Lua for Programmers Part 2: Data and Standard Libraries

Posted on Aug 27, 2012 in Tutorials

If you haven't read part one 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, the current part; 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.

Tables

Tables are essentially associative arrays which can have keys and values of any type. Tables are incredibly flexible, and are the *only* data structure in Lua, so knowledge of them is crucial.

Here's an example of how to use them:

```
x = 5
a = {} -- empty table
b = { key = x, anotherKey = 10 } -- strings as keys
c = { [x] = b, ["string"] = 10, [34] = 10, [b] = x } -- variables and literals

-- assignment
a[1] = 20
a["foo"] = 50
a[x] = "bar"

-- retrieval
print(b["key"]) -- 5
print(c["string"]) -- 10
print(c[34]) -- 10
print(c[b]) -- 5
```

Curly brackets are used to create them. You can use any kind of expression as a key as long as it's surrounded by square brackets. If you'd like the key to be a string then you don't need square brackets or quotes, as demonstrated by the definition of `b`.

Syntactic Sugar

Tables have some awesome syntax when it comes to string keys:

```
t = { foo = 1, bar = 2 }
print(t.foo) -- 1
t.bar = 3
```

This makes many uses of tables a lot more visually pleasing. One example is the Lua standard libraries; instead of writing something like `math["sqrt"](x)` we can write `math.sqrt(x)`.

Arrays and Indices

Lua has additional support for traditional, integer-indexed arrays:

```
a = { 11, 22, "foo", "bar" }
a[3] = "foooo"

print(a[1]) -- 11
print(a[3]) -- foooo
print(#a) -- 4
```

If you don't supply keys along with values when creating a table they will be automatically assigned indices. Also notice how the length operator returns the number of elements in a table.

Lua is different from most languages when it comes to indices; they start at 1 rather than 0. That means that instead of half-open ranges (`[0, length)`), closed ranges (`[1, length]`) are used. This concept is built into Lua at multiple levels, so although it's possible to still use indices starting at 0, it's highly impractical.

Generic `for` Loop

Now that we know about tables, we can take a look at Lua's final loop type: generic `for`. As Programming in Lua states, it "allows you to traverse all values returned by an iterator function." The details of implementing an iterator function, and how this all works is a little complicated, but using it with tables is very simple. Here's some examples:

```lua
a = { x = 400, y = 300, [20] = "foo" }
b = { 20, 30, 40 }

for key, value in pairs(a) do
  print(key, value)
end

for index, value in ipairs(b) do
  print(index, value)
end
```

`pairs` is an iterator function which loops through each item and gives you both the key and the value. `ipairs` is like `pairs` except it only loops through items with indices, starting at 1 and continuing until it encounters a `nil` value. Here's what that example outputs when run:

```
y    300
x    400
20   foo
1    20
2    30
3    40
```

`table` module

If you're working with tables as arrays then the `table` module, part of Lua's standard library, will definitely help out. Here's an example of a few of the functions:

```lua
t = { 24, 25, 8, 13, 1, 40 }
table.insert(t, 50) -- inserts 50 at end
table.insert(t, 3, 89) -- inserts 89 at index 3
table.remove(t, 2) -- removes item at index 2
table.sort(t) -- sorts via the < operator
```

Strings

Strings work as you'd expect; you can declare them with double or single quotes, and use backslash to escape characters. For strings spanning multiple lines you can use square brackets in the format `[[...]]`. Here's an example:

```lua
s = "foo\nbar"
t = 'he said "hello world"'
u = "Hello \"world\""
```

```
v = [[
<html>
  <body>
    <p>Hello world!</p>
  </body>
</html>
]]
```

Programming in Lua's section on strings lists all the escape characters and gives more information.

`string` module

The standard library provides the `string` module for working with strings. Let's take a look at a few of the more simple functions:

```
string.lower("HeLLO") -- hello
string.upper("Hello") -- HELLO
string.reverse("world") -- dlrow
string.char(87) -- W
string.sub("hello world", 2, 5) -- ello
```

There's also a number of functions that work with Lua's string patterns. Strings patterns are somewhat like regular expressions, except less powerful and with a different syntax. I won't give an explanation for them here, the manual does a pretty good job of that, but rather some examples:

```
string.gsub("hello 42", "(%d+)", "%1 3") -- hello 42 3
string.gsub("heLLo", "(%u)", "") -- heo

-- 4 + 4 = 8
string.gsub("2 + 2 = 4", "(%d)", function(s)
  return s * 2
end)

-- prints each word
for w in string.gmatch("good morning chaps", "%w+") do
  print(w)
end
```

Numbers and Maths

Numbers are quite simple in Lua. There's no distinction between doubles, floats, integers, or unsigned integers; everything is a double (or, double-precision floating-point). Programming in Lua states that doubles have no rounding errors when representing integers, and are also very efficient on modern CPUs. Since all numbers are the same type, there's no type conversion to worry about when performing arithmetic.

Here's a few examples of how numbers can be written: `4`, `0.4`, `.4`, `4.3`, `0xFF`, `0xA33FF0E`, `8.7e12`, `8.7e+12`, `8e-12`.

The `math` module contains many of the mathematical functions you would expect from a language such as `math.sin`, `math.cos`, `math.floor`, etc. There's a few things you'll want to be aware of:

- `math.huge` is the largest value a number can be.
- Although `math.floor` and `math.ceil` are present, there's no `math.round`. Not to worry though, you can easily achieve the same thing with code like `math.floor(x + .5)`.
- You can use `math.random` to generate pseudo-random numbers. Before using it, it's a good idea to set the seed with `math.randomseed`. A common thing to do is `math.randomseed(os.time())`.

Conclusion

There's many other parts of the library that I haven't covered, such as the `os`, `io`, and `coroutine` modules. I recommend taking a look at the manual's index for documentation on the entire standard library.

This part and the previous part have covered the absolute essentials of Lua. Be sure to check part three to delve a bit deeper into the language. As always, if you have any feedback, I'd love to hear from you.

programming lua

« Lua for Programmers Part 1: Language Essentials | Lua for Programmers Part 3: More Advanced Concepts »

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)