Package ‘testthat’

October 13, 2018

Title  Unit Testing for R
Version  2.0.1
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 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  cli, crayon, digest, magrittr, methods, praise, R6 (>= 2.2.0), rlang, withr (>= 2.0.0)
Suggests  covr, devtools, knitr, rmarkdown, usethis, xml2
VignetteBuilder  knitr
Encoding  UTF-8
RoxygenNote  6.0.1
Collate  'auto-test.R' 'capture-condition.R' 'capture-output.R'
'colour-text.R' 'compare.R' 'compare-character.R'
'compare-numeric.R' 'compare-time.R' 'context.R' 'describe.R'
'evaluate-promise.R' 'expect-comparison.R' 'expect-equality.R'
'expect-inheritance.R' 'expect-known.R' 'expect-length.R'
'expect-logical.R' 'expect-named.R' 'expect-output.R'
'reporter.R' 'expect-self-test.R' 'expect-that.R'
'expectation.R' 'expectations-matches.R' 'make-expectation.R'
'mock.R' 'old-school.R' 'praise.R' 'recover.R'
'reporter-check.R' 'reporter-debug.R' 'reporter-fail.R'
'reporter-junit.R' 'reporter-list.R' 'reporter-location.R'
'reporter-minimal.R' 'reporter-multi.R' 'stack.R'
'reporter-progress.R' 'reporter-rstudio.R' 'reporter-silent.R'
'reporter-stop.R' 'reporter-summary.R' 'reporter-tap.R'
'reporter-teamcity.R' 'reporter-zzz.R' 'skip.R' 'source.R'
teardown.R' 'test-compiled-code.R' 'test-directory.R'
test-example.R' 'test-files.R' 'test-path.R' 'test-that.R'
'traceback.R' 'try-again.R' 'utilzio.R' 'utils.R' 'watcher.R'


**NeedsCompilation**: yes

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

**Maintainer**: Hadley Wickham <hadley@rstudio.com>

**Repository**: CRAN

**Date/Publication**: 2018-10-13 08:40:03 UTC

**R topics documented:**

current

---

auto_test ................................. 3
auto_test_package .................. 4
CheckReporter ....................... 4
comparison-expectations .......... 5
context .............................. 6
DebugReporter ...................... 6
describe ................................ 7
equality-expectations .......... 8
expect .................................. 10
expect_cpp_tests_pass ............ 10
expect_known_output .............. 11
expect_length ..................... 12
expect_match ...................... 13
expect_named ...................... 14
fail .................................. 15
FailReporter ..................... 15
inheritance-expectations ...... 16
JunitReporter ...................... 17
ListReporter ....................... 18
LocationReporter ................ 18
logical-expectations .......... 19
MinimalReporter ................ 20
MultiReporter .................... 20
output-expectations .......... 21
ProgressReporter ................ 23
RstudioReporter .................. 24
SilentReporter .................... 24
skip ................................ 25
StopReporter ...................... 26
SummaryReporter ................ 27
TapReporter .......................... 27
TeamcityReporter ................ 28
tear down ......................... 28
test_dir .............................. 29
test_examples .................. 31
test_file .............................. 31
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

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

Arguments

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

Details

The current strategy for rerunning tests is as follows:

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

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

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

See Also

`auto_test_package()`
### auto_test_package

_Watches a package for changes, rerunning tests as appropriate._

**Description**

Watches a package for changes, rerunning tests as appropriate.

**Usage**

```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 design to ensure that you see something useful there.

**Usage**

- `CheckReporter`

**Format**

An object of class `R6ClassGenerator` of length 24.

**See Also**

**comparison-expectations**

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

---

**Description**

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

**Usage**

```r
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` object to test
- `expected` Single numeric value to compare.
- `label` object label. When NULL, computed from deparsed object.
- `expected.label` Equivalent of `label` for shortcut form.
- `...` other values passed to `all.equal()`

**See Also**

Other expectations: `equality-expectations`, `expect_length`, `expect_match`, `expect_named`, `inheritance-expectations`, `logical-expectations`, `output-expectations`

**Examples**

```r
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)
```
### context

*Describe the context of a set of tests.*

#### Description

A context defines a set of tests that test related functionality. Usually you will have one context per file, but you may have multiple contexts in a single file if you so choose.

