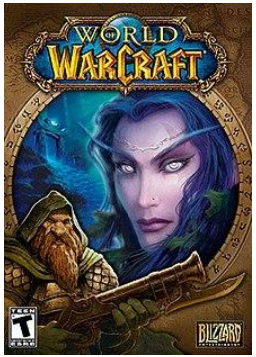
Lua

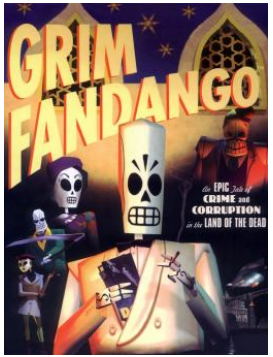
By: Dan Imbimbo

Introduction

What is Lua?

- Lightweight, embeddable, user-friendly scripting language
- Many similarities to Python
- Can interact with C Languages
- Primarily for video game development





Getting Started

Lua.org

1.

about
news
get started
download
documentation
community
site map
português

2.

❖ Building

Lua is implemented in pure ANSI C and compiles unmodified in all platforms that have an ANSI C compiler. Lua also compiles cleanly as C++.

Lua is very easy to build and install. There are detailed instructions in the package but here is a simple terminal session that downloads the current release of Lua and builds it in Linux:

```
curl -R -O http://www.lua.org/ftp/lua-5.4.4.tar.gz
tar zxf lua-5.4.4.tar.gz
cd lua-5.4.4
make all test
```

If you have trouble building Lua, read the FAQ.

If you don't have the time or the inclination to compile Lua yourself, get a binary or try the live demo.

Download Binaries

3.

LuaBinaries

- Home
  - Overview
  - Motivation
  - Installation
  - History
  - Credits
  - Contact us
- Manual
- Configuration
- Packaging
- Download
- License

Overview

LuaBinaries is a distribution of the **Lua** libraries a

This distribution offers a standard set of Lua librar
Kepler Project components.

LuaBinaries is free software and uses the same li

Motivation

Since Lua offers no standard set of libraries, man
resulting products.

Tecgraf/PUC-Rio and Kepler Project teams have
be nice to have a single standard for the whole Lu
are called LuaBinaries and are available here for

4.

Download

All the binaries, source code and documentation are available from the SourceForge project files page:

https://sourceforge.net/projects/luabinaries/files/

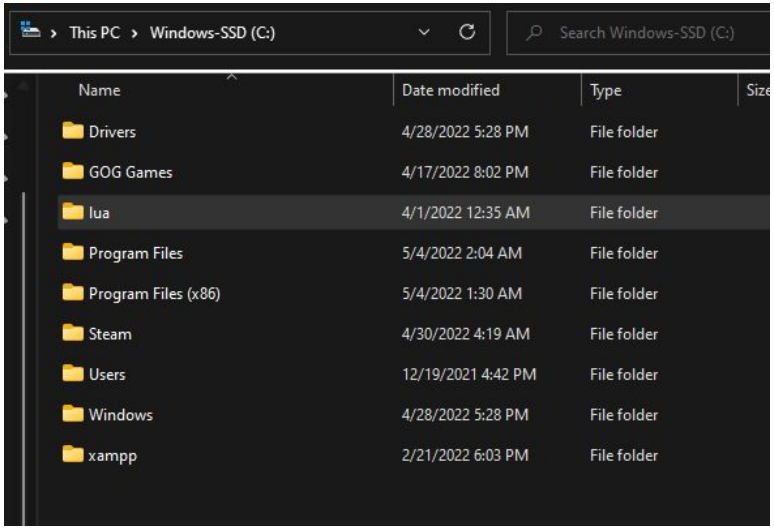
But here are shortcuts for the most popular downloads:

