Lua for TerraME: A Short Introduction

Vesion 0.2, August 16, 2010

*This tutorial a rather simplified collection of some freely available material on Lua language. It is based mainly on the book "**Programming in Lua**" (1st edition) and on Lua's webpage (**lua.org**), being adapted for those that already have some programming background and need to learn Lua in order to start developing models in TerraME. Because this tutorial has almost no originality in its content, instead of citing it, please use its original sources.*

Lua[1] is a powerful, fast, lightweight, embeddable scripting language. It combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection. Some properties that make Lua recommendable amongst all other scripting languages are:

Robust: Lua has been used in many industrial applications, with an emphasis on embedded systems and games. It is currently the leading scripting language in games. Lua has a solid reference manual and there are several books about it.

Fast: Lua has a deserved reputation for performance. To claim to be "as fast as Lua" is an aspiration of other scripting languages. Several benchmarks show Lua as the fastest language in the realm of interpreted scripting languages. It is fast not only in fine-tuned benchmarks, but a substantial fraction of large applications has been written in Lua.

Portable: Lua is distributed in a small package and builds out-of-the-box in all platforms that have an ANSI/ISO C compiler. Lua runs on all flavours of Unix and Windows, and also on mobile devices and embedded microprocessors.

Embeddable: Lua is a fast language engine with small footprint that you can embed easily into other applications. Lua has a simple and well documented API that allows strong integration with code written in other languages.

Powerful (but simple): Lua provides meta-mechanisms for implementing features, instead of having a host of features directly in the language. For example, although it is not purely object-oriented, Lua provides meta-mechanisms to implement classes and inheritance.

Small: The tarball for Lua 5.1.4 takes 860K uncompressed. The source contains around 17000 lines of C. Under Linux, the interpreter takes 153K and the library, 203K.

Free: Lua is free software, distributed under a very liberal license (the well-known MIT license). It can be used for any purpose at absolutely no cost.

---

[1]Lua (pronounced LOO-AH) means "Moon" in Portuguese. As such, it is neither an acronym nor an abbreviation, but a noun. More specifically, "Lua" is a name, the name of the Earth's moon and the name of the language. Like most names, it should be written in lower case with an initial capital, that is, "Lua". Please do not write it as "LUA", which is both ugly and confusing, because then it becomes an acronym with different meanings for different people. So, please, write "Lua" right! (Source: http://www.lua.org/about.html)

New Moon, Different Day

Lua has a conventional sintax, having lots of similarities with other languages. Lua is *case sensitive*, meaning that x and X are different variables. Each piece of code Lua executes is a chunk. More specifically, a chunk is a sequence of statements. A semicolon may *optionally* follow any statement. Usually, I use semicolons only to separate two or more statements written in the same line, but this is just a convention. Line breaks play no role in Lua's syntax; for instance, the following chunks are valid and equivalent:

```
a = 1
b = a*2

a = 1; b = a*2

a = 1   b = a*2          -- ugly, but valid
```

Variables have no predefined types; any variable may contain values of any type. Also, there are no type definitions in Lua; each value carries its own type. This tutorial introduces six basic types of Lua: **nil**, **boolean**, **number**, **string**, **table**, and **function**. The type function gives the type name of a given value:

```
print(type(c))              --> nil ('c' is not initialized)
c = 10
print(type(c))              --> number
print(type(c > 2))          --> boolean
print(type("a string!!"))   --> string
print(type(print))          --> function
```

nil: a type with a single value, **nil**, whose main property is to be different from any other value, representing the absence of a useful value. A variable has a nil value by default, before a first assignment, and you can assign nil to a variable to delete it.

