Spoken in Portuguese 

A Gentle Introduction to Lua

The minimalistic embeddable language, not the Earth's Moon

PiXELS Camp

Carlos Rodrigues — October 2016

First, Some History…

- Evolved from earlier languages used in engineering applications at Petrobras
- Designed from the start as a minimalistic scripting language for embedded uses
- First public release in **1994**
- First used to script a (large) game in **1997***

*www.grimfandango.net/features/articles/lua-in-grim-fandango

Features

- Simple and easy to learn syntax

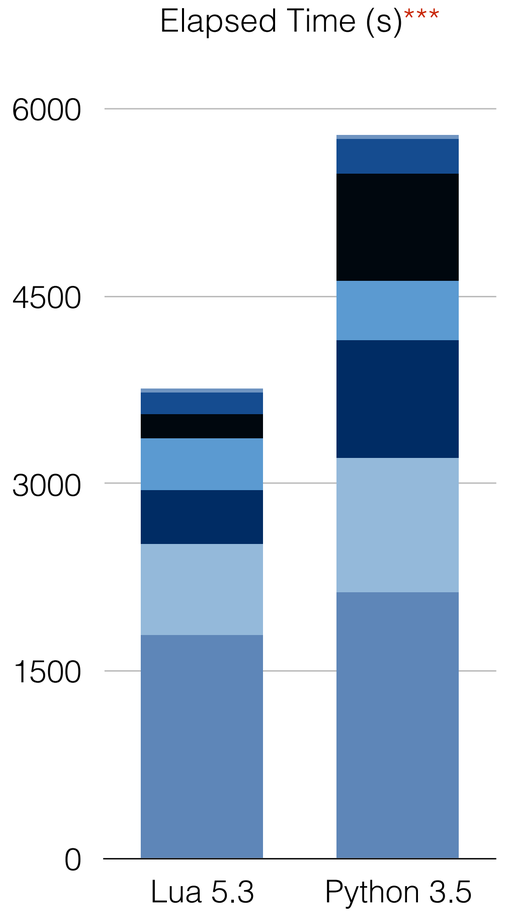
  …but flexible enough for advanced usage
  …**minimalism** is a primary design goal

- Small and fast implementations

  …the complete reference runtime* is **~200 KB**
  …and LuaJIT** can be **many times** faster
  …with no restrictions on commercial use (MIT licensed)

- Runs everywhere

  …only requirement is a standard C compiler



*lua.org
**luajit.org
***bit.ly/2d9kRA4

About Versions…

- Continuous development of new features

  …which may break source and binary compatibility
  …older versions only get bug fixes

- 5.1 is still the most popular language version

  …although no longer maintained by the Lua core team
  …because LuaJIT is **source** and **binary** compatible to it

The Lua **experience** is defined by the **environment** it is embedded in.

The Basics

```lua
-- unknown variables are nil
print(a)

-- semicolons are optional
a = 1; a = a + 1; print(a)

-- only nil and false are false
if "" and 0 and {} then
    print("true")
end

-- there's only one number type
if type(1) == type(1.1) then
    print("true")
end
```

```lua
--[[ tables are the only structured type and
     can pose as arrays and dictionaries ]]
local t1 = {1, 2, 3, 4}
local t2 = {["key one"]=1, key_two=2, [3]=3}

-- array indexes start at "1" (don't ask)
for i=1, #t1 do
    print(i .. "->" .. t1[i])
end

-- nil deletes things
a = nil
t1 = nil
t2["key one"] = nil
```

Scoping Rules

```lua
-- variables are global by default
a = 1
do
    local a = a + 1
end
print(a)  -- “1”

-- loop variables are local
for i=0, 10, 1 do
    print(string.format("i=%d", i))
end
print(i)  -- “nil”
```

```lua
-- scoping is lexical (static)
local f = (function()
    local n = 0

    return function()
        n = n + 1  -- “n” is an upvalue
        return n
    end
end)()  -- “f” becomes a closure

print(f())  -- “1”
print(f())  -- “2”
```

About Functions

```lua
-- same as "f = function(…)"
function f(n)
    return n + 1
end

-- same as "local f = function(…)"
local function f(n)
    return n + 1
end

-- returning multiple values...
function f(a, b)
    return a, b
end
```

```lua
-- ...is assignment, not destructuring
a, b = f(1, 2)

-- extra captures are nil
a, b, c = f(1, 2)  -- "c" is nil

-- missing captures are discarded
a = f(1, 2)

-- same rules for function arguments
a, b = f(1)        -- "b" is nil
a, b = f(1, 2, 3)  -- "3" is discarded
```