LuaBinaries 5.4.2 - Release 1

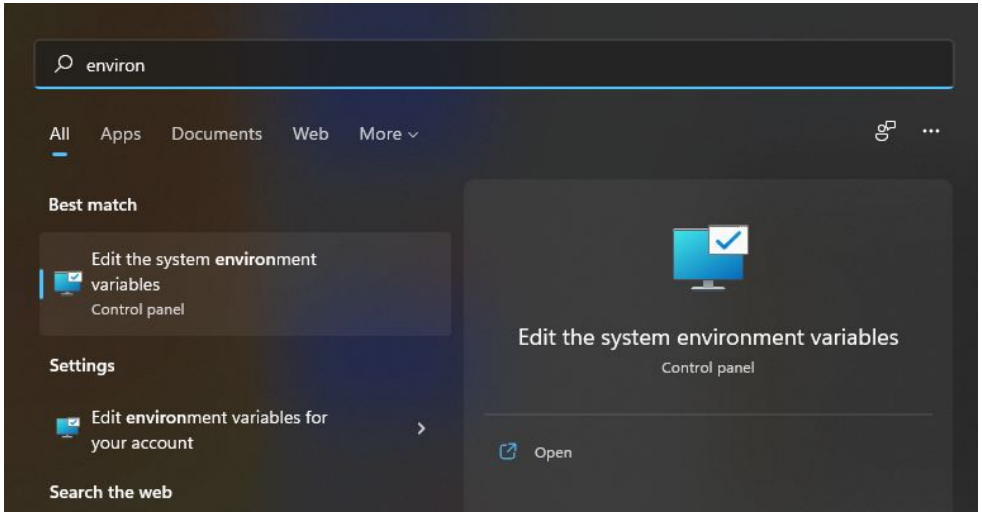
lua-5.4.2_Sources.tar.gz	Source Code and Makefiles
lua-5.4.2_Sources.zip	Source Code and Makefiles
lua-5.4.2_Win32_bin.zip	Windows x86 Executables
lua-5.4.2_Win32_dllw6_lib.zip	Windows x86 DLL and Includes (MingW-w64 6 Built)
lua-5.4.2_Win64_bin.zip	Windows x64 Executables
lua-5.4.2_Win64_dllw6_lib.zip	Windows x64 DLL and Includes (MingW-w64 6 Built)
lua-5.4.2_MacOS1011_bin.tar.gz	MacOS X Intel Executables
lua-5.4.2_MacOS1011_lib.tar.gz	MacOS X Intel Library and Includes
lua-5.4.2_Linux54_64_bin.tar.gz	Linux x64 Executables
lua-5.4.2_Linux54_64_lib.tar.gz	Linux x64 Library and Includes

Extract to folder

5.

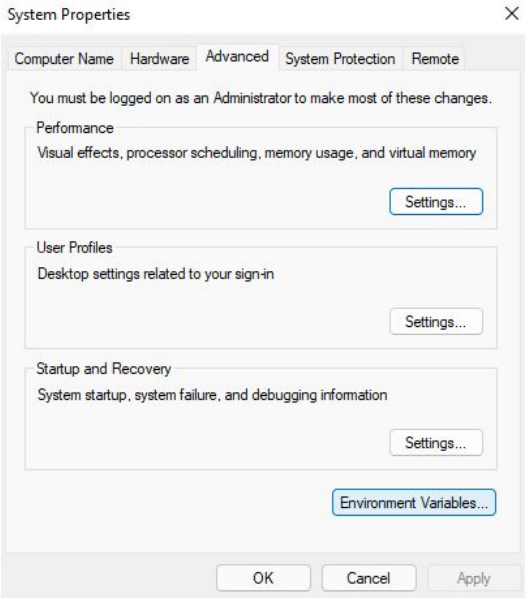


6.

