WIKIPEDIA

Lua (programming language)

Lua (/ˈluːə/ *LOO-ə*; from Portuguese: *lua* [ˈlu.(w)ɐ] meaning *moon*)[a] is a lightweight, high-level, multi-paradigm programming language designed primarily for embedded use in applications.[3] Lua is cross-platform, since the interpreter of compiled bytecode is written in ANSI C,[4] and Lua has a relatively simple C API to embed it into applications.[5]

Lua was originally designed in 1993 as a language for extending software applications to meet the increasing demand for customization at the time. It provided the basic facilities of most procedural programming languages, but more complicated or domain-specific features were not included; rather, it included mechanisms for extending the language, allowing programmers to implement such features. As Lua was intended to be a general embeddable extension language, the designers of Lua focused on improving its speed, portability, extensibility, and ease-of-use in development.

Lua

Paradigm	Multi-paradigm: scripting, imperative (procedural, prototype-based, object-oriented), functional
Designed by	Roberto Ierusalimschy Waldemar Celes Luiz Henrique de Figueiredo
First appeared	1993
Stable release	5.4.3[1] / 29 March 2021
Typing discipline	Dynamic, strong, duck
Implementation language	ANSI C
OS	Cross-platform
License	MIT License
Filename extensions	`.lua`
Website	www.lua.org (https://www.lua.org/)

Major implementations

Lua (https://www.lua.org/download.html), LuaJIT (https://luajit.org/luajit.html), LuaVela (https://eliasdaler.github.io/luavela/), MoonSharp (https://www.moonsharp.org/), Luvit (https://luvit.io), LuaRT (https://www.luart.org)

Contents

Dialects
Metalua (http://metalua.luaforge.net/), Idle (http://idle.thomaslauer.com), GSL Shell (https://www.nongnu.org/gsl-shell/), Luau (https://luau-lang.org)
Influenced by
C++, CLU, Modula, Scheme, SNOBOL
Influenced
GameMonkey, Io, JavaScript, Julia, MiniD, Red, Ring,[2] Ruby, Squirrel, MoonScript, C--

History

Lua was created in 1992 by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes, members of the Computer Graphics Technology Group (Tecgraf) at the Pontifical Catholic University of Rio de Janeiro, in Brazil.

From 1977 until 1992, Brazil had a policy of strong trade barriers (called a market reserve) for computer hardware and software. In that atmosphere, Tecgraf's clients could not afford, either politically or financially, to buy customized software from abroad. Those reasons led Tecgraf to implement the basic tools it needed from scratch.[6]

Lua's predecessors were the data-description/configuration languages *SOL* (Simple Object Language) and *DEL* (data-entry language).[7] They had been independently developed at Tecgraf in 1992–1993 to add some flexibility into two different projects (both were interactive graphical programs for engineering applications at Petrobras company). There was a lack of any flow-control structures in SOL and DEL, and Petrobras felt a growing need to add full programming power to them.

In *The Evolution of Lua*, the language's authors wrote:[6]

> In 1993, the only real contender was Tcl, which had been explicitly designed to be embedded into applications. However, Tcl had unfamiliar syntax, did not offer good support for data description, and ran only on Unix platforms. We did not consider LISP or Scheme because of their unfriendly syntax. Python was still in its infancy. In the free, do-it-yourself atmosphere that then reigned in Tecgraf, it was quite natural that we should try to develop our own scripting language ... Because many potential users of the language were not professional programmers, the language should avoid cryptic syntax and semantics. The implementation of the new language should be highly portable, because Tecgraf's clients had a very diverse collection of computer platforms. Finally, since we expected that other Tecgraf products would also need to embed a scripting language, the new language should follow the example of SOL and be provided as a library with a C API.

Lua 1.0 was designed in such a way that its object constructors, being then slightly different from the current light and flexible style, incorporated the data-description syntax of SOL (hence the name Lua: *Sol* meaning "Sun" in Portuguese, and *Lua* meaning "Moon"). Lua syntax for control structures was mostly borrowed from Modula (`if`, `while`, `repeat`/`until`), but also had taken influence from CLU (multiple assignments and multiple returns from function calls, as a simpler alternative to reference parameters or explicit pointers), C++ ("neat idea of allowing a local variable to be declared only where we need it"[6]), SNOBOL and AWK (associative arrays). In an article published in *Dr. Dobb's Journal*, Lua's creators also state that LISP and Scheme with their single, ubiquitous data-structure mechanism (the list) were a major influence on their decision to develop the table as the primary data structure of Lua.[8]

Lua semantics have been increasingly influenced by Scheme over time,[6] especially with the introduction of anonymous functions and full lexical scoping. Several features were added in new Lua versions.

Versions of Lua prior to version 5.0 were released under a license similar to the BSD license. From version 5.0 onwards, Lua has been licensed under the MIT License. Both are permissive free software licences and are almost identical.