About Tables

```lua
-- they stand in for named parameters
function f(t)
    return t.x + t.y
end

-- parentheses optional for one table
local a = f{x=1, y=2}

-- they stand in for lists/sequences
local t = {“a"}

table.insert(t, "b")  -- {a, b}
table.insert(t, "c")  -- {a, b, c}
table.remove(t, 2)    -- {a, c}

print(table.concat(t, ":"))  -- “a:c”
```

```lua
-- tables have a special array part
local t = {1, 2, b="bee", 3, a="aye"}

-- loops over numeric keys*
for i=1, #t do
    print(t[i])  -- "1", "2", "3"
end

-- iterates over numeric keys* (ordered)
for i, v in ipairs(t) do
    print(i .. "->" .. v)
end

-- iterates over all keys (unordered)
for k, v in pairs(t) do
    print(k .. "->" .. v)
end
```


*Watch out for “holes.”

Modules

```lua
-- contents of “module.lua”

local function add(a, i)
    return a + i
end

return {  -- export
    add = add
}
```

```lua
local t = {}

-- same as “t.add = function(…)”
function t.add(a, i)
    return a + i
end

return t  -- export
```

```lua
-- load the module (in another file)
local m = require "module"

print(m.add(2, 1))  -- “3”
```

Classes

```lua
-- contents of "class.lua"
local Class = {}

-- look for unknown attributes in "Class"
Class.__index = Class

function Class.new(n)
    -- the new instance is an empty table
    local self = setmetatable({}, Class)
    self.n = n; return self
end

function Class.increment(self, i)
    self.n = self.n + i; return self.n
end

return Class
```

```lua
-- load the module (i.e. class)
local Class = require “class"

local obj = Class.new(2)
print(obj.increment(obj, 1))  -- “3”

-- more convenient syntax
print(obj:increment(1))       -- “4"
```

About Metamethods

- many metamethods available
  `__index`, `__newindex`
  `__call`, `__tostring`
  `__len` (Lua 5.2)
  `__add`, `__sub`, `__mul`, …
  `__eq`, `__lt`, `__le`

- metatables can be chained
  …think class inheritance

- metamethods can involve C

```
local mt = {
    __sub = function(a, b)
        if #a ~= #b then
            error("length mismatch", 2)
        end

        local r = {}
        for i=1, #a do r[i] = a[i] - b[i] end
        return r
    end
}

local t1 = {1, 2, 3}; local t2 = {1, 1, 1}
setmetatable(t1, mt); setmetatable(t2, mt)

print(table.concat(t1 - t2, ", "))  -- “0, 1, 2”
print(table.concat(t2 - t1, ", "))  -- “0, -1, -2”
```

Coroutines

```lua
-- coroutines are interruptible functions
function f(limit)
    local i = 1
    while i <= limit do
        print(i)
        coroutine.yield()
        i = i + 1
    end
end

local co = coroutine.create(f)
local status, msg = coroutine.resume(co, 10)

while status do  -- resume until finished (dead)
    status, msg = coroutine.resume(co)  -- “1”, “2”, ..., “10”
end
print("finished: " .. msg)  -- "cannot resume dead coroutine"
```

Coroutines as Iterators

```lua
function range(start, stop)
    local function generator(start, stop)
        for n=start, stop - 1 do
            coroutine.yield(n)
        end
    end

    -- create doesn't take extra arguments
    local co = coroutine.create(function()
        generator(start, stop)
    end)

    return function()
        local _, n = coroutine.resume(co)
        return n
    end
end
```

```lua
function range(start, stop)
    --[[...]]

    -- shortcut for double wrapping
    return coroutine.wrap(function()
        generator(start, stop)
    end)
end
```

```lua
for i in range(0, 10) do
    print(i)  -- "0", ..., "9"
end
```

Other Iterators

```lua
-- keep state using closures
function range(start, stop)
    local n = start - 1

    return function()
        n = n + 1

        if n >= stop then
            return nil
        end

        return n
    end
end
```

```lua
-- implicit state in the “generic for”
function range(start, stop)
    local function iterator(stop, current)
        if current >= stop then
            return nil
        end

        return current + 1
    end

    -- “stop” is the loop invariant
    return iterator, stop, start
end
```

```lua
for i in range(0, 10) do
    print(i)  -- “0”, ..., “9”
end
```

Next time you bump into something that embeds Lua, take a **closer look**.

Thanks!

Any questions?

Carlos Rodrigues
cer@brpx.com

These slides are available at speakerdeck.com/carlosefr