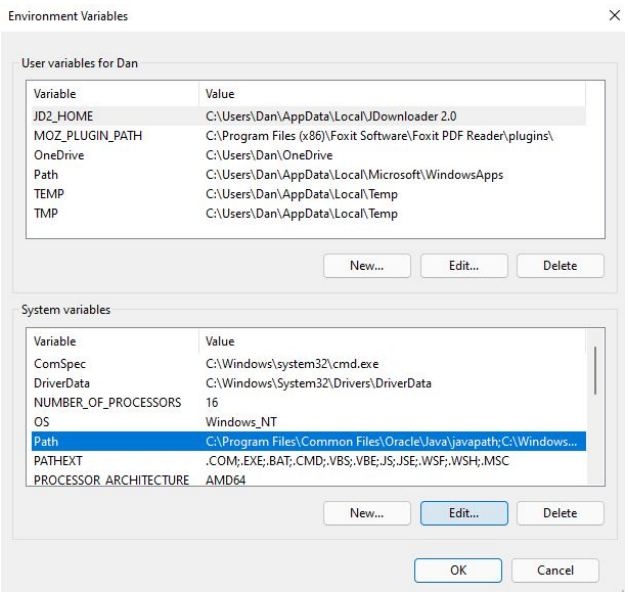


Add to System PATH variable

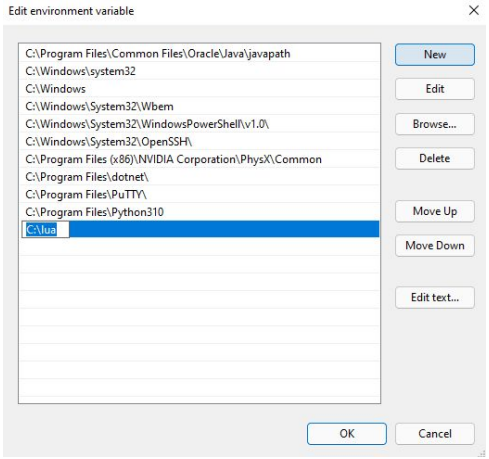
7.



8.



9.



Lua can now be executed from any directory with ‘lua54’

Command Prompt - lua54

```
Microsoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Dan>lua54
Lua 5.4.2  Copyright (C) 1994-2020 Lua.org, PUC-Rio
>
```

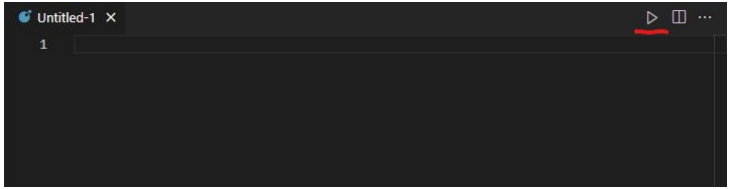
Optional: VS Code as programming environment

1. Download extensions

2. Edit settings.json

```
{
"code-runner.runInTerminal": true,
    "code-runner.executorMap": {
        "lua":"lua54"
    }
}
```

Execute lua scripts via button



Programming Under Lua

Reserved Words & Commenting

Basic Information

- Lua is a case sensitive language

The following *keywords* are reserved and cannot be used as names:

and	break	do	else	elseif	end
false	for	function	goto	if	in
local	nil	not	or	repeat	return
then	true	until	while		

- Single line & Multiple line commenting

```
--This is a single line comment.
--[[
    This is a block comment.
]]
```

Variables & Data Types

Nil

- Effectively NULL values
  - ‘None’ in Python

Input:

```
a=nil
print(type(a)) --prints the data type of a (nil)
```

Output:

```
nil
```

Number

- Numerical values
  - Subtypes of float and integer

Input:

```
b=5
print(type(b)) --prints the data type of b (5)
c=3.14
print(type(c)) --prints the data type of c (3.14)
```

Output:

```
number
number
```

Input:

```
b=5
print(math.type(b)) --prints the subtype of b (5)
c=3.14
print(math.type(c)) --prints the subtype of c (3.14)
```

Output:

```
integer
float
```

String

- Collection of alphanumeric characters
  - No separate ‘char’ type

Input:

```
d="This is a string."
print(type(d)) --prints the data type of d ("This is a string.")

e='a'
print(type(e)) --prints the data type of e ('a')

print(d..e) --concatenation of d and e

print(#d) --prints the length of the string d ("This is a string.")
```

Output:

```
string
string
This is a string.a
17
```

- ‘..’ Concatenation operator
  - Used to append one string to another
- ‘#’ Length operator
  - Used to get the character length of a given string

Boolean

- Conditional values
  - nil is also considered ‘false’

Input:

```
f=true
g=false
print(type(f)) --prints the data type of f (true)
print(type(g)) --prints the data type of g (false)
```

Output:

```
boolean
boolean
```

Table

- Only data structure in Lua
  - Used for arrays, records, lists, queues, sets, etc.

