# Package 'testthat'

August 29, 2016

**Version** 1.0.2

**Title** Unit Testing for R

**Description** A unit testing system designed to be fun, flexible and easy to set up.

**URL** https://github.com/hadley/testthat

**BugReports** https://github.com/hadley/testthat/issues

**Depends** R (>= 3.1.0)

**Imports** digest, crayon, praise, magrittr, R6, methods

**Suggests** devtools, withr, covr

**License** MIT + file LICENSE

**Collate** 'auto-test.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-equal-to-reference.R' 'expect-equality.R' 'expect-inheritance.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' 'reporter-check.R' 'reporter-fail.R' 'reporter-list.R' 'reporter-minimal.R' 'reporter-multi.R' 'reporter-rstudio.R' 'reporter-silent.R' 'reporter-stop.R' 'stack.R' 'reporter-summary.R' 'reporter-tap.R' 'reporter-teamcity.R' 'reporter-zzz.R' 'skip.R' 'source.R' 'test-compiled-code.R' 'test-example.R' 'test-files.R' 'test-package.R' 'test-path.R' 'test-that.R' 'traceback.R' 'try-again.R' 'utils.R' 'watcher.R'

**RoxygenNote** 5.0.1

**NeedsCompilation** yes

**Author** Hadley Wickham [aut, cre], RStudio [cph]

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

**Repository** CRAN

**Date/Publication** 2016-04-23 08:37:40

# R topics documented:

---

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

---

## Description

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

## Usage

```
auto_test(code_path, test_path, reporter = "summary", env = test_env())
```

## 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. |

## Details

The current strategy for rerunning tests is as follows:

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

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

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

## See Also

`auto_test_package`

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

### Description

Watches a package for changes, rerunning tests as appropriate.

### Usage

```
auto_test_package(pkg = ".", reporter = "summary")
```

### Arguments

| | |
|---|---|
| pkg | path to package |
| reporter | test reporter to use |

### 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.

compare *Provide human-readable comparison of two objects*

## Description

compare is similar to all.equal(), but shows you examples of where the failures occured.

## Usage

```
compare(x, y, ...)

## Default S3 method:
compare(x, y, ...)

## S3 method for class 'character'
compare(x, y, check.attributes = TRUE, ...,
  max_diffs = 5, max_lines = 5, width = getOption("width"))

## S3 method for class 'numeric'
compare(x, y, tolerance = .Machine$double.eps^0.5, ...,
  max_diffs = 9)

## S3 method for class 'POSIXt'
compare(x, y, tolerance = 0.001, ..., max_diffs = 9)
```

## Arguments

| | |
|---|---|
| x, y | Objects to compare |
| ... | Additional arguments used to control specifics of comparison |
| check.attributes | If TRUE, also checks values of attributes. |
| max_diffs | Maximum number of differences to show |
| max_lines | Maximum number of lines to show from each difference |
| width | Width of output device |
| tolerance | Numerical tolerance: any differences smaller than this value will be ignored. |

## Examples

```
# Character ------------------------------------------------------------------
x <- c("abc", "def", "jih")
compare(x, x)

y <- paste0(x, "y")
compare(x, y)

compare(letters, paste0(letters, "-"))
```

```
x <- "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis cursus
 tincidunt auctor. Vestibulum ac metus bibendum, facilisis nisi non, pulvinar
 dolor. Donec pretium iaculis nulla, ut interdum sapien ultricies a. "
y <- "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis cursus
 tincidunt auctor. Vestibulum ac metus1 bibendum, facilisis nisi non, pulvinar
 dolor. Donec pretium iaculis nulla, ut interdum sapien ultricies a. "
compare(x, y)
compare(c(x, x), c(y, y))

# Numeric ------------------------------------------------------------------

x <- y <- runif(100)
y[sample(100, 10)] <- 5
compare(x, y)

x <- y <- 1:10
x[5] <- NA
x[6] <- 6.5
compare(x, y)

# Compare ignores minor numeric differences in the same way
# as all.equal.
compare(x, x + 1e-9)
```

---

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

---

## Description

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

## Usage

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

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

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

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

## Arguments

| | |
|---|---|
| `object` | 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_equal_to_reference, expect_length, expect_match, expect_named, inheritance-expectations, logical-expectations, output-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)
```

---

# 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

| | |
|---|---|
| desc | description of context. Should start with a capital letter. |

## Examples

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

describe *describe: a BDD testing language*

## Description

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

## Usage

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

## Arguments

| | |
|---|---|
| description | description of the feature |
| code | test code containing the specs |

## Details

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

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

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

## Examples

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

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

## specs
describe("math library", {
```

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

---

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

---

## Description

- expect_identical tests with identical
- expect_equal tests with all.equal
- expect_equivalent tests with all.equal and check.attributes = FALSE

## Usage

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

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

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

## 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_equal_to_reference, expect_length, expect_match, expect_named, inheritance-expectations, logical-expectations, output-expectations

## Examples

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

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

## End(Not run)

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

## End(Not run)

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

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

---

`evaluate_promise` *Evaluate a promise, capturing all types of output.*

---

## Description

These functions allow you to capture the side-effects of a function call including printed output, messages and warnings. They are used to evaluate code for `expect_output`, `expect_message`, `expect_warning`, and `expect_silent`.