#### Usage

```
context(desc)
```

#### Arguments


#### Examples

```
context("String processing")
context("Remote procedure calls")
```

### DebugReporter

*Test reporter: start recovery.*

#### Description

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

#### Usage

```
DebugReporter
```

#### Format

An object of class `R6ClassGenerator` of length 24.

#### 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**

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

**Arguments**

- **description**: description of the feature
- **code**: test code containing the specs

**Details**

Tests using the `describe` syntax not only verify the tested code, but also document its intended behaviour. Each `describe` block specifies a larger component or function and contains a set of specifications. A specification is 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.

**Examples**

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

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

## specs
describe("math library", {
```
describe("addition()", {  
  it("can add two numbers", {  
    expect_equivalent(1 + 1, addition(1, 1))  
  })  
})

describe("division()", {  
  it("can divide two numbers", {  
    expect_equivalent(10 / 2, division(10, 2))  
  })  
  it("can handle division by 0") #not yet implemented  
})

---

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

Description

- `expect_identical` tests with `identical()`
- `expect_equal` tests with `all.equal()`
- `expect_setequal` ignores order and duplicates
- `expect_equivalent` tests with `all.equal()` and `check.attributes = FALSE`
- `expect_reference` tests if two symbols point to the same underlying object in memory (requires rlang 1.2.9000 or greater)

Usage

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

```r
expect_setequal(object, expected)
```

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

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

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

```r
expect_reference(object, expected, info = NULL, label = NULL, expected.label = NULL)
```
equality-expectations

Arguments

- **object**: object to test
- **expected**: Expected value
- **...**: other values passed to `all.equal()`
- **info**: extra information to be included in the message (useful when writing tests in loops).
- **label**: object label. When NULL, computed from deparsed object.
- **expected.label**: Equivalent of `label` for shortcut form.

See Also

Other expectations: `comparison-expectations`, `expect_length`, `expect_match`, `expect_named`, `inheritance-expectations`, `logical-expectations`, `output-expectations`

Examples

```r
a <- 10
equal(a, 10)

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

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

# Test the RELATIVE difference is within .002
x <- 10
equal(10.01, expected = x, tolerance = 0.002, scale = x)
# `expect_equivalent` ignores attributes
a <- b <- 1:3
names(b) <- letters[1:3]
equal_equivalent(a, b)
```
expect

The building block of all expect_functions

Description
Use this if you are writing your own expectation. See vignette("custom-expectation") for details.

Usage
expect(ok, failure_message, info = NULL, srcref = NULL)

Arguments
- ok: Was the expectation successful?
- failure_message: What message should be shown if the expectation was not successful?
- info: Additional information. Included for backward compatibility only and new expectations should not use it.
- srcref: Only needed in very rare circumstances where you need to forward a srcref captured elsewhere.

expect_cpp_tests_pass
Test Compiled Code in a Package

Description
Test compiled code in the package package. See use_catch() for more details.

Usage
expect_cpp_tests_pass(package)

Arguments
- package: The name of the package to test.

Note
A call to this function will automatically be generated for you in tests/testthat/test-cpp.R after calling use_catch(); you should not need to manually call this expectation yourself.
**Description**

For complex printed output and objects, it is often challenging to describe exactly what you expect to see. `expect_known_value()` and `expect_known_output()` provide a slightly weaker guarantee, simply asserting that the values have no changed since the last time that you ran them.

**Usage**

```r
expect_known_output(object, file, update = TRUE, ..., info = NULL, label = NULL, print = FALSE, width = 80)
```

```r
expect_known_value(object, file, update = TRUE, ..., info = NULL, label = NULL)
```

```r
expect_known_hash(object, hash = NULL)
```

**Arguments**

- `object`  
  object to test

- `file`  
  File path where known value/output will be stored.

- `update`  
  Should the file be updated? Defaults to `TRUE`, with the expectation that you’ll notice changes because of the first failure, and then see the modified files in git.

- `...`  
  other values passed to `all.equal()`

- `info`  
  extra information to be included in the message (useful when writing tests in loops).

- `label`  
  object label. When `NULL`, computed from deparsed object.

- `print`  
  If `TRUE` and the result of evaluating code is visible this will print the result, ensuring that the output of printing the object is included in the overall output.

- `width`  
  Number of characters per line of output

- `hash`  
  Known hash value. Leave empty and you’ll be informed what to use in the test output.

**Details**

These expectations should be used in conjunction with git, as otherwise there is no way to revert to previous values. Git is particularly useful in conjunction with `expect_known_output()` as the diffs will show you exactly what has changed.

Note that known values updates will only updated when running tests interactively. `R CMD check` clones the package source so any changes to the reference files will occurred a temporary directory, and will not be synchronised back to the source package.
### Examples