Input:

```
h={1,2,3}
print(type(h)) --prints the data type of g ({1,2,3})

h["key"]="example of (key,value)"
print(h["key"]) --prints the value of h["key"]

i={10, 8, 6, 4, 2}
k={}
print(#i) --prints a border value (index) of i
print(#k) --prints a border value (index) of k

table.sort(i) --sorts the given table in ascending order
print(table.unpack(i)) --returns each of the elements of a table
```

Output:

```
table
example of (key,value)
5
0
2        4        6        8        10
```

- No fixed size
- Can be assigned key as index
- Indexing automatically starts at 1
- Many functions for table manipulation
  - table.remove()
  - table.insert()
  - table.sort()
  - table.unpack()

Function

- Individually defined procedures
- Considered as first-class values
  - Storable in variables, can be passed to other functions, and/or returned

Input:

```
print(type(print)) --prints the data type of print
```

Output:

```
function
```

(Functions will be discussed further later on…)

Userdata & Threads

- Userdata stores data from C programs
- Threads store coroutines
  - Allowing for separate threads of execution

(Threads/Coroutines will also be discussed further later on…)

Numeric Operators

1. Arithmetic

a. Unary

Negation & Exponentiation

- '-' Negation reverses the sign of a value

Input:

```
a=3
print(-a)
```

Output:

```
-3
```

- '^' Exponentiation raises one number by another
  - Utilizes C's 'pow' function
  - Coerces result to 'float'

Input:

```
a = 3
print(a^2)
```

Output:

```
9.0
```

b. Binary

Addition & Subtraction

- ‘+’ Addition evaluates the sum of two values
- ‘-’ Subtraction evaluates the difference between two values

Input:

```
a, b = 3, 2
print("Addition:"..(a+b))
b=2.0
print("Subtraction:"..(a-b))
```

Output:

```
Addition:5
Subtraction:1.0
```

Operations between a float and integer coerce the integer to a float

<u>General</u>

- Can assign multiple values to multiple variables on one line

Multiplication

- ‘*’ Multiplication evaluates the product of two numbers

Input:

```
a, b = 3, 2
print(a*b)
```

Output:

```
6
```

Float/Floor Division & Modulo

- ‘/’ Float division computes the quotient of two values coerced to a float
- ‘//’ Float division computes the quotient rounded down to the nearest integer

Input:

```
a, b = 3, 2
print("Float Division:"..(a/b))
print("Floor Division:"..(a//b))
```

Output:

```
Float Division:1.5
Floor Division:1
```

- ‘%’ Modulo calculates the remainder of a quotient using floor division

Input:

```
a, b = 3, 2
print(a%b)
```

Output:

```
1
```

2. Bitwise

Bitwise AND & OR

- '&' Bitwise AND returns a 1 in binary for each 1 shared by both binary values
  - Final result is an integer

Input:

```
a, b = 5, 3
print(a & b)
```

Output:

```
1
```

5 in binary = 101

3 in binary = 011

101 AND 011 = 001 = 1

101 OR 011 = 111 = 7

- '|' Bitwise OR returns a 1 in binary for each 1 of either binary values
  - Final result is an integer

Input:

```
a, b = 5, 3
print(a | b)
```

Output:

```
7
```

Bitwise XOR & NOT

- '~' Bitwise XOR returns a 1 in binary for each 1 that is not shared by both binary values
  - Final result is an integer

Input:

```
a, b = 5, 3
print(a ~ b)
```

Output:

```
6
```

5 in binary = 101

3 in binary = 011

101 XOR 011= 110 = 6

NOT 101 = -(101 + 1)= -110= -6

- '~' Bitwise NOT returns the binary value as a negative with an additional bit
  - Final result is an integer

Input:

```
a = 5
print(~a)
```

Output:

```
-6
```

Left Shift & Right Shift

- '<<' Left shift shifts each bit of the binary value to the left by a given number
  - Final result is an integer

Input:

```
a, b = 5, 3
print(a << b)
```

Output:

```
40
```

5 in binary = 101

101 << 3 = 0010 1000 = 40

101 >> 3 = 0000 = 0

- '>>' Right shift shifts each bit of the binary value to the right by a given number
  - Final result is an integer

Input:

```
a, b = 5, 3
print(a >> b)
```

Output:

```
0
```

  - bits in 101 = 3 bits
  - Right shift = 3
  - Abs value of shift >= bits in 101
    - ➢ Will cause integer result of 0

3. Relational

Equality & Inequality

- '==' Equality determines if two values of the same type are equivalent
  - Comparisons between two values of different types will always return false

Input:

```
a, b = 4,8
print(a==b)
print(a==print)
```

Output:

```
false
false
```

Only:

- Numbers
- Strings
- Objects
- Functions

- '~=' Inequality evaluates to the direct negation of the equality operator
  - Comparisons between two values of different types will always return true

