Lua

scripting language

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com


https://www.facebook.com/tutorialspointindia


https://twitter.com/tutorialspoint

About the Tutorial

Lua is an open source language built on top of C programming language. Lua has its value across multiple platforms ranging from large server systems to small mobile applications.

This tutorial covers various topics ranging from the basics of Lua to its scope in various applications.

Audience

This tutorial is designed for all those readers who are looking for a starting point to learn Lua. It has topics suitable for both beginners as well as advanced users.

Prerequisites

It is a self-contained tutorial and you should be able to grasp the concepts easily even if you are a total beginner. However it would help if you have a basic understanding of working with a simple text editor and command line.

Copyright & Disclaimer

Table of Contents

1. OVERVIEW

Lua is an extensible, lightweight programming language written in C. It started as an in-house project in 1993 by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes.

It was designed from the beginning to be a software that can be integrated with the code written in C and other conventional languages. This integration brings many benefits. It does not try to do what C can already do but aims at offering what C is not good at: a good distance from the hardware, dynamic structures, no redundancies, ease of testing and debugging. For this, Lua has a safe environment, automatic memory management, and good facilities for handling strings and other kinds of data with dynamic size.

Features

Lua provides a set of unique features that makes it distinct from other languages. These include:

- Extensible
- Simple
- Efficient
- Portable
- Free and open

Example Code

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

How Lua is Implemented?

Lua consists of two parts - the Lua interpreter part and the functioning software system. The functioning software system is an actual computer application that can interpret programs written in the Lua programming language. The Lua interpreter is

written in ANSI C, hence it is highly portable and can run on a vast spectrum of devices from high-end network servers to small devices.

Both Lua's language and its interpreter are mature, small, and fast. It has evolved from other programming languages and top software standards. Being small in size makes it possible for it to run on small devices with low memory.

Learning Lua

The most important point while learning Lua is to focus on the concepts without getting lost in its technical details.

The purpose of learning a programming language is to become a better programmer; that is, to become more effective in designing and implementing new systems and at maintaining old ones.

Some Uses of Lua

- Game Programming
- Scripting in Standalone Applications
- Scripting in Web
- Extensions and add-ons for databases like MySQL Proxy and MySQL WorkBench
- Security systems like Intrusion Detection System.

2. ENVIRONMENT

Try it Option Online

We have already set up the Lua Programming environment online, so that you can build and execute all the available examples online at the same time when you are doing your theory work. This gives you confidence in what you are reading and to check the result with different options. Feel free to modify any example and execute it online.

Try the following example using our online compiler option available at http://www.compileonline.com/

```
#!/usr/local/bin/lua


print("Hello World!")
```

For most of the examples given in this tutorial, you will find a **Try it** option in our website code sections at the top right corner that will take you to the online compiler. So, just make use of it and enjoy your learning.

Local Environment Setup

If you are still willing to set up your environment for Lua programming language, you need the following softwares available on your computer - (a) Text Editor, (b) The Lua Interpreter, and (c) Lua Compiler.

Text Editor

You need a text editor to type your program. Examples of a few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

Name and version of the text editor can vary on different operating systems. For example, Notepad will be used on Windows, and vim or vi can be used on Windows as well as Linux or UNIX.

The files you create with your editor are called source files and these files contain the program source code. The source files for Lua programs are typically named with the extension "**.lua**".

The Lua Interpreter

It is just a small program that enables you to type Lua commands and have them executed immediately. It stops the execution of a Lua file in case it encounters an error unlike a compiler that executes fully.

The Lua Compiler

When we extend Lua to other languages/applications, we need a Software Development Kit with a compiler that is compatible with the Lua Application Program Interface.

Installation on Windows

There is a separate IDE named "SciTE" developed for the windows environment, which can be downloaded from http://code.google.com/p/luaforwindows/ download section.

Run the downloaded executable to install the Lua IDE.

Since it's an IDE, you can both create and build the Lua code using the same.

In case, you are interested in installing Lua in command line mode, you need to install MinGW or Cygwin and then compile and install Lua in windows.

Installation on Linux

To download and build Lua, use the following command:

```
$ wget http://www.lua.org/ftp/lua-5.2.3.tar.gz
$ tar zxf lua-5.2.3.tar.gz
$ cd lua-5.2.3
$ make linux test
```

In order to install on other platforms like aix, ansi, bsd, generic linux, mingw, posix, solaris by replacing Linux in make Linux, test with the corresponding platform name.

