Download Lua (PDF)

Getting started with Lua

Remarks

Lua is minimalistic, lightweight and embeddable scripting language. It's being designed, implemented, and maintained by a team at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil. The mailing list is open to get involved.

Common use-cases for Lua includes scripting video games, extending applications with plugins and configs, wrapping some high-level business logic or just embedding into devices like TVs, cars, etc.

For high performance tasks there is independent implementation using just-in-time-compiler available called LuaJIT.

Versions

Version	Notes	Release Date
1.0	Initial, non-public release.	1993-07-28
1.1	First public release. Conference paper describing it.	1994-07-08
2.1	Starting with Lua 2.1, Lua became freely available for all purposes, including commercial uses. Journal paper describing it.	1995-02-07
2.2	Long strings, the debug interface, better stack tracebacks	1995-11-28
2.4	External `luac` compiler	1996-05-14
2.5	Pattern matching and vararg functions.	1996-11-19
3.0	Introduced auxlib, a library for helping writing Lua libraries	1997-07-01
3.1	Anonymous functions and function closures via "upvalues".	1998-07-11
3.2	Debug library and new table functions	1999-07-08

Version	Notes	Release Date
3.2.2		2000-02-22
4.0	Multiple states, "for" statements, API revamp.	2000-11-06
4.0.1		2002-07-04
5.0	Coroutines, metatables, full lexical scoping, tail calls, booleans move to MIT license.	2003-04-11
5.0.3		2006-06-26
5.1	Module system revamp, incremental garbage collector, metatables for all types, luaconf.h revamp, fully reentrant parser, variadic arguments.	2006-02-21
5.1.5		2012-02-17
5.2	Emergency garbage collector, goto, finalizers for tables.	2011-12-16
5.2.4		2015-03-07
5.3	Basic UTF-8 support, bitwise ops, 32/64bit integers.	2015-01-12
5.3.4	Latest version.	2017-01-12

Comments

Single-line comments in Lua start with `--` and continue until the end of line:

```
-- this is single line comment
-- need another line
-- huh?
```

Block comments start with `--[[` and end with `]]` :

```
--[[
    This is block comment.
    So, it can go on...
    and on...
    and on....
]]
```

Block comments use the same style of delimiters as long strings; any number of equal signs can be added between the brackets to delimit a comment:

```
--[=[
    This is also a block comment
    We can include "]]" inside this comment
--]=]

--[==[
    This is also a block comment
    We can include "]=]" inside this comment
--]==]
```

A neat trick to comment out chunks of code is to surround it with `--[[` and `--]]` :

```
--[[
    print'Lua is lovely'
--]]
```

To reactivate the chunk, simply append a `-` to the comment opening sequence:

```
---[[
    print'Lua is lovely'
--]]
```

This way, the sequence `--` in the first line starts a single-line comment, just like the last line, and the `print` statement is not commented out.

Taking this a step further, two blocks of code can be setup in such a way that if the first block is commented out the second won't be, and visa versa:

```
---[[
  print 'Lua is love'
--[=[]]
  print 'Lua is life'
--]=]
```

To active the second chunk while disabling the first chunk, delete the leading `-` on the first line:

```
--[[
  print 'Lua is love'
--[=[]]
  print 'Lua is life'
--]=]
```

Executing Lua programs

Usually, Lua is being shipped with two binaries:

- `lua` - standalone interpreter and interactive shell
- `luac` - bytecode compiler

Lets say we have an example program (`bottles_of_mate.lua`) like this:

```
local string = require "string"

function bottle_take(bottles_available)

    local count_str = "%d bottles of mate on the wall."
    local take_str = "Take one down, pass it around, " .. count_str
    local end_str = "Oh noes, " .. count_str
    local buy_str = "Get some from the store, " .. count_str
    local bottles_left = 0

    if bottles_available > 0 then
         print(string.format(count_str, bottles_available))
         bottles_left = bottles_available - 1
         print(string.format(take_str, bottles_left))
    else
        print(string.format(end_str, bottles_available))
        bottles_left = 99
        print(string.format(buy_str, bottles_left))
    end

    return bottles_left
end

local bottle_count = 99

while true do
    bottle_count = bottle_take(bottle_count)
end
```

The program itself can be ran by executing following on Your shell:

```
$ lua bottles_of_mate.lua
```