## Usage

```
evaluate_promise(code, print = FALSE)

capture_messages(code)

capture_warnings(code)

capture_output(code, print = FALSE)
```

### Arguments

| | |
|---|---|
| code | Code to evaluate. This should be an unevaluated expression. |
| 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 |

### Value

A list containing

| | |
|---|---|
| result | The result of the function |
| output | A string containing all the output from the function |
| warnings | A character vector containing the text from each warning |
| messages | A character vector containing the text from each message |

### Examples

```
evaluate_promise({
  print("1")
  message("2")
  warning("3")
  4
})
```

---

## 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.

`expect_equal_to_reference` *Expectation: is the object equal to a reference value stored in a file?*

## Description

This expectation is equivalent to `expect_equal`, except that the expected value is stored in an RDS file instead of being specified literally. This can be helpful when the value is necessarily complex. If the file does not exist then it will be created using the value of the specified object, and subsequent tests will check for consistency against that generated value. The test can be reset by deleting the RDS file.

## Usage

```
expect_equal_to_reference(object, file, ..., info = NULL, label = NULL,
  expected.label = NULL)
```

## Arguments

| | |
|---|---|
| `object` | object to test |
| `file` | The file name used to store the object. Should have an "rds" extension. |
| `...` | other values passed to `expect_equal` |
| `info` | extra information to be included in the message (useful when writing tests in loops). |
| `label` | For the full form, a label for the expected object, which is used in error messages. Useful to override the default (which is based on the file name), when doing tests in a loop. For the short-cut form, the object label, which is computed from the deparsed object by default. |
| `expected.label` | Equivalent of `label` for shortcut form. |

## Details

It is important to initialize the reference RDS file within the source package, most likely in the `tests/testthat/` directory. Testing spawned by `devtools::test()`, for example, will accomplish this. But note that testing spawned by `R CMD check` and `devtools::check()` will NOT. In the latter cases, the package source is copied to an external location before tests are run. The resulting RDS file will not make its way back into the package source and will not be available for subsequent comparisons.

## See Also

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

## Examples

```
## Not run:
expect_equal_to_reference(1, "one.rds")

## End(Not run)
```

---

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

---

## Description

Expectation: does a vector have the specified length?

## Usage

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

## Arguments

| | |
|---|---|
| `object` | object to test |
| `n` | Expected length. |

## See Also

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

## Examples

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

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

## End(Not run)
```

`expect_match` *Expectation: does string/output/message/warning/error match a regular expression?*

## Description

Expectation: does string/output/message/warning/error match a regular expression?

## Usage

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

## Arguments

| | |
|---|---|
| `object` | object to test |
| `regexp` | Regular expression to test against. |
| `...` | Additional arguments passed on to `grepl`, e.g. `ignore.case` or `fixed`. |
| `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_equal_to_reference`, `expect_length`, `expect_named`, `inheritance-expectations`, `logical-expectations`, `output-expectations`

## Examples

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

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

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

## End(Not run)
```

expect_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 onto `has_names`. |

## See Also

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

## Examples

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

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

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

## 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

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

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

### Arguments

| | |
|---|---|
| message | a string to display. |

### Examples

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

## End(Not run)
```

## FailReporter *Test reporter: fail at end.*

### Description

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

### Usage

```
FailReporter
```

### Format

An object of class R6ClassGenerator of length 24.

inheritance-expectations

*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

```
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 convery intent.

## See Also

`inherits`

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

## Examples

```
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)
```

---

`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.

---

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

---

## Description

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

## Usage

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

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

### Arguments

| | |
|---|---|
| object | object to test |
| 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_equal_to_reference, 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)
```

---

## make_expectation *Make an equality test.*

---

### Description

This a convenience function to make a expectation that checks that input stays the same.

### Usage

```
make_expectation(x, expectation = "equals")
```

### Arguments

| | |
|---|---|
| x | a vector of values |
| expectation | the type of equality you want to test for (equals, is_equivalent_to, is_identical_to) |

## Examples

```
x <- 1:10
make_expectation(x)

make_expectation(mtcars$mpg)

df <- data.frame(x = 2)
make_expectation(df)
```

---

`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.

---

`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.

output-expectations *Expectation: does code produce output/message/warning/error?*

## Description

Use `expect_output()`, `expect_message()`, `expect_warning()`, or `expect_error()` to check for specific outputs. Use `expect_silent()` to assert that there should be no output of any type. The file-based`expect_output_file()` compares the output to the contents of a text file and optionally updates it.

## Usage

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

expect_output_file(object, file, update = FALSE, ..., info = NULL,
  label = NULL)

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

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

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

expect_silent(object)
```

