Dan Imbimbo

Professor Robila

CSIT 313-02 Project: Lua

Introduction

For my project, I chose the Lua programming language (utilizing the current 5.4 version). Lua is a fast, lightweight, embeddable, and user-friendly scripting language [1] that I would say bears many similarities to Python's syntax in a lot of ways, which most CSIT students at MSU are certainly familiar with. As just stated, Lua is embeddable and is typically done so alongside C languages, due to Lua's integration of the C API which allows C code to interact with that of Lua scripts [2]. Also, the language, much like Java, is a sort of hybrid as it compiles its code into bytecode to then be interpreted by the Lua Virtual Machine [3]. In the programming industry, Lua is most well known for its widespread use within video games such as Roblox, Angry Birds, World of Warcraft, Warframe, and even Grim Fandango (which was one of the first video games to show how Lua could be utilized during game development) [3][4]. The language hit its peak in popularity during 2011 as it reached a top 10 position in the TIOBE Programming Community Index (which measures the popularity of programming languages based on search engine results) [5], before dropping significantly in the years that followed [6]. However, Lua has recently seen a bit of a resurgence in the community over the past year, as it has reentered the top 20 most popular programming languages, as of March 2022 [7].

Getting Started

First and foremost, a user will want to download the precompiled binaries of the language by navigating to the sourceforge link on the official website (http://luabinaries.sourceforge.net/).

From there, the user should select the ‘Download’ option and choose the binaries for their respective operating system. Once the download has been completed, the user should extract the binaries to a folder and then add that folder to their system’s PATH variable; in Windows, this is done by going to ‘Environmental Variables…’ under their System Properties, then selecting ‘Edit’ on the ‘PATH’ variable listed under System variables, and then adding a new entry which points to the directory of the folder holding the Lua binaries. Once that is done, entering “lua54” (for Lua 5.4) anywhere under the Windows Command Prompt will start the Lua interpreter.

From here, I would recommend that the user downloads VS Code to use as their programming environment and, once they have installed and run VS Code, add the “Lua” extension. This will provide many helpful features when coding with the language, such as Auto Completion and IntelliSense. After this, I would also recommend they download the “Code Runner” extension, which will add a button to the top right of the coding window for simple execution of scripts as well as allowing for the ability to run only the segment of a script that is currently highlighted (alternatively the user could just manually open the ‘Terminal’ in VS Code and type “lua54 ./script.lua” in the directory of their script to run it in its entirety). With Code Runner, however, by default it will execute code under the ‘Output’ window of VS Code which does not allow for user input, so this should be changed to the ‘Terminal’ window of VS Code; and also, for Lua, the user will have to alter the settings of the extension’s ‘Executor Map’ so that it executes the “lua54” command instead of simply “lua”. These both can be done by adding the following lines to VSCode’s settings.json file:

```
{
"code-runner.runInTerminal": true,
    "code-runner.executorMap": {
        "lua":"lua54"
    }
}
```

And now with all of that out of the way, Lua scripts should be easy to run under VS Code!

Programming Under Lua:

-Reserved Words and Commenting

Lua is a case sensitive language with the following words being reserved and unable to be used as identifiers: and; break; do; else; elseif; end; false; for; function; goto; if; in; local; nil; not; or; repeat; return; then; true; until; while [8]. Also of note is how comments are delimited within Lua, which is done with two hyphens "--" for single line commenting, and with an opening "--[[" followed by a closing "]]" for block commenting; both being displayed in the following example [8]:

```
--This is a single line comment.
--[[
    This is a block comment.
]]
```

-Variables and Data Types

Lua happens to be a dynamically typed programming language, much like Python, so the data types for its variables are not explicitly declared by the programmer, but instead implicitly done so at runtime [9]. That said, the data type of a variable can be identified using the predefined "type()" function, which is demonstrated further below [9]. Lua also sets all variables (that are not nil) to be global variables by default [10], however a variable's scope can be limited to the code block it is declared in by placing the 'local' keyword in front of the variable's declaration statement [11].

Introduced in Lua 5.4 was the ability to create local constant variables by using the '<const>' attribute after the variable's name during its declaration [12], as well as to-be-closed

variables by using the '<close>' attribute which acts similarly to constants but have their values closed whenever the variable goes out of scope (which is useful for releasing resources) [13].

In Lua 5.4 the following eight data types exist: nil; number; string; boolean; table; function; userdata; threads [9].

- **Nil**

  NULL values in Lua are represented by the 'nil' data type, which acts almost exactly the same as 'None' in Python by defining the absence of a value [14].

  Input:

```
a=nil
print(type(a)) --prints the data type of a (nil)
```

  Output:

```
nil
```

- **Number**

  In Lua, all numerical values are categorized under the 'number' data type using the two subtypes 'integer' and 'float' [15]. That said, it did not originally allow for integers, instead only utilizing floating point numbers [16]. This functionality was only added in 2015 with the third revision of the fifth iteration of the language (Lua 5.3) [17].

  Input:

```
b=5
print(type(b)) --prints the data type of b (5)
c=3.14
print(type(c)) --prints the data type of c (3.14)
```

  Output:

```
number
number
```

Lua 5.3 also brought with it the ability to distinguish between the two subtypes of the 'number' data type, using the type function provided by the standard 'math' library [18].

