LuaUnit Documentation

Release 3.3

Philippe Fremy

Oct 02, 2018

Contents

CHAPTER 1

Introduction

LuaUnit is a popular unit-testing framework for Lua, with an interface typical of xUnit libraries (Python unittest, Junit, NUnit, ...). It supports several output formats (Text, TAP, JUnit, ...) to be used directly or work with Continuous Integration platforms (Jenkins, Hudson, ...).

For simplicity, LuaUnit is contained into a single-file and has no external dependency. To start using it, just add the file *luaunit.lua* to your project. A LuaRocks package is also available.

Tutorial and reference documentation is available on Read-the-docs .

LuaUnit also provides some dedicated support to scientific computing. See the section *Scientific computing and LuaUnit*

LuaUnit may also be used as an assertion library. In that case, you will call the assertion functions, which generate errors when the assertion fails. The error includes a detailed analysis of the failed assertion, like when executing a test suite.

LuaUnit provides another generic usage function: `prettystr()` which converts any value to a nicely formatted string. It supports in particular tables, nested table and even recursive tables.

CHAPTER 2

More details

LuaUnit provides a wide range of assertions and goes into great efforts to provide the most useful output. For example since version 3.3 , comparing lists will provide a detailed difference analysis:

```
-- lua test code. Can you spot the difference ?
function TestListCompare:test1()
    local A = { 121221, 122211, 121221, 122211, 121221, 122212, 121212, 122112,␣
↪122121, 121212, 122121 }
    local B = { 121221, 122211, 121221, 122211, 121221, 122212, 121212, 122112,␣
↪121221, 121212, 122121 }
    lu.assertEquals( A, B )
end

$ lua test_some_lists_comparison.lua

TestListCompare.test1 ... FAIL
test/some_lists_comparisons.lua:22: expected:

List difference analysis:
* lists A (actual) and B (expected) have the same size
* lists A and B start differing at index 9
* lists A and B are equal again from index 10
* Common parts:
  = A[1], B[1]: 121221
  = A[2], B[2]: 122211
  = A[3], B[3]: 121221
  = A[4], B[4]: 122211
  = A[5], B[5]: 121221
  = A[6], B[6]: 122212
  = A[7], B[7]: 121212
  = A[8], B[8]: 122112
* Differing parts:
  - A[9]: 122121
  + B[9]: 121221
* Common parts at the end of the lists
```

(continues on next page)

(continued from previous page)

```
  = A[10], B[10]: 121212
  = A[11], B[11]: 122121
```

The command-line options provide a flexible interface to select tests by name or patterns, control output format, set verbosity and more. See *Using the command-line* .

LuaUnit is very well tested: code coverage is 99.5% . The test suite is run on every version of Lua (Lua 5.1 to 5.3, LuaJIT 2.0 and 2.1 beta) and on many OS (Windows Seven, Windows Server 2012, MacOs X and Ubuntu). You can check the continuous build results on Travis-CI and AppVeyor .

LuaUnit is maintained on GitHub: https://github.com/bluebird75/luaunit . We gladly accept feature requests and even better Pull Requests.

LuaUnit is released under the BSD license.

2.1 Installation

LuaUnit is packed into a single-file. To make start using it, just add the file to your project.

Several installation methods are available.

2.1.1 LuaRocks

LuaUnit v3.3 is available as a LuaRocks package .

2.1.2 GitHub

The simplest way to install LuaUnit is to fetch the GitHub version:

```
git clone git@github.com:bluebird75/luaunit.git
```

Then copy the file luaunit.lua into your project or the Lua libs directory.

The version in development on GitHub is always stable and can be used safely.

On Linux, you can also install it into your Lua directories

```
sudo python doit.py install
```

If that fail, edit the function *install()* in the file *doit.py* to adjust the Lua version and installation directory. It uses, by default, Linux paths that depend on the version.

2.2 Upgrade note

Important note when upgrading to version 3.1 and above : there is a break of backward compatibility in version 3.1, assertions functions are no longer exported directly to the global namespace. See *Enabling global or module-level functions* on how to adjust or restore previous behavior.

2.3 LuaUnit development

See *Developing luaunit*

2.4 Version and Changelog

This documentation describes the functionality of LuaUnit v3.2 .

2.4.1 New in version 3.3 - 6. Mar 2018

- **General**
  - when comparing lists with *assertEquals()*, failure message provides an advanced comparison of the lists
  - *assertErrorMsgEquals()* can check for error raised as tables
  - tests may be finished early with *fail()*, *failIf()*, *success()* or *successIf()*
  - improve printing of recursive tables
  - improvements and fixes to JUnit and TAP output
  - stricter *assertTrue()* and *assertFalse()*: they only succeed with boolean values
  - add *assertEvalToTrue()* and *assertEvalToFalse()* with previous *assertTrue()*/*assertFalse()* behavior of coercing to boolean before asserting
  - all assertion functions accept an optional extra message, to be printed along the failure
- **New command-line arguments:**
  - can now shuffle tests with `--shuffle` or `-s`
  - possibility to repeat tests (for example to trigger a JIT), with `--repeat NUM` or `-r NUM`
  - more flexible test selection with inclusion (`--pattern` / `-p`) or exclusion (`--exclude` / `-x`) or combination of both
- **Scientific computing dedicated support (see documentation):**
  - provide the machine epsilon in EPS
  - new functions: *assertNan()*, *assertInf()*, *assertPlusInf()*, *assertMinusInf()*, *assertPlusZero()*, *assertMinusZero()* and their negative version
  - in *assertAlmostEquals()*, margin no longer provides a default value of 1E-11, the machine epsilon is used instead
- **Platform and continuous integration support:**
  - validate LuaUnit on MacOs platform (thank to Travis CI)
  - validate LuaUnit with 32 bits numbers (floats) and 64 bits numbers (double)
  - add test coverage measurements thank to coveralls.io . Status: 99.76% of the code is verified.
  - use cache for AppVeyor and Travis builds
  - support for `luarocks doc` command
- General doc improvements (detailed description of all output, more cross-linking between sections)

2.4.2 New in version 3.2 - 12. Jul 2016

- Add command-line option to stop on first error or failure. See *Other options*
- Distinguish between failures (failed assertion) and errors
- Support for new versions: Lua 5.3 and LuaJIT (2.0, 2.1 beta)
- Validation of all lua versions on Travis CI and AppVeyor
- Add compatibility layer with forked luaunit v2.x
- Added documentation about development process. See *Developing luaUnit*
- Improved support for table containing keys of type table. See *Annex B: Comparing tables with keys of type table*
- Small bug fixes, several internal improvements
- Availability of a Luarock package. See *https://luarocks.org/modules/bluebird75/luaunit* .

2.4.3 New in version 3.1 - 10. Mar 2015

- luaunit no longer pollutes global namespace, unless defining EXPORT_ASSERT_TO_GLOBALS to true. See *Enabling global or module-level functions*
- fixes and validation of JUnit XML generation
- strip luaunit internal information from stacktrace
- general improvements of test results with duration and other details
- improve printing for tables, with an option to always print table id. See *Annex A: More on table printing*
- fix printing of recursive tables

Important note when upgrading to version 3.1 : assertions functions are no longer exported directly to the global namespace. See *Enabling global or module-level functions*

2.4.4 New in version 3.0 - 9. Oct 2014

Because LuaUnit was forked and released as some 2.x version, version number is now jumping to 3.0 .

- full documentation available in text, html and pdf at http://luaunit.read-the-docs.org
- new output format: JUnit, compatible with Bamboo and other CI platforms. See *Output formats*
- much better table assertions
- new assertions for strings, with patterns and case insensitivity: assertStrContains, assertNotStrContains, assertNotStrIContains, assertStrIContains, assertStrMatches
- new assertions for floats: assertAlmostEquals, assertNotAlmostEquals
- type assertions: assertIsString, assertIsNumber, ...
- error assertions: assertErrorMsgEquals, assertErrorMsgContains, assertErrorMsgMatches
- improved error messages for several assertions
- command-line options to select test, control output type and verbosity

2.4.5 New in version 1.5 - 8. Nov 2012

- compatibility with Lua 5.1 and 5.2
- better object model internally
- a lot more of internal tests
- several internal bug fixes
- make it easy to customize the test output
- running test functions no longer requires a wrapper
- several level of verbosity

2.4.6 New in version 1.4 - 26. Jul 2012

- switch from X11 to more popular BSD license
- add TAP output format for integration into Jenkins. See *Output formats*
- official repository now on GitHub

2.4.7 New in version 1.3 - 30. Oct 2007

- port to lua 5.1
- iterate over the test classes, methods and functions in the alphabetical order
- change the default order of expected, actual in assertEquals. See *Equality assertions*

2.4.8 Version 1.2 - 13. Jun 2005

- first public release

2.4.9 Version 1.1

- move global variables to internal variables
- assertion order is configurable between expected/actual or actual/expected. See *Equality assertions*
- new assertion to check that a function call returns an error
- display the calling stack when an error is spotted
- two verbosity level, like in python unittest

CHAPTER 3

Getting started with LuaUnit

This section will guide you through a step by step usage of *LuaUnit* . The full source code of the example below is available in the : *Annex C: Source code of example* or in the file *my_test_suite.lua* in the doc directory.

