Lua Progamming Language

Lua is a powerful, fast, lightweight and embeddable programming language. It is used by many frameworks, games and other applications. While it can be used by itself, it has been designed to be easy to embed in another application. It is implemented in ANSI C, a subset of C programming language that is very portable, which means it can run on many systems and many devices where the most other scripting languages would not be able to run.

> "Lua" (pronounced LOO-ah) means "Moon" in Portuguese. As such, it is neither an acronym nor an abbreviation, but a noun. Lua comes from two languages: DEL and Sol. DEL means "Data Entry Language", While Sol means "Simple object language" and also means Sun in Portuguese.

One of main advantages of Lua is its simplicity. Programming which is also called scripting in the case of programs that run inside an embedded applications, is the process of writing computer program. A programming language is a language used to give instructions to a computer through computer code that is contained in a computer program. A programming language consists of two things: a syntax, which is like grammar in English, and libraries, basic functions provided with the language. These libraries could be compared with vocabulary in English.

1. Hello World!

Lua can either be used embedded in an application or by itself. The first example of Lua code in this book will be the basic and traditional hello world program.

> A "Hello world" program is a computer program that outputs "Hello, world" on a display device. Because it is typically one of the simplest programs possible in most programming languages, it is by tradition often used to illustrate to beginners the most basic syntax of a programming language, or to verify that a language or system is operating correctly.

```
print("Hello World!")
```

The code above prints the text Hello, world! to the output. It does so by calling the print function with the string "Hello, world!" as an argument.

2. Syntax

Syntax of a programming language defines how statements and expressions must be written in that programming language, just like grammar defines how sentences and words must be written. Statements and expressions can be respectively compared to sentences and words.

Lua is a dynamically typed language intended for use as an extension language or scripting language.

Statements and expressions can be respectively compared to sentences and words. Expressions are pieces of code that have a value and that can be evaluated, while statements are pieces of code that can be executed and contain an instruction with one or many expressions to use that instruction with.

For example, `3 + 5` is an expression and `variable = 3 + 5` is a statement that sets the value of variable to that expression.

```lua
-- Different types
local x = 10 -- number
local name = "Neo" -- string
local is_alive = false -- boolean
local a = nil -- no value or invalid value
```

3. Obtaining Lua

Lua can be obtained on the official Lua website, on the download page.

4. Basics

If you are using the stand-alone Lua interpreter, all you have to do to run your first program is to call the interpreter -usually named `lua` or `lua5.3`/`lua5.4`- with the name of the text file that contains your program. If you save the *Hello World* program in a file `hello.lua`, the following command should run it:

```
$ lua hello.lua
```

As a more complex example, the next program defines a function to compute the factorial of a given number, asks the user for a number, and prints its factorial:

```lua
-- defines a factorial function
function fact (n)
    if n == 0 then
        return 1
    else
        return n * fact(n - 1)
    end
end

print("enter a number:")
a = io.read("*n")        -- reads a number
print(fact(a))
```

5. Chuncks

We call each piece of code that Lua executes, such as a file or a single line in interactive mode, a *chunk*. A chunk is simply a sequence of commands (or statements).

A chunk can be as simple as a single statement, such as in the "Hello World" example, or it can be composed of a mix of statements and function definitions (which are actually assignments, as we will see later), such as the factorial example.

Instead of writing your program to a file, you can run the stand-alone interpreter in interactive mode. If you call `lua` without any arguments, you will get its prompt:

```
$ lua
Lua 5.4.4  Copyright (C) 1994-2022 Lua.org, PUC-Rio
>
```

Thereafter, each command that you type (such as `print "Hello World"`) executes immediately after you enter it. To exit the interactive mode and the interpreter, just type the end-of-file control character (`ctrl-D` in POSIX, `ctrl-Z` in Windows), or call the function `os.exit`, from the Operating System library - you have to type `os.exit()`.

Starting in version 5.4, we can enter expressions directly in the interactive mode, and Lua will print their values:

```
$ lua
Lua 5.4.4  Copyright (C) 1994-2022 Lua.org, PUC-Rio
> math.pi / 4   --> 0.78539816339745
> a = 15
> a ^ 2         --> 225
> a + 2         --> 17
```

In older versions, we need to precede these expressions with an equals sign:

```
$ lua5.2
Lua 5.2.3  Copyright (C) 1994-2013 Lua.org, PUC-Rio
> a = 15
> = a ^ 2         --> 225
```

For compatibility, Lua 5.4 still accepts these equals signs.