Output should look like this, running in the endless loop:

```
Get some from the store, 99 bottles of mate on the wall.
99 bottles of mate on the wall.
Take one down, pass it around, 98 bottles of mate on the wall.
98 bottles of mate on the wall.
Take one down, pass it around, 97 bottles of mate on the wall.
97 bottles of mate on the wall.
...
...
3 bottles of mate on the wall.
Take one down, pass it around, 2 bottles of mate on the wall.
2 bottles of mate on the wall.
Take one down, pass it around, 1 bottles of mate on the wall.
1 bottles of mate on the wall.
Take one down, pass it around, 0 bottles of mate on the wall.
Oh noes, 0 bottles of mate on the wall.
Get some from the store, 99 bottles of mate on the wall.
99 bottles of mate on the wall.
Take one down, pass it around, 98 bottles of mate on the wall.
...
```

You can compile the program into Lua's bytecode by executing following on Your shell:

```
$ luac bottles_of_mate.lua -o bottles_of_mate.luac
```

Also bytecode listing is available by executing following:

```
$ luac -l bottles_of_mate.lua


main <./bottles.lua:0,0> (13 instructions, 52 bytes at 0x101d530)
0+ params, 4 slots, 0 upvalues, 2 locals, 4 constants, 1 function
    1    [1]    GETGLOBAL    0 -1    ; require
    2    [1]    LOADK        1 -2    ; "string"
    3    [1]    CALL         0 2 2
    4    [22]    CLOSURE      1 0    ; 0x101d710
    5    [22]    MOVE         0 0
    6    [3]    SETGLOBAL    1 -3    ; bottle_take
    7    [24]    LOADK        1 -4    ; 99
    8    [27]    GETGLOBAL    2 -3    ; bottle_take
    9    [27]    MOVE         3 1
    10    [27]    CALL         2 2 2
    11    [27]    MOVE         1 2
    12    [27]    JMP          -5    ; to 8
    13    [28]    RETURN       0 1

function <./bottles.lua:3,22> (46 instructions, 184 bytes at 0x101d710)
1 param, 10 slots, 1 upvalue, 6 locals, 9 constants, 0 functions
    1    [5]    LOADK        1 -1    ; "%d bottles of mate on the wall."
    2    [6]    LOADK        2 -2    ; "Take one down, pass it around, "
    3    [6]    MOVE         3 1
    4    [6]    CONCAT       2 2 3
    5    [7]    LOADK        3 -3    ; "Oh noes, "
    6    [7]    MOVE         4 1
    7    [7]    CONCAT       3 3 4
    8    [8]    LOADK        4 -4    ; "Get some from the store, "
    9    [8]    MOVE         5 1
    10    [8]    CONCAT       4 4 5
    11    [9]    LOADK        5 -5    ; 0
    12    [11]    EQ           1 0 -5    ; - 0
    13    [11]    JMP          16    ; to 30
    14    [12]    GETGLOBAL    6 -6    ; print
    15    [12]    GETUPVAL     7 0    ; string
    16    [12]    GETTABLE     7 7 -7    ; "format"
    17    [12]    MOVE         8 1
    18    [12]    MOVE         9 0
    19    [12]    CALL         7 3 0
    20    [12]    CALL         6 0 1
    21    [13]    SUB          5 0 -8    ; - 1
    22    [14]    GETGLOBAL    6 -6    ; print
    23    [14]    GETUPVAL     7 0    ; string
    24    [14]    GETTABLE     7 7 -7    ; "format"
    25    [14]    MOVE         8 2
    26    [14]    MOVE         9 5
    27    [14]    CALL         7 3 0
    28    [14]    CALL         6 0 1
    29    [14]    JMP          15    ; to 45
    30    [16]    GETGLOBAL    6 -6    ; print
    31    [16]    GETUPVAL     7 0    ; string
    32    [16]    GETTABLE     7 7 -7    ; "format"
    33    [16]    MOVE         8 3
    34    [16]    MOVE         9 0
    35    [16]    CALL         7 3 0
    36    [16]    CALL         6 0 1
    37    [17]    LOADK        5 -9    ; 99
    38    [18]    GETGLOBAL    6 -6    ; print
    39    [18]    GETUPVAL     7 0    ; string
    40    [18]    GETTABLE     7 7 -7    ; "format"
    41    [18]    MOVE         8 4
    42    [18]    MOVE         9 5
    43    [18]    CALL         7 3 0
    44    [18]    CALL         6 0 1
    45    [21]    RETURN       5 2
    46    [22]    RETURN       0 1
```

