LUA TRAINING

INTRO

Lua is an extensible, light-weight programming language written in C.

Features

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

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.

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
```

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

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

Now, we can build and run a Lua file say helloWorld.lua,

```
$ lua helloWorld
```

We can see the following output.

```
hello world
```

Indentifiers

- case sensitive language.
- Identifier starts with letter 'A to Z' or 'a to z' or an underscore '_'
- Lua does not allow punctuation characters such as @, $, and % within identifiers

Whitespace in Lua

- Describes blanks, tabs, newline characters and comments.
- Seperates one part of a statement from another.

Variables

In Lua 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.

```
local d , f = 5 ,10 --declaration of d and f as local variables
```

Type Function

Type enables us to know the type of the variable.

```
print(type("What is my type")) --> string
```

Data Types

- Variables don't have types, only the values have types.

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

Lua – Operators

Lua provides the following type of operators

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

Lua - Loops

Lua provides the following types of loops to handle looping requirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
repeat...until loop	Repeats the operation of group of statements till the until condition is met.
nested loops	You can use one or more loop inside any another *while, for or do..while* loop.

while loop

```
while( true )
do
    print("This loop will run forever.")
end
```

Nested loops

Lua programming language allows to use one loop inside another loop.

Break statement

When the break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

DECISION MAKING

Lua programming language provides the following types of decision making statements.

if statement	An if statement consists of a boolean expression followed by one or more statements.
if...else statement	An if statement can be followed by an optional else statement, which executes when the boolean expression is false.
nested if statements	You can use one if or else if statement inside another if or else if statement(s).

if statement

Syntax

```
if(boolean_expression)

then

 --[ statement(s) will execute if the boolean expression is true --]

end
```

Functions

Defining a Function

The general form of a method definition in Lua programming language is as follows:

```
optional_function_scope function function_name( argument1, argument2, argument3..., argumentn)

function_body

return result_params_comma_separated

end
```

Example

```
--[[ function returning the max between two numbers --]]

 function max(num1, num2)

          if (num1 > num2) then

                   result = num1;

          else

                   result = num2;

          end

          return result;
```

```
end
```

- Functions can be assigned to variables and also can pass them as parameters of another function
- It is possible to create functions with variable arguments in Lua using '...' as its parameter.

```
function average(...)
    result = 0
    local arg={...}
    for i,v in ipairs(arg) do
        result = result + v
    end
    return result/#arg --[[ #arg gives the count]]
end
print("The average is",average(10,5,3,4,5,6))
```

STRINGS

String is a sequence of characters

String can be initialized with three forms which includes:

- Characters between single quotes
- Characters between double quotes
- Characters between [[and]]

An example for the above three forms are shown below.

```
string1 = "Lua"
print("\"String 1 is\"",string1)
string2 = 'Tutorial'
print("String 2 is",string2)
string3 = [["Lua Tutorial"]]
print("String 3 is",string3)
```

Replacing a Substring

A sample code for replacing occurrences of one string with another is given below.

```
string = "Lua Tutorial"

-- replacing strings

newstring = string.gsub(string,"Tutorial","Language")

print("The new string is",newstring)
```

ARRAYS

One-Dimensional Array

A one-dimensional array can be represented using a simple table structure and can be initialized and read using a simple for loop. An example is shown below.

```
array = {"Lua", "Tutorial"}

 for i= 0, 2

do

        rint(array[i])

end
```

Multi-Dimensional Array

Multi-dimensional arrays can be implemented in two ways:

- Array of arrays
- Single dimensional array by manipulating indices

An example for multidimensional array of 3. 3 is shown below using array of arrays.

```
-- Initializing the array

array = {}

for i=1,3 do

        array[i] = {}

        for j=1,3 do

                array[i][j] = i*j

        end

end

-- Accessing the array

for i=1,3 do

        for j=1,3

        do
```

```
                    print(array[i][j])
          end
end
```

ITERATORS

Iterator is a construct that enables you to traverse through the elements of the so called collection or container.

Stateless Iterators

This type of iterator function does not retain any state.

Stateful Iterators

Each This type of iterator retains the state. To hold the state of the current element, closures are used.

```
function newCounter ()
      local i = 0
      return function ()   -- anonymous function
               i = i + 1
               return i
             end
    end

    c1 = newCounter()
    print(c1())  --> 1
    print(c1())  --> 2
```

- a closure is a function plus all it needs to access its upvalues correctly

FILE I/O

I/O library is used for reading and manipulating files in Lua.

Implicit File Descriptors

Implicit file descriptors use the standard input/output modes or using a single input and single output file.

MODE	DESCRIPTION
"*n"	Reads from the current file position and returns a number if exists at the file position or returns nil.
"*a"	Returns all the contents of file from the current file position.
"*l"	Reads the line from the current file position, and moves file position to next line.
number	Reads number of bytes specified in the function.

Example:

```
file = io.open("test.lua", "r")

io.input(file)

print(io.read()) -- prints the first line of the file

io.close(file)


file = io.open("test.lua", "a")

io.output(file) -- sets the default output file as test.lua

io.write("-- End of the test.lua file") -- appends a word test to the last line of the file

io.close(file)
```

Explicit File Descriptors

which allows us to manipulate multiple files at a time. These functions are quite similar to implicit file descriptors. Here, we use file:function_name instead of io.function_name.

- file:seek(optional whence, optional offset): Whence parameter is "set", "cur" or "end".
- file:flush(): Clears the default output buffer

TABLES

An associative array is an array that 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 to a table dynamically. Tables are the main (in fact, the only) data structuring mechanism in Lua, and a powerful one.

Representation and Usage

Tables are called objects and they are neither values nor variables. Lua uses a constructor expression {} to create an empty table. It is to be known that there is no fixed relationship between a variable that holds reference of table and the table itself.

```
Mytable = {}

mytable[1]= "Lua"

--removing reference

mytable = nil

-- lua garbage collection will take care of releasing memory
```

```
    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
```

Table Manipulation

There are in built functions for table manipulation and they are listed in the following table.

S.NO	Method & Purpose
1	table.concat (table [, sep [, i [, j]]]): Concatenates the strings in the tables based on the parameters given. See example for detail.
2	table.insert (table, [pos,] value): Inserts a value into the table at specified position.
3	table.maxn (table) Returns the largest numeric index.
4	table.remove (table [, pos]) Removes the value from the table
5	table.sort (table [, comp])

	Sorts the table based on optional comparator argument.