Features

Lua is commonly described as a "multi-paradigm" language, providing a small set of general features that can be extended to fit different problem types. Lua does not contain explicit support for inheritance, but allows it to be implemented with metatables. Similarly, Lua allows programmers to implement namespaces, classes, and other related features using its single table implementation; first-class functions allow the employment of many techniques from functional programming; and full lexical scoping allows fine-grained information hiding to enforce the principle of least privilege.

In general, Lua strives to provide simple, flexible meta-features that can be extended as needed, rather than supply a feature-set specific to one programming paradigm. As a result, the base language is light—the full reference interpreter is only about 247 kB compiled[4]—and easily adaptable to a broad range of applications.

A dynamically typed language intended for use as an extension language or scripting language, Lua is compact enough to fit on a variety of host platforms. It supports only a small number of atomic data structures such as boolean values, numbers (double-precision floating point and 64-bit integers by default), and strings. Typical data structures such as arrays, sets, lists, and records can be represented using Lua's single native data structure, the table, which is essentially a heterogeneous associative array.

Lua implements a small set of advanced features such as first-class functions, garbage collection, closures, proper tail calls, coercion (automatic conversion between string and number values at run time), coroutines (cooperative multitasking) and dynamic module loading.

Syntax

The classic "Hello, World!" program can be written as follows:[9]

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

or as:

```
print 'Hello, World!'
```

A comment in Lua starts with a double-hyphen and runs to the end of the line, similar to Ada, Eiffel, Haskell, SQL and VHDL. Multi-line strings and comments are adorned with double square brackets.

The factorial function is implemented as a function in this example:

```
function factorial(n)
  local x = 1
```

```
  for i = 2, n do
    x = x * i
  end
  return x
end
```

Control flow

Lua has four types of loops: the `while` loop, the `repeat` loop (similar to a `do while` loop), the numeric `for` loop, and the generic `for` loop.

```
--condition = true

while condition do
  --statements
end

repeat
  --statements
until condition

for i = first, last, delta do  --delta may be negative, allowing the for loop to count down or up
  --statements
  --example: print(i)
end
```

The generic `for` loop:

```
for key, value in pairs(_G) do
  print(key, value)
end
```

would iterate over the table `_G` using the standard iterator function `pairs`, until it returns `nil`.

Loops can also be nested (put inside of another loop).

```
local grid = {
  { 11, 12, 13 },
  { 21, 22, 23 },
  { 31, 32, 33 }
}

for y, row in ipairs(grid) do
  for x, value in ipairs(row) do
    print(x, y, value)
  end
end
```

Functions

Lua's treatment of functions as first-class values is shown in the following example, where the print function's behavior is modified:

```
do
  local oldprint = print
  -- Store current print function as oldprint
  function print(s)
    --[[ Redefine print function. The usual print function can still be used
      through oldprint. The new one has only one argument.]]
    oldprint(s == "foo" and "bar" or s)
```

```
  end
end
```

Any future calls to `print` will now be routed through the new function, and because of Lua's lexical scoping, the old print function will only be accessible by the new, modified print.

Lua also supports closures, as demonstrated below:

```
function addto(x)
  -- Return a new function that adds x to the argument
  return function(y)
    --[=[ When we refer to the variable x, which is outside the current
      scope and whose lifetime would be shorter than that of this anonymous
      function, Lua creates a closure.]=]
    return x + y
  end
end
fourplus = addto(4)
print(fourplus(3))  -- Prints 7

--This can also be achieved by calling the function in the following way:
print(addto(4)(3))
--[[ This is because we are calling the returned function from 'addto(4)' with the argument '3'
directly.
  This also helps to reduce data cost and up performance if being called iteratively.
]]
```

A new closure for the variable `x` is created every time `addto` is called, so that each new anonymous function returned will always access its own `x` parameter. The closure is managed by Lua's garbage collector, just like any other object.

Tables

Tables are the most important data structures (and, by design, the only built-in composite data type) in Lua and are the foundation of all user-created types. They are associative arrays with addition of automatic numeric key and special syntax.

A table is a collection of key and data pairs, where the data is referenced by key; in other words, it is a hashed heterogeneous associative array.

Tables are created using the `{}` constructor syntax.

```
a_table = {} -- Creates a new, empty table
```

Tables are always passed by reference (see Call by sharing).

