A Look at Lua

Joseph Quigley

Abstract

An overview of the compact yet powerful Lua programming language.

Lua is a free and open-source multi-paradigm programming language released under the MIT license. Created in 1993 by Roberto Lerusalimschy, Luiz Henrique de Figueiredo and Waldemar Celes, Lua is a dynamically typed language. Extremely compact (only 150KB compiled), it is primarily used as a scripting language or an extension to another language (mainly C/C++).

What Is Lua and How Is It Used?

Lua is implemented as a library and has no “main” program. It works only when embedded in a host client. The interpreter is a C program that uses the Lua library to offer a standalone Lua interpreter. Rather than provide a complex and rigid specification for a single paradigm, Lua is intended to be extended to fit different problem types.

Being small, Lua fits on many host platforms and has been ported and used in video games for both the PlayStation Portable and the Nintendo DS, and it is used in larger games, such as FarCry and World of Warcraft. The Adobe Photoshop Lightroom photography program and the lighthttpd Web server have incorporated Lua as well. Lua has a few advanced features, primarily coercion, coroutines, garbage collection, first-class functions and dynamic module loading. Because Lua is small, it includes only a few data types. It attempts to maintain a balance between power and small size.

What's Different about Lua?

Lua is comparably as easy as Python in terms of learning how to write code. Of the two, Lua is usually the better choice for embedded systems, simply because it's smaller. Lua's strength is in processing strings and tables. It handles logical equations more adeptly than Python.

For a quick hack, a Lua programmer can process complicated data more quickly and easily than a Python programmer can (although a Ruby programmer can do so almost as quickly). But, for a large application that handles many chunks of complex data, a heavier language such as Ruby or Python may be a better choice.

There is no need to worry about different types of integers. You may have found that the different types of integers and numbers (such as floats, longs or doubles) can screw up the output of your program or even make it crash if you are absent-minded. Lua uses coercion for every integer and number type to convert it into a single type. You can add a float, long integer or a double to any other type of integer or number without a hitch in Lua. In contrast, doing this can cause programs written in Python 2.4 or older versions to crash. Lua is extremely forgiving syntactically. What if, for some reason, you are
programming on an embedded device with a four-inch wide screen? You can reduce the amount of lines and other characters, which in turn enables easy reading of the code to make up for the small screen.

Small is beautiful. A programmer can embed Lua into several other languages, such as C/C++ and Java, without bloating the host language, because Lua has a tiny API. Similar to Lisp's single data structure, tables are the only data structuring mechanism that Lua has. This makes tables very powerful, because with a little work, they can emulate data structures in larger languages.

Object-oriented programming implementation is minimalistic. Lua uses tables and functions rather than classes.

In contrast to Python, Lua does not focus on 100% backward compatibility. Many newer releases of Lua break programs written in previous versions. Fortunately, the Lua developers always announce what the new versions of Lua will break.

Lua supports threading. Multiple Lua interpreters can coexist in the same process, and each one can run independently in its own thread. This often makes Lua desirable for multithreaded programs in embedded systems.

**Installing Lua**

To compile and install Lua from the source code, grab a copy of Lua 5.1 from the Lua.org Web site, and untar, configure, make and install it:

```bash
tar -xvzf lua-5.1.1.tar.gz
cd lua-5.1.1
make xyz
make xyz install
```

(\textit{xyz} is your platform name.)

Lua should now be installed and working. To test your install, type \texttt{lua} on the command line. An interactive interpreter should appear.

**Syntax**

Lua is a dynamically typed language whose syntax is very similar to that of Python and even more similar to Ruby. Line breaks do not play any role in Lua's syntax, like that of Python or Ruby. Take, for example, the following ugly, but valid code:

```lua
foo = 89
bar = foo+2
print(bar)
```

Because it is small, Lua has only eight basic data types:

1. nil (similar to Python's None)

2. booleans
Lua supports only a few data structures, including arrays, lists and hash tables.