boolean: a type with the two traditional boolean values, **false** and **true**. However, they do not hold a monopoly of condition values: In Lua, any value may represent a condition. Conditionals consider false and nil as false and anything else as true. Beware: Lua considers both zero and the empty string as true in conditional tests. There are three logical operators for working with condition values: **and**, **or**, and **not**. Both and and or use short-cut evaluation, evaluating their second operand only when necessary. The operator and returns its first argument if it is false; otherwise, it returns its second argument. The operator or returns its first argument if it is not false; otherwise, it returns its second argument.

```
print(4 and 5)          --> 5
print(nil and 13)       --> nil
print(4 or 5)           --> 4
print(false or 5)       --> 5
```

number: a real (double-precision floating-point) number. Lua has no integer type, as it does not need it. There is a misconception about floating-point arithmetic errors and some people fear that even a simple increment can go weird with floating-point numbers. The fact is that, when you use a double to represent an integer, there is no rounding error at all (unless the number is greater than 10^14). Specifically, a Lua

number can represent any long integer without rounding problems. Moreover, most modern CPUs do floating-point arithmetic at least as fast as integer arithmetic. Besides the traditional arithmetic operators (+, –, *, /), Lua provides the exponent (^) and the modulo (%) operators for working with number values.

string: a sequence of immutable characters. You cannot change a character inside a string; instead, you create a new string with the desired modifications. We can delimit literal strings by matching single or double quotes. As a matter of style, you should use always the same kind of quotes in a program, unless the string itself has quotes; then you use the other quote, or escape those quotes with backslashes. Strings can contain escape sequences such as \n, \t, \\, and \". We can concatenate strings by using the operator "**..**" (two dots). If any of its operands is a number, Lua converts that number to a string.

```
print("Hello " .. "World")      --> Hello World
print(0 .. 1)                   --> 01
```

Lua also has the traditional relational operators (<, >, <=, >=, ~=, and ==). They compare numbers, strings and booleans by their values. However, tables and functions (to be presented in the next sections) are compared by reference, that is, two such values are considered equal if and only if they are the very same object.

Operator precedence in Lua follows the table below, from the higher to the lower priority:

```
^
not  – (unary)
*  /  %
+  –
.. (two dots)
<  >  <= >= ~= ==
and
or
```

Based on this, the following expressions on the left are equivalent to those on the right:

```
a+-i < b/2+1          <-->      (a+(-i)) < ((b/2)+1)
5+x^2*8               <-->      5+((x^2)*8)
a < y and y <= z      <-->      (a < y) and (y <= z)
–x^y^z                <-->      –(x^(y^z))
```

When in doubt, always use explicit parentheses. It is easier than looking up in the manual and probably you will have the same doubt when you read the code again.

A comment starts anywhere with a double hyphen (--) and runs until the end of the line. Block comments start with --[[and run until the corresponding]]. A common trick, when we want to comment out a piece of code, is to write the following:

```
--[[
print(10)         -- no action (comment)
--]]
```

Now, if we add a single hyphen in the beginning of the first line, it becomes an end-of-line comment, and then the code is in again.

Lua provides a small and conventional set of control structures, with **if** for conditional and **for**, **while**, and **repeat** for iteration. All control structures have an explicit terminator: **end** terminates the if, for and while structures; and **until** terminates the repeat structure.

An **if** statement tests its condition and executes its then-part or its else-part accordingly. The else-part is optional.

```
if a<0 then a = 0 end
if a<b then a = 5 else a = b end
```

When you write nested ifs, you can use **elseif**. It is similar to an else followed by an if, but it avoids the need for multiple ends.

```
if      op == "+" then r = a + b
elseif  op == "–" then r = a – b
else
        error("invalid operation")
end
```

Iterations with **while** start with a test of the condition; if the condition is false, then the loop ends; otherwise, Lua executes the body of the loop and repeats the process.

```
i = 1
while a[i] do
        print(a[i])
        i = i + 1
end
```

A **repeat-until** statement repeats its body until its condition is true. The test is done after the body, so the body is always executed at least once.

```
repeat
        line = os.read()
until line ~= ""
print(line)       -- print the first non-empty line
```

A numeric **for** has the following syntax:

```
for var = exp1, exp2, exp3 do
        something
end
```

That loop will execute something for each value of var from exp1 to exp2, using exp3 as the step to increment var. This third expression is optional; when absent, Lua assumes one as the step value. As typical examples of such loops, we have

```
for i = 1, f(x) do print(i) end
for i = 10, 1, –1 do print(i) end
```

All three expressions are evaluated once, before the loop starts (for example, f(x) in the example above). You should never change the value of the control variable: The effect of such changes is unpredictable. If you want to break a for loop before its normal termination, use **break**.

The Table Type

The **table** type implements associative arrays, which can be indexed not only with numbers, but also with strings or any other value of the language, except nil. Moreover, tables have no fixed size; you can add as many elements as you want dynamically. Tables are the main (in fact, the only) data structuring mechanism. We use tables to represent ordinary arrays, symbol tables, sets, records, queues, and other data structures, in a simple, uniform, and efficient way.

Tables in Lua are neither values nor variables; they are dynamically allocated objects which contain references (or pointers) to their fields. You create tables by means of a *constructor expression*, which in its simplest form is written as {}:

```
a = {}                  -- create a table and store its reference in `a'
k = "x"
a[k] = 10               -- new entry, with key="x" and value=10
a[20] = "great"         -- new entry, with key=20 and value="great"
print(a["x"])           --> 10
k = 20
print(a[k])             --> "great"
a["x"] = a["x"] + 1     -- increments entry "x"
print(a["x"])           --> 11
```

A table is always anonymous. There is no fixed relationship between a variable that holds a table and the table itself:

```
a = {}
a["x"] = 10
b = a                   -- `b' refers to the same table as `a'
print(b["x"])           --> 10
b["x"] = 20
print(a["x"])           --> 20
a = nil                 -- now only `b' still refers to the table
b = nil                 -- now there are no references left to the table
```

When a program has no references to a table left, Lua memory management will eventually delete the table and reuse its memory.

Each table may store values with different types of indices and it grows as it needs to accommodate new entries. For example, to represent a conventional array, you simply use a table with integer keys. There is no way to declare its size; you just initialize the elements as you need:

```
a = {}                              -- empty table
for i=1,1000 do a[i] = i*2 end      -- create 1000 new entries
print(a[9])                         --> 18
a["x"] = 10
print(a["x"])                       --> 10
print(a["y"])                       --> nil
```

Notice the last line: table fields evaluate to nil if they are not initialized. Also like variables, you can assign nil to a table field to delete it. Since you can index a table with any value,

you can start the indices of an array with any number that pleases you. However, it is customary in Lua to start arrays with one.

To simplify the access to table records, Lua provides a.name as syntactic sugar for a["name"]. For Lua, the two forms are equivalent and can be intermixed freely; but for a human reader, each form may signal a different intention. So, we could write the last lines of the previous example in a cleanlier manner:

```
a.x = 10        -- same as a["x"] = 10
print(a.x)      -- same as print(a["x"])
print(a.y)      -- same as print(a["y"])
```

A common mistake for beginners is to confuse a.x with a[x]. The first form represents a["x"], that is, a table indexed by the string "x". The second form is a table indexed by the value of the variable x. See the difference:

```
a = {}
x = "y"
a[x] = 10       -- put 10 in field "y"
print(a[x])     --> 10   -- value of field "y"
print(a.x)      --> nil   -- value of field "x" (undefined)
print(a.y)      --> 10   -- value of field "y"
```

Because we can index a table with any type, when indexing a table we have the same subtleties that arise in equality. Although we can index a table both with the number 0 and with the string "0", these two values are different (according to equality) and therefore denote different positions in a table. This way, "+1", "01", and "1" are different positions. When in doubt about the actual types of your indices, use an explicit conversion to be sure, because you can introduce subtle bugs in your program if you do not pay attention to it.

```
i = 10; j = "10"; k = "+10"
a = {}
a[i] = "one value"
a[j] = "another value"
a[k] = "yet another value"
print(a[j])                 --> another value
print(a[k])                 --> yet another value
print(a[tonumber(j)])       --> one value
print(a[tonumber(k)])       --> one value
```

Finally, we can simplify the creation of tables by specifying entries and their values directly in the construction expression, such as:

```
loc = {
        cover = "forest",
        distRoad = 0.3,
        distUrban = 2
}

print(loc.cover)
loc.distRoad = loc.distRoad^2
loc.distTotal = loc.distRoad + loc.distUrban      -- adding a fourth entry
loc["deforestationPot"] = 1/loc.distTotal         -- adding a fifth entry
```

The Function Type

A **function** is a first-class value in Lua. It means that a function is a value with the same rights as conventional values like numbers and strings. Functions can be stored in variables and in tables, can be passed as arguments, and can be returned by other functions, giving great flexibility to the language. A function can carry out a specific task (commonly called *procedure*) or compute and return values. In the first case, we use a function call as a statement; in the second case, we use it as an expression:

```
print(8*9, 9/8)
a = math.sin(3) + math.cos(10)
```

In both cases, we write a list of arguments enclosed in parentheses. If the function call has no arguments, we must write an empty list of arguments () to indicate the call. There is a special case to this rule: If the function has one single argument and this argument is either a literal string or a table constructor, then the parentheses are optional:

```
print "Hello World"     --> same as print("Hello World")
f{x=10, y=20}           --> same as f({x=10, y=20})
```

A function definition has a conventional syntax, with a name, a list of parameters, a body (a list of statements), and the explicit terminator **end**. For instance, the following code creates a function that sums two numbers:

```
function add(a, b)
        return a + b
end
```

Because functions are first-class values, the code above is just an instance of what we call *syntactic sugar*; in other words, it is just a pretty way to write

```
add = function(a, b)
        return a + b
end
```

That is, a function definition is in fact a statement that assigns a value of type function to a variable. We can see the expression function (...) ... end as a function constructor, just as {} is a table constructor.

Function parameters are local variables, initialized with the arguments of the function call. It is possible to call a function with a number of arguments different from its number of parameters. Lua adjusts the number of arguments to the number of parameters: Extra arguments are thrown away; extra parameters get nil. For instance, if we have

```
function f(a, b) return a or b end
```

we will have the following mapping from arguments to parameters:

CALL	PARAMETERS
f(3)	a=3, b=nil
f(3, 4)	a=3, b=4
f(3, 4, 5)	a=3, b=4 (5 is discarded)

Although this behavior can lead to programming errors (easily spotted at run time), it is also useful, especially for default arguments. For instance, consider the following function, to increment a global counter.

```
function incCount(n)
        n = n or 1
        count = count + n
end
```

This function has 1 as its default argument; that is, the call incCount(), without arguments, increments count by one. When you call incCount(), Lua first initializes n with nil; the or results in its second operand; and as a result Lua assigns a default 1 to n.

An unconventional, but quite convenient, feature of Lua is that functions may return multiple results. Several predefined functions in Lua return multiple values. An example is the string.find function, which locates a pattern in a string. It returns two indices: the index of the character where the pattern match starts and the one where it ends (or nil if it cannot find the pattern). A multiple assignment allows the program to get both results:

```
s, e = string.find("hello Lua users", "Lua")
print(s, e)                --> 7     9
```

To return multiple results inside of a function, we need only to list them all after the return keyword separated by commas, in the same way we do in multiple assignament. Lua adjusts the number of results to the circumstances of the call. When we call a function as a statement, Lua discards all of its results. When we use a call as an expression, Lua keeps only the first result. We get all results only when the call is the last (or the only) expression in a list of expressions. For instance:

```
function foo() return 'a', 'b' end       -- returns 2 results

x,y = foo()              -- x='a', y='b'
x,y,z = foo()            -- x='a', y='b', z=nil
x,y = foo(), 20          -- x='a', y=20
print(foo())             --> a   b
print(foo(), 1)          --> a   1
print(foo() .. "x")      --> ax
a = {foo()}              -- a = {'a', 'b'}  (use "{}" to get all returned values as a table)
print((foo()))           --> a  (only one value by enclosing the call with an extra "()")
```

Functions can also be parameters to other functions. This kind of function is what we call a *higher-order function*, a great source of flexibility on the language. For example, the table library provides a function table.foreach, that applies a function to each element of a given table. The function used as argument receives one single argument, and is called once for each value of the table. For instance:

```
x={1, 3, 2, 6, 4}
table.foreach(x, function(a)    -- print each value of x
        print(a)
end)                            -- do not forget to close the parenthesis!
table.foreach(x, print)         -- a slightly different result
```

Mixing Functions and Tables

The parameter passing mechanism in Lua is positional: When we call a function, arguments match parameters by their positions. The first argument gives the value to the first parameter, and so on. Sometimes, however, it is useful to specify the arguments by name. To illustrate this point, let us consider the function rename (from the os library), which renames a file. Quite often, we forget which name comes first, the new or the old; therefore, we may want to redefine this function to receive its two arguments by name:

```
-- invalid code
rename(old="temp.lua", new="temp1.lua")
```

Lua has no direct support for that syntax, but we can have the same final effect, with a small syntax change. The idea is to pack all arguments into a table and use that table as *the only* argument to the function. The special syntax that Lua provides for function calls, with just one table constructor as argument, helps the trick:

```
rename{old="temp.lua", new="temp1.lua"}
```

Accordingly, we define rename with only one parameter and get the actual arguments from this parameter:

```
function rename(arg)
        return os.rename(arg.old, arg.new)
end
```

This style of parameter passing is especially helpful when the function has many parameters, and most of them are optional. This is also the way Lua allows object-oriented programming, mainly because a table in Lua is an object in more than one sense. Like objects, tables have a state. Like objects, tables have an identity (a *selfness*) that is independent of their values; specifically, two objects (tables) with the same value are different objects, whereas an object can have different values at different times, but it is always the same object. Like objects, tables have a life cycle that is independent of who created them or where they were created.

Objects have their own operations. Tables also can have operations:

```
loc = {cover = "forest", distRoad = 0.3, distUrban = 2}
loc.deforestPot = function()
        return 1/(loc.distRoad + loc.distUrban)
end
```

This definition creates a new function and stores it in the field deforestPot of the loc object. Then, we can call it as

```
loc.deforestPot()
```

This kind of function is almost what we call a method. However, the use of the name loc inside the function is a bad programming practice. First, this function will work only for this particular object. Second, even for this particular object the function will work only as long as the object is stored in that particular variable; if we change the name of this object, deforestPot does not work any more:

```
a = loc; loc = nil
a. deforestPot ()   -- ERROR!
```

Such behaviour violates the previous principle that objects have independent life cycles.

A more flexible approach is to operate on the receiver of the operation. For that, we would have to define our method with an extra parameter, which tells the method on which object it has to operate. This parameter usually has the name *self* or *this*:

```
loc. deforestPot = function(self)
        return 1/(self.distRoad + self.distUrban)
end
```

Now, when we call the method we have to specify on which object it has to operate:

```
a1 = loc; loc = nil
a1. deforestPot(a1)   -- OK
```

This use of a self parameter is a central point in any object-oriented language. Most OO languages have this mechanism partly hidden from the programmer, so that she does not have to declare this parameter. Lua can also hide this parameter, using the *colon operator*. We can rewrite the previous method definition as

```
function loc:deforestPot() return 1/(self.distRoad + self.distUrban) end
```

and the method call as

```
loc:deforestPot()
```

The effect of the colon is to add an extra hidden parameter in a method definition and to add an extra argument in a method call. The colon is only a syntactic facility; there is nothing really new here. We can define a function with the dot syntax and call it with the colon syntax, or vice-versa, as long as we handle the extra parameter correctly.

