Lua for Programmers Part 1: Language Essentials

Posted on Aug 27, 2012 in Tutorials

In this series we'll be taking a look at most everything you should know to program in Lua. The series is targeted at people who already know how to program, and as such I'll aim to be brief in my explanations.

Here's an overview of the four parts the compose the series:

- Part 1: Language Essentials, the current part; 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, 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.

What is Lua?

In case you didn't already know, Lua is an interpreted programming language. It's fast, flexible, embeddable, simple, and easy to learn. You can get Lua from its download page.

Some Learning Resources

Two resources that I would recommend for further learning of Lua is the Programming in Lua book (they have the first edition available online) coupled with the manual. However, since Lua 5.1 is still the dominant Lua version, I'll be referring to the 5.1 manual.

A couple other websites you might find useful are the lua-users wiki, and the Unofficial Lua FAQ.

If you've downloaded Lua and run it from the terminal you'll be able to use an interactive interpreter. This is a very handy tool for learning and experimentation. Here's an example:

```
$ lua
Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio
> x = 0
> while x < 10 do
>> x = x + 2
>> print(x)
>> end
2
4
6
8
10
> return x
10
>
```

General Syntax

```
--[[
This is
a block comment
]]

if state == 5 then
  doSomething() -- this is a line comment
elseif foo then
  a = true
else
  b = false
end
```

There's a small example of Lua syntax. Lua doesn't use semicolons, nor curly braces. However, it does have braces of a sort in the form of keywords like `then` and `do`. The code above also demonstrates conditional statements.

Variables and Types

Like many other interpreted languages, Lua is typeless, meaning that declaring a variable is as simple as `var = value`. This creates the variable *in the global scope*; we'll soon see how to declare local variables. You can also declare multiple variables at once: `var1, var2, var3 = val1, val2, val3`.

There are a few constants, or, types, that you should be aware of:

- `nil` is the nothing value, and also represents undefined variables. While languages such as ActionScript have both a `null` and `undefined` value, Lua combines these two things into one with `nil`.
- `true` and `false` are present. Beware that in conditional statements, the only things that will equate to `false` are `nil` and `false` itself; everything else is considered `true`.

We'll delve more into Lua's native types in part two.

Operators

Lua has a mostly standard collection of operators; here they are in increasing precedence (source):

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

Here are the differences between your standard C-based operator set:

- The logical AND, OR, and NOT operators are the keywords `and`, `or`, and `not`, respectively.
- The inequality operator is `~=` instead of `!=`.
- `..` is the operator for string concatenation.
- `^` raises a number to a power, for example `10 ^ 2` would yield 100 (or, ten-squared). In most other languages, this performs the binary XOR operation.
- `#` is an operator used to get the length of tables and strings. We'll see this used in part two.

If you're wondering where bitwise operators are, there aren't any. However, Lua 5.2 added the `bit32` library which you can use for bitwise operations. If you're using earlier versions of Lua check out this page for some solutions.

Finally, there are two things to note about assignment. First, compound assignment isn't supported, thus an expression such as `i += 3` must be written as `i = i + 3`. Second, the syntax of the operator itself is quite limited. You can't do things like `foo(a = 3)` or `a = b = 1`.

Loops

You've already seen `if` statements, so let's move on to loops. There are four different kinds in Lua.

`while`

```lua
i = 1

while i <= 5 do
  i = i + 1
  print(i)
end
```

Your traditional `while` loop. As you might guess, `print` is the function used to write to the standard output stream.

`repeat`

```lua
i = 5

repeat
  i = i - 1
  print(i)
until i == 1
```

Repeats its body *until* the condition is true, and is guaranteed to run the body at least once. Quite like a `do..while` loop.

Numeric `for`

Lua is a bit different when it comes to the `for` loop. There are two different variations of it, numeric and generic. We'll see the generic variant in part two.

```lua
x = 1

for i = 1, 11, 2 do
  x = x * i
  print(x)
end
```

This loop has the following syntax:

```lua
for var = start, limit, step do
  -- code here
end
```

The equivalent `while` loop would be:

```lua
var = start

while var <= limit do
  -- code here
  var = var + step
end
```

Or in a C `for` loop:

```c
for (int var = start; var <= limit; var += step) {}
```

Remember that `step` also takes negative numbers, so you can progress backwards too.

Loop Termination

If you want to prematurely exit from a loop you can use the `break` keyword:

```lua
while true do
  if condition then
    x = x ^ 2
  else
    break
  end
end
```

Lua doesn't have the `continue` keyword that many languages have.

Functions

```lua
function foo()
  local x, y = something(4, 5)
  return x ^ y
end

function something(arg1, arg2)
  local ret1 = (arg1 * arg2) ^ 2
  local ret2 = (arg1 - arg2) ^ 2
  return ret1 + ret2, ret1 * ret2
end
```

That's how you declare functions. As you can see, the order of declaration in a file doesn't matter as it does in some compiled languages such as C and C++. There are three things I want to point out:

- Lua is a bit different in how local variables are declared. If you want to do so, you need to prefix the declaration with the `local` keyword. We'll learn more about variable scope in part three.
- Functions can return multiple values. `foo` demonstrates how to capture these values.
- Default arguments aren't directly supported, but there are ways of achieving the same effect which we'll see in part three.

Conclusion

Well, that's it for now. Be sure to check out part two and the other parts of the series as well. And if you have any feedback, I'd love to hear it.

programming lua

« Illusive Dreams: Test Build 5 | Lua for Programmers Part 2: Data and Standard Libraries »

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)