Input:

```
b=5
print(math.type(b)) --prints the subtype of b (5)
c=3.14
print(math.type(c)) --prints the subtype of c (3.14)
```

Output:

```
integer
float
```

- **String**

Sequences of alphanumeric characters and symbols are represented in Lua through strings (delimited by either single or double quotation marks) and, just like Python, there is no separate data type for individual characters, as they are also simply categorized as strings [19]. Another parallel that Lua shares with Python is that its strings are immutable, meaning that the value of a string cannot be altered after it is defined [19]. Multiple strings and/or numbers (which are converted to strings) can also be joined together through concatenation, which is performed by using the '..' operator between the values or variables [20]. Lua also includes a length operator (denoted with a pound sign) which can be used to return the length of a given string [21].

Input:

```
d="This is a string."
print(type(d)) --prints the data type of d ("This is a string.")

e='a'
print(type(e)) --prints the data type of e ('a')

print(d..e) --concatenation of d and e
```

```
print(#d) --prints the length of the string d ("This is a string.")
```

Output:

```
string
string
This is a string.a
17
```

- **Boolean**

  The conditional values ‘true’ and ‘false’ are represented by the boolean data type in Lua, which acts the same as most programming languages with the caveat that ‘nil’ also acts as ‘false’ in conditional statements (both being called “false values”) [15].

  Input:

```
f=true
g=false
print(type(f)) --prints the data type of f (true)
print(type(g)) --prints the data type of g (false)
```

  Output:

```
boolean
boolean
```

- **Table**

  The table data type represents the primary data-structure of Lua, which is used for implementing arrays, records, lists, queues, sets, and more [15][22]. Tables are objects which have no fixed size (comparable to arrayLists in Java) and also bear similarities to associative arrays, as values can be indexed with non-numeric array keys [22]. The length operator can also be used on tables in order to return a border value of the table, meaning “any positive integer index present in the table that is followed by an absent index,” which is demonstrated in the input below [21]. (One important thing to note, however, is

that in Lua indexing of tables automatically begins with a ‘1’ instead of a ‘0’ as it does in most every other traditional programming language) [23]. There are many readily available functions for manipulating tables too, such as: ‘table.insert’ which allows you to *insert* an element at a specified location in a given table or, if a location is not specified, simply *insert* it directly after the border value; ‘table.remove’ which allows you to *remove* a specified element from a given table or, if a location is not specified, simply *remove* the element located at its border value; ‘table.sort’ which automatically sorts the elements of a given table in ascending order or through a comparison function; and ‘table.unpack’ which returns all of the elements from a given table [24]. (That said, each table function which accepts a table as an argument will ignore any of its elements which have been indexed with non-numerical values, i.e., array keys) [24].

Input:

```
h={1,2,3}
print(type(h)) --prints the data type of g ({1,2,3})

h["key"]="example of (key,value)"
print(h["key"]) --prints the value of h["key"]

i={10, 8, 6, 4, 2}
k={}
print(#i) --prints a border value (index) of i
print(#k) --prints a border value (index) of k

table.sort(i) --sorts the given table in ascending order
print(table.unpack(i)) --returns each of the elements of a table
```

Output:

```
table
example of (key,value)
5
0
2       4       6       8       10
```

- **Function**

  The function data type references individual procedures which have already been defined in either Lua or C [15]. In the Lua language, functions are treated as “first-class values” and can be “stored in variables, passed as arguments to other functions, and returned as results” [25]. (Functions will be discussed further within the section dedicated to them.)

  Input:

```
print(type(print)) --prints the data type of print
```

  Output:

```
function
```

- **Userdata & Threads**

  The userdata data type represents variables which are storing arbitrary data for a host C program, and as such there are no predefined operations for userdata within Lua [15]; infact, userdata values cannot even be created or altered within Lua but only through use of the C API [15]. Meanwhile, the thread data type represents separate threads of execution that are utilized for the implementation of coroutines [15], which allows for multiple functions to be suspended as well as resumed in order to coordinate their execution (these will also be discussed further in the ‘Functions’ section) [26].

-Operators

- **Arithmetic**

  Lua 5.4 has all of the traditional arithmetic operators that one would expect to be able to perform on numerical values, namely the unary operations of negation & exponentiation and the binary operations of addition, subtraction, multiplication, (float) division, and modulo [27]. The language, much like Python, also includes an operator for

‘floor division’ as demonstrated below [27], and performs automatic conversion on the operands of certain operations via coercion (which is evidenced by float division and exponentiation below) [28]. Also exhibited below is the ability to assign multiple values to multiple variables on one line via lists delimited by commas [29].

- **Unary**
  - **Negation**

    The negation operation, also known as “unary minus” [27], gives the value it is being performed on but with the reversed sign (meaning, a negative for a positive or a positive for a negative). This is done by placing a minus symbol before a variable/value as shown below.

    Input:

```
a=3
print(-a)
```

    Output:

```
-3
```

  - **Exponentiation**

    The exponentiation operation in Lua actually utilizes the C function of ‘pow’, which multiplies a given value by itself for the number of times that is specified. This is accessed by placing a caret symbol after a variable/value [27], which is then followed by the desired exponent as exhibited below. (Lua differs from Python where exponentiation is denoted by two asterisk symbols followed by the exponent.) And, as touched upon earlier, exponentiation coerces the result of the operation into a float [28].

Input:

```
a = 3
print(a^2)
```

Output:

```
9.0
```

➢ **Binary**

- **Addition & Subtraction**

  Addition, as well as subtraction, in Lua are performed as one would expect by providing