A key (index) can be any value except `nil` and NaN, including functions.

```
a_table = {x = 10}  -- Creates a new table, with one entry mapping "x" to the number 10.
print(a_table["x"]) -- Prints the value associated with the string key, in this case 10.
b_table = a_table
b_table["x"] = 20   -- The value in the table has been changed to 20.
print(b_table["x"]) -- Prints 20.
print(a_table["x"]) -- Also prints 20, because a_table and b_table both refer to the same table.
```

A table is often used as structure (or record) by using strings as keys. Because such use is very common, Lua features a special syntax for accessing such fields.[10]

```
point = { x = 10, y = 20 }   -- Create new table
print(point["x"])            -- Prints 10
print(point.x)               -- Has exactly the same meaning as line above. The easier-to-read dot
notation is just syntactic sugar.
```

By using a table to store related functions, it can act as a namespace.

```
Point = {}

Point.new = function(x, y)
  return {x = x, y = y}  --  return {["x"] = x, ["y"] = y}
end

Point.set_x = function(point, x)
  point.x = x  --  point["x"] = x;
end
```

Tables are automatically assigned a numerical key, enabling them to be used as an array data type. The first automatic index is 1 rather than 0 as it is for many other programming languages (though an explicit index of 0 is allowed).

A numeric key `1` is distinct from a string key `"1"`.

```
array = { "a", "b", "c", "d" }   -- Indices are assigned automatically.
print(array[2])                  -- Prints "b". Automatic indexing in Lua starts at 1.
print(#array)                    -- Prints 4.  # is the length operator for tables and strings.
array[0] = "z"                   -- Zero is a legal index.
print(#array)                    -- Still prints 4, as Lua arrays are 1-based.
```

The length of a table `t` is defined to be any integer index `n` such that `t[n]` is not `nil` and `t[n+1]` is `nil`; moreover, if `t[1]` is `nil`, `n` can be zero. For a regular array, with non-nil values from 1 to a given `n`, its length is exactly that `n`, the index of its last value. If the array has "holes" (that is, nil values between other non-nil values), then `#t` can be any of the indices that directly precedes a `nil` value (that is, it may consider any such nil value as the end of the array).[11]

```
ExampleTable =
{
  {1, 2, 3, 4},
  {5, 6, 7, 8}
}
print(ExampleTable[1][3]) -- Prints "3"
print(ExampleTable[2][4]) -- Prints "8"
```

A table can be an array of objects.

```
function Point(x, y)        -- "Point" object constructor
  return { x = x, y = y }   -- Creates and returns a new object (table)
end
array = { Point(10, 20), Point(30, 40), Point(50, 60) }   -- Creates array of points
                        -- array = { { x = 10, y = 20 }, { x = 30, y = 40 }, { x = 50, y = 60 } };
print(array[2].y)                                         -- Prints 40
```

Using a hash map to emulate an array is normally slower than using an actual array; however, Lua tables are optimized for use as arrays to help avoid this issue.[12]

Metatables

Extensible semantics is a key feature of Lua, and the metatable concept allows powerful customization of tables. The following example demonstrates an "infinite" table. For any `n`, `fibs[n]` will give the `n`-th Fibonacci number using dynamic programming and memoization.

```
fibs = { 1, 1 }                                -- Initial values for fibs[1] and fibs[2].
setmetatable(fibs, {
  __index = function(values, n)                --[[__index is a function predefined by Lua,
                                                   it is called if key "n" does not exist.]]
    values[n] = values[n - 1] + values[n - 2]  -- Calculate and memorize fibs[n].
    return values[n]
  end
})
```

Object-oriented programming

Although Lua does not have a built-in concept of classes, object-oriented programming can be emulated using functions and tables. An object is formed by putting methods and fields in a table. Inheritance (both single and multiple) can be implemented with metatables, delegating nonexistent methods and fields to a parent object.

There is no such concept as "class" with these techniques; rather, prototypes are used, similar to Self or JavaScript. New objects are created either with a factory method (that constructs new objects from scratch) or by cloning an existing object.

Creating a basic vector object:

```
local Vector = {}
local VectorMeta = { __index = Vector}

function Vector.new(x, y, z)    -- The constructor
  return setmetatable({x = x, y = y, z = z}, VectorMeta)
end

function Vector.magnitude(self)     -- Another method
  return math.sqrt(self.x^2 + self.y^2 + self.z^2)
end

local vec = Vector.new(0, 1, 0) -- Create a vector
print(vec.magnitude(vec))       -- Call a method (output: 1)
print(vec.x)                    -- Access a member variable (output: 0)
```

Here, `setmetatable` tells Lua to look for an element in the `Vector` table if it is not present in the `vec` table. `vec.magnitude`, which is equivalent to `vec["magnitude"]`, first looks in the `vec` table for the `magnitude` element. The `vec` table does not have a `magnitude` element, but its metatable delegates to the `Vector` table for the `magnitude` element when it's not found in the `vec` table.

Lua provides some syntactic sugar to facilitate object orientation. To declare member functions inside a prototype table, one can use `function table:func(args)`, which is equivalent to `function table.func(self, args)`. Calling class methods also makes use of the colon: `object:func(args)` is equivalent to `object.func(object, args)`.

