Lua for Programmers Part 3: More Advanced Concepts

Posted on Sep 07, 2012 in Tutorials

If you haven't read part one and two already, I'd highly recommend doing so. For reference, here's a list of the parts in this series:

- Part 1: Language Essentials, covers fundamental syntax and concepts such as operators, loops, and functions.
- Part 2: Data and Standard Libraries, covers Lua's built-in data types and some of the standard libraries.
- Part 3: More Advanced Concepts, the current part; deals with things like variable scope, advanced functions, and file loading.
- Part 4: Tips and Tricks, a collection of small things that you may find useful.

Blocks and Scope

In part one I stated that Lua has "braces of a sort in the form of keywords like `then` and `do`." Much like a language with C-based syntax, these "braces" represent blocks, which are essentially sequences of statements. An example of some blocks in Lua:

```
if x then
  stuff()
  moreStuff()
end

for i = 1, 10 do
  local x = "foo"
end

function foo(x, y)
  -- ...
end

-- explicit block
do
  local x = 3
```

```
  local y = 4
end
```

Conditionals, loops, and functions are all blocks. You can explicitly define a block as shown by the last example.

Lua uses lexical scoping, which means that every block has its own scope:

```
x = 5 -- global

function foo()
  local x = 6
  print(x) -- 6

  if x == 6 then
    local x = 7
    y = 10 -- global
    print(x) -- 7
  end

  print(x, y) -- 6, 10

  do
    x = 3
    print(x) -- 3
  end

  print(x) -- 3
end

foo()
print(x, y) -- 5, 10
```

This example demonstrates the effect of variable shadowing. It also demonstrates how global variables can be defined at any scope if the `local` keyword is absent and no local variables exist by that name (as is the case with `y`).

Collapsing Blocks

Lua allows you to collapse blocks onto one line:

```
function foo(x) return x * 5 end
if x then print(x) end
do foo() end
```

You'll see this used a lot in Lua code.

More On Functions

Default Arguments

As mentioned in part one, Lua doesn't directly support default arguments, but there are ways of getting the same behaviour. If a value isn't supplied for a function argument it will default to `nil`, as all undefined variables do. Therefore:

```
function func(x, y, z)
  if not y then y = 0 end
  if not z then z = 1 end
  -- code
end
```

A more common method is to use the `or` operator instead:

```
function func(x, y, z)
  y = y or 0
  z = z or 1
end
```

Table Syntax

There's an interesting syntax for calling functions that take a table as their only argument:

```
function foo(t)
  return t[1] * t.x + t[2] * t.y
end

foo{3, 4, x = 5, y = 6} -- 39
```

That function call is equivalent to:

```
foo({ 3, 4, x = 5, y = 6 })
```

Variable Arguments

Functions support the capturing of a varying number of arguments:

```lua
function sum(...)
  local ret = 0
  for i, v in ipairs{...} do ret = ret + v end
  return ret
end

sum(3, 4, 5, 6) -- 18
```

To do so, just put `...` at the end of your argument list. You can access members of the list of arguments by putting them in a table (`{...}`) or through the `select` function:

```lua
function sum(...)
  local ret = 0
  for i = 1, select("#", ...) do ret = ret + select(i, ...) end
  return ret
end
```

To put it briefly, `select` returns all values after the specified index, or the length of the list if `"#"` is provided instead. See the documentation for more.

Finally, just to be clear, `...` isn't a data structure of any kind, it's merely a list of values, like what you get from functions returning multiple values.

`self`

Lua gives us a neat piece of syntactic sugar for calling and defining functions. When you have a function inside a table, you can do stuff like this:

```lua
t = {}

function t:func(x, y)
  self.x = x
  self.y = y
end

t:func(1, 1)
print(t.x) -- 1
```

The definition and call translate to:

```lua
function t.func(self, x, y)
  self.x = x
```

```
  self.y = y
end

t.func(t, 1, 1)
```

Although this is unnecessary in the example above, it becomes incredibly useful when mixed with metatables to implement things like object-oriented programming.

Loading Files

An important part of any language is how code is separated into multiple files. As you may know, Lua files generally end with the `.lua` extension. There are a few ways to load and run a file from inside a Lua script. Two of those ways are `dofile` and `loadfile`:

```
dofile("test.lua")
loadfile("test.lua")()

func = loadfile("test.lua")
func()
```

All of those methods are essentially equivalent. Lua files are loaded as functions; this explains why `loadfile` returns a function. In addition to doing the job of `loadfile`, `dofile` also calls the function (if no errors occurred when compiling the file).

This method of loading files as functions opens up some interesting possibilities:

```
local message = "Cheese for everyone!"
local t = { 1, 2, 3, 4, 5 }
print(message)
return t
```

As you can see, files can return values and define local variables. Here's how we might go about loading such a file:

```
x = dofile("cheese.lua")
```

`x` would be set to `t`, and the message from `cheese.lua` would be printed to the console.

Package System

A much better and more common way to load files is via the package system:

```
require("cheese")
require("folder.subfolder.file")
```

The `require` function takes a path to a Lua file and searches for it in a number of ways. The manual explains the whole process, so I'll just tell you what you need to know. `require` replaces each dot with the directory separator ("/" on Unix systems). It then looks for this file in the locations specified by `package.path`, which includes the current directory.

Once the file's been loaded, `require` adds the path you specified to the `package.loaded` table. `require` won't load any path found in this table, ensuring that files will only be loaded once.

That's the essence of the package system, but there's a lot more to it, so be sure to check out the documentation.

Executing Strings

If you want to compile a string as a function within a Lua script, `loadstring` does the job:

```
loadstring("print('hello')")()
```

It works the same way as `loadfile`. This means that you can have local variables and return values inside the strings too.

Metatables

Metatables are a very powerful and flexible feature of Lua that allow you to modify how tables behave. With them you can implement object-oriented programming, advanced data structures, and a lot more. Since I've already written a rather comprehensive tutorial on the subject, I'll direct you to that instead of explaining metatables here. Another source you could read is, of course, Programming in Lua.

Conclusion

As with the last part, there are many things I didn't cover that you might want to check out yourself; some examples are iterators, coroutines, and error handling. Be sure to check out the final part of the series too.

As always, if you have any feedback, I'd love to hear from you.

« Lua for Programmers Part 2: Data and Standard Libraries | Lua for Programmers Part 4: Tips and Tricks »

Latest Posts

Manually Installing phpMyAdmin (Ubuntu 18.04)
Jan 17, 2019

Installing an Up-To-Date LAMP Stack (Ubuntu 18.04)
Jan 12, 2019

Projects Over the Last Few Years
Jul 17, 2018

Website Overhaul
Jul 10, 2018

Ludum Dare 27: Triad
Aug 29, 2013

Categories

Tutorials (27)

Projects (21)

General (4)

Posts by Year

2019 (2)

2018 (2)

2013 (1)

2012 (12)

2011 (34)

2010 (1)

Copyright © 2019 Michael Ebens.
Please read the licence page for more details about the licencing of this website's content.