The table type implements an associative array that can be indexed with any value (similar to Python), except nil (dissimilar to Python). Nil's goal is to be different from any other value, as well as the default value for global variables. Nil also plays a much more important role in Lua than None does in Python. Although tables are the only data structuring mechanism in Lua (which may seem like a disadvantage), the table is just as powerful as Python's dictionary and list, and it's even as powerful as Ruby's hash. Tables are used to represent many different types of arrays, sets, trees and several other data structures. One handy feature of tables is to use strings as keys—for example:

```lua
x = { ["hello world!"] = "ciao world!" }
print(x["hello world!"])  
```

When running this example, Lua outputs “ciao world!” and not “hello world!” as it might appear.

For a more in-depth look at Lua's tables go to [http://lua-users.org/wiki/TablesTutorial](http://lua-users.org/wiki/TablesTutorial).

**Variables and Identifiers**

Because any value can represent a condition, booleans in Lua differ from those in many other languages. Both false and nil are considered false in Lua, but Lua considers everything else true (including zero and an empty string).

Unlike Python, global variables do not need to be declared. To create one, assign a value to it. To delete it, give it the nil value. A global variable exists only if it has a non-nil value. Exactly the opposite of Python, most variables in Lua are global by default, and you must declare the variable “local” to make it a local variable rather than assuming that all variables are local.

Because most CPUs perform floating-point arithmetic just as fast as integer arithmetic, numbers in Lua represent real, double-precision, floating-point numbers rather than common integers. Because Lua doesn't need integer types, it doesn't have them. This eliminates rounding errors, floating-point numbers and long integers.

Lua handles strings very adeptly and has been used for strings that are several megabytes long. It converts between strings and numbers; any numeric operation applied to a string converts the string to a number. This conversion principle applies only to numbers, as Lua converts numbers to strings when strings are expected. Even with automatic conversion, Lua still can differentiate between numbers and strings in cases like 90 == “90” (which always is false).
Identifiers starting with an underscore (such as _FOO) are not recommended for use in Lua, because many are reserved for special uses. As long as an identifier does not begin with a digit, the identifier can be made up of a combination of underscores, letters and digits.

You can basically rename anything in Lua, even to the point of making it un-callable. Take, for example, the following code:

```lua
x = io
x.read()
io = "Hello world!"
x = "Let's make io uncallable!"
io.read()
```

The second line gets keyboard input through the io module. Because io is essentially a variable with a function as a value, you can give it a different value so that io does not relate to the input/output functions anymore. When you try to get keyboard input from the io module again, Lua returns an error. The program is unable to call the input/output functions now that io's value has been reassigned. In order to use io again, you must restart the Lua program.

### Operators and Assignment

Lua concatenates strings with the .. operator. Note that `print("Hello"..'World!")` is valid, but `print("I've said 'Hello World'"..5.."or more times.")` is not. This is because Lua sees the periods as decimals after the integer. The operator must have a space between the strings and the integer. Otherwise, it won't return an error. The following code validly concatenates the strings and the integer:

```lua
print("I've said 'Hello World' 5 or more times.")
```

Lua uses many of the common operators that Python, Ruby and most every other language use. For the Python/Ruby logical not operator, Lua can either use it or use ~= for the negation of equality. Always remember that Lua treats strings and integers differently: "1" < 2 is always false, and strings are compared alphabetically.

Lua ends while loops if the condition is false. Repeat-until statements are the opposite of while loops; they loop until the condition is true. For loops have some hidden twists, which can be annoying to Python or Ruby programmers. Local variables created in the for loop are visible only inside the loop. The variable does not exist when the loop ends, so if you need the value of the control variable, you have to save its value into another loop. Breaks or returns should appear only as the last statement before an end, an else or an until in a loop for syntactic reasons.

Lua treats functions as “first class” values and uses them for OOP (object-oriented programming). Lua can call its own functions or C functions, and it handles functions as a type. You can give a variable the function property or create it with the function() method. Functions written in Lua can return multiple results if you list them after a return keyword.

Lua supports OOP, but due to Lua's size, its implementation of OOP lacks a few features. Lua uses tables and functions for OOP rather than classes. In the same way that Python accesses a function or variable in a class, Lua accesses it with Table.function or Table.variable.

Lua can be picky when it comes to multiple assignment, because it adjusts the number of values on the
assignment. If the amount of values is less than the list of variables, all remaining values are given the nil value. If the list of values is longer than the amount of variables, Lua silently discards them.

### Object-Oriented Programming

Lua has some basic OOP capabilities. The self parameter is an integral concept in any object-oriented language, and it is one of the few OOP concepts that Lua has. Many object-oriented languages tend to hide the self mechanism from you so that you do not have to declare this parameter. Lua hides this parameter with the colon operator. You also can use the colon, a function and a table to emulate a class. Because Lua does not have the class concept, each object defines its own behavior and shape: Garrick, small font below.

```plaintext
Earth = {martians = 5389}
function Earth:casualties (survivors)
    Earth.martians = Earth.martians - survivors
    print("Earth is free! ". Earth.martians.." martians survived!")
end
Earth:casualties(5380)
```

The colon in the above example is used to add an extra parameter in the method definition. It also adds an extra argument in the method call. You don't have to use the colon. Lua programmers can define a function with the dot syntax and call it with the colon syntax, or vice versa if they add an extra parameter: Garrick, small font below.

```plaintext
Earth = {martians = 5389,
    casualties = function (self, survivors)
        self.martians = self.martians - survivors
        print("Earth is free! ". self.martians.." martians survived!")
    end
}
Earth.casualties(Earth, 5380)
Earth.martians = 5389
Earth:casualties(5380)
```

In this case, the function had to be part of the table so that it could be called via the dot or the colon syntax. Note that I also had to give the function the self parameter for either calling method to work. Although these are simple OOP examples that scratch only the surface of OOP, you can find out about inheritance and other OOP implementations in the Lua Reference Manual or in the book *Programming in Lua* (both are available for free from the Lua Web site).

### Show and Tell

Now, let's compare programming in Lua to programming in Python. First, let's write a trivia game. Here is some simple Lua code that uses a table as a dictionary to store both the questions and the answers:

```plaintext
print("What's your name?")
name = io.read()
questions = {
    ["Which came first? Minix or Unix?"] = "Unix",
    ["Who created Linux?"] = "Linus Torvalds",
    ["In what year was Linux created?"] = "1991"
}
```
correct_answers = 0
for key, value in pairs(questions) do
    print(key)
    answer = io.read()
    if answer == value then
        correct_answers = correct_answers + 1
    end
end
if correct_answers == 0 then
    print("You need to browse Wikipedia!")
else
    print("\nGood job, ", name, "!")
    print("Correct answers: ", correct_answers)
end

Next, let's break it down and analyze it line by line. On the second line, the variable name is given the
value io.read(). The io library has many functions that handle all sorts of input and output, but I'm using
it only for keyboard input.

On the next line is the variable questions. The questions variable's value is a table that I have used like a
dictionary to store both the questions and the answers. The questions are in brackets to tell Lua that they
are the table's key.

Skipping the third line, there is a for loop whose function here is to use pairs() to find the key and the
values of each item in the table. It then needs to place the value of the key and the value of the key's
value into the variables key and value.

After printing the key (which contains the question), Lua places the user's answer through io.read() into
the answer variable and checks to see whether it equals the proper answer. If the answer is correct, it
adds 1 to the value of the correct_answers variable and repeats the process until there are no more items
in the table to go through.

Next, Lua checks to see whether users got any of the questions correct and then prints a message telling
users to learn more about UNIX (and its variants) hacker history or congratulates users on how many
questions they answered correctly. Notice the concatenation operators on the 16th line.