3.1 Setting up your test script

To get started, create your file *my_test_suite.lua* .

The script should import LuaUnit:

```
lu = require('luaunit')
```

The last line executes your script with LuaUnit and exit with the proper error code:

```
os.exit( lu.LuaUnit.run() )
```

Now, run your file with:

```
lua my_test_suite.lua
```

It prints something like:

```
Ran 0 tests in 0.000 seconds, 0 successes, 0 failures
OK
```

Now, your testing framework is in place, you can start writing tests.

3.2 Writing tests

LuaUnit scans all variables that start with *test* or *Test*. If they are functions, or if they are tables that contain functions that start with *test* or *Test*, they are run as part of the test suite.

So just write a function whose name starts with test. Inside test functions, use the assertions functions provided by LuaUnit, such as *assertEquals()*.

Let's see that in practice.

Suppose you want to test the following add function:

```
function add(v1,v2)
    -- add positive numbers
    -- return 0 if any of the numbers are 0
    -- error if any of the two numbers are negative
    if v1 < 0 or v2 < 0 then
        error('Can only add positive or null numbers, received '..v1..' and '..v2)
    end
    if v1 == 0 or v2 == 0 then
        return 0
    end
    return v1+v2
end
```

You write the following tests:

```
function testAddPositive()
    lu.assertEquals(add(1,1),2)
end

function testAddZero()
    lu.assertEquals(add(1,0),0)
    lu.assertEquals(add(0,5),0)
    lu.assertEquals(add(0,0),0)
end
```

assertEquals() is the most commonly used assertion function. It verifies that both argument are equals, in the order actual value, expected value.

Rerun your test script (-v is to activate a more verbose output):

```
$ lua my_test_suite.lua -v
```

It now prints:

```
Started on 02/19/17 22:15:53
    TestAdd.testAddPositive ... Ok
    TestAdd.testAddZero ... Ok
=========================================================
Ran 2 tests in 0.003 seconds, 2 successes, 0 failures
OK
```

You always have:

- the date at which the test suite was started
- the group to which the function belongs (usually, the name of the function table, and <*TestFunctions*> for all direct test functions)
- the name of the function being executed
- a report at the end, with number of executed test, number of non selected tests if any, number of failures, number of errors (if any) and duration.

The difference between failures and errors are:

- luaunit assertion functions generate failures
- any unexpected error during execution generates an error
- failures or errors during setup() or teardown() always generate errors

If we continue with our example, we also want to test that when the function receives negative numbers, it generates an error. Use *assertError()* or even better, *assertErrorMsgContains()* to also validate the content of the error message. There are other types or error checking functions, see *Error assertions* . Here we use *assertErrorMsgContains()* . First argument is the expected message, then the function to call and the optional arguments:

```
function testAddError()
    lu.assertErrorMsgContains('Can only add positive or null numbers, received 2 and -
↪3', add, 2, -3)
end
```

Now, suppose we also have the following function to test:

```
function adder(v)
    -- return a function that adds v to its argument using add
    function closure( x ) return x+v end
    return closure
end
```

We want to test the type of the value returned by adder and its behavior. LuaUnit provides assertion for type testing (see *Type assertions*). In this case, we use *assertIsFunction()*:

```
function testAdder()
    f = adder(3)
    lu.assertIsFunction( f )
    lu.assertEquals( f(2), 5 )
end
```

3.3 Grouping tests, setup/teardown functionality

When the number of tests starts to grow, you usually organise them into separate groups. You can do that with LuaUnit by putting them inside a table (whose name must start with *Test* or *test*).

For example, assume we have a second function to test:

```
function div(v1,v2)
    -- divide positive numbers
    -- return 0 if any of the numbers are 0
    -- error if any of the two numbers are negative
    if v1 < 0 or v2 < 0 then
        error('Can only divide positive or null numbers, received '..v1..' and '..v2)
    end
    if v1 == 0 or v2 == 0 then
        return 0
    end
    return v1/v2
end
```

We move the tests related to the function add into their own table:

```
TestAdd = {}
    function TestAdd:testAddPositive()
        lu.assertEquals(add(1,1),2)
    end

    function TestAdd:testAddZero()
        lu.assertEquals(add(1,0),0)
        lu.assertEquals(add(0,5),0)
        lu.assertEquals(add(0,0),0)
    end

    function TestAdd:testAddError()
        lu.assertErrorMsgContains('Can only add positive or null numbers, received 2␣
↪and -3', add, 2, -3)
    end

    function TestAdd:testAdder()
        f = adder(3)
        lu.assertIsFunction( f )
        lu.assertEquals( f(2), 5 )
    end
-- end of table TestAdd
```

Then we create a second set of tests for div:

```
TestDiv = {}
    function TestDiv:testDivPositive()
        lu.assertEquals(div(4,2),2)
    end

    function TestDiv:testDivZero()
        lu.assertEquals(div(4,0),0)
        lu.assertEquals(div(0,5),0)
        lu.assertEquals(div(0,0),0)
    end

    function TestDiv:testDivError()
        lu.assertErrorMsgContains('Can only divide positive or null numbers, received␣
↪2 and -3', div, 2, -3)
    end
-- end of table TestDiv
```

Execution of the test suite now looks like this:

```
Started on 02/19/17 22:15:53
    TestAdd.testAddError ... Ok
    TestAdd.testAddPositive ... Ok
    TestAdd.testAddZero ... Ok
    TestAdd.testAdder ... Ok
    TestDiv.testDivError ... Ok
    TestDiv.testDivPositive ... Ok
    TestDiv.testDivZero ... Ok
=========================================================
Ran 7 tests in 0.006 seconds, 7 successes, 0 failures
OK
```

When tests are defined in tables, you can optionally define two special functions, *setUp()* and *tearDown()*, which will be executed respectively before and after every test.

These function may be used to create specific resources for the test being executed and cleanup the test environment.

For a practical example, imagine that we have a *log()* function that writes strings to a log file on disk. The file is created upon first usage of the function, and the filename is defined by calling the function *initLog()*.

The tests for these functions would take advantage of the *setup/teardown* functionality to prepare a log filename shared by all tests, make sure that all tests start with a non existing log file name, and delete the log filename after every test:

```
TestLogger = {}
    function TestLogger:setUp()
        -- define the fname to use for logging
        self.fname = 'mytmplog.log'
        -- make sure the file does not already exists
        os.remove(self.fname)
    end

    function TestLogger:testLoggerCreatesFile()
        initLog(self.fname)
        log('toto')
        -- make sure that our log file was created
        f = io.open(self.fname, 'r')
        lu.assertNotNil( f )
        f:close()
    end

    function TestLogger:tearDown()
        -- cleanup our log file after all tests
        os.remove(self.fname)
    end
```

Note: *Errors generated during execution of setUp() or tearDown() functions are considered test failures.*

Note: *For compatibility with luaunit v2 and other lua unit-test frameworks, setUp() and tearDown() may also be named setup(), SetUp(), Setup() and teardown(), TearDown(), Teardown().*

3.4 Using the command-line

You can control the LuaUnit execution from the command-line:

Output format

Choose the test output format with `-o` or `--output`. Available formats are:

- text: the default output format
- nil: no output at all
- tap: TAP format
- junit: output junit xml

Example of non-verbose text format:

```
$ lua doc/my_test_suite.lua
.......
Ran 7 tests in 0.003 seconds, 7 successes, 0 failures
OK
```

Example of TAP format:

```
$ lua doc/my_test_suite.lua -o TAP
1..7
# Started on 02/19/17 22:15:53
# Starting class: TestAdd
ok     1        TestAdd.testAddError
ok     2        TestAdd.testAddPositive
ok     3        TestAdd.testAddZero
ok     4        TestAdd.testAdder
# Starting class: TestDiv
ok     5        TestDiv.testDivError
ok     6        TestDiv.testDivPositive
ok     7        TestDiv.testDivZero
# Ran 7 tests in 0.007 seconds, 7 successes, 0 failures
```

For a more detailed overview of all formats and their verbosity see the section *Output formats* .

List of tests to run

You can list some test names on the command-line to run only those tests. The name must be the exact match of either the test table, the test function or the test table and the test method. The option may be repeated.

Example:

```
-- Run all TestAdd table tests and one test of TestDiv table.
$ lua doc/my_test_suite.lua TestAdd TestDiv.testDivError -v
Started on 02/19/17 22:15:53
    TestAdd.testAddError ... Ok
    TestAdd.testAddPositive ... Ok
    TestAdd.testAddZero ... Ok
    TestAdd.testAdder ... Ok
    TestDiv.testDivError ... Ok
========================================================
Ran 5 tests in 0.003 seconds, 5 successes, 0 failures
OK
```

Including / excluding tests

The most flexible approach for selecting tests to use the include and exclude functionality. With `--pattern` or `-p`, you can provide a lua pattern and only the tests that contain the pattern will actually be run.