```r
tmp <- tempfile()

# The first run always succeeds
expect_known_output(mtcars[1:10, ], tmp, print = TRUE)

# Subsequent runs will succeed only if the file is unchanged
# This will succeed:
expect_known_output(mtcars[1:10, ], tmp, print = TRUE)

## Not run:
# This will fail
expect_known_output(mtcars[1:9, ], tmp, print = TRUE)

## End(Not run)
```

---

### expect_length

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

### Description

Expectation: does a vector have the specified length?

### Usage

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

### Arguments

- **object**: object to test
- **n**: Expected length.

### See Also

Other expectations: `comparison-expectations`, `equality-expectations`, `expect_match`, `expect_named`, `inheritance-expectations`, `logical-expectations`, `output-expectations`

### Examples

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

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

## End(Not run)
```
expect_match

**Description**

Expectation: does string match a regular expression?

**Usage**

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

**Arguments**

- **object**: object to test
- **regexp**: Regular expression to test against.
- **perl**: logical. Should Perl-compatible regexps be used?
- **fixed**: logical. If TRUE, pattern is a string to be matched as is. Overrides all conflicting arguments.
- **...**: Arguments passed on to base::grepl
- **ignore.case**: if FALSE, the pattern matching is case sensitive and if TRUE, case is ignored during matching.
- **useBytes**: logical. If TRUE the matching is done byte-by-byte rather than character-by-character. See ‘Details’.
- **all**: Should all elements of actual value match regexp (TRUE), or does only one need to match (FALSE)
- **info**: extra information to be included in the message (useful when writing tests in loops).
- **label**: object label. When NULL, computed from deparsed object.

**See Also**

Other expectations: `comparison-expectations`, `equality-expectations`, `expect_length`, `expect_named`, `inheritance-expectations`, `logical-expectations`, `output-expectations`

**Examples**

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

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

# Zero-length inputs always fail
expect_match(character(0), ".")

# End(Not run)"
expect_named

Expectation: does object have names?

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

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

Arguments
- **object**: object to test
- **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 (useful when writing tests in loops).
- **label**: object label. When NULL, computed from deparsed object.
- ... Other arguments passed on to has_names().

See Also
Other expectations: comparison-expectations, equality-expectations, expect_length, expect_match, inheritance-expectations, logical-expectations, output-expectations

Examples
```r
x <- c(a = 1, b = 2, c = 3)
extpect_named(x)
extpect_named(x, c("a", "b", "c"))

# Use options to control sensitivity
extpect_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)
```
**fail**

*Default expectations that alway succeed or fail.*

---

**Description**

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

**Usage**

```r
fail(message = "Failure has been forced")
```

```r
succeed(message = "Success has been forced")
```

**Arguments**

- `message`: a string to display.

**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`.

**Usage**

```r
FailReporter
```

**Format**

An object of class `R6ClassGenerator` of length 24.

**See Also**

**Expectation:** does the object inherit from a S3 or S4 class, or a base type?

**Description**

Tests whether or not an object inherits from any of a list of classes, or is an instance of a base type. `expect_null()` is a special case designed for detecting NULL.

**Usage**

```r
expect_null(object, info = NULL, label = NULL)
expect_type(object, type)
expect_is(object, class, info = NULL, label = NULL)
expect_s3_class(object, class)
expect_s4_class(object, class)
```

**Arguments**

- `object` object to test
- `info` extra information to be included in the message (useful when writing tests in loops).
- `label` object label. When NULL, computed from deparsed object.
- `type` String giving base type (as returned by `typeof()`).
- `class` character vector of class names

**Details**

`expect_is()` is an older form. I’d recommend using `expect_s3_class()` or `expect_s4_class()` in order to more clearly convey intent.

**See Also**

- `inherits()`

Other expectations: `comparison-expectations, equality-expectations, expect_length, expect_match, expect_named, logical-expectations, output-expectations`
Examples

```r
expect_is(1, "numeric")
a <- matrix(1:10, nrow = 5)
expect_is(a, "matrix")

expect_is(mtcars, "data.frame")
# alternatively for classes that have an is method
expect_true(is.data.frame(mtcars))

f <- factor("a")
expect_is(f, "factor")
expect_s3_class(f, "factor")
expect_type(f, "integer")

expect_null(NULL)
```

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.