That in mind, here is a corresponding class with `:` syntactic sugar:

```
local Vector = {}
Vector.__index = Vector

function Vector:new(x, y, z)    -- The constructor
  -- Since the function definition uses a colon,
  -- its first argument is "self" which refers
```

```lua
  -- to "Vector"
  return setmetatable({x = x, y = y, z = z}, self)
end

function Vector:magnitude()     -- Another method
  -- Reference the implicit object using self
  return math.sqrt(self.x^2 + self.y^2 + self.z^2)
end

local vec = Vector:new(0, 1, 0) -- Create a vector
print(vec:magnitude())          -- Call a method (output: 1)
print(vec.x)                    -- Access a member variable (output: 0)
```

Inheritance

Lua supports using metatables to give Lua class inheritance.[13] In this example, we allow vectors to have their values multiplied by a constant in a derived class.

```lua
local Vector = {}
Vector.__index = Vector

function Vector:new(x, y, z)    -- The constructor
  -- Here, self refers to whatever class's "new"
  -- method we call.  In a derived class, self will
  -- be the derived class; in the Vector class, self
  -- will be Vector
  return setmetatable({x = x, y = y, z = z}, self)
end

function Vector:magnitude()     -- Another method
  -- Reference the implicit object using self
  return math.sqrt(self.x^2 + self.y^2 + self.z^2)
end

-- Example of class inheritance
local VectorMult = {}
VectorMult.__index = VectorMult
setmetatable(VectorMult, Vector) -- Make VectorMult a child of Vector

function VectorMult:multiply(value)
  self.x = self.x * value
  self.y = self.y * value
  self.z = self.z * value
  return self
end

local vec = VectorMult:new(0, 1, 0) -- Create a vector
print(vec:magnitude())          -- Call a method (output: 1)
print(vec.y)                    -- Access a member variable (output: 1)
vec:multiply(2)                 -- Multiply all components of vector by 2
print(vec.y)                    -- Access member again (output: 2)
```

Lua also supports multiple inheritance; `__index` can either be a function or a table.[14] Operator overloading can also be done; Lua metatables can have elements such as `__add`, `__sub`, and so on.[15]

Implementation

Lua programs are not interpreted directly from the textual Lua file, but are compiled into bytecode, which is then run on the Lua virtual machine. The compilation process is typically invisible to the user and is performed during run-time, especially when a JIT compiler is used, but it can be done offline in order to increase loading performance or reduce the memory footprint of the host environment by leaving out the compiler. Lua bytecode can also be produced and executed from within Lua, using the `dump` function from the string library and the `load/loadstring/loadfile` functions. Lua version 5.3.4 is implemented in approximately 24,000 lines of C code.[3][4]

Like most CPUs, and unlike most virtual machines (which are stack-based), the Lua VM is register-based, and therefore more closely resembles an actual hardware design. The register architecture both avoids excessive copying of values and reduces the total number of instructions per function. The virtual machine of Lua 5 is one of the first register-based pure VMs to have a wide use.[16] Parrot and Android's Dalvik are two other well-known register-based VMs. PCScheme's VM was also register-based.[17]

This example is the bytecode listing of the factorial function defined above (as shown by the `luac` 5.1 compiler):[18]

```
function <factorial.lua:1,7> (9 instructions, 36 bytes at 0x8063c60)
1 param, 6 slots, 0 upvalues, 6 locals, 2 constants, 0 functions
        1       [2]     LOADK           1 -1    ; 1
        2       [3]     LOADK           2 -2    ; 2
        3       [3]     MOVE            3 0
        4       [3]     LOADK           4 -1    ; 1
        5       [3]     FORPREP         2 1     ; to 7
        6       [4]     MUL             1 1 5
        7       [3]     FORLOOP         2 -2    ; to 6
        8       [6]     RETURN          1 2
        9       [7]     RETURN          0 1
```

C API

Lua is intended to be embedded into other applications, and provides a C API for this purpose. The API is divided into two parts: the Lua core and the Lua auxiliary library.[19] The Lua API's design eliminates the need for manual reference management in C code, unlike Python's API. The API, like the language, is minimalistic. Advanced functionality is provided by the auxiliary library, which consists largely of preprocessor macros which assist with complex table operations.

The Lua C API is stack based. Lua provides functions to push and pop most simple C data types (integers, floats, etc.) to and from the stack, as well as functions for manipulating tables through the stack. The Lua stack is somewhat different from a traditional stack; the stack can be indexed directly, for example. Negative indices indicate offsets from the top of the stack. For example, −1 is the top (most recently pushed value), while positive indices indicate offsets from the bottom (oldest value). Marshalling data between C and Lua functions is also done using the stack. To call a Lua function, arguments are pushed onto the stack, and then the `lua_call` is used to call the actual function. When writing a C function to be directly called from Lua, the arguments are read from the stack.