Input:

```
a, b = 4,8
print(a~=b)
print(a~=print)
```

Output:

```
true
true
```

Less than /equal to & Greater than / equal to

- ‘<’, ‘<=’ Less than / equal to determines if the first value is smaller than (or equal to) the second

Input:

```
a, b = 4,8
print(a<b)
print(a<=b)
print(a<=a)
```

Output:

```
true
true
true
```

- ‘>’, ‘>=’ Greater than / equal to determines if the first value is larger than (or equal to) the second

Input:

```
a, b = 4,8
print(a>b)
print(a>=b)
print(a>=a)
```

Output:

```
false
false
true
```

Only:

- Numbers
- Strings

3. Logical

Logical AND, OR & NOT

- ‘and’ Logical AND returns the 1st argument if it is false, otherwise it returns the 2nd
  - Uses short-circuit evaluation

Input:

```
print(true and true)
print(false and (print("Short Circuit AND: Does not execute")))
print(1 and 2)
```

Output:

```
true
false
2
```

- ‘or’ Logical OR returns the 1st argument if it is true, otherwise it returns the 2nd
  - Uses short-circuit evaluation

Input:

```
print(true or false)
print(true or (print("Short Circuit OR: Does not execute")))
print(1 or 2)
```

Output:

```
true
true
1
```

- ‘not’ Logical NOT returns true or false based on the negation of the expression

Input:

```
print(not (1>2))
```

Output:

```
true
```

Operator Precedence

Order of Precedence

- All binary operations are left associative but exponentiation and concatenation
- Operations enclosed within parentheses are always evaluated first

Operator precedence in Lua follows the table below, from lower to higher priority:

```
or
and
<     >     <=    >=    ~=    ==
|
~
&
<<    >>
..
+     -
*     /     //    %
unary operators (not   #     -     ~)
^
```

Control Structures

If Statements - part 1

- All values except ‘nil’ and ‘false’ are considered ‘true’
- Condition doesn’t require parentheses
- Body is delimited by ‘then’ keyword and ‘end’ terminator

Input:

```
--if (true)
flag = 0; --optional semicolon separator
if (flag) then print("flag is not false") end --all on one line
```

Output:

```
flag is true
```

- ‘else-part’ of ‘if’
  - Only executes when ‘if’ is false

Input:

```
--if (false)
flag = false
if (flag) then print("flag is not false") else print("flag is
false") end
```

Output:

```
flag is false
```

If Statements - part 2

- Can nest other ‘if’ statements inside the ‘then-part’ or ‘else-part’ of an existing ‘if’ statement
  - Each ‘if’ statement requires its own ‘end’ terminator

Input:

```
--nested ifs
a, b = 1, 2
if a < 0 then
    print("a is less than 0")
    if b < a then
        print("b is less than a; b is also less than 0")
    end
else
    print("a is greater than 0")
    if b > a then
        print("b is greater a; b is also greater than 0")
    end
end
```

Output:

```
a is greater than 0
b is greater a; b is also greater than 0
```

- Supports ‘elseif’ statements
  - Only evaluates when ‘if’ is false

Input:

```
--elseifs
b = 2
if b < 0 then
    print("b is less than 0")
elseif b > 1 then
    print("b is greater than 1")
else
    print("b is between 0 and 1")
end
```

Output:

```
b is greater than 1
```

While & Repeat-until Loops

- ‘While’ loops are delimited by ‘do’ keyword and ‘end’ terminator

Input:

```
counter = 1
while counter <= 10 do
    if counter == 5 then
        break
    end
    print(counter)
    counter = counter + 1
end
```

Output:

```
1
2
3
4
```

- Lua doesn’t have ‘do-while’ loops but its ‘repeat-until’ loops are functionally very similar
  - Body always executes at least once

Input:

```
counter = 1
repeat
    print("loop executes at least one, even though condition is
already true")
until counter >= 1
```

Output:

```
loop executes at least one, even though condition is already true
```

- Lua supports ‘break’ statements

Numeric For Loops

- Lua has two types of ‘for’ loops
- ‘Numeric for’ loops are similar to traditional ‘for’ loops (Java, C, etc.)
  - Three components: initial value, limit, and step
  - (Limit is similar to ‘>=’ when decrementing initial value & similar to ‘<=’ when incrementing it)
  - Loop delimited by ‘do’ keyword and ‘end’ terminator