Getting Started

variables

```
var = 50 -- a global variable
print(var) --> 50
do
  local var = 100 -- a local variable
  print(var) --> 100
end
print(var) --> 50
-- The global var (50) still exists
-- The local var (100) has gone out of scope and can't be accessed any longer.
```

types

```
num = 20 -- a number
num = 20.001 -- still a number
str = "zaldrizes buzdari iksos daor" -- a string
tab = {1, 2, 3} -- a table (these have their own category)
bool = true -- a boolean value
bool = false -- the only other boolean value
print(type(num)) --> 'number'
print(type(str)) --> 'string'
print(type(bool)) --> 'boolean'
type(type(num)) --> 'string'

-- Functions are a type too, and first-class values in Lua.
print(type(print)) --> prints 'function'
old_print = print
print = function (x) old_print "I'm ignoring the param you passed me!" end
old_print(type(print)) --> Still prints 'function' since it's still a function.
-- But we've (unhelpfully) redefined the behavior of print.
print("Hello, world!") --> prints "I'm ignoring the param you passed me!"
```

The special type `nil`

Another type in Lua is `nil` . The only value in the `nil` type is `nil` . `nil` exists to be different from all other values in Lua. It is a kind of non-value value.

```
print(foo) -- This prints nil since there's nothing stored in the variable 'foo'.
foo = 20
print(foo) -- Now this prints 20 since we've assigned 'foo' a value of 20.

-- We can also use `nil` to undefine a variable
foo = nil -- Here we set 'foo' to nil so that it can be garbage-collected.

if nil then print "nil" end --> (prints nothing)
-- Only false and nil are considered false; every other value is true.
if 0 then print "0" end --> 0
if "" then print "Empty string!" --> Empty string!
```

expressions

```
a = 3
b = a + 20 a = 2 print(b, a) -- hard to read, can also be written as
b = a + 20; a = 2; print(a, b) -- easier to read, ; are optional though
true and true --> returns true
true and 20 --> 20
false and 20 --> false
false or 20 --> 20
true or 20 --> true
tab or {}
  --> returns tab if it is defined
  --> returns {} if tab is undefined
  -- This is useful when we don't know if a variable exists
tab = tab or {} -- tab stays unchanged if it exists; tab becomes {} if it was previously nil.

a, b = 20, 30 -- this also works
a, b = b, a -- switches values
```

Defining functions

```
function name(parameter)
    return parameter
end
print(name(20)) --> 20
-- see function category for more information
name = function(parameter) return parameter end -- Same as above
```

booleans

Only `false` and `nil` evaluate as false, everything else, including `0` and the empty string evaluate as true.

garbage-collection

```
tab = {"lots", "of", "data"}
tab = nil; collectgarbage()
-- tab does no longer exist, and doesn't take up memory anymore.
```

tables

```
tab1 = {"a", "b", "c"}
tab2 = tab1
tab2[1] = "d"
print(tab1[1]) --> 'd' -- table values only store references.
--> assigning tables does not copy its content, only the reference.

tab2 = nil; collectgarbage()
print(tab1) --> (prints table address) -- tab1 still exists; it didn't get garbage-collected.

tab1 = nil; collectgarbage()
-- No more references. Now it should actually be gone from memory.
```

These are the basics, but there's a section about tables with more information.

conditions

```
if (condition) then
    -- do something
elseif (other_condition) then
    -- do something else
else
    -- do something
end
```

for loops

There are two types of `for` loop in Lua: a numeric `for` loop and a generic `for` loop.

- A numeric `for` loop has the following form:

  ```
  for a=1, 10, 2 do -- for a starting at 1, ending at 10, in steps of 2
    print(a) --> 1, 3, 5, 7, 9
  end
  ```

  The third expression in a numeric `for` loop is the step by which the loop will increment. This makes it easy to do reverse loops:

  ```
  for a=10, 1, -1 do
    print(a) --> 10, 9, 8, 7, 6, etc.
  end
  ```

  If the step expression is left out, Lua assumes a default step of 1.

  ```
  for a=1, 10 do
    print(a) --> 1, 2, 3, 4, 5, etc.
  end
  ```

  Also note that the loop variable is local to the `for` loop. It will not exist after the loop is over.

- Generic `for` loops work through all values that an iterator function returns:

  ```
  for key, value in pairs({"some", "table"}) do
    print(key, value)
    --> 1 some
    --> 2 table
  end
  ```

  Lua provides several built in iterators (e.g., `pairs` , `ipairs`), and users can define their own custom iterators as well to use with generic `for` loops.

do blocks

```
local a = 10
do
    print(a) --> 10
    local a = 20
    print(a) --> 20
end
print(a) --> 10
```

Hello World

This is hello world code:

```
print("Hello World!")
```

How it works? It's simple! Lua executes `print()` function and uses `"Hello World"` string as argument.