In Python, the easiest way to do the above game would be like this:

name = raw_input("What's your name?\n")
questions = {"Which came first? Minix or Unix?": "Unix",
            "Who created Linux?": "Linus Torvalds",
            "In what year was Linux created?": "1991"}
correct_answers = 0

for key in questions:
    print key
    answer = raw_input()
    if answer == questions[key]:
        correct_answers += 1
if correct_answers == 0:
    print "You need to browse Wikipedia!" 
else:
    print "\nGood job, " + name + "!"
    print "Correct answers: ", correct_answers

You may notice that it's easier to get keyboard input in Python than in Lua, but dictionaries are easier to
identify and look much prettier in Lua. The for loop is a little more complex in Python than it is in Lua, because Python needs to know the key to be able to get the key’s value. In Lua, the pairs() function breaks apart the key and its value from the dictionary table, making it much cleaner and easier to get data from tables than in Python. As for lines of code, not counting the many “ends”, Lua wins hands down with 13 lines of code versus 17 in Python. Even though Lua programmers would be typing more, their code is much easier to sift through, especially when it's thousands of lines long, because of Lua's use of end rather than colons (as in Python).

Now, how about a GUI? Is programming a GUI in Lua the same, easier or more difficult than in Python? Before you try to answer that question, determine which program in the two languages will be easier to maintain, read and understand without comments. Here, I use the WxGTK library for a GUI. The only hitch with wxLua is that it is an entirely separate program. A wxLua application will not run with the regular Lua interpreter, so you must run it with the `wxlua` program.

Here's a GTK GUI program made with wxLua:

```lua
frame = wx.wxFrame(wx.wxNull, wx.wxID_ANY, "wxLua App", wx.wxDefaultPosition, wx.wxSize(250, 50), wx.wxDEFAULT_FRAME_STYLE)
frame:Show(true)
```

Now, here's the code for a program that does the same job in GTK, but with wxPython:

```python
from wxPython.wx import *
class Main(wxApp):
    def OnInit(wxApp):
        frame = wxFrame(NULL, -1, "wxPython App")
        frame.Show(true)
        return true
Main().MainLoop()
```

This time, Lua's lack of necessary formatting and full-fledged OOP makes an easy job easier. Rather than create a class and function, wxLua incorporates everything needed for this GUI application in a single system function, whereas Python and wxPython require a Class as well as a function. Also note that Lua imports system libraries automatically. This wxLua application exercises some of Lua's OOP features that I discussed previously. The application creates the frame, sets the frame's name and the frame values, and then it calls the Show() function from within the wxFrame method using the colon. You also can call the frame with the period syntax rather than the colon:

```lua
frame:show(true)
```

**Embedding and Extending**

Although taking a look at embedding and extending Lua is outside the scope of this article, I touch on a few concepts here. First, the Lua API is very straightforward. Its design eliminates the need for manual reference when embedded in C code (unlike Python's API). Like the language, Lua's C API (for embedding) is fairly minimalistic. If you need advanced functionality, you can use a secondary library that is primarily made up of preprocessor macros.

Second, C and C++ are not the only languages in which Lua can be embedded. Tao.Lua provides
straight .NET and Mono bindings to Lua, and LuaJava allows scripts written in Lua to manipulate Java components. LuaJava allows Java components to be accessed from Lua with the same syntax that Lua uses for accessing its native objects. It also allows Java to use a Lua interface so that any interface can be implemented in Lua and passed as a parameter to any method. The method's result (when called in the Java program) is called in Lua, and the result is sent back to Java.

**Conclusion**

Lua is a flexible, powerful, compact language that can be used and extended in myriad situations. Its focus on simplicity makes for easy debugging and has attracted many users. Its simple, powerful syntax provides flexibility because of Lua's metamechanisms. The small, fast interpreter uses less resources than Python, and its syntax allows for easier code readability. Its simple C API makes embedding a breeze. Whether you are doing data processing, GUIs or game programming, you will find a use for Lua.

**Resources**

*Programming in Lua*: [http://www.lua.org/pil](http://www.lua.org/pil)

lua-users: [http://www.lua-users.org](http://www.lua-users.org)

wxPython: [http://www.wxpython.org](http://www.wxpython.org)


wxLua: [http://wxlua.sourceforge.net](http://wxlua.sourceforge.net)

LuaJava: [http://www.keplerproject.org/luajava](http://www.keplerproject.org/luajava)

The Tao Framework: [http://www.taoframework.com](http://www.taoframework.com)

Lua Versus Python: [http://www.lua-users.org/wiki/LuaVersusPython](http://www.lua-users.org/wiki/LuaVersusPython)