Example:

```
-- Run all tests of zero testing and error testing
-- by using the magic character .
$ lua my_test_suite.lua -v -p Err.r -p Z.ro
```

For our test suite, it gives the following output:

```
Started on 02/19/17 22:15:53
    TestAdd.testAddError ... Ok
    TestAdd.testAddZero ... Ok
    TestDiv.testDivError ... Ok
```

(continues on next page)

(continued from previous page)

```
    TestDiv.testDivZero ... Ok
============================================================
Ran 4 tests in 0.003 seconds, 4 successes, 0 failures, 3 non-selected
OK
```

The number of tests ignored by the selection is printed, along with the test result. The pattern can be any lua pattern. Be sure to exclude all magic characters with % (like -+?*) and protect your pattern from the shell interpretation by putting it in quotes.

You can also exclude tests that match some patterns:

Example:

```
-- Run all tests except zero testing and except error testing
$ lua my_test_suite.lua -v -x Error -x Zero
```

For our test suite, it gives the following output:

```
Started on 02/19/17 22:29:45
    TestAdd.testAddPositive ... Ok
    TestAdd.testAdder ... Ok
    TestDiv.testDivPositive ... Ok
============================================================
Ran 3 tests in 0.003 seconds, 3 successes, 0 failures, 4 non-selected
OK
```

You can also combine test selection and test exclusion. See *Flexible test selection*

3.5 Conclusion

You now know enough of LuaUnit to start writing your test suite. Check the reference documentation for a complete list of assertions, command-line options and specific behavior.

CHAPTER 4

Reference documentation

4.1 Enabling global or module-level functions

Versions of LuaUnit before version 3.1 would export all assertions functions to the global namespace. A typical lua test file would look like this:

```
require('luaunit')

TestToto = {} --class

    function TestToto:test1_withFailure()
        local a = 1
        assertEquals( a , 1 )
        -- will fail
        assertEquals( a , 2 )
    end

[...]
```

However, this is an obsolete practice in Lua. It is now recommended to keep all functions inside the module. Starting from version 3.1 LuaUnit follows this practice and the code should be adapted to look like this:

```
-- the imported module must be stored
lu = require('luaunit')

TestToto = {} --class

    function TestToto:test1_withFailure()
        local a = 1
        lu.assertEquals( a , 1 )
        -- will fail
        lu.assertEquals( a , 2 )
    end
```

(continues on next page)

(continued from previous page)

```
[...]
```

If you prefer the old way, LuaUnit can continue to export assertions functions if you set the following global variable **prior** to importing LuaUnit:

```
-- this works
EXPORT_ASSERT_TO_GLOBALS = true
require('luaunit')

TestToto = {} --class

    function TestToto:test1_withFailure()
        local a = 1
        assertEquals( a , 1 )
        -- will fail
        assertEquals( a , 2 )
    end

[...]
```

4.2 LuaUnit.run() function

Return value

Run your test suite with the following line:

```
os.exit(lu.LauaUnit.run())
```

The *run()* function returns the number of failures of the test suite. This is good for an exit code, 0 meaning success.

Arguments

If no arguments are supplied, it parses the command-line arguments of the script and interpret them. If arguments are supplied to the function, they are parsed instead of the command-line. It uses the same syntax.

Example:

```
-- execute tests matching the 'withXY' pattern
os.exit(lu.LuaUnit.run('--pattern', 'withXY'))
```

Choice of tests

If test names were supplied, only those tests are executed. When test names are supplied as arguments, they don't have to start with *test*, they are run anyway.

If no test names were supplied, a general test collection process starts under the following rules:

- all variable starting with *test* or *Test* are scanned.
- if the variable is a function it is collected for testing
- if the variable is a table:
  - all keys starting with *test* or *Test* are collected (provided that they are functions)
  - keys with name *setUp* and *tearDown* are also collected

If one or more pattern were supplied, the test are then filtered according the pattern(s). Only the test that match the pattern(s) are actually executed.

setup and teardown

The function *setUp()* is executed before each test if it exists in the table. The function *tearDown()* is executed after every test if it exists in the table.

Note: *tearDown()* is always executed if it exists, even if there was a failure in the test or in the *setUp()* function. Failures in *setUp()* or *tearDown()* are considered as a general test failures.

4.3 LuaUnit.runSuite() function

If you want to keep the flexibility of the command-line parsing, but want to force some parameters, like the output format, you must use a slightly different syntax:

```
runner = lu.LuaUnit.new()
runner:setOutputType("tap")
os.exit( runner:runSuite() )
```

runSuite() behaves like *run()* except that it must be started with a LuaUnit instance as first argument, and it will use the LuaUnit instance settings.

4.4 Command-line options

Usage: lua <your_test_suite.lua> [options] [testname1 [testname2] ...]

Test names

When no test names are supplied, all tests are collected.

The syntax for supplying test names can be either: name of the function, name of the table or [name of the table].[name of the function]. Only the supplied tests will be executed.

Selecting tests with –pattern and –exclude is usually more flexible. See *Flexible test selection*

Options

--output, -o FORMAT Set output format to FORMAT. Possible values: text, tap, junit, nil . See *Output formats*

--name, -n FILENAME For junit format only, mandatory name of xml file. Ignored for other formats.

--pattern, -p PATTERN Execute all test names matching the Lua PATTERN. May be repeated to include severals patterns. See *Flexible test selection*

--exclude, -x PATTERN Exclude all test names matching the Lua PATTERN. May be repeated to exclude severals patterns. See *Flexible test selection*

--repeat, -r NUM Repeat all tests NUM times, e.g. to trigger the JIT. See *Other options*

--shuffle, -s Shuffle tests before running them. See *Other options*

--error, -e Stop on first error. See *Other options*

--failure, -f Stop on first failure or error. See *Other options*

--verbose, -v Increase verbosity

--quiet, -q	Set verbosity to minimum
--help, -h	Print help
--version	Version information of LuaUnit

4.4.1 Output formats

Choose the output format with the syntax `-o FORMAT` or `--output FORMAT`.

Formats available:

- `text`: the default output format of LuaUnit
- `tap`: output compatible with the Test Anything Protocol
- `junit`: output compatible with the *JUnit XML* format (used by many CI platforms). The XML is written to the file provided with the `--name` or `-n` option.
- `nil`: no output at all

To demonstrate the different output formats, we will take the example of the *Getting started with LuaUnit* section and add the following two failing cases:

```
TestWithFailures = {}
    -- two failing tests

    function TestWithFailures:testFail1()
        local a="toto"
        local b="titi"
        lu.assertEquals( a, b ) --oops, two values are not equal
    end

    function TestWithFailures:testFail2()
        local a=1
        local b='toto'
        local c = a + b --oops, can not add string and numbers
        return c
    end
```

Text format

By default, LuaUnit uses the output format TEXT, with minimum verbosity:

```
$ lua my_test_suite.lua
.......FE
Failed tests:
-------------
1) TestWithFailures.testFail1
doc\my_test_suite_with_failures.lua:79: expected: "titi"
actual: "toto"
stack traceback:
        doc\my_test_suite_with_failures.lua:79: in function 'TestWithFailures.
↪testFail1'

2) TestWithFailures.testFail2
doc\my_test_suite_with_failures.lua:85: attempt to perform arithmetic on local 'b' (a␣
↪string value)
stack traceback:
        [C]: in function 'xpcall'
```

(continues on next page)

(continued from previous page)

```
Ran 9 tests in 0.001 seconds, 7 successes, 1 failure, 1 error
```

This format is heavily inspired by python unit-test library. One character is printed for every test executed, a dot for a successful test, a **F** for a test with failure and a **E** for a test with an error.

At the end of the test suite execution, the details of the failures or errors are given, with an informative message and a full stack trace.

The last line sums up the number of test executed, successful, failed, in error and not selected if any. When all tests are successful, a line with just OK is added:

```
$ lua doc\my_test_suite.lua
.......
Ran 7 tests in 0.002 seconds, 7 successes, 0 failures
OK
```

The text format is also available as a more verbose version, by adding the `--verbose` flag:

```
$ lua doc\my_test_suite_with_failures.lua --verbose
Started on 02/20/17 21:47:21
    TestAdd.testAddError ... Ok
    TestAdd.testAddPositive ... Ok
    TestAdd.testAddZero ... Ok
    TestAdd.testAdder ... Ok
    TestDiv.testDivError ... Ok
    TestDiv.testDivPositive ... Ok
    TestDiv.testDivZero ... Ok
    TestWithFailures.testFail1 ... FAIL
doc\my_test_suite_with_failures.lua:79: expected: "titi"
actual: "toto"
    TestWithFailures.testFail2 ... ERROR
doc\my_test_suite_with_failures.lua:85: attempt to perform arithmetic on local 'b' (a␣
↪string value)
=========================================================
Failed tests:
-------------
1) TestWithFailures.testFail1
doc\my_test_suite_with_failures.lua:79: expected: "titi"
actual: "toto"
stack traceback:
        doc\my_test_suite_with_failures.lua:79: in function 'TestWithFailures.
↪testFail1'

2) TestWithFailures.testFail2
doc\my_test_suite_with_failures.lua:85: attempt to perform arithmetic on local 'b' (a␣
↪string value)
stack traceback:
        [C]: in function 'xpcall'

Ran 9 tests in 0.008 seconds, 7 successes, 1 failure, 1 error
```

In this format, you get:

- a first line with date-time at which the test was started
- one line per test executed
- the test line is ended by **Ok**, **FAIL**, or **ERROR** in case the test is not successful

- a summary of the failed tests with all details, like in the compact version.