Here is an example of calling a Lua function from C:

```c
#include <stdio.h>
#include <lua.h> // Lua main library (lua_*)
#include <lauxlib.h> // Lua auxiliary library (luaL_*)

int main(void)
{
    // create a Lua state
    lua_State *L = luaL_newstate();

    // load and execute a string
    if (luaL_dostring(L, "function foo (x,y) return x+y end")) {
        lua_close(L);
        return -1;
    }
```

```
    // push value of global "foo" (the function defined above)
    // to the stack, followed by integers 5 and 3
    lua_getglobal(L, "foo");
    lua_pushinteger(L, 5);
    lua_pushinteger(L, 3);
    lua_call(L, 2, 1); // call a function with two arguments and one return value
    printf("Result: %d\n", lua_tointeger(L, -1)); // print integer value of item at stack top
    lua_pop(L, 1); // return stack to original state
    lua_close(L); // close Lua state
    return 0;
}
```

Running this example gives:

```
$ cc -o example example.c -llua
$ ./example
Result: 8
```

The C API also provides some special tables, located at various "pseudo-indices" in the Lua stack. At `LUA_GLOBALSINDEX` prior to Lua 5.2[20] is the globals table, `_G` from within Lua, which is the main namespace. There is also a registry located at `LUA_REGISTRYINDEX` where C programs can store Lua values for later retrieval.

It is possible to write extension modules using the Lua API. Extension modules are shared objects which can be used to extend the functionality of the interpreter by providing native facilities to Lua scripts. From the Lua side, such a module appears as a namespace table holding its functions and variables. Lua scripts may load extension modules using `require`,[19] just like modules written in Lua itself. A growing collection of modules known as *rocks* are available through a package management system called LuaRocks,[21] in the spirit of CPAN, RubyGems and Python eggs. Prewritten Lua bindings exist for most popular programming languages, including other scripting languages.[22] For C++, there are a number of template-based approaches and some automatic binding generators.

Applications

In video game development, Lua is widely used as a scripting language by programmers, mainly due to its perceived easiness to embed, fast execution, and short learning curve.[23] One of the notable gaming platforms is Roblox in which their own dialect, Luau, is used for scripting quick development of games.[24] Another is World of Warcraft which also uses a scaled down version of Lua. [25]

In 2003, a poll conducted by GameDev.net showed Lua was the most popular scripting language for game programming.[26] On 12 January 2012, Lua was announced as a winner of the Front Line Award 2011 from the magazine *Game Developer* in the category Programming Tools.[27]

A large number of non-game applications also use Lua for extensibility, such as LuaTeX, an implementation of the TeX type-setting language, Redis, a key-value database, Neovim, a text editor, and Nginx, a web server.

Through the Scribunto extension, Lua is available as a server-side scripting language in the MediaWiki software that powers Wikipedia and other wikis.[28] Among its uses are allowing the integration of data from Wikidata into articles,[29] and powering the automated taxobox system.

Derived languages

Languages that compile to Lua

- MoonScript is a dynamic, whitespace-sensitive scripting language inspired by CoffeeScript, which is compiled into Lua. This means that instead of using `do` and `end` (or `{` and `}`) to delimit sections of code it uses line breaks and indentation style.[30][31][32] A notable usage of MoonScript is a video game distribution website Itch.io.
- Haxe supports compilation to a Lua target, supporting Lua 5.1-5.3 as well as LuaJIT 2.0 and 2.1.
- Fennel, a Lisp dialect that targets Lua.[32]
- Urn, a Lisp dialect that is built on Lua.[33]
- Amulet, an ML-like functional language, whose compiler outputs Lua files.[34]

Dialects

- LuaJIT (see below), JIT-enabled Lua 5.1 language with `goto` (from Lua 5.2) and a C FFI.
- Luau from Roblox, Lua 5.1 language with gradual typing and ergonomic additions.[35]
- Ravi, JIT-enabled Lua 5.3 language with optional static typing. JIT is guided by type information.[36]
- Shine, a fork of LuaJIT with many extensions, including a module system and a macro system.[37]

In addition, the Lua users community provides some *power patches* on top of the reference C implementation.[38]

LuaJIT

LuaJIT	
Developer(s)	Mike Pall
Stable release	2.0.5 / May 1, 2017
Repository	repo.or.cz/w/luajit-2.0.git (https://repo.or.cz/w/luajit-2.0.git)
Written in	C, Lua
Operating system	see list
Type	Just in time compiler
License	MIT License[39]
Website	luajit.org (https://luajit.org)

LuaJIT is a just in time compiler for Lua. It has been used for embedding or for general purposes. In version 2.0 of LuaJIT, the project has been rewritten for better optimizations for performance.[40]

History

The LuaJIT project has started in 2005 by developer Mike Pall, released under the MIT open source license.[41] The latest release, 2.0.5 is released in 2017. Since then, the project is not currently maintained by developers other than contributors.[42]

Installation

LuaJIT is open source and the project has to be compiled in order for it to be used. The repository will have to be download with Git or other methods of downloading repositories.[42] Then it is compiled with any C compiler, usually with GNU make, but other options are available.[43] Finally, the LuaJIT executable and the Lua 5.1 DLL have to be in the same directory in order for the LuaJIT compiler to be used.