## Arguments

| | |
|---|---|
| `object` | object to test |
| `regexp` | regular expression to test against.<br>If `NULL`, the default, asserts that there should be an output, a messsage, a warning, or an error, but does not test for specific value.<br>If `NA`, asserts that there should be no output, messages, warnings, or errors. |
| `...` | Additional arguments passed on to `grepl`, e.g. `ignore.case` or `fixed`. |
| `info` | extra information to be included in the message (useful when writing tests in loops). |
| `label` | object label. When `NULL`, computed from deparsed object. |
| `file` | Path to a "golden" text file that contains the desired output. |
| `update` | Should the "golden" text file be updated? Default: `FALSE`. |
| `all` | For messages and warnings, do all need to the `regexp` (TRUE), or does only one need to match (FALSE) |

## See Also

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

## Examples

```
# 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), "*x*", 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), "*x*", 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 ------------------------------------------------------------------
expect_silent("123")
```

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

## End(Not run)
```

---

`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.

---

`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.

skip *Skip a test.*

## Description

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

## Usage

```
skip(message)

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

skip_if_not_installed(pkg)

skip_on_cran()

skip_on_os(os)

skip_on_travis()

skip_on_appveyor()

skip_on_bioc()
```

## Arguments

| | |
|---|---|
| message | A message describing why the test was skipped. |
| condition | Boolean condition to check. If not TRUE, will skip the test. |
| pkg | Name of package to check for |
| os | Character vector of system names. Supported values are "windows", "mac", "linux" and "solaris". |

## 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 (useful for suggested packages).

### Examples

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

---

## source_file *Source a file, directory, or all helpers.*

---

### Description

The expectation is that the files can be sourced in alphabetical order. Helper scripts are R scripts accompanying test scripts but prefixed by `helper`. These scripts are once before the tests are run.

### Usage

```
source_file(path, env = test_env(), chdir = TRUE)

source_dir(path, pattern = "\\.[rR]$", env = test_env(), chdir = TRUE)

source_test_helpers(path = "tests/testthat", env = test_env())
```

### Arguments

| | |
|---|---|
| `path` | Path to tests |
| `env` | Environment in which to evaluate code. |
| `chdir` | Change working directory to `dirname(path)`? |
| `pattern` | Regular expression used to filter files |

---

## StopReporter *Test reporter: stop on error.*

---

### Description

The default reporter, executed when `expect_that` is run interactively. It responds by stop()ing 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.

## SummaryReporter *Test reporter: summary of errors.*

### Description

This is the most useful reporting reporter as it lets you know both which tests have run successfully, as well as fully reporting information about failures and errors. It is the default reporting reporter used by `test_dir` and `test_file`.

### 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.

## 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

### Usage

```
TapReporter
```

### Format

An object of class `R6ClassGenerator` of length 24.

## 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.

## testthat *R package to make testing fun!*

### Description

Try the example below. Have a look at the references and learn more from function documentation such as `expect_that`.

### Details

Software testing is important, but, in part because it is frustrating and boring, many of us avoid it.

testthat is a new testing framework for R that is easy learn and use, and integrates with your existing workflow.

### Options

`testthat.use_colours`: Should the output be coloured? (Default: `TRUE`).

`testthat.summary.max_reports`: The maximum number of detailed test reports printed for the summary reporter (default: 15).

### References

Wickham, H (2011). testthat: Get Started with Testing. **The R Journal** *3/1* 5-10. http://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf

https://github.com/hadley/testthat

http://adv-r.had.co.nz/Testing.html

## Examples

```
library(testthat)
a <- 9
expect_that(a, is_less_than(10))
expect_less_than(a, 10)
```

---

`testthat_results` *Create a 'testthat_results' object from the test results as stored in the ListReporter results field.*

---

## Description

Create a 'testthat_results' object from the test results as stored in the ListReporter results field.

## Usage

```
testthat_results(results)
```

## Arguments

| | |
|---|---|
| `results` | a list as stored in ListReporter |

## Value

its list argument as a 'testthat_results' object

## See Also

ListReporter

---

`test_dir` *Run all of the tests in a directory.*

---

## Description

Test files start with `test` and are executed in alphabetical order (but they shouldn't have dependencies). Helper files start with `helper` and loaded before any tests are run.

## Usage

```
test_dir(path, filter = NULL, reporter = "summary", env = test_env(), ...)
```

### Arguments

| | |
|---|---|
| path | path to tests |
| filter | 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 "test-" and ".R". |
| reporter | reporter to use |
| env | environment in which to execute test suite. |
| ... | Additional arguments passed to grepl to control filtering. |

### Value

the results as a "testthat_results" (list)

---

## 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