We have a helloWorld.lua, in Lua as follows:

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

Now, we can build and run a Lua file say helloWorld.lua, by switching to the folder containing the file using cd, and then using the following command:

```
$ lua helloWorld

We can see the following output.

hello world
```

Installation on Mac OS X

To build/test Lua in the Mac OS X, use the following command:

```
$ curl -R -O http://www.lua.org/ftp/lua-5.2.3.tar.gz

$ tar zxf lua-5.2.3.tar.gz

$ cd lua-5.2.3

$ make macosx test
```

In certain cases, you may not have installed the Xcode and command line tools. In such cases, you won't be able to use the make command. Install Xcode from mac app store. Then go to Preferences of Xcode, and then switch to Downloads and install the component named "Command Line Tools". Once the process is completed, make command will be available to you.

It is not mandatory for you to execute the "make macosx test" statement. Even without executing this command, you can still use Lua in Mac OS X.

We have a helloWorld.lua, in Lua, as follows:

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

Now, we can build and run a Lua file say helloWorld.lua by switching to the folder containing the file using cd and then using the following command:

```
$ lua helloWorld
```

We can see the following output:

```
hello world
```

Lua IDE

As mentioned earlier, for Windows SciTE, Lua IDE is the default IDE provided by the Lua creator team. The alternate IDE available is from ZeroBrane Studio, which is available across multiple platforms like Windows, Mac and Linux.

There are also plugins for eclipse that enable the Lua development. Using IDE makes it easier for development with features like code completion and is highly recommended. The IDE also provides interactive mode programming similar to the command line version of Lua.

3. BASIC SYNTAX

Let us start creating our first Lua program!

First Lua Program

Interactive Mode Programming

Lua provides a mode called interactive mode. In this mode, you can type in instructions one after the other and get instant results. This can be invoked in the shell by using the lua -i or just the lua command. Once you type in this, press Enter and the interactive mode will be started as shown below.

```
$ lua -i
$ Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio
quit to end; cd, dir and edit also available
```

You can print something using the following statement:

```
> print("test")
```

Once you press enter, you will get the following output:

```
'test'
```

Default Mode Programming

Invoking the interpreter with a Lua file name parameter begins execution of the file and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Lua program. All Lua files will have extension .lua. So put the following source code in a test.lua file.

```
print("test")
```

Assuming, lua environment is setup correctly, let's run the program using the following code:

```
$ lua test.lua
```

We will get the following output:

```
test
```

Let's try another way to execute a Lua program. Below is the modified test.lua file:

```
#!/usr/local/bin/lua

print("test")
```

Here, we have assumed that you have Lua interpreter available in your /usr/local/bin directory. The first line is ignored by the interpreter, if it starts with # sign. Now, try to run this program as follows:

```
$ chmod a+rx test.lua
$./test.lua
```

We will get the following output.

```
test
```

Let us now see the basic structure of Lua program, so that it will be easy for you to understand the basic building blocks of the Lua programming language.

Tokens in Lua

A Lua program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following Lua statement consists of three tokens:

```
io.write("Hello world, from ",_VERSION,"!\n")
```

The individual tokens are:

```
io.write
(
"Hello world, from ",_VERSION,"!\n"
)
```

Comments

Comments are like helping text in your Lua program and they are ignored by the interpreter. They start with --[[and terminates with the characters --]] as shown below:

```
--[[ my first program in Lua --]]
```

Identifiers

A Lua identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter 'A to Z' or 'a to z' or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9).

Lua does not allow punctuation characters such as @, $, and % within identifiers. Lua is a **case sensitive** programming language. Thus *Manpower* and *manpower* are two different identifiers in Lua. Here are some examples of the acceptable identifiers:

```
mohd       zara    abc   move_name  a_123
myname50   _temp   j     a23b9      retVal
```

Keywords

The following list shows few of the reserved words in Lua. These reserved words may not be used as constants or variables or any other identifier names.

and	break	do	else
elseif	end	false	for

function	if	in	local
nil	not	or	repeat
return	then	true	until
while			

Whitespace in Lua

A line containing only whitespace, possibly with a comment, is known as a blank line, and a Lua interpreter totally ignores it.

Whitespace is the term used in Lua to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the interpreter to identify where one element in a statement, such as int ends, and the next element begins. Therefore, in the following statement:

```
local age
```

There must be at least one whitespace character (usually a space) between local and age for the interpreter to be able to distinguish them. On the other hand, in the following statement:

```
fruit = apples + oranges   --get the total fruit
```

No whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish for readability purpose.

4. VARIABLES

A variable is nothing but a name given to a storage area that our programs can manipulate. It can hold different types of values including functions and tables.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because Lua is case-sensitive. There are eight basic types of values in Lua:

In Lua, though we don't have variable data types, we have three types based on the scope of the variable.

