test_file(), 21, 35
test_local (test_package), 42
test_package, 42
test_path, 43
test_that, 44
test_that(), 6, 36
testing_package (is_testing), 28
typeof(), 27

use_catch, 45

vctrs::vec_assert(), 25
waldo::compare(), 8, 24
watch(), 3, 4
with_mocked_bindings
  (local_mocked_bindings), 29
withr::defer(), 40
withr::deferred_run(), 31