To run that code as a chunk (not in interactive mode), we must enclose the expressions inside calls to `print`:

```
print(math.pi / 4)  --> 0.78539816339745
a = 15
print(a ^ 2)        --> 225
print(a + 2)        --> 17
```

Lua usually interprets each line that we type in interactive mode as a complete chunk or expression. However, if it detects that the line is not complete, it waits for more input, until it has a complete chunk. This way, we can enter a multi-line definition, such as the factorial function, directly in interactive mode. However, it is usually more convenient to put such definitions in a file and then call Lua to run the file.

[Multiline Code in Intractive Mode] | *figs/intractive-multiline.png*

We can use the `-i` option to instruct Lua to start an interactive session after running a given chunk:

```
$ lua -i prog
```

A command line like this one will run the chunk in the file `prog` and then prompt for interaction. This is especially useful for debugging and manual testing.

Another way to run chunks is with the function `dofile`, which immediately executes a file. For instance, suppose we have a file `lib1.lua` with the following code:

```
function norm (x, y)
    return math.sqrt(x^2 + y^2)
end

function twice (x)
    return 2.0 * x
end
```

Then, in interactive mode, we can type this code:

```
> dofile("lib1.lua")    -- load our library
> n = norm(3.4, 1.0)
> twice(n)              --> 7.0880180586677
```

The function `dofile` is useful also when we are testing a piece of code. We can work with two windows: one is a text editor with our program (in a file `prog.lua`, say) and the other is a console running Lua in interactive mode. After saving a modification in our program, we execute `dofile("prog.lua")` in the Lua console to load the new code; then we can exercise the new code, calling its functions and printing the results.

5.1. Some Lexical Conventions

Identifiers (or names) in Lua can be any string of letters, digits, and underscores, not beginning with a digit; for instance

- `i`
- `j`
- `i10`
- `_ij`
- `aSomewhatLongName`
- `_INPUT`

You should avoid identifiers starting with an underscore followed by one or more upper-case letters (e.g., `_VERSION`); they are reserved for special uses in Lua. Usually, I reserve the identifier `_` (a single underscore) for dummy variables.

The following words are reserved; we cannot use them as identifiers:

```
and end if or until break false in repeat while do for local
return else function nil then elseif goto not true
```

Lua is case-sensitive

`and` is a reserved word, but `And` and `AND` are two different identifiers.

6. Comments

A comment is a code annotation that is ignored by the programming language. Comments can be used to describe one or many lines of code, to document a program, to temporarily disable code, or for any other reason.

They need to be prefixed by two hyphens to be recognized by Lua and they can be put either on their own line or at the end of another line:

```
print("This is normal code.")
-- This is a comment
print("This is still normal code.") -- Comment at the end of a line of code.
```

These comments are called short comments. It is also possible to create long comments, which start with a long bracket and can continue on many lines:

```
print("This is normal code")
--[[Line 1
Line 2
]]
```

Long brackets consist of two brackets in the middle of which any number of equality signs may be put. That number is called the level of the long bracket. Long brackets will continue until the next bracket of the same level, if there is one.

A long bracket with no equal sign is called a long bracket of level 0. This approach makes it possible to use closing double brackets inside of long comments by adding equal signs in the middle of the two brackets. It is often useful to do this when using comments to disable blocks of code.

```
--[==[
This is a comment that contains a closing long bracket of level 0 which is here:
 ]]
However, the closing double bracket doesn't make the comment end, because the
  comment was opened with an opening long bracket of level 2, and only a closing
  long bracket of level 2 can close it.
]==]
```

In the example above, the closing long bracket of level 0 (`]]`) does not close the comment, but the closing long bracket of level 2 (`]==]`) does.

> Long comments can be more complex than that, as we will see in the section called "Long Strings".

7. Expressions

Expressions are pieces of code that have a value and that can be evaluated. They cannot be executed directly (with the exception of function calls), and thus, a script that would contain only the following code, which consists of an expression, would be erroneous:

```
3 + 5
```

The code above is erroneous because all it contains is an expression. The computer cannot execute `3 + 5`, since that does not make sense.

Code must be comprised of a sequence of statements. These statements can contain expressions which will be values the statement has to manipulate or use to execute the instruction.

Some code examples in this chapter do not constitute valid code, because they consist of only expressions. In the next chapter, statements will be covered and it will be possible to start writing valid code.