- **Global variables:** All variables are considered global unless explicitly declared as a local.
- **Local variables:** When the type is specified as local for a variable then its scope is limited with the functions inside their scope.
- **Table fields:** This is a special type of variable that can hold anything except nil including functions.

Variable Definition in Lua

A variable definition means to tell the interpreter where and how much to create the storage for the variable. A variable definition have an optional type and contains a list of one or more variables of that type as follows:

```
type variable_list;
```

Here, **type** is optionally local or type specified making it global, and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here:

```
local    i, j
local    i
local    a,c
```

The line **local i, j** both declares and defines the variables i and j; which instructs the interpreter to create variables named i, j and limits the scope to be local.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:

```
type variable_list = value_list;
```

Some examples are:

```
local d , f = 5 ,10 --declaration of d and f as local variables.
d , f = 5, 10;      --declaration of d and f as global variables.
d, f = 10           --[[declaration of d and f as global variables.
                        Here value of f is nil --]]
```

For definition without an initializer: variables with static storage duration are implicitly initialized with nil.

Variable Declaration in Lua

As you can see in the above examples, assignments for multiples variables follows a variable_list and value_list format. In the above example **local d , f = 5 ,10**, we have d and f in variable_list and 5 and 10 in values list.

Value assigning in Lua takes place like first variable in the variable_list with first value in the value_list and so on. Hence, the value of d is 5 and the value of f is 10.

Example

Try the following example, where variables have been declared at the top, but they have been defined and initialized inside the main function:

```
-- Variable definition:
local a, b
-- Initialization
a = 10
b = 30
```

```
print("value of a:", a)

print("value of b:", b)

-- Swapping of variables
b, a = a, b
print("value of a:", a)

print("value of b:", b)

f = 70.0/3.0
print("value of f", f)
```

When the above code is built and executed, it produces the following result:

```
value of a:     10
value of b:     30
value of a:     30
value of b:     10
value of f 23.333333333333
```

Lvalues and Rvalues in Lua

There are two kinds of expressions in Lua:

- **lvalue:** Expressions that refer to a memory location is called "lvalue" expression. An lvalue may appear as either the left-hand or right-hand side of an assignment.
- **rvalue:** The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it, which means an rvalue may appear on the right-hand side, but not on the left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and cannot appear on the left-hand side. Following is a valid statement:

```
g = 20
```

But following is not a valid statement and would generate a build-time error:

```
10 = 20
```

In Lua programming language, apart from the above types of assignment, it is possible to have multiple lvalues and rvalues in the same single statement. It is shown below.

```
g,l = 20,30
```

In the above statement, 20 is assigned to g and 30 is assigned to l.

5. DATA TYPES

Lua is a dynamically typed language, so the variables don't have types, only the values have types. Values can be stored in variables, passed as parameters and returned as results.

In Lua, though we don't have variable data types, but we have types for the values. The list of data types for values are given below.

Value 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 co-routines.
table	Represent ordinary arrays, symbol tables, sets, records, graphs, trees, etc., and implements associative arrays. It can hold any value (except nil).

Type Function

In Lua, there is a function called 'type' that enables us to know the type of the variable. Some examples are given in the following code.

```
print(type("What is my type"))   --> string
t=10
print(type(5.8*t))               --> number
print(type(true))                --> boolean
print(type(print))               --> function
print(type(type))                --> function
print(type(nil))                 --> nil
print(type(type(ABC)))           --> string
```

When you build and execute the above program, it produces the following result on Linux:

```
string
number
function
function
boolean
nil
string
```

By default, all the variables will point to nil until they are assigned a value or initialized. In Lua, zero and empty strings are considered to be true in case of condition checks. Hence, you have to be careful when using Boolean operations. We will know more using these types in the next chapters.

6. OPERATORS

An operator is a symbol that tells the interpreter to perform specific mathematical or logical manipulations. Lua language is rich in built-in operators and provides the following type of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Misc Operators

This tutorial will explain the arithmetic, relational, logical, and other miscellaneous operators one by one.

Arithmetic Operators

Following table shows all the arithmetic operators supported by Lua language. Assume variable **A** holds 10 and variable **B** holds 20, then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiply both operands	A * B will give 200
/	Divide numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0

^	Exponent Operator takes the exponents	A^2 will give 100
-	Unary - operator acts as negation	-A will give -10

Example

Try the following example to understand all the arithmetic operators available in the Lua programming language:

```
a = 21
b = 10
c = a + b
print("Line 1 - Value of c is ", c )
c = a - b
print("Line 2 - Value of c is ", c )
c = a * b
print("Line 3 - Value of c is ", c )
c = a / b
print("Line 4 - Value of c is ", c )
c = a % b
print("Line 5 - Value of c is ", c )
c = a^2
print("Line 6 - Value of c is ", c )
c = -a
print("Line 7 - Value of c is ", c )
```