This format is usually interesting if some tests print debug output, to match the output to the test.

JUNIT format

The Junit XML format was introduced by the Java testing framework JUnit and has been then used by many continuous integration platform as an interoperability format between test suites and the platform.

To output in the JUnit XML format, you use the format junit with `--output junit` and specify the XML filename with `--name <filename>`. On the standard output, LuaUnit will print information about the test progress in a simple format.

Let's see with a simple example:

```
$ lua my_test_suite_with_failures.lua -o junit -n toto.xml
# XML output to toto.xml
# Started on 02/24/17 09:54:59
# Starting class: TestAdd
# Starting test: TestAdd.testAddError
# Starting test: TestAdd.testAddPositive
# Starting test: TestAdd.testAddZero
# Starting test: TestAdd.testAdder
# Starting class: TestDiv
# Starting test: TestDiv.testDivError
# Starting test: TestDiv.testDivPositive
# Starting test: TestDiv.testDivZero
# Starting class: TestWithFailures
# Starting test: TestWithFailures.testFail1
# Failure: doc/my_test_suite_with_failures.lua:79: expected: "titi"
# actual: "toto"
# Starting test: TestWithFailures.testFail2
# Error: doc/my_test_suite_with_failures.lua:85: attempt to perform arithmetic on␣
↪local 'b' (a string value)
# Ran 9 tests in 0.007 seconds, 7 successes, 1 failure, 1 error
```

On the standard output, you will see the date-time, the name of the XML file, one line for each test started, a summary of the failure or errors when they occurs and the usual one line summary of the test execution: number of tests run, successful, failed, in error and number of non selected tests if any.

The XML file generated by this execution is the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<testsuites>
    <testsuite name="LuaUnit" id="00001" package="" hostname="localhost" tests="9"␣
↪timestamp="2017-02-24T09:54:59" time="0.007" errors="1" failures="1">
        <properties>
            <property name="Lua Version" value="Lua 5.2"/>
            <property name="LuaUnit Version" value="3.2"/>
        </properties>
        <testcase classname="TestAdd" name="TestAdd.testAddError" time="0.001">
        </testcase>
        <testcase classname="TestAdd" name="TestAdd.testAddPositive" time="0.001">
        </testcase>
        <testcase classname="TestAdd" name="TestAdd.testAddZero" time="0.000">
        </testcase>
        <testcase classname="TestAdd" name="TestAdd.testAdder" time="0.000">
        </testcase>
        <testcase classname="TestDiv" name="TestDiv.testDivError" time="0.000">
        </testcase>
```

(continues on next page)

(continued from previous page)

```
        <testcase classname="TestDiv" name="TestDiv.testDivPositive" time="0.000">
        </testcase>
        <testcase classname="TestDiv" name="TestDiv.testDivZero" time="0.001">
        </testcase>
        <testcase classname="TestWithFailures" name="TestWithFailures.testFail1" time=
↪"0.000">
            <failure type="doc/my_test_suite_with_failures.lua:79: expected: &quot;
↪titi&quot;
actual: &quot;toto&quot;">
                <![CDATA[stack traceback:
        doc/my_test_suite_with_failures.lua:79: in function 'TestWithFailures.
↪testFail1']]></failure>
        </testcase>
        <testcase classname="TestWithFailures" name="TestWithFailures.testFail2" time=
↪"0.000">
            <error type="doc/my_test_suite_with_failures.lua:85: attempt to perform␣
↪arithmetic on local &apos;b&apos; (a string value)">
                <![CDATA[stack traceback:
        [C]: in function 'xpcall']]></error>
        </testcase>
    <system-out/>
    <system-err/>
    </testsuite>
</testsuites>
```

As you can see, the XML file is quite rich in terms of information. The verbosity level has no effect on junit output, all verbosity give the same output.

Slight inconsistencies exist in the exact XML format in the different continuous integration suites. LuaUnit provides a compatible output which is validated against Jenkins/Hudson schema and Ant/Maven schema . If you ever find an problem in the XML formats, please report a bug to us, more testing is always welcome.

TAP format

The TAP format for test results has been around since 1988. LuaUnit produces TAP reports compatible with version 12 of the specification.

Example with minimal verbosiy:

```
$ lua my_test_suite_with_failures.lua -o tap --quiet
1..9
# Started on 02/24/17 22:09:31
# Starting class: TestAdd
ok     1        TestAdd.testAddError
ok     2        TestAdd.testAddPositive
ok     3        TestAdd.testAddZero
ok     4        TestAdd.testAdder
# Starting class: TestDiv
ok     5        TestDiv.testDivError
ok     6        TestDiv.testDivPositive
ok     7        TestDiv.testDivZero
# Starting class: TestWithFailures
not ok 8        TestWithFailures.testFail1
not ok 9        TestWithFailures.testFail2
# Ran 9 tests in 0.003 seconds, 7 successes, 1 failure, 1 error
```

With minimal verbosity, you have one line for each test run, with the status of the test, and one comment line when starting the test suite, when starting a new class or when finishing the test.

Example with default verbosiy:

```
$ lua my_test_suite_with_failures.lua -o tap
1..9
# Started on 02/24/17 22:09:31
# Starting class: TestAdd
ok     1        TestAdd.testAddError
ok     2        TestAdd.testAddPositive
ok     3        TestAdd.testAddZero
ok     4        TestAdd.testAdder
# Starting class: TestDiv
ok     5        TestDiv.testDivError
ok     6        TestDiv.testDivPositive
ok     7        TestDiv.testDivZero
# Starting class: TestWithFailures
not ok 8        TestWithFailures.testFail1
    doc/my_test_suite_with_failures.lua:79: expected: "titi"
    actual: "toto"
not ok 9        TestWithFailures.testFail2
    doc/my_test_suite_with_failures.lua:85: attempt to perform arithmetic on local 'b
↪' (a string value)
# Ran 9 tests in 0.005 seconds, 7 successes, 1 failure, 1 error
```

In the default mode, the failure or error message is displayed in the failing test diagnostic part.

Example with full verbosiy:

```
$ lua my_test_suite_with_failures.lua -o tap --verbose
1..9
# Started on 02/24/17 22:09:31
# Starting class: TestAdd
ok     1        TestAdd.testAddError
ok     2        TestAdd.testAddPositive
ok     3        TestAdd.testAddZero
ok     4        TestAdd.testAdder
# Starting class: TestDiv
ok     5        TestDiv.testDivError
ok     6        TestDiv.testDivPositive
ok     7        TestDiv.testDivZero
# Starting class: TestWithFailures
not ok 8        TestWithFailures.testFail1
    doc/my_test_suite_with_failures.lua:79: expected: "titi"
    actual: "toto"
    stack traceback:
        doc/my_test_suite_with_failures.lua:79: in function 'TestWithFailures.
↪testFail1'
not ok 9        TestWithFailures.testFail2
    doc/my_test_suite_with_failures.lua:85: attempt to perform arithmetic on local 'b
↪' (a string value)
    stack traceback:
        [C]: in function 'xpcall'
# Ran 9 tests in 0.007 seconds, 7 successes, 1 failure, 1 error
```

With maximum verbosity, the stack trace is also displayed in the test diagnostic.

NIL format

With the nil format output, absolutely nothing is displayed while running the tests. Only the exit code of the command can tell whether the test was successful or not:

```
$ lua my_test_suite_with_failures.lua -o nil --verbose
$
```

This mode is used by LuaUnit for its internal validation.

4.4.2 Other options

Stopping on first error or failure

If `--failure` or `-f` is passed as an option, LuaUnit will stop on the first failure or error and display the test results.

If `--error` or `-e` is passed as an option, LuaUnit will stop on the first error (but continue on failures).

Randomize test order

If `--shuffle` or `-s` is passed as an option, LuaUnit will execute tests in random order. The randomisation works on all test functions and methods. As a consequence test methods of a given class may be splitted into multiple location, generating several test class creation and destruction.

Repeat test

When using luajit, the just-in-time compiler will kick in only after a given function has been executed a sufficient number of times. To make sure that the JIT is not introducing any bug, LuaUnit provides a way to repeat a test may times, with `--repeat` or `-r` followed by a number.

4.4.3 Flexible test selection

LuaUnit provides very flexible way to select which tests to execute. We will illustrate this with several examples.

In the examples, we use a test suite composed of the following test funcions:

```
-- class: TestAdd
TestAdd.testAddError
TestAdd.testAddPositive
TestAdd.testAddZero
TestAdd.testAdder

-- class: TestDiv
TestDiv.testDivError
TestDiv.testDivPositive
TestDiv.testDivZero
```

With `--pattern` or `-p`, you can provide a lua pattern and only the tests that contain the pattern will actually be run.

Example:

```
-- Run all tests of zero testing and error testing
-- by using the magic character .
$ lua mytest_suite.lua -v -p Err.r -p Z.ro
Started on 02/19/17 22:29:45
    TestAdd.testAddError ... Ok
    TestAdd.testAddZero ... Ok
    TestDiv.testDivError ... Ok
    TestDiv.testDivZero ... Ok
=========================================================
Ran 4 tests in 0.004 seconds, 4 successes, 0 failures, 3 non-selected
OK
```

The number of tests ignored by the selection is printed, along with the test result. The tests *TestAdd.testAdder testAdd.testPositive and testDiv.testDivPositive* have been correctly ignored.

The pattern can be any lua pattern. Be sure to exclude all magic characters with % (like `-+?*`) and protect your pattern from the shell interpretation by putting it in quotes.

With `--exclude` or `-x`, you can provide a lua pattern of tests which should be excluded from execution.

Example:

```
-- Run all tests except zero testing and except error testing
$ lua mytest_suite.lua -v -x Error -x Zero
Started on 02/19/17 22:29:45
    TestAdd.testAddPositive ... Ok
    TestAdd.testAdder ... Ok
    TestDiv.testDivPositive ... Ok
=========================================================
Ran 3 tests in 0.003 seconds, 3 successes, 0 failures, 4 non-selected
OK
```

You can also combine test selection and test exclusion. The rules are the following:

- if the first argument encountered is a inclusion pattern, the list of tests start empty
- if the first argument encountered is an exclusion pattern, the list of tests start with all tests of the suite
- each subsequent inclusion pattern will add new tests to the list
- each subsequent exclusion pattern will remove test from the list
- the final list is the list of tests executed

In pure logic term, inclusion is the equivalent of `or match(pattern)` and exclusion is `and not match(pattern)`.

Let's look at some practical examples:

```
-- Add all tests which include the word Add
-- except the test Adder
-- and also include the Zero tests
$ lua my_test_suite.lua -v --pattern Add --exclude Adder --pattern Zero
Started on 02/19/17 22:29:45
    TestAdd.testAddError ... Ok
    TestAdd.testAddPositive ... Ok
    TestAdd.testAddZero ... Ok
    TestDiv.testDivZero ... Ok
=========================================================
Ran 4 tests in 0.003 seconds, 4 successes, 0 failures, 3 non-selected
OK
```

4.5 Assertions functions

We will now list all assertion functions. For every functions, the failure message tries to be as informative as possible, by displaying the expectation and value that caused the failure. It relies on the *prettystr()* for printing nicely formatted values.

All function accept an optional extra message which if provided, is printed along with the failure message.

Note: see *Annex A: More on table printing* and *Annex B: Comparing tables with keys of type table* for more dealing with recursive tables and tables containing keys of type table.

4.5.1 Equality assertions

All equality assertions functions take two arguments, in the order *actual value* then *expected value*. Some people are more familiar with the order *expected value* then *actual value*. It is possible to configure LuaUnit to use the opposite order for all equality assertions, by setting up a module variable:

```
lu.ORDER_ACTUAL_EXPECTED=false
```

The order only matters for the message that is displayed in case of failures. It does not influence the test itself.

assertEquals (*actual*, *expected*[, *extra_msg*])
Alias: *assert_equals()*

Assert that two values are equal.

For tables, the comparison is a deep comparison :

- number of elements must be the same
- tables must contain the same keys
- each key must contain the same values. The values are also compared recursively with deep comparison.

If provided, *extra_msg* is a string which will be printed along with the failure message.

LuaUnit provides other table-related assertions, see *Table assertions* .

assertNotEquals (*actual*, *expected*[, *extra_msg*])
Alias: *assert_not_equals()*

Assert that two values are different. The assertion fails if the two values are identical. Like the previous function, it uses table deep comparison.

If provided, *extra_msg* is a string which will be printed along with the failure message.

4.5.2 Value assertions

LuaUnit contains several flavours of true/false assertions, to be used in different contexts. Usually, when asserting for *true* or *false*, you want strict assertions (*nil* should not assert to *false*); *assertTrue()* and *assertFalse()* are the functions for this purpose. In some cases though, you want Lua coercion rules to apply (e.g. value *1* or string *"hello"* yields *true*) and the right functions to use are *assertEvalToTrue()* and *assertEvalToFalse()*. Finally, you have the *assertNotTrue()* and *assertNotFalse()* to verify that a value is anything but the boolean *true* or *false*.

The below table sums it up:

True assertion family

Input Value	assertTrue()	assertEvalToTrue()	assertNotTrue()
true	OK	OK	OK
false	Fail	Fail	Fail
nil	Fail	Fail	OK
0	Fail	OK	OK
1	Fail	OK	OK
"hello"	Fail	OK	OK

False assertion family

Input Value	assertNotFalse()	assertFalse()	assertEvalToFalse()
true	Fail	Fail	Fail
false	OK	OK	OK
nil	Fail	OK	OK
0	Fail	Fail	Fail
1	Fail	Fail	Fail
"hello"	Fail	Fail	Fail

assertEvalToTrue (*value*[, *extra_msg*])
Alias: *assert_eval_to_true()*

Assert that a given value evals to `true`. Lua coercion rules are applied so that values like `0`, `""`, `1.17` **succeed** in this assertion. If provided, extra_msg is a string which will be printed along with the failure message.

See *assertTrue()* for a strict assertion to boolean `true`.

assertEvalToFalse (*value*[, *extra_msg*])
Alias: *assert_eval_to_false()*

Assert that a given value eval to `false`. Lua coercion rules are applied so that `nil` and `false` **succeed** in this assertion. If provided, extra_msg is a string which will be printed along with the failure message.

See *assertFalse()* for a strict assertion to boolean `false`.

assertTrue (*value*[, *extra_msg*])
Alias: *assert_true()*

Assert that a given value is strictly `true`. Lua coercion rules do not apply so that values like `0`, `""`, `1.17` **fail** in this assertion. If provided, extra_msg is a string which will be printed along with the failure message.

See *assertEvalToTrue()* for an assertion to `true` where Lua coercion rules apply.

assertFalse (*value*[, *extra_msg*])
Alias: *assert_false()*

Assert that a given value is strictly `false`. Lua coercion rules do not apply so that `nil` **fails** in this assertion. If provided, *extra_msg* is a string which will be printed along with the failure message.

See *assertEvalToFalse()* for an assertion to `false` where Lua coertion fules apply.

assertNil (*value*[, *extra_msg*])
Aliases: *assert_nil()*, *assertIsNil()*, *assert_is_nil()*

Assert that a given value is *nil* . If provided, *extra_msg* is a string which will be printed along with the failure message.

assertNotNil (*value*[, *extra_msg*])
Aliases: *assert_not_nil()*, *assertNotIsNil()*, *assert_not_is_nil()*

Assert that a given value is not *nil* . Lua coercion rules are applied so that values like `0`, `""`, `false` all validate the assertion. If provided, *extra_msg* is a string which will be printed along with the failure message.

assertIs (*actual*, *expected*[, *extra_msg*])
Alias: *assert_is()*

Assert that two variables are identical. For string, numbers, boolean and for nil, this gives the same result as *assertEquals()* . For the other types, identity means that the two variables refer to the same object. If provided, *extra_msg* is a string which will be printed along with the failure message.

Example :