Input:

```
for i=2,20,2 do
    print(i)
    if(i == 10) then
        goto skip
    end
end
print("*loop end*")

::skip::

print("*program end*")
```

Output:

```
2
4
6
8
10
*program end*
```

- Lua supports ‘goto’ statements
  - Transfers control of program to specified label
    - Label is delimited by ‘:: label_name ::’

Generic For Loops

- ‘Generic for’ loops are somewhat similar to ‘for’ loops in Python
  - Iterates through a sequence with two variables
    - Two iterator functions for tables
      - ipairs: variables returned as index and value
      - pairs: variables returned as key and value

Input:

```
table = {"dog", "cat", "monkey", "lion", "bear"}

for i, v in ipairs(table) do
    print(i, v)
end
```

Output:

```
1       dog
2       cat
3       monkey
4       lion
5       bear
```

Functions

Functions

- Defined with the ‘function’ keyword followed by its name, arguments (which are enclosed in parentheses), body, and ‘end’ terminator
  - Arguments can be specified as ‘...’ to indicate a variable number of arguments like print() has

Input:

```
function getSum(array)
    sum = 0
    for i,v in ipairs(array) do
        sum = sum + v
    end
    return sum
end

odds = {1, 3, 5, 7, 9}
print(getSum(odds))
```

Output:

```
25
```

- Lua allows functions to return multiple values
  - The limit on the number is “greater than 1000”
  - Values which exceed the number of specified variables will be discarded

Input:

```
function multiReturn()
    s1="This "
    s2="is "
    s3="all "
    s4="one "
    s5="return."
    return s1, s2, s3, s4, s5
end

a,b,c,d = multiReturn()
print(a..b..c..d)
```

Output:

```
This is all one
```

Coroutines

- Coroutines utilize threads for “collaborative multithreading”
  - They do not run concurrently but can yield execution so other threads can run
    - The suspended thread will be later be resumed from the position it had yielded at
  - Stores functions which are performed when the coroutine is called to
  - Thread is marked ‘dead’ after coroutine has completed its function
    - The ‘dead’ memory will automatically be freed by Lua’s automated garbage collection

Input:

```
thread_1 = coroutine.create(
    function()
        print("Coroutine is executing. (x1)")
        coroutine.yield() --coroutine yields for other threads
        print("Coroutine is executing. (x2)")
        coroutine.yield() --coroutine again yields for other threads
        print("Coroutine is executing. (x3)")
    end
)

--Thread Information
print("data type: "..type(thread_1))
print(thread_1) --prints the thread's address
print("thread’s coroutine status: "..coroutine.status(thread_1)..."\n")

coroutine.resume(thread_1) --starts the thread's coroutine
coroutine.resume(thread_1) --restarts the thread's coroutine
coroutine.resume(thread_1) --restarts the thread's coroutine
print("thread’s coroutine status: "..coroutine.status(thread_1)) --now
that the coroutine has completed
```

Output:

```
data type: thread
thread: 000000000142e2f8
thread’s coroutine status: suspended

Coroutine is executing. (x1)
Coroutine is executing. (x2)
Coroutine is executing. (x3)
thread’s coroutine status: dead
```

Conclusion

The End

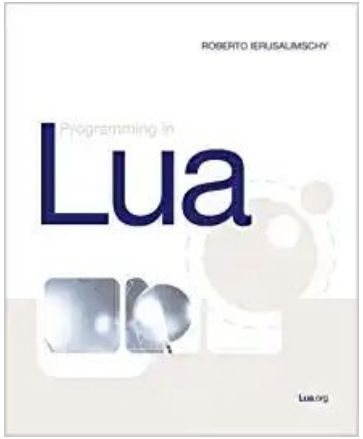
- Lua has many unique capabilities
  - Expressive tables
  - Threads and coroutines
  - Multiple returns from a single function
  - Two types of for loops
  - and more…

Lua 5.4 Reference Manual

The reference manual is the official definition of the Lua language.
For a complete introduction to Lua programming, see the book Programming in Lua.

start · contents · index · other versions

Contents



- To learn more about Lua check out the ‘Lua 5.4 Reference Manual’ and the ‘Programming in Lua’ online textbook provided at lua.org