Lua needs no separator between consecutive statements, but we can use a semicolon if we wish. Line breaks play no role in Lua's syntax; for instance, the following four chunks are all valid and equivalent:

```
a = 1
b = a * 2

a = 1;
b = a * 2;

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

8. Global Variables

Global variables do not need declarations; we simply use them. It is not an error to access a non-initialized variable; we just get the value nil as the result:

```
> b   --> nil
> b = 10
> b   --> 10
```

If we assign nil to a global variable, Lua behaves as if we have never used the variable:

```
> b = nil
> b   --> nil
```

Lua does not differentiate a non-initialized variable from one that we assigned nil. After the assignment, Lua can eventually reclaim the memory used by the variable.

9. Types and Values

Lua is a dynamically-typed language. There are no type definitions in the language; each value carries its own type.

There are eight basic types in Lua: *nil*, *Boolean*, *number*, *string*, *userdata*, *function*, *thread*, and *table*. The function `type` gives the type name of any given value:

```
> type(nil)           --> nil
> type(true)          --> boolean
> type(10.4 * 3)      --> number
> type("Hello world") --> string
> type(io.stdin)      --> userdata
> type(print)         --> function
> type(type)          --> function
> type({})            --> table
> type(type(X))       --> string
```

The last line will result in “string” no matter the value of `X`, because the result of `type` is always a string.

The userdata type allows arbitrary C data to be stored in Lua variables. It has no predefined operations in Lua, except assignment and equality test. Userdata are used to represent new types created by an application program or a library written in C; for instance, the standard I/O library uses them to represent open files. We will discuss more about userdata later, when we get to the C API.

Variables have no predefined types; any variable can contain values of any type:

```
> type(a)          --> nil    ('a' is not initialized)
> a = 10
> type(a)          --> number
> a = "a string!!"
> type(a)          --> string
> a = nil
> type(a)          --> nil
```

The list of data types for values are given below.

Type	Description

`nil`	Used to differentiate the value from having some data or no(nil) data.
`boolean`	Includes true and false as values. Generally used for condition checking.
`number`	Represents real(double precision floating point) numbers.
`string`	Represents array of characters.
`function`	Represents a method that is written in C or Lua.
`userdata`	Represents arbitrary C data.
`thread`	Represents independent threads of execution and it is used to implement coroutines.
`table`	Represent ordinary arrays, symbol tables, sets, records, graphs, trees, etc., and implements associative arrays. It can hold any value (except nil).

Usually, when we use a single variable for different types, the result is messy code. However, sometimes the judicious use of this facility is helpful, for instance in the use of nil to differentiate a normal return value from an abnormal condition.

By default, all the variables will point to nil until they are assigned a value or initialized.

We will discuss now the simple types nil and Boolean.

9.1. Nil

Nil is a type with a single value, nil, whose main property is to be different from any other value. Lua uses nil as a kind of non-value, to represent the absence of a useful value. As we have seen, a global variable has a nil value by default, before its first assignment, and we can assign nil to a global variable to delete it.

9.2. Booleans

The Boolean type has two values, `false` and `true`, which represent the traditional Boolean values. However, Booleans do not hold a monopoly of condition values: in Lua, any value can represent a condition. Conditional tests (e.g., conditions in control structures) consider both the Boolean `false` and `nil` as false and anything else as true. In particular, Lua considers both zero and the empty string as `true` in conditional tests.

Throughout this book, I will write "false" to mean any false value, that is, the Boolean `false` or nil. When I mean specifically the Boolean value, I will write "false". The same holds for "true" and “true”.

Lua supports a conventional set of logical operators: `and`, `or`, and `not`. Like control structures, all logical operators consider both the Boolean `false` and nil as false, and anything else as true. The result of the `and` operator is its first operand if that operand is false; otherwise, the result is its second operand. The result of the `or` operator is its first operand if it is not false; otherwise, the result is its second operand:

```
> 4 and 5           --> 5
> nil and 13        --> nil
> false and 13      --> false
> 0 or 5            --> 0
> false or "hi"     --> "hi"
> nil or false      --> false
```

Both **and** and **or** use short-circuit evaluation, that is, they evaluate their second operand only when necessary. Short-circuit evaluation ensures that expressions like (`i ~= 0 and a/i > b`) do not cause run-time errors: Lua will not try to evaluate `a / i` when `i` is zero.

A useful Lua idiom is `x = x or v`, which is equivalent to

```
if not x then x = v end
```

That is, it sets `x` to a default value `v` when `x` is not set (provided that `x` is not set to **false**).

Another useful idiom is `((a and b) or c)` or simply `(a and b or c)` (given that `and` has a higher precedence than `or`). It is equivalent to the C expression `a ? b : c`, provided that `b` is not false. For instance, we can select the maximum of two numbers `x` and `y` with the expression `(x > y) and x or y`. When `x > y`, the first expression of the **and** is true, so the **and** results in its second operand (`x`), which is always true (because it is a number), and then the **or** expression results in the value of its first operand, `x`. When `x > y` is false, the **and** expression is false and so the **or** results in its second operand, `y`.

The not operator always gives a Boolean value:

```
> not nil        --> true
> not false      --> true
> not 0          --> false
> not not 1      --> true
> not not nil    --> false
```

The following operators are often used with boolean values, but can also be used with values of any data type:

Operation	Syntax	Description
Boolean negation	`not a`	If a is false or nil, returns true. Otherwise, returns false.
Logical conjunction	`a and b`	Returns the first argument if it is false or nil. Otherwise, returns the second argument.
Logical disjunction	`a or b`	Returns the first argument if it is neither false nor nil. Otherwise, returns the second argument.

Essentially, the `not` operator just negates the boolean value (makes it false if it is true and makes it true if it is false), the `and` operator returns true if both are true and false if not and the `or` operator returns true if either of arguments is true and false otherwise.