```
s1='toto'
s2='to'..'to'
t1={1,2}
t2={1,2}
v1=nil
v2=false

lu.assertIs(s1,s1) -- ok
lu.assertIs(s1,s2) -- ok
lu.assertIs(t1,t1) -- ok
lu.assertIs(t1,t2) -- fail
lu.assertIs(v1,v2) -- fail
```

assertNotIs (*actual*, *expected*[, *extra_msg*])
Alias: *assert_not_is()*

Assert that two variables are not identical, in the sense that they do not refer to the same value. If provided, *extra_msg* is a string which will be printed along with the failure message.

See *assertIs()* for more details.

4.5.3 String assertions

Assertions related to string and patterns.

assertStrContains (*str*, *sub*[, *isPattern*[, *extra_msg*]])
Alias: *assert_str_contains()*

Assert that the string *str* contains the substring or pattern *sub*. If provided, *extra_msg* is a string which will be printed along with the failure message.

By default, substring is searched in the string. If *isPattern* is provided and is true, *sub* is treated as a pattern which is searched inside the string *str* .

assertStrIContains (*str*, *sub*[, *extra_msg*])
Alias: *assert_str_icontains()*

Assert that the string *str* contains the given substring *sub*, irrespective of the case. If provided, *extra_msg* is a string which will be printed along with the failure message.

Note that unlike `assertStrcontains()`, you can not search for a pattern.

assertNotStrContains (*str*, *sub*[, *isPattern*[, *extra_msg*]])
Alias: *assert_not_str_contains()*

Assert that the string *str* does not contain the substring or pattern *sub*. If provided, *extra_msg* is a string which will be printed along with the failure message.

By default, the substring is searched in the string. If *isPattern* is provided and is true, *sub* is treated as a pattern which is searched inside the string *str* .

assertNotStrIContains (*str*, *sub*[, *extra_msg*])
Alias: *assert_not_str_icontains()*

Assert that the string *str* does not contain the substring *sub*, irrespective of the case. If provided, *extra_msg* is a string which will be printed along with the failure message.

Note that unlike `assertNotStrcontains()`, you can not search for a pattern.

assertStrMatches (*str*, *pattern*[, *start*[, *final*[, *extra_msg*]]])
Alias: *assert_str_matches()*

Assert that the string *str* matches the full pattern *pattern*.

If *start* and *final* are not provided or are *nil*, the pattern must match the full string, from start to end. The function allows to specify the expected start and end position of the pattern in the string. If provided, *extra_msg* is a string which will be printed along with the failure message.

4.5.4 Error assertions

Error related assertions, to verify error generation and error messages.

assertError (*func*, *...*)
Alias: *assert_error()*

Assert that calling functions *func* with the arguments yields an error. If the function does not yield an error, the assertion fails.

Note that the error message itself is not checked, which means that this function does not distinguish between the legitimate error that you expect and another error that might be triggered by mistake.

The next functions provide a better approach to error testing, by checking explicitly the error message content.

Note: When testing LuaUnit, switching from *assertError()* to *assertErrorMsgEquals()* revealed quite a few bugs!

assertErrorMsgEquals (*expectedMsg*, *func*, *...*)
Alias: *assert_error_msg_equals()*

Assert that calling function *func* will generate exactly the given error message. If the function does not yield an error, or if the error message is not identical, the assertion fails.

Be careful when using this function that error messages usually contain the file name and line number information of where the error was generated. This is usually inconvenient. To ignore the filename and line number information, you can either use a pattern with `assertErrorMsgMatches()` or simply check for the message content with `assertErrorMsgContains()` .

assertErrorMsgContains (*partialMsg*, *func*, *...*)
Alias: *assert_error_msg_contains()*

Assert that calling function *func* will generate an error message containing *partialMsg* . If the function does not yield an error, or if the expected message is not contained in the error message, the assertion fails.

assertErrorMsgMatches (*expectedPattern*, *func*, *...*)
Alias: *assert_error_msg_matches()*

Assert that calling function *func* will generate an error message matching *expectedPattern* . If the function does not yield an error, or if the error message does not match the provided patternm the assertion fails.

Note that matching is done from the start to the end of the error message. Be sure to escape magic all magic characters with `%` (like `-+.?*`) .

4.5.5 Type assertions

The following functions all perform type checking on their argument. If the received value is not of the right type, the failure message will contain the expected type, the received type and the received value to help you identify better the problem.

assertIsNumber (*value*[, *extra_msg*])
Aliases: *assertNumber()*, *assert_is_number()*, *assert_number()*

Assert that the argument is a number (integer or float). If provided, *extra_msg* is a string which will be printed along with the failure message.

assertIsString (*value*[, *extra_msg*])
Aliases: *assertString()*, *assert_is_string()*, *assert_string()*

Assert that the argument is a string. If provided, *extra_msg* is a string which will be printed along with the failure message.

assertIsTable (*value*[, *extra_msg*])
Aliases: *assertTable()*, *assert_is_table()*, *assert_table()*

Assert that the argument is a table. If provided, *extra_msg* is a string which will be printed along with the failure message.

assertIsBoolean (*value*[, *extra_msg*])
Aliases: *assertBoolean()*, *assert_is_boolean()*, *assert_boolean()*

Assert that the argument is a boolean. If provided, *extra_msg* is a string which will be printed along with the failure message.

assertIsNil (*value*[, *extra_msg*])
Aliases: *assertNil()*, *assert_is_nil()*, *assert_nil()*

Assert that the argument is nil. If provided, *extra_msg* is a string which will be printed along with the failure message.

assertIsFunction (*value*[, *extra_msg*])
Aliases: *assertFunction()*, *assert_is_function()*, *assert_function()*

Assert that the argument is a function. If provided, *extra_msg* is a string which will be printed along with the failure message.

assertIsUserdata (*value*[, *extra_msg*])
Aliases: *assertUserdata()*, *assert_is_userdata()*, *assert_userdata()*

Assert that the argument is a userdata. If provided, *extra_msg* is a string which will be printed along with the failure message.

assertIsCoroutine (*value*[, *extra_msg*])
Aliases: *assertCoroutine()*, *assert_is_coroutine()*, *assert_coroutine()*

Assert that the argument is a coroutine (an object with type *thread*). If provided, *extra_msg* is a string which will be printed along with the failure message.

assertIsThread (*value*[, *extra_msg*])
Aliases: *assertIsThread()*, *assertThread()*, *assert_is_thread()*, *assert_thread()*

Same function as :func:assertIsCoroutine . Since Lua coroutines have the type thread, it's not clear which name is the clearer, so we provide syntax for both names. If provided, *extra_msg* is a string which will be printed along with the failure message.

4.5.6 Table assertions

assertItemsEquals (*actual*, *expected*[, *extra_msg*])
Alias: *assert_items_equals()*

Assert that two tables contain the same items, irrespective of their keys. If provided, *extra_msg* is a string which will be printed along with the failure message.

This function is practical for example if you want to compare two lists but where items are not in the same order:

```
lu.assertItemsEquals( {1,2,3}, {3,2,1} ) -- assertion succeeds
```

The comparison is not recursive on the items: if any of the items are tables, they are compared using table equality (like as in *assertEquals()*), where the key matters.