Usage

`JUnitReporter`

Format

An object of class `R6ClassGenerator` of length 24.

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/](http://llg.cubic.org/docs/junit/)
ListReporter

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

Description

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

Usage

ListReporter

Format

An object of class R6ClassGenerator of length 24.

See Also


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.

Usage

LocationReporter

Format

An object of class R6ClassGenerator of length 24.

See Also

Description

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

Usage

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

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

Arguments

- object: object to test
- info: extra information to be included in the message (useful when writing tests in loops).
- label: object label. When NULL, computed from deparsed object.

Details

Attributes are ignored.

See Also

is_false() for complement

Other expectations: comparison-expectations, equality-expectations, expect_length, expect_match, expect_named, inheritance-expectations, output-expectations

Examples

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

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

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

Test reporter: minimal.

Description

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

Usage

MinimalReporter

Format

An object of class R6ClassGenerator of length 24.

See Also


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.

Usage

MultiReporter

Format

An object of class R6ClassGenerator of length 24.

See Also

Description

Use `expect_output()`, `expect_message()` and `expect_warning()` to match specified outputs. Use `expect_error()` or `expect_condition()` to match individual errors or conditions. Use `expect_silent()` to assert that there should be no output of any type.

Usage

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

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

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

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

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

```r
expect_silent(object)
```

Arguments

- **object**: object to test.
- **regexp**: regular expression to test against. If `NULL`, the default, asserts that there should be an output, a message, a warning, or an error, but does not test for specific value. If `NA`, asserts that there should be no output, messages, warnings, or errors.
- **all**: Should all elements of actual value match `regexp` (TRUE), or does only one need to match (FALSE)?
- **perl**: logical. Should Perl-compatible regexps be used?
- **fixed**: logical. If TRUE, pattern is a string to be matched as is. Overrides all conflicting arguments.
- **info**: extra information to be included in the message (useful when writing tests in loops).
- **label**: object label. When `NULL`, computed from deparsed object.
- **class**: Instead of supplying a regular expression, you can also supply a class name. This is useful for "classed" conditions.
For messages and warnings, do all need to match the regexp (TRUE), or does only one need to match (FALSE)

Details

Note that warnings are captured by a custom signal handler: this means that options(warn) has no effect.

Value

The first argument, invisibly. If expect_error() captures an error, that is returned instead of the value.

See Also

Other expectations: comparison-expectations, equality-expectations, expect_length, expect_match, expect_named, inheritance-expectations, logical-expectations

Examples

```r
# Output -----------------------------------------------
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)

# Messages -----------------------------------------------

f <- function(x) {
  if (x < 0) message("*x* is already negative")
  -x
}
expect_message(f(-1))
expect_message(f(-1), "already negative")
expect_message(f(1), NA)

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

# Warnings -----------------------------------------------

f <- function(x) {
  if (x < 0) warning("*x* is already negative")
  -x
}
expect_warning(f(-1))
expect_warning(f(-1), "already negative")
expect_warning(f(1), NA)
```
# You can use the arguments of grepl to control the matching
expect_warning(f(-1), "*xx*, fixed = TRUE)
expect_warning(f(-1), "NEGATIVE", ignore.case = TRUE)

# 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)

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

## End(Not run)

---

**ProgressReporter**

*Test reporter: interactive progress bar of errors.*

**Description**

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

**Usage**

`ProgressReporter`

**Format**

An object of class `R6ClassGenerator` of length 24.

**Details**

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

Test reporter: RStudio

Description

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

Usage

Rstudioreporter

Format

An object of class R6ClassGenerator of length 24.

See Also


SilentReporter

Test reporter: gather all errors silently.

Description

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

Usage

SilentReporter

Format

An object of class R6ClassGenerator of length 24.

See Also

**Description**

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

**Usage**

```r
skip(message)

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

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

skip_if_not_installed(pkg, minimum_version = NULL)

skip_on_cran()

skip_on_os(os)

skip_on_travis()

skip_on_appveyor()

skip_on_bioc()

skip_if_translated()
```

**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
- `os` Character vector of system names. Supported values are "windows", "mac", "linux" and "solaris".