There is a guide on using the LuaJIT compiler which includes command line options.[44]

Performance

When compared to other Lua run-times, LuaJIT is often the fastest Lua compiler.[45]

Platforms

LuaJIT can be used in:[46]

- Windows
- Xbox 360
- Linux
- Android
- BSD
- Sony PlayStation:
  - PS3
  - PS4
  - PS Vita
- macOS
- iOS

It can be compiled using either GCC, Clang, or MSVC.[46]

Examples

The FFi Library can be used to call C functions and use C data structures from Lua Code.[47] There is a guide provided from LuaJIT about using this library.[48] As such, there are multiple LuaJIT bindings to C libraries that use the FFI Library. This example would call a C function, printf from pure Lua code and will output Hello world!.

```
local ffi = require("ffi")
ffi.cdef[[
int printf(const char *fmt, ...);
]]
ffi.C.printf("Hello world!\n")
```

The LuaJIT compiler has also added some extensions to Lua's C API.[49] This example written in C++ would be used for debugging purposes.

```
#include <exception>
#include "lua.hpp"

// Catch C++ exceptions and convert them to Lua error messages.
// Customize as needed for your own exception classes.
static int wrap_exceptions(lua_State *L, lua_CFunction f)
{
  try {
    return f(L);  // Call wrapped function and return result.
  } catch (const char *s) {  // Catch and convert exceptions.
    lua_pushstring(L, s);
  } catch (std::exception& e) {
    lua_pushstring(L, e.what());
  } catch (...) {
    lua_pushliteral(L, "caught (...)");
  }
  return lua_error(L);  // Rethrow as a Lua error.
}
```

```
static int myinit(lua_State *L)
{
  ...
  // Define wrapper function and enable it.
  lua_pushlightuserdata(L, (void *)wrap_exceptions);
  luaJIT_setmode(L, -1, LUAJIT_MODE_WRAPCFUNC|LUAJIT_MODE_ON);
  lua_pop(L, 1);
  ...
}
```

See also

- Comparison of programming languages

Notes

a. The name is commonly mis-capitalised as "LUA". This is incorrect because the name is not an acronym.

References

1. "Release 5.4.3" (https://github.com/lua/lua/releases/tag/v5.4.3). 29 March 2021. Retrieved 30 March 2021.
2. Ring Team (5 December 2017). "The Ring programming language and other languages" (http://ring-lang.sourceforge.net/doc1.6/introduction.html#ring-and-other-languages). *ring-lang.net*. ring-lang.
3. Ierusalimschy, Roberto; de Figueiredo, Luiz Henrique; Filho, Waldemar Celes (June 1996). "Lua—An Extensible Extension Language" (https://www.lua.org/spe.html). *Software: Practice and Experience*. **26** (6): 635–652. doi:10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P (https://doi.org/10.1002%2F%28SICI%291097-024X%28199606%2926%3A6%3C635%3A%3AAID-SPE26%3E3.0.CO%3B2-P). Retrieved 24 October 2015.
4. "About Lua" (https://www.lua.org/about.html#why). Lua.org. Retrieved 11 August 2011.
5. Yuri Takhteyev (21 April 2013). "From Brazil to Wikipedia" (https://www.foreignaffairs.com/articles/139332/yuri-takhteyev/from-brazil-to-wikipedia?page=2). *Foreign Affairs*. Retrieved 25 April 2013.
6. Ierusalimschy, R.; Figueiredo, L. H.; Celes, W. (2007). "The evolution of Lua" (https://www.lua.org/doc/hopl.pdf) (PDF). *Proc. of ACM HOPL III* (https://www.acm.org/sigs/sigplan/hopl). pp. 2–1–2–26. doi:10.1145/1238844.1238846 (https://doi.org/10.1145%2F1238844.1238846). ISBN 978-1-59593-766-7. S2CID 475143 (https://api.semanticscholar.org/CorpusID:475143).
7. "The evolution of an extension language: a history of Lua" (https://www.lua.org/history.html). 2001. Retrieved 18 December 2008.
8. Figueiredo, L. H.; Ierusalimschy, R.; Celes, W. (December 1996). "Lua: an Extensible Embedded Language. A few metamechanisms replace a host of features" (https://www.lua.org/ddj.html). *Dr. Dobb's Journal*. **21** (12). pp. 26–33.
9. "Programming in Lua : 1" (https://www.lua.org/pil/1.html).
10. "Lua 5.1 Reference Manual" (https://www.lua.org/manual/5.1/manual.html#2.3). 2014. Retrieved 27 February 2014.
11. "Lua 5.1 Reference Manual" (https://www.lua.org/manual/5.1/manual.html#2.5.5). 2012. Retrieved 16 October 2012.
12. "Lua 5.1 Source Code" (https://www.lua.org/source/5.1/lobject.h.html#array). 2006. Retrieved 24 March 2011.
13. Roberto Ierusalimschy. *Programming in Lua, 4th Edition*. p. 165.