Installation

Binaries

Lua binaries are provided by most GNU/Linux distributions as a package.

For example, on Debian, Ubuntu, and their derivatives it can be acquired by executing this:

```
sudo apt-get install lua50
```

```
sudo apt-get install lua51
```

```
sudo apt-get install lua52
```

There are some semi-official builds provided for Windows, MacOS and some other operating systems hosted at SourceForge.

Apple users can also install Lua easily using Homebrew:

```
brew install lua
```

(Currently Homebrew has 5.2.4, for 5.3 see Homebrew/versions.)

Source

Source is available at the official page. Acquisition of sources and build itself should be trivial. On Linux systems the following should be sufficient:

```
$ wget http://lua.org/ftp/lua-5.3.3.tar.gz
$ echo "a0341bc3d1415b814cc738b2ec01ae56045d64ef ./lua-5.3.3.tar.gz" | sha1sum -c -
$ tar -xvf ./lua-5.3.3.tar.gz
$ make -C ./lua-5.3.3/ linux
```

In the example above we're basically downloading a source `tarball` from the official site, verifying its checksum, and extracting and executing `make` . (Double check the checksum at the official page.)

Note: you must specify what build target you want. In the example, we specified `linux` . Other available build targets include `solaris` , `aix` , `bsd` , `freebsd` , `macosx` , `mingw` , etc. Check out `doc/readme.html` , which is included in the source, for more details. (You can also find the latest version of the README online.)

Modules

Standard libraries are limited to primitives:

- `coroutine` - coroutine management functionality
- `debug` - debug hooks and tools
- `io` - basic IO primitives
- `package` - module management functionality
- `string` - string and Lua specific pattern matching functionality
- `table` - primitives for dealing with an essential but complex Lua type - tables
- `os` - basic OS operations

- `utf8` - basic UTF-8 primitives (since Lua 5.3)

All of those libraries can be disabled for a specific build or loaded at run-time.

Third-party Lua libraries and infrastructure for distributing modules is sparse, but improving. Projects like LuaRocks, Lua Toolbox, and LuaDist are improving the situation. A lot of information and many suggestions can be found on the older Lua Wiki, but be aware that some of this information is quite old and out of date.

Some tricky things

Sometimes Lua doesn't behave the way one would think after reading the documentation. Some of these cases are:

Nil and Nothing aren't the same (COMMON PITFALL!)

As expected, `table.insert(my_table, 20)` adds the value `20` to the table, and `table.insert(my_table, 5, 20)` adds the value 20 at the 5th position. What does `table.insert(my_table, 5, nil)` do though? One might expect it to treat `nil` as no argument at all, and insert the value `5` at the end of the table, but it actually adds the value `nil` at the 5th position of the table. When is this a problem?

```
(function(tab, value, position)
    table.insert(tab, position or value, position and value)
end)({}, 20)
-- This ends up calling table.insert({}, 20, nil)
-- and this doesn't do what it should (insert 20 at the end)
```

A similar thing happens with `tostring()` :

```
print (tostring(nil)) -- this prints "nil"
table.insert({}, 20) -- this returns nothing
-- (not nil, but actually nothing (yes, I know, in lua those two SHOULD
-- be the same thing, but they aren't))

-- wrong:
print (tostring( table.insert({}, 20) ))
-- throws error because nothing ~= nil

--right:
local _tmp = table.insert({}, 20) -- after this _tmp contains nil
print(tostring(_tmp)) -- prints "nil" because suddenly nothing == nil
```

This may also lead to errors when using third party code. If, for example, the documentation of some function states "returns donuts if lucky, nil otherwise", the implementation *might* looks somewhat like this

```
function func(lucky)
    if lucky then
        return "donuts"
    end
end
```

this implementation might seem reasonable at first; it returns donuts when it has to, and when you type `result = func(false)` result will contain the value `nil` .

However, if one were to write `print(tostring(func(false)))` lua would throw an error that looks somewhat like this one `stdin:1: bad argument #1 to 'tostring' (value expected)`

Why is that? `tostring` clearly gets an argument, even though it's `nil` . Wrong. func returns nothing at all, so `tostring(func(false))` is the same as `tostring()` and NOT the same as `tostring(nil)` .

Errors saying "value expected" are a strong indication that this might be the source of the problem.

Leaving gaps in arrays

This is a huge pitfall if you're new to lua, and there's a lot of information in the tables category