Learning Lua

This introduction to Lua is a fast and furious overview of the main parts of Lua that you will need for creating your first game with Lua and LÖVE.

It's largely inspired by Learn Lua in 15 minutes and it mainly focused on two type of readers:

- You already know other programming languages and just want to have a peek at what Lua looks like.
- You want to start as fast as possible to program the game and want to learn the language by doing it.

In both cases you will want to come back to this section each time you don't understand the code snippets for the game. If you don't find the answer in this chapter, you might want to have a look at the Lua Manual or you might need a more gentle and didactical introduction to Lua.

If you want a more gently and didactical learning experience, you might want to read Programming in Lua before further reading this manual.

Running Lua programs

As explained in the Getting started chapters for Linux, Mac OS X, and Windows, depending on your operating system and your personal preferences you can run your program by:

- Linux / terminal
- Windows / OS X / UI

You're invited to create a test Lua file and retype in there each snippet you'll find in this chapter and try it out to check how it works by running the file.

Comments

Comments are "programmer-readable annotation[s] in the source code of a computer program. They are added with the purpose of making the source code easier to understand, and are generally ignored" when runnning the program (Wikipedia, Comment)

Lua has single line and multiple line comments

```
-- Two dashes start a one-line comment.

--[[
     Adding two ['s and ]'s makes it a
     multi-line comment.
--]]
```

Variables

```
num = 42  -- All numbers are doubles.
-- Don't freak out, 64-bit doubles have 52 bits for
-- storing exact int values; machine precision is
-- not a problem for ints that need < 52 bits.

s = 'walternate'  -- Immutable strings like Python.
t = "double-quotes are also fine"
u = [[ Double brackets
       start and end
       multi-line strings.]]
t = nil  -- Undefines t; Lua has garbage collection.
```

Flow control

```
-- Blocks are denoted with keywords like do/end:
while num < 50 do
  num = num + 1  -- No ++ or += type operators.
end

-- If clauses:
if num > 40 then
  print('over 40')
elseif s ~= 'walternate' then  -- ~= is not equals.
  -- Equality check is == like Python; ok for strs.
  io.write('not over 40\n')  -- Defaults to stdout.
else
  -- Variables are global by default.
  thisIsGlobal = 5  -- Camel case is common.

  -- How to make a variable local:
  local line = io.read()  -- Reads next stdin line.

  -- String concatenation uses the .. operator:
  print('Winter is coming, ' .. line)
end

-- Undefined variables return nil.
-- This is not an error:
foo = anUnknownVariable  -- Now foo = nil.

aBoolValue = false

-- Only nil and false are falsy; 0 and '' are true!
if not aBoolValue then print('twas false') end

-- 'or' and 'and' are short-circuited.
-- This is similar to the a?b:c operator in C/js:
ans = aBoolValue and 'yes' or 'no'  --> 'no'
```

```
karlSum = 0
for i = 1, 100 do  -- The range includes both ends.
  karlSum = karlSum + i
end

-- Use "100, 1, -1" as the range to count down:
fredSum = 0
for j = 100, 1, -1 do fredSum = fredSum + j end

-- In general, the range is begin, end[, step].

-- Another loop construct:
repeat
  print('the way of the future')
  num = num - 1
until num == 0
```

Functions

~.lua function fib(n) if n < 2 then return 1 end return fib(n - 2) + fib(n - 1) end

-- Closures and anonymous functions are ok: function adder(x) -- The returned function is created when adder is -- called, and remembers the value of x: return function (y) return x + y end end a1 = adder(9) a2 = adder(36) print(a1(16)) --> 25 print(a2(64)) --> 100

-- Returns, func calls, and assignments all work -- with lists that may be mismatched in length. -- Unmatched receivers are nil; -- unmatched senders are discarded.

x, y, z = 1, 2, 3, 4 -- Now x = 1, y = 2, z = 3, and 4 is thrown away.

function bar(a, b, c) print(a, b, c) return 4, 8, 15, 16, 23, 42 end

x, y = bar('zaphod') --> prints "zaphod nil nil" -- Now x = 4, y = 8, values 15..42 are discarded.

-- Functions are first-class, may be local/global. -- These are the same: function f(x) return x * x end f = function (x) return x * x end

-- And so are these: local function g(x) return math.sin(x) end local g; g = function (x) return math.sin(x) end -- the 'local g' decl makes g-self-references ok.

-- Trig funcs work in radians, by the way.

-- Calls with one string param don't need parens: print 'hello' -- Works fine.

Scope

Lua has global and local variables.

```
value = 1 -- define a global value with value 1

function testScope()
    value = 100 -- set the global value to 100
    for value = 1, 5 do -- define a local value that is valid inside of the
for loop
        print(value) -- print the local value (1,2,3,4,5)
    end
    print(value) -- print the global value (100)
    local value -- define a local value valid inside of the function
    value = 10 -- set the local value to 10
    print(value) -- print the local value (10)
end

print(value) -- print the global value (1)
testScope()
print(value) -- print the global value (100, modified by function)
```

Lists and structures

Lua is a very simple language and what you learn in this chapter should be enough for creating your first game. But there is more to discover than what you've seen in this short chapter: in the "Learning more Lua" chapter you can find a few further constructs that you will probably need if you want to create more complex games with Lua and LÖVE.

We finish this chapter with a short overview of lists and structures, which are implemented as tables. You'll find more information on tables in the Learning more Lua chapter.

```
-- Tables = Lua's only compound data structure;
--          they are associative arrays.
-- Similar to php arrays or js objects, they are
-- hash-lookup dicts that can also be used as lists.

-- Using tables as dictionaries / maps:

-- Dict literals have string keys by default:
t = {key1 = 'value1', key2 = false}

-- String keys can use js-like dot notation:
print(t.key1)  -- Prints 'value1'.
t.newKey = {}  -- Adds a new key/value pair.
t.key2 = nil   -- Removes key2 from the table.

for key, val in pairs(u) do  -- Table iteration.
  print(key, val)
end
```

```
-- Using tables as lists / arrays:

-- List literals implicitly set up int keys:
v = {'value1', 'value2', 1.21, 'gigawatts'}
for i = 1, #v do  -- #v is the size of v for lists.
  print(v[i])  -- Indices start at 1 !! SO CRAZY!
end
-- A 'list' is not a real type. v is just a table
-- with consecutive integer keys, treated as a list.
table.insert(v, 'newValue')
```