14. "Programming in Lua : 16.3" (https://www.lua.org/pil/16.3.html). *www.lua.org*. Retrieved 16 September 2021.
15. "lua-users wiki: Metamethods Tutorial" (http://lua-users.org/wiki/MetamethodsTutorial). *lua-users.org*. Retrieved 16 September 2021.
16. Ierusalimschy, R.; Figueiredo, L. H.; Celes, W. (2005). "The implementation of Lua 5.0" (http://www.jucs.org/jucs_11_7/the_implementation_of_lua/jucs_11_7_1159_1176_defigueiredo.html). *J. Of Universal Comp. Sci*. **11** (7): 1159–1176.
17. Texas Instruments (1990). *PC Scheme: Users Guide and Language Reference Manual, Trade Edition*. ISBN 0-262-70040-9.
18. Kein-Hong Man (2006). "A No-Frills Introduction to Lua 5.1 VM Instructions" (https://talk.pokitto.com/uploads/default/original/2X/7/716c67a0b5b1636cbc4dc1fec232ca2536cb74d1.pdf) (PDF).
19. "Lua 5.2 Reference Manual" (https://www.lua.org/manual/5.2/). Lua.org. Retrieved 23 October 2012.
20. "Changes in the API" (https://www.lua.org/manual/5.2/manual.html#8.3). *Lua 5.2 Reference Manual*. Lua.org. Retrieved 9 May 2014.
21. "LuaRocks" (https://luarocks.org/). LuaRocks wiki. Retrieved 24 May 2009.
22. "Binding Code To Lua" (http://lua-users.org/wiki/BindingCodeToLua). Lua-users wiki. Retrieved 24 May 2009.
23. "Why is Lua considered a game language?" (https://web.archive.org/web/20130820131611/http://stackoverflow.com/questions/38338/why-is-lua-considered-a-game-language). Archived from the original on 20 August 2013. Retrieved 22 April 2017.
24. "Why Luau?" (https://roblox.github.io/luau/why.html). *Luau*. Retrieved 10 February 2021.
25. "Lua Functions" (https://wow.gamepedia.com/Lua_functions). *wow.gamepedia.com*. Retrieved 1 March 2021.
26. "Poll Results" (https://web.archive.org/web/20031207171619/http://gamedev.net/gdpolls/viewpoll.asp?ID=163). Archived from the original on 7 December 2003. Retrieved 22 April 2017.
27. "Front Line Award Winners Announced" (https://web.archive.org/web/20130615013638/http://www.gdmag.com/blog/2012/01/front-line-award-winners.php). Archived from the original on 15 June 2013. Retrieved 22 April 2017.
28. "Extension:Scribunto - MediaWiki" (https://www.mediawiki.org/wiki/Extension:Scribunto). *MediaWiki.org*. Retrieved 21 February 2019.
29. "Wikidata:Infobox Tutorial - Wikidata" (https://www.wikidata.org/wiki/Wikidata:Infobox_Tutorial). *www.wikidata.org*. Retrieved 21 December 2018.
30. "Language Guide - MoonScript 0.5.0" (https://moonscript.org/reference/). *moonscript.org*. Retrieved 25 September 2020.
31. leaf (23 September 2020), *leafo/moonscript* (https://github.com/leafo/moonscript), retrieved 25 September 2020
32. Andre Alves Garzia. "Languages that compile to Lua" (https://andregarzia.com/2020/06/languages-that-compile-to-lua.html). *AndreGarzia.com*. Retrieved 25 September 2020.
33. "Urn: A Lisp implementation for Lua | Urn" (https://urn-lang.com/). *urn-lang.com*. Retrieved 12 January 2021.
34. "Amulet ML" (https://amulet.works/). *amulet.works*. Retrieved 12 January 2021.
35. "Luau" (https://roblox.github.io/luau/). *Roblox.GitHub.io*.
36. "Ravi Programming Language" (http://ravilang.github.io/). *GitHub*.
37. Hundt, Richard (22 April 2021). "richardhundt/shine" (https://github.com/richardhundt/shine).
38. "Lua Power Patches" (http://lua-users.org/wiki/LuaPowerPatches). *lua-users.org*.
39. https://github.com/LuaJIT/LuaJIT/blob/v2.1/COPYRIGHT
40. http://luajit.org/luajit.html
41. https://luajit.org

42. https://luajit.org/download.html
43. https://luajit.org/install.html
44. http://luajit.org/running.html
45. https://staff.fnwi.uva.nl/h.vandermeer/docs/lua/luajit/luajit_performance.html
46. https://luajit.org/luajit.html
47. http://luajit.org/ext_ffi.html
48. http://luajit.org/ext_ffi_tutorial.html
49. http://luajit.org/ext_c_api.html