```
lu.assertItemsEquals( {1,{2,3},4}, {4,{3,2,},1} ) -- assertion fails because {2,3} ~=
↪{3,2}
```

4.5.7 Ending test

LuaUnit allows to force test ending, either positevely or negatively, with the following functions.

fail (*message*)
Stops the ongoing test and mark it as failed with the given message.

failIf (*cond*, *message*)
If the condition *cond* evaluates to *true*, stops the ongoing test and mark it as failed with the given message. Else, continue the test execution normally.

success ()
Stops the ongoing test and mark it as successful.

successIf (*cond*)
If the condition *cond* evaluates to *true*, stops the ongoing test and mark it as successful. Else, continue the test execution normally.

4.5.8 Scientific computing and LuaUnit

LuaUnit is used by the CERN for the MAD-NG program, the forefront of computational physics in the field of particle accelerator design and simulation (See MAD). Thank to the feedback of a scientific computing developer, LuaUnit has been enhanced with some facilities for scientific applications (see all assertions functions below).

The floating point library used by Lua is the one provided by the C compiler which built Lua. It is usually compliant with IEEE-754 . As such, it can yields results such as *plus infinity*, *minus infinity* or *Not a Number* (NaN). The precision of any calculation performed in Lua is related to the smallest representable floating point value (typically called *EPS*): 2^-52 for 64 bits floats (type double in the C language) and 2^-23 for 32 bits float (type float in C).

Note: Lua may be compiled with numbers represented either as 32 bits floats or 64 bits double (as defined by the macro LUA_FLOAT_TYPE in luaconf.h). LuaUnit has been validated in both these configurations and in particuluar, the epsilon value *EPS* is adjusted accordingly.

For more information about performing calculations on computers, please read the reference paper What Every Computer Scientist Should Know About Floating-Point Arithmetic

If your calculation shall be portable to multiple OS or compilers, you may get different calculation errors depending on the OS/compiler. It is therefore important to verify them on every target.

Note: If you need to deal with value *minus zero*, be very careful because Lua versions are inconsistent on how they treat the syntax *-0* : it creates either a *plus zero* or a *minus zero* . Multiplying or dividing *0* by *-1* also yields inconsistent results. The reliable way to create the *-0* value is : minusZero = -1 / (1/0)

EPS *constant*

The machine epsilon, to be used with `assertAlmostEquals()` .

This is either:

- 2^-52 or ~2.22E-16 (with lua number defined as double)
- 2^-23 or ~1.19E-07 (with lua number defined as float)

assertNan (*value*[, *extra_msg*])
Alias: *assert_nan()*

Assert that a given number is a *NaN* (Not a Number), according to the definition of IEEE-754 . If provided, *extra_msg* is a string which will be printed along with the failure message.

assertNotNan (*value*[, *extra_msg*])
Alias: *assert_not_nan()*

Assert that a given number is NOT a *NaN* (Not a Number), according to the definition of IEEE-754 . If provided, *extra_msg* is a string which will be printed along with the failure message.

assertPlusInf (*value*[, *extra_msg*])
Alias: *assert_plus_inf()*

Assert that a given number is *plus infinity*, according to the definition of IEEE-754 . If provided, *extra_msg* is a string which will be printed along with the failure message.

assertMinusInf (*value*[, *extra_msg*])
Alias: *assert_minus_inf()*

Assert that a given number is *minus infinity*, according to the definition of IEEE-754 . If provided, *extra_msg* is a string which will be printed along with the failure message.

assertInf (*value*[, *extra_msg*])
Alias: *assert_inf()*

Assert that a given number is *infinity* (either positive or negative), according to the definition of IEEE-754 . If provided, *extra_msg* is a string which will be printed along with the failure message.

assertNotPlusInf (*value*[, *extra_msg*])
Alias: *assert_not_plus_inf()*

Assert that a given number is NOT *plus infinity*, according to the definition of IEEE-754 . If provided, *extra_msg* is a string which will be printed along with the failure message.

assertNotMinusInf (*value*[, *extra_msg*])
Alias: *assert_not_minus_inf()*

Assert that a given number is NOT *minus infinity*, according to the definition of IEEE-754 . If provided, *extra_msg* is a string which will be printed along with the failure message.

assertNotInf (*value*[, *extra_msg*])
Alias: *assert_not_inf()*

Assert that a given number is neither *infinity* nor *minus infinity*, according to the definition of IEEE-754 . If provided, *extra_msg* is a string which will be printed along with the failure message.

assertPlusZero (*value*[, *extra_msg*])
Alias: *assert_plus_zero()*

Assert that a given number is *+0*, according to the definition of IEEE-754 . The verification is done by dividing by the provided number and verifying that it yields *infinity* . If provided, *extra_msg* is a string which will be printed along with the failure message.

Be careful when dealing with *+0* and *-0*, see note above.

assertMinusZero (*value*[, *extra_msg*])
Alias: *assert_minus_zero()*

Assert that a given number is *-0*, according to the definition of IEEE-754 . The verification is done by dividing by the provided number and verifying that it yields *minus infinity* . If provided, *extra_msg* is a string which will be printed along with the failure message.

Be careful when dealing with *+0* and *-0*, see MinusZero

assertNotPlusZero (*value*[, *extra_msg*])
Alias: *assert_not_plus_zero()*

Assert that a given number is NOT *+0*, according to the definition of IEEE-754 . If provided, *extra_msg* is a string which will be printed along with the failure message.

Be careful when dealing with *+0* and *-0*, see MinusZero

assertNotMinusZero (*value*[, *extra_msg*])
Alias: *assert_not_minus_zero()*

Assert that a given number is NOT *-0*, according to the definition of IEEE-754 . If provided, *extra_msg* is a string which will be printed along with the failure message.

Be careful when dealing with *+0* and *-0*, see MinusZero

assertAlmostEquals (*actual*, *expected*[, *margin=EPS*[, *extra_msg*]])
Alias: *assert_almost_equals()*

Assert that two floating point numbers are equal by the defined margin. If margin is not provided, the machine epsilon *EPS* is used. If provided, *extra_msg* is a string which will be printed along with the failure message.

Be careful that depending on the calculation, it might make more sense to measure the absolute error or the relative error (see below):

assertNotAlmostEquals (*actual*, *expected*[, *margin=EPS*[, *extra_msg*]])
Alias: *assert_not_almost_equals()*

Assert that two floating point numbers are not equal by the defined margin. If margin is not provided, the machine epsilon *EPS* is used. If provided, *extra_msg* is a string which will be printed along with the failure message.

Be careful that depending on the calculation, it might make more sense to measure the absolute error or the relative error (see below).

Example of absolute versus relative error

```
-- convert pi/6 radian to 30 degree
pi_div_6_deg_calculated = math.deg(math.pi/6)
pi_div_6_deg_expected = 30

-- convert pi/3 radian to 60 degree
pi_div_3_deg_calculated = math.deg(math.pi/3)
pi_div_3_deg_expected = 60

-- check absolute error: it is not constant
print( (pi_div_6_deg_expected - pi_div_6_deg_calculated) / lu.EPS ) -- prints: 16
print( (pi_div_3_deg_expected - pi_div_3_deg_calculated) / lu.EPS ) -- prints: 32

-- Better use relative error:
print( ( (pi_div_6_deg_expected - pi_div_6_deg_calculated) / pi_div_6_deg_expected) /␣
↪lu.EPS ) -- prints: 0.53333
print( ( (pi_div_3_deg_expected - pi_div_3_deg_calculated) / pi_div_3_deg_expected) /␣
↪lu.EPS ) -- prints: 0.53333
```

(continues on next page)

(continued from previous page)

```
-- relative error is constant. Assertion can take the form of:
assertAlmostEquals( (pi_div_6_deg_expected - pi_div_6_deg_calculated) / pi_div_6_deg_
↪expected, lu.EPS )
assertAlmostEquals( (pi_div_3_deg_expected - pi_div_3_deg_calculated) / pi_div_3_deg_
↪expected, lu.EPS )
```

4.5.9 Pretty printing

prettystr(*value*)

Converts *value* to a nicely formatted string, whatever the type of the value. It supports in particular tables, nested table and even recursive tables.

You can use it in your code to replace calls to *tostring()* .

Example of prettystr()

```
> lu = require('luaunit')
> t1 = {1,2,3}
> t1['toto'] = 'titi'
> t1.f = function () end
> t1.fa = (1 == 0)
> t1.tr = (1 == 1)
> print( lu.prettystr(t1) )
{1, 2, 3, f=function: 00635d68, fa=false, toto="titi", tr=true}
```

CHAPTER 5

Developing LuaUnit

5.1 Development ecosystem

LuaUnit is developed on GitHub.

Bugs or feature requests should be reported using GitHub issues.

LuaUnit is released under the BSD license.

This documentation is available at Read-the-docs.

5.2 Contributing

You may contribute to LuaUnit by reporting bugs or wishes, or by contributing code directly with a pull request.

Some issues on GitHub are marked with label *enhancement*. Feel also free to pick up such tasks and implement them.

Changes should be proposed as *Pull Requests* on GitHub.

Thank to our continuous integration setup with Travis-Ci and AppVeyor, all unit-tests and functional tests are run on Linux, Windows and MacOs, with all versions of Lua. So any *Pull Request* will show immediately if anything is going unexpected.

5.2.1 Unit-tests

All proposed changes should pass all unit-tests and if needed, add more unit-tests to cover the bug or the new functionality. Usage is pretty simple:

```
$ lua run_unit_tests.lua
.....................................................................................
.............................
Ran 111 tests in 0.120 seconds
OK
```

5.2.2 Functional tests

Functional tests also exist to validate LuaUnit. Their management is slightly more complicated.

The main goal of functional tests is to validate that LuaUnit output has not been altered. Since LuaUnit supports some standard compliant output (TAP, junitxml), this is very important (and it has been broken in the past)

Functional tests perform the following actions:

- Run the 2 suites: example_with_luaunit.lua, test_with_err_fail_pass.lua (with various options to have successe, failure and/or errors)
- Run every suite with all output format, all verbosity
- Validate the XML output with jenkins/hudson and junit schema
- Compare the results with the previous output (archived in test/ref), with some tricks to make the comparison possible :
  - adjustment of the file separator to use the same output on Windows and Unix
  - date and test duration is zeroed so that it does not impact the comparison
  - adjust the stack trace format which has changed between Lua 5.1, 5.2 and 5.3
- Run some legacy suites or tricky output to manage and verify output: test_with_xml.lua, , compat_luaunit_v2x.lua, legacy_example_with_luaunit.lua

For functional tests to run, *diff* must be available on the command line. *xmllint* is needed to perform the xml validation but this step is skipped if *xmllint* can not be found.

When functional tests work well, it looks like this:

```
$ lua run_functional_tests.lua
..............
Ran 15 tests in 9.676 seconds
OK
```

When functional test fail, a diff of the comparison between the reference output and the current output is displayed (it can be quite long). The list of faulty files is summed-up at the end.

5.2.3 Modifying reference files for functional tests

The script run_functional_tests.lua supports a –update option, with an optional argument.

- *–update* without argument **overwrites all reference output** with the current output. Use only if you know what you are doing, this is usually a very bad idea!
- The following argument overwrite a specific subset of reference files, select the one that fits your change:
  - TestXml: XML output of test_with_xml
  - ExampleXml: XML output of example_with_luaunit
  - ExampleTap: TAP output of example_with_luaunit
  - ExampleText: text output of example_with_luaunit
  - ExampleNil: nil output of example_with_luaunit
  - ErrFailPassText: text output of test_with_err_fail_pass
  - ErrFailPassTap: TAP output of test_with_err_fail_pass
  - ErrFailPassXml: XML output of test_with_err_fail_pass

  - StopOnError: errFailPassTextStopOnError-1.txt, -2.txt, -3.txt, -4.txt

For example to record a change in the test_with_err_fail_pass output

```
$ lua run_functional_tests.lua --update ErrFailPassXml ErrFailPassTap ErrFailPassText