Finally, we can declare a class in Lua by creating a function that takes a table constructor as argument. This function typically initializes, checks properties values and adds auxiliary data structure or methods. For instance, the next example builds the type MyLoc:

```
function MyLoc(locdata)
        locdata.cover = locdata.cover or "forest"
        locdata.deforestationPot = function(self)
                return 1/(self.distRoad + self.distUrban)
        end
        return locdata
end

loc = MyLoc{distRoad = 0.3, distUrban = 2}

print(loc.cover)
print(loc:deforestationPot())
```

When loc is instantiated, the constructor initializes the attribute cover and the function deforestationPot. TerraME includes new value types in Lua using this constructor mechanism for building spatial dynamic models.

Annex: Some Useful Lua Functions

Function	Description	Example
math.sin, **math.cos**, **math.tan**, etc.	Trigonometric functions, always working with radians.	math.sin(2 * math.pi)
math.deg, **math.rad**	Converts between degrees and radians.	math.deg(math.pi)
math.exp, **math.log**, **math.log10**	Exponentiation and logarithms.	math.exp(1) math.log(2.71)
math.floor, **math.ceil**	Rounding functions.	math.floor(2.3)
math.max, **math.min**	The maximum (minimum) of two numbers.	math.max(4, 7)
math.random	Generates pseudo-random real (when called without arguments) or integer (otherwise) numbers.	math.random() -- *[0,1)* math.random(n) -- *1<=x<=n* math.random(a,b) --*a<=x<=b*
math.randomseed	Set a seed for the pseudo-random generator.	math.randomseed(0) math.randomseed(os.time())
string.upper, **string.lower**	Upper (lower) case.	string.upper("moon")
string.sub	Cuts a string, from the i-th to the j-th character inclusive. The first character of a string has index 1. Value -1 refers to the last character, -2 to the previous one, and so on. If you do not provide a third argument, it has default -1.	s = "[in brackets]" string.sub(s, 4) string.sub(s, 2, 7) string.sub(s, 2, -2)
string.format	Formats a string, with rules similar to those of the standard C printf function.	string.format("pi = %.4f", math.pi)
string.gsub	Replaces the occurrences of a given pattern inside of the subject string. Returns the new string and the number of occurrences. An optional fourth parameter limits the number of substitutions to be made.	string.gsub("Lua is cute", "cute", "great") string.gsub("all lii", "l", "x", 1)
table.insert	Inserts an element in a given position of an array, moving up other elements to open space and incrementing the size of the array.	a = {3, 7, 5} table.insert(a, 1, 15) table.insert(a, 3, 10)
table.remove	Removes (and returns) an element from a given position in an array, moving down other elements. When called without a position, it removes the last element.	a = {3, 7, 5} table.remove(a, 1) table.remove(a)
table.getn	Returns the number of elements of a table with a linear sequence of non-nil elements with numeric indices starting at 1. Arrays that do not follow this convention can produce undefined results.	x = {3, 5} table.getn(x) print(#x) -- *sugar*
table.sort	Orders the elements of a table according to an order function, that has two arguments and must return true if the first argument should come first in the sorted array. This function has the less-than operation (corresponding to the `<´ operator) as default.	a = {3, 7, 4, 1, 2} table.sort(a) table.sort(a, function(v1, v2) return string.lower(v1) < string.lower(v2) end)
tostring	Converts a number to a string.	tostring(12)
tonumber	Converts a string to a number.	tonumber("11")

Lua has many other functionalities not presented in this short tutorial. More about Lua can be found at http://www.lua.org/pil.