Further reading

- Ierusalimschy, R. (2013). *Programming in Lua* (https://www.lua.org/pil/) (3rd ed.). Lua.org. ISBN 978-85-903798-5-0. (The 1st ed. is available online (https://www.lua.org/pil/contents.html).)
- Gutschmidt, T. (2003). *Game Programming with Python, Lua, and Ruby*. Course Technology PTR. ISBN 978-1-59200-077-7.
- Schuytema, P.; Manyen, M. (2005). *Game Development with Lua*. Charles River Media. ISBN 978-1-58450-404-7.
- Jung, K.; Brown, A. (2007). *Beginning Lua Programming* (https://web.archive.org/web/20180708015602/https://www.wrox.com/WileyCDA/WroxTitle/productCd-0470069171.html). Wrox Press. ISBN 978-0-470-06917-2. Archived from the original (https://www.wrox.com/WileyCDA/WroxTitle/productCd-0470069171.html) on 8 July 2018. Retrieved 7 July 2018.
- Figueiredo, L. H.; Celes, W.; Ierusalimschy, R., eds. (2008). *Lua Programming Gems* (https://www.lua.org/gems/). Lua.org. ISBN 978-85-903798-4-3.
- Takhteyev, Yuri (2012). *Coding Places: Software Practice in a South American City* (https://web.archive.org/web/20121102000628/http://codingplaces.net/). The MIT Press. ISBN 978-0-262-01807-4. Archived from the original (https://codingplaces.net/) on 2 November 2012. Chapters 6 and 7 are dedicated to Lua, while others look at software in Brazil more broadly.
- Varma, Jayant (2012). *Learn Lua for iOS Game Development* (https://www.amazon.com/exec/obidos/ASIN/1430246626/lua-docs-20). Apress. ISBN 978-1-4302-4662-6.
- Matheson, Ash (29 April 2003). "An Introduction to Lua" (https://web.archive.org/web/20121218104442/http://www.gamedev.net/page/resources/_/technical/game-programming/an-introduction-to-lua-r1932). *GameDev.net*. Archived from the original (https://www.gamedev.net/page/resources/_/technical/game-programming/an-introduction-to-lua-r1932) on 18 December 2012. Retrieved 3 January 2013.
- Fieldhouse, Keith (16 February 2006). "Introducing Lua" (https://web.archive.org/web/20060312124121/http://www.onlamp.com/pub/a/onlamp/2006/02/16/introducing-lua.html). *ONLamp.com*. O'Reilly Media. Archived from the original (http://www.onlamp.com/pub/a/onlamp/2006/02/16/introducing-lua.html) on 12 March 2006. Retrieved 28 February 2006.
- Streicher, Martin (28 April 2006). "Embeddable scripting with Lua" (https://www.ibm.com/developerworks/linux/library/l-lua.html). *developerWorks*. IBM.
- Quigley, Joseph (1 June 2007). "A Look at Lua" (https://www.linuxjournal.com/article/9605). *Linux Journal*.
- Hamilton, Naomi (11 September 2008). "The A-Z of Programming Languages: Lua" (https://web.archive.org/web/20180708020030/https://www.computerworld.com.au/article/260022/a-z_programming_languages_lua/). *Computerworld*. IDG. Archived from the original (https://www.computerworld.com.au/article/260022/a-z_programming_languages_lua/) on 8 July 2018. Retrieved 7 July 2018. Interview with Roberto Ierusalimschy.
- Ierusalimschy, Roberto; de Figueiredo, Luiz Henrique; Celes, Waldemar (12 May 2011). "Passing a Language through the Eye of a Needle" (https://doi.org/10.1145%2F1978862.1983083). *ACM Queue*. **9** (5): 20–29. doi:10.1145/1978862.1983083 (https://doi.org/10.1145%2F19

78862.1983083). S2CID 19484689 (https://api.semanticscholar.org/CorpusID:19484689). How the embeddability of Lua impacted its design.

- Ierusalimschy, Roberto; de Figueiredo, Luiz Henrique; Celes, Waldemar (November 2018). "A Look at the Design of Lua" (https://cacm.acm.org/magazines/2018/11/232214-a-look-at-the-design-of-lua). *Communications of the ACM*. **61** (11): 114–123. doi:10.1145/3186277 (https://doi.org/10.1145%2F3186277). S2CID 53114923 (https://api.semanticscholar.org/CorpusID:53114923).
- Lua papers and theses (https://www.lua.org/papers.html)

External links

- Official website (https://www.lua.org/)
- Lua Users (http://lua-users.org/), Community
- Lua Forum (https://luaforum.com)
- Lua Rocks - Package manager (https://luarocks.org/)
- Projects in Lua (https://web.archive.org/web/20070202005230/http://luaforge.net/)