>>>>>>> Generating test/ref/errFailPassXmlDefault.txt
>>>>>>> Generating test/ref/errFailPassXmlDefault-success.txt
>>>>>>> Generating test/ref/errFailPassXmlDefault-failures.txt
>>>>>>> Generating test/ref/errFailPassXmlQuiet.txt
>>>>>>> Generating test/ref/errFailPassXmlQuiet-success.txt
>>>>>>> Generating test/ref/errFailPassXmlQuiet-failures.txt
>>>>>>> Generating test/ref/errFailPassXmlVerbose.txt
>>>>>>> Generating test/ref/errFailPassXmlVerbose-success.txt
>>>>>>> Generating test/ref/errFailPassXmlVerbose-failures.txt
>>>>>>> Generating test/ref/errFailPassTapDefault.txt
>>>>>>> Generating test/ref/errFailPassTapDefault-success.txt
>>>>>>> Generating test/ref/errFailPassTapDefault-failures.txt
>>>>>>> Generating test/ref/errFailPassTapQuiet.txt
>>>>>>> Generating test/ref/errFailPassTapQuiet-success.txt
>>>>>>> Generating test/ref/errFailPassTapQuiet-failures.txt
>>>>>>> Generating test/ref/errFailPassTapVerbose.txt
>>>>>>> Generating test/ref/errFailPassTapVerbose-success.txt
>>>>>>> Generating test/ref/errFailPassTapVerbose-failures.txt
>>>>>>> Generating test/ref/errFailPassTextDefault.txt
>>>>>>> Generating test/ref/errFailPassTextDefault-success.txt
>>>>>>> Generating test/ref/errFailPassTextDefault-failures.txt
>>>>>>> Generating test/ref/errFailPassTextQuiet.txt
>>>>>>> Generating test/ref/errFailPassTextQuiet-success.txt
>>>>>>> Generating test/ref/errFailPassTextQuiet-failures.txt
>>>>>>> Generating test/ref/errFailPassTextVerbose.txt
>>>>>>> Generating test/ref/errFailPassTextVerbose-success.txt
>>>>>>> Generating test/ref/errFailPassTextVerbose-failures.txt
$
```

You can then commit the new files into git.

Note: how to commit updated reference outputs

When committing those changes into git, please use if possible an intelligent git committing tool to commit only the interesting changes. With SourceTree for example, in case of XML changes, I can select only the lines relevant to the change and avoid committing all the updates to test duration and test datestamp.

5.2.4 Typical failures for functional tests

Functional tests may start failing when:

1. Increasing LuaUnit version
2. Improving or breaking LuaUnit output

This a good place to start looking if you see failures occurring.

CHAPTER 6

Annexes

6.1 Annex A: More on table printing

When asserting tables equality, by default, the table content is printed in case of failures. LuaUnit tries to print tables in a readable format. It is possible to always display the table id along with the content, by setting a module parameter PRINT_TABLE_REF_IN_ERROR_MSG . This helps identifying tables:

```
local lu = require('luaunit')

local t1 = {1,2,3}
-- normally, t1 is dispalyed as: "{1,2,3}"

-- if setting this:
lu.PRINT_TABLE_REF_IN_ERROR_MSG = true

-- display of table t1 becomes: "<table: 0x29ab56> {1,2,3}"
```

Note: table loops

When displaying table content, it is possible to encounter loops, if for example two table references eachother. In such cases, LuaUnit display the full table content once, along with the table id, and displays only the table id for the looping reference.

Example: displaying a table with reference loop

```
local t1 = {}
local t2 = {}
t1.t2 = t2
t1.a = {1,2,3}
t2.t1 = t1

-- when displaying table t1:
```

(continues on next page)

(continued from previous page)

```
--   table t1 inside t2 is only displayed by its id because t1 is already being␣
↪displayed
--   table t2 is displayed along with its id because it is part of a loop.
-- t1: "<table: 0x29ab56> { a={1,2,3}, t2=<table: 0x27ab23> {t1=<table: 0x29ab56>} }"
```

6.2 Annex B: Comparing tables with keys of type table

If provided, *extra_msg* is a string which will be printed along with the failure message.

This is a very uncommon scenario but there are a few programs out there which use tables as keys for other tables. LuaUnit has been adjusted to deal intelligently with this scenario.

A small code block is worth a thousand pictures :

```
local lu = require('luaunit')

-- let's define two tables
t1 = { 1, 2 }
t2 = { 1, 2 }
lu.assertEquals( t1, t2 ) -- succeeds

-- let's define three tables, with the two above tables as keys
t3 = { t1='a' }
t4 = { t2='a' }
t5 = { t2='a' }
```

There are two ways to treat comparison of tables t3 and t4:

Method 1: table keys are compared by content

- t3 contain one key: t1
- t4 contain one key: t2, which has exactly the same content as t1
- the two keys compare equally with assertEquals, so assertEquals(t3, t4) succeeds

Method 2: table keys are compared by reference

- t3 contain one key: t1
- t4 contain one key: t2, which is not the same table as t1, its reference is different
- the two keys are different because t1 is a different object than t2 so assertEquals(t3, t4) fails

Whether method 1 or method 2 is more appropriate is up for debate.

LuaUnit has been adjusted to support both scenarios, with the config variable: *TABLE_EQUALS_KEYBYCONTENT*

- TABLE_EQUALS_KEYBYCONTENT = true (default): method 1 - table keys compared by content
- TABLE_EQUALS_KEYBYCONTENT = false: method 2 - table keys compared by reference

In any case, assertEquals(t4, t5) always succeeds.

To adjust the config, change it into the luaunit table before running any tests:

```
local lu = require('luaunit')

-- define all your tests
-- ...
```

(continues on next page)

(continued from previous page)

```
lu.TABLE_EQUALS_KEYBYCONTENT = false
-- run your tests:
os.exit( lu.LuaUnit.run() )
```

6.3 Annex C: Source code of example

Source code of the example used in the *Getting started with LuaUnit*

```
--
-- The examples described in the documentation are below.
--

lu = require('luaunit')

function add(v1,v2)
    -- add positive numbers
    -- return 0 if any of the numbers are 0
    -- error if any of the two numbers are negative
    if v1 < 0 or v2 < 0 then
        error('Can only add positive or null numbers, received '..v1..' and '..v2)
    end
    if v1 == 0 or v2 == 0 then
        return 0
    end
    return v1+v2
end

function adder(v)
    -- return a function that adds v to its argument using add
    function closure( x ) return x+v end
    return closure
end

function div(v1,v2)
    -- divide positive numbers
    -- return 0 if any of the numbers are 0
    -- error if any of the two numbers are negative
    if v1 < 0 or v2 < 0 then
        error('Can only divide positive or null numbers, received '..v1..' and '..v2)
    end
    if v1 == 0 or v2 == 0 then
        return 0
    end
    return v1/v2
end



TestAdd = {}
    function TestAdd:testAddPositive()
        lu.assertEquals(add(1,1),2)
    end
```

(continues on next page)

(continued from previous page)

```
    function TestAdd:testAddZero()
        lu.assertEquals(add(1,0),0)
        lu.assertEquals(add(0,5),0)
        lu.assertEquals(add(0,0),0)
    end

    function TestAdd:testAddError()
        lu.assertErrorMsgContains('Can only add positive or null numbers, received 2␣
↪and -3', add, 2, -3)
    end

    function TestAdd:testAdder()
        f = adder(3)
        lu.assertIsFunction( f )
        lu.assertEquals( f(2), 5 )
    end
-- end of table TestAdd

TestDiv = {}
    function TestDiv:testDivPositive()
        lu.assertEquals(div(4,2),2)
    end

    function TestDiv:testDivZero()
        lu.assertEquals(div(4,0),0)
        lu.assertEquals(div(0,5),0)
        lu.assertEquals(div(0,0),0)
    end

    function TestDiv:testDivError()
        lu.assertErrorMsgContains('Can only divide positive or null numbers, received␣
↪2 and -3', div, 2, -3)
    end
-- end of table TestDiv

--[[
--
--      Uncomment this section to see how failures are displayed
--
TestWithFailures = {}
    -- two failing tests

    function TestWithFailures:testFail1()
        lu.assertEquals( "toto", "titi")
    end

    function TestWithFailures:testFail2()
        local a=1
        local b='toto'
        local c = a + b -- oops, can not add string and numbers
        return c
    end
-- end of table TestWithFailures
]]


--[[
```

(continues on next page)

(continued from previous page)

```
TestLogger = {}
    function TestLogger:setUp()
        -- define the fname to use for logging
        self.fname = 'mytmplog.log'
        -- make sure the file does not already exists
        os.remove(self.fname)
    end

    function TestLogger:testLoggerCreatesFile()
        initLog(self.fname)
        log('toto')
        f = io.open(self.fname, 'r')
        lu.assertNotNil( f )
        f:close()
    end

    function TestLogger:tearDown()
        self.fname = 'mytmplog.log'
        -- cleanup our log file after all tests
        os.remove(self.fname)
    end
-- end of table TestLogger

]]

os.exit(lu.LuaUnit.run())
```

6.4 Annex D: BSD License

CHAPTER 7

Index and Search page

- genindex
- search

Index