**Details**

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

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

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

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

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

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

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

Examples

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

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

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

StopReporter

Test reporter: stop on error.

Description

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

Usage

StopReporter

Format

An object of class R6ClassGenerator of length 24.

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


<table>
<thead>
<tr>
<th>SummaryReporter</th>
<th>Test reporter: summary of errors.</th>
</tr>
</thead>
</table>

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.

Usage

SummaryReporter

Format

An object of class R6ClassGenerator of length 24.

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


<table>
<thead>
<tr>
<th>TapReporter</th>
<th>Test reporter: TAP format.</th>
</tr>
</thead>
</table>

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

Usage

TapReporter
teardown

Format
An object of class R6ClassGenerator of length 24.

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

Usage
TeamcityReporter

Format
An object of class R6ClassGenerator of length 24.

See Also

teardown Run code on setup/teardown

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

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

setup(code, env = parent.frame())
**Description**

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

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

**Usage**

```r

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

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

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

is_testing()
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>path</code></td>
<td>path to tests</td>
</tr>
<tr>
<td><code>filter</code></td>
<td>If not NULL, only tests with file names matching this regular expression will be executed. Matching will take on the file name after it has been stripped of &quot;test=&quot; and &quot;.R&quot;.</td>
</tr>
</tbody>
</table>
reporter

reporter to use

env

environment in which to execute the tests

... Additional arguments passed to `grep1()` to control filtering.

encoding

File encoding, default is "unknown" unknown.

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.

wrap

Automatically wrap all code within `test_that()`? This ensures that all expectations are reported, even if outside a test block.

package

package name

Value

The results of the reporter function on all test results.
The results as a "testthat_results" (list)

Test files

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

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

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

Environments

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

R CMD check

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

```r
library(testthat)
library(yourpackage)
test_check("yourpackage")
```

Examples

```r
## Not run: test_package("testthat")
```
**test_examples**  
*Test package examples*

**Description**

These helper functions make it easier to test the examples in a package. Each example counts as one test, and it succeeds if the code runs without an error.

**Usage**

```r
test_examples(path = "../..")
test_example(path)
test_rd(rd)
```

**Arguments**

- **path**
  For `test_examples()`, path to directory containing Rd files. For `test_example()`, path to a single Rd file. Remember the working directory for tests is `tests/testthat`.
- **rd**
  A parsed Rd object, obtained from `tools::Rd_db()` or otherwise.

**test_file**  
*Run all tests in specified file.*

**Description**

Run all tests in specified file.

**Usage**

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

**Arguments**

- **path**
  path to file
- **reporter**
  reporter to use
- **env**
  environment in which to execute the tests
- **start_end_reporter**
  whether to start and end the reporter
- **load_helpers**
  Source helper files before running the tests?
- **encoding**
  File encoding, default is "unknown"
- **wrap**
  Automatically wrap all code within `test_that()`? This ensures that all expectations are reported, even if outside a test block.
Value
the results as a "testthat_results" (list)

---

test_path

Locate file in testing directory.

---

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

Usage
test_path(...)

Arguments
... Character vectors giving path component.

Value
A character vector giving the path.

---

test_that

Create a test.

---

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

Usage
test_that(desc, code)

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

Details
Tests are evaluated in their own environments, and should not affect global state.
When run from the command line, tests return NULL if all expectations are met, otherwise it raises an error.
Examples

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

## End(Not run)
```

use_catch  

Use Catch for C++ Unit Testing

Description

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

Usage

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

Arguments

dir  
The directory containing an R package.

Details

Calling use_catch() will:

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

C++ unit tests can be added to C++ source files within the src directory of your package, with a format similar to R code tested with testthat. Here’s a simple example of a unit test written with testthat + Catch:
context("C++ Unit Test") {
  test_that("two plus two is four") {
    int result = 2 + 2;
    expect_true(result == 4);
  }
}

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

Functions

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

<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:
```c
#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

*Topic datasets
  CheckReporter, 4
  DebugReporter, 6
  FailReporter, 15
  JunitReporter, 17
  ListReporter, 18
  LocationReporter, 18
  MinimalReporter, 20
  MultiReporter, 20
  ProgressReporter, 23
  RstudioReporter, 24
  SilentReporter, 24
  StopReporter, 26
  SummaryReporter, 27
  TapReporter, 27
  TeamcityReporter, 28

*Topic debugging
  auto_test, 3
  auto_test_package, 4

all.equal(), 5, 8, 9, 11
auto_test, 3
auto_test(), 4, 27
auto_test_package, 4
auto_test_package(), 3

CheckReporter, 4, 6, 15, 18, 20, 24, 27, 28
comparison-expectations, 5
context, 6, 32

DebugReporter, 4, 6, 15, 18, 20, 24, 27, 28
describe, 7
equality-expectations, 8
expect, 10
expect_condition (output-expectations), 21
expect_cpp_tests_pass, 10
expect_equal (equality-expectations), 8
expect_equal_to_reference
  (expect_known_output), 11
expect_equivalent
  (equality-expectations), 8
expect_error (output-expectations), 21
expect_false (logical-expectations), 19
expect_gt (comparison-expectations), 5
expect_gte (comparison-expectations), 5
expect_identical
  (equality-expectations), 8
expect_is (inheritance-expectations), 16
expect_known_hash
  (expect_known_output), 11
expect_known_output, 11
expect_known_value
  (expect_known_output), 11
expect_length, 5, 9, 12, 13, 14, 16, 19, 22
expect_less_than
  (comparison-expectations), 5
expect_lt (comparison-expectations), 5
expect_lte (comparison-expectations), 5
expect_match, 5, 9, 12, 13, 14, 16, 19, 22
expect_message (output-expectations), 21
expect_more_than
  (comparison-expectations), 5
expect_named, 5, 9, 12, 13, 14, 16, 19, 22
expect_null (inheritance-expectations), 16
expect_output (output-expectations), 21
expect_output_file
  (expect_known_output), 11
expect_reference
  (expect_known_output), 11
expect_sS_class
  (inheritance-expectations), 16
expect_sT_class
  (inheritance-expectations), 16
expect_setequal
  (equality-expectations), 8
expect_s3_class
  (inheritance-expectations), 16
expect_s4_class
  (inheritance-expectations), 16
expect_setequal
  (equality-expectations), 8
expect_silent (output-expectations), 21
expect_true (logical-expectations), 19
expect_type (inheritance-expectations), 16
expect_warning (output-expectations), 21
fail, 15
FailReporter, 4, 6, 15, 18, 20, 24, 27, 28
grepl(), 30
has_names(), 14
identical(), 8
inheritance-expectations, 16
inherits(), 16
is_false(), 19
is_testing (test_dir), 29
JUnitReporter, 17
ListReporter, 4, 6, 15, 18, 18, 20, 24, 27, 28
LocationReporter, 4, 6, 15, 18, 18, 20, 24,
27, 28
logical-expectations, 19
MinimalReporter, 4, 6, 15, 18, 20, 20, 24, 27,
28
MultiReporter, 4, 6, 15, 18, 20, 20, 24, 27,
28
output-expectations, 21
ProgressReporter, 4, 6, 15, 18, 20, 23, 24,
27, 28
recover(), 6
RstudioReporter, 4, 6, 15, 18, 20, 24, 24, 27,
28
setup (teardown), 28
SilentReporter, 4, 6, 15, 18, 20, 24, 24, 27,
28
skip, 25
skip_if (skip), 25
skip_if_not (skip), 25
skip_if_not_installed (skip), 25
skip_if_translated (skip), 25
skip_on_appveyor (skip), 25
skip_on_bioc (skip), 25
skip_on_cran (skip), 25
skip_on_os (skip), 25
skip_on_travis (skip), 25
stop(), 26
stopifnot(), 26
StopReporter, 4, 6, 15, 18, 20, 24, 26, 27, 28
succeed (fail), 15
SummaryReporter, 4, 6, 15, 18, 20, 23, 24, 27,
27, 28
TapReporter, 4, 6, 15, 18, 20, 24, 27, 27, 28
TeamcityReporter, 4, 6, 15, 18, 20, 24, 27,
28, 28
tardown, 28
test_check (test_dir), 29
test_dir, 29
test_dir(), 23
test_example (test_examples), 31
test_examples, 31
test_file, 31
test_file(), 23
test_package (test_dir), 29
test_path, 32
test_rd (test_examples), 31
test_that, 32
test_that(), 7, 25, 30, 31
tools: :Rd_db(), 31
typeof(), 16
use_catch, 33
use_catch(), 10
watch(), 3, 4