```
local is_alive = true
print(is_alive) -- true

local is_alive = false
print(is_alive) -- false
```

9.3. Numbers

Numbers generally represent quantities, but they can be used for many other things. The number type in Lua works mostly in the same way as real numbers.

Numbers can be constructed as integers, decimal numbers, decimal exponents or even in hexadecimal.

Here are some valid numbers:

- `3`
- `3.0`
- `3.1416`
- `314.16e-2`
- `0.31416E1`
- `0xff`
- `0x56`

```
-- The Lua
local a = 1
local b = 2
local c = a + b
print(c) -- 3

local d = b - a
print(d)

local x = 1 * 3 * 4 -- 12
print(x)

local y = (1+3) * 2 -- 8
print(y)

print(10 / 2) -- 5
print(2 ^ 2) -- 4
print(5 % 2) -- 1
print(-b) -- -2

-- Incerment
local level = 1
level = level + 1
print(level)
```

9.3.1. Arithmetic operations

The operators for numbers in Lua are the following:

Operation	Syntax	Description	Example
Arithmetic negation	`-a`	Changes the sign of a and returns the value	`-3.14159`
Addition	`a + b`	Returns the sum of `a` and `b`	`5.2 + 3.6`
Subtraction	`a - b`	Subtracts b from a and returns the result	`5.2 + 3.6`
Multiplication	`a * b`	Returns the product of a and b	`3.2 _ 1.5`
Exponentiation	`a ^ b`	Returns a to the power b, or the exponentiation of a by b	`5 ^ 2`
Division	`a / b`	Divides a by b and returns the result	`6.4 / 2`
Modulus operation	`a % b`	Returns the remainder of the division of a by b	`5 % 3`

9.3.2. Integers

A new subtype of numbers, integers, was added in Lua 5.3. Numbers can be either integers or floats. Floats are similar to numbers as described above, while integers are numbers with no decimal part.

Float division (`/`) and exponentiation always convert their operands to floats, while all other operators give integers if their two operands were integers. In other cases, with the exception of the floor division operator (`//`) the result is a float.

9.4. Strings

Strings are sequences of characters that can be used to represent text. They can be written in Lua by being contained in double quotes, single quotes or long brackets (it should be noted that comments and strings have nothing in common other than the fact they can both be delimited by long brackets, preceded by two hyphens in the case of comments).

Strings that aren't contained in long brackets will only continue for one line. Because of this, the only way to make a string that contains many lines without using long brackets is to use escape sequences. This is also the only way to insert single or double quotes in certain cases.

1. `' '`
2. `" "`
3. `[[ ]]`

```
local phrase = [[My name is ]]
local name = 'P J'
print(phrase .. name) -- My name is P J

-- Strings and Numbers
local age = 21
local name = "Billy"
print(name .. " is " .. age .. " Years old")
```

Escape sequence characters are used in string to change the normal interpretation of characters.

For example, to print double inverted commas (`""`), we have to use `\"` in the string.

The escape sequence and its use is listed below in the table.

Escape Sequence	Use
`\a`	Bell
`\b`	Backspace
`\f`	Formfeed
`\n`	New line
`\r`	Carriage return
`\t`	Tab
`\v`	Vertical tab

`\\`	Backslash
`\"`	Double quotes
`\'`	Single quotes
`\[`	Left square bracket
`\]`	Right square bracket

It is possible to get the length of a string, as a number, by using the unary length operator (#):

```
print(#("This is a string")) --> 16
```

9.4.1. Concatenation

> In formal language theory and computer programming *string concatenation* is the operation of joining two character strings end-to-end.
>
> *Example 1. "snowball"*
>
> The concatenation of "snow" and "ball".

9.5. Other types

The four basic types in Lua (numbers, booleans, nil and strings) have been described in the previous sections, but four types are missing: functions, tables, userdata and threads.

Functions	Pieces of code that can be called, receive values and return values back.
Tables	Data structures that can be used for data manipulation.
Userdata	Used internally by applications Lua is embedded in to allow Lua to communicate with that program through objects controlled by the application.
Threads	Used by coroutines, which allow many functions to run at the *same time*.