```
test_examples(path = "../../man")

test_example(path)
```

### 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. |

---

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

---

### Description

Run all tests in specified file.

### Usage

```
test_file(path, reporter = "summary", env = test_env(),
  start_end_reporter = TRUE, load_helpers = 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? |

### Value

the results as a "testthat_results" (list)

---

## test_package *Run all tests in an installed package.*

---

### Description

Test are run in an environment that inherits from the package's namespace environment, so that tests can access non-exported functions and variables. Tests should be placed in `tests/testthat`. Use `test_check` with `R CMD check` and `test_package` interactively at the console.

### Usage

```
test_package(package, filter = NULL, reporter = "summary", ...)

test_check(package, filter = NULL, reporter = "check", ...)
```

### Arguments

| | |
|---|---|
| package | package name |
| filter | 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 `"test-"` and `".R"`. |
| reporter | reporter to use |
| ... | Additional arguments passed to `grepl` to control filtering. |

### Value

the results as a "testthat_results" (list)

### R CMD check

Create `tests/testthat.R` that contains:

```
library(testthat)
library(yourpackage)

test_check("yourpackage")
```

### Examples

```
## Not run: test_package("testthat")
```

---

`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. An error will be thrown if the path doesn't exist.

---

`test_that` *Create a test.*

---

### Description

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

### Usage

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

### Arguments

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

### Details

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

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

## Examples

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

## End(Not run)
```

---

`try_again` *Try evaluating an expressing multiple times until it succeeds.*

---

## Description

Try evaluating an expressing multiple times until it succeeds.

## Usage

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

## Arguments

| | |
|---|---|
| `times` | Maximum number of attempts. |
| `code` | Code to evaluate |

## Examples

```
third_try <- local({
  i <- 3
  function() {
    i <<- i - 1
    if (i > 0) fail(paste0("i is ", i))
  }
})
try_again(3, third_try())
```

use_catch *Use Catch for C++ Unit Testing*

## Description

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

## Usage

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

## Arguments

| | |
|---|---|
| `dir` | The directory containing an R package. |

## Details

Calling `use_catch()` will:

1. Create a file `src/test-runner.cpp`, which ensures that the `testthat` package will understand how to run your package's unit tests,
2. Create an example test file `src/test-example.cpp`, which showcases how you might use Catch to write a unit test, and
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`.

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

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

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

## Functions

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

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

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

## Advanced Usage

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

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

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

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

## Standalone Usage

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

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

as necessary within your unit test suite.

## See Also

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

## watch *Watch a directory for changes (additions, deletions & modifications).*

### Description

This is used to power the auto_test and auto_test_package functions which are used to rerun tests whenever source code changes.

### Usage

```
watch(path, callback, pattern = NULL, hash = TRUE)
```

### Arguments

| | |
|---|---|
| path | character vector of paths to watch. Omit trailing backslash. |
| callback | function called everytime a change occurs. It should have three parameters: added, deleted, modified, and should return TRUE to keep watching, or FALSE to stop. |
| pattern | file pattern passed to dir |
| hash | hashes are more accurate at detecting changes, but are slower for large files. When FALSE, uses modification time stamps |

### Details

Use Ctrl + break (windows), Esc (mac gui) or Ctrl + C (command line) to stop the watcher.

## with_mock *Mock functions in a package.*

### Description

Executes code after temporarily substituting implementations of package functions. This is useful for testing code that relies on functions that are slow, have unintended side effects or access resources that may not be available when testing.

### Usage

```
with_mock(..., .env = topenv())
```

### Arguments

| | |
|---|---|
| ... | named parameters redefine mocked functions, unnamed parameters will be evaluated after mocking the functions |
| .env | the environment in which to patch the functions, defaults to the top-level environment. A character is interpreted as package name. |

## Details

This works by using some C code to temporarily modify the mocked function *in place*. On exit (regular or error), all functions are restored to their previous state. This is somewhat abusive of R's internals, and is still experimental, so use with care.

Primitives (such as `interactive`) cannot be mocked, but this can be worked around easily by defining a wrapper function with the same name.

## Value

The result of the last unnamed parameter

## References

Suraj Gupta (2012): How R Searches And Finds Stuff

## Examples

```
with_mock(
  all.equal = function(x, y, ...) TRUE,
  expect_equal(2 * 3, 4),
  .env = "base"
)
with_mock(
  `base::identical` = function(x, y, ...) TRUE,
  `base::all.equal` = function(x, y, ...) TRUE,
  expect_equal(x <- 3 * 3, 6),
  expect_identical(x + 4, 9)
)
## Not run:
expect_equal(3, 5)
expect_identical(3, 5)

## End(Not run)
```

# Index