When you execute the above program, it produces the following result:

```
Line 1 - Value of c is      31
Line 2 - Value of c is      11
Line 3 - Value of c is      210
Line 4 - Value of c is      2.1
```

```
Line 5 - Value of c is      1
Line 6 - Value of c is      441
Line 7 - Value of c is      -21
```

Relational Operators

Following table shows all the relational operators supported by Lua language. Assume variable **A** holds 10 and variable **B** holds 20 then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
~=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A ~= B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Example

Try the following example to understand all the relational operators available in the Lua programming language:

```
a = 21
b = 10

if( a == b )
then
   print("Line 1 - a is equal to b" )
else
   print("Line 1 - a is not equal to b" )
end

if( a ~= b )
then
   print("Line 2 - a is not equal to b" )
else
   print("Line 2 - a is equal to b" )
end

if ( a < b )
then
   print("Line 3 - a is less than b" )
else
   print("Line 3 - a is not less than b" )
end

if ( a > b )
then
```

```
   print("Line 4 - a is greater than b" )
else
   print("Line 5 - a is not greater than b" )
end

-- Lets change value of a and b
a = 5
b = 20
if ( a <= b )
then
   print("Line 5 - a is either less than or equal to  b" )
end

if ( b >= a )
then
   print("Line 6 - b is either greater than  or equal to b" )
end
```

When you build and execute the above program, it produces the following result:

```
Line 1 - a is not equal to b
Line 2 - a is not equal to b
Line 3 - a is not less than b
Line 4 - a is greater than b
Line 5 - a is either less than or equal to  b
Line 6 - b is either greater than  or equal to b
```

Logical Operators

Following table shows all the logical operators supported by Lua language. Assume variable **A** holds true and variable **B** holds false then:

Operator	Description	Example
and	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A and B) is false.
or	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A or B) is true.
not	Called Logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A and B) is true.

Example

Try the following example to understand all the logical operators available in the Lua programming language:

```
a = 5
b = 20

if ( a and b )
then
   print("Line 1 - Condition is true" )
end

if ( a or b )
then
   print("Line 2 - Condition is true" )
end

--lets change the value ofa and b
```

```
a = 0
b = 10

if ( a and b )
then
   print("Line 3 - Condition is true" )
else
   print("Line 3 - Condition is not true" )
end

if ( not( a and b) )
then
   print("Line 4 - Condition is true" )
else
   print("Line 3 - Condition is not true" )
end
```

When you build and execute the above program, it produces the following result:

```
Line 1 - Condition is true
Line 2 - Condition is true
Line 3 - Condition is true
Line 3 - Condition is not true
```

Misc Operators

Miscellaneous operators supported by Lua Language include **concatenation** and **length**.

Operator	Description	Example

..	Concatenates two strings.	a..b where a is "Hello " and b is "World", will return "Hello World".
#	A unary operator that returns the length of the a string or a table.	#"Hello" will return 5

Example

Try the following example to understand the miscellaneous operators available in the Lua programming language:

```
a = "Hello "
b = "World"

print("Concatenation of string a with b is ", a..b )

print("Length of b is ",#b )

print("Length of b is ",#"Test" )
```

When you build and execute the above program, it produces the following result:

```
Concatenation of string a with b is      Hello World
Length of b is   5
Length of b is   4
```

Operators Precedence in Lua

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example, x = 7 + 3 * 2; Here x is assigned 13, not 20 because operator * has higher precedence than + so, it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Unary	not # -	Right to left
Concatenation	..	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Relational	< > <= >= == ~=	Left to right
Equality	== ~=	Left to right
Logical AND	and	Left to right
Logical OR	or	Left to right

Example

Try the following example to understand all the precedence of operators in Lua programming language:

```
a = 20
b = 10
c = 15
d = 5

e = (a + b) * c / d;-- ( 30 * 15 ) / 5
print("Value of (a + b) * c / d is   :",e )
```

```
e = ((a + b) * c) / d; -- (30 * 15 ) / 5
print("Value of ((a + b) * c) / d is :",e )

e = (a + b) * (c / d);-- (30) * (15/5)
print("Value of (a + b) * (c / d) is :",e )

e = a + (b * c) / d;  -- 20 + (150/5)
print("Value of a + (b * c) / d is   :",e )
```

When you build and execute the above program, it produces the following result:

```
Value of (a + b) * c / d is   :    90
Value of ((a + b) * c) / d is :    90
Value of (a + b) * (c / d) is :    90
Value of a + (b * c) / d is   :    50
```

7. LOOPS

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: the first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Following is the general form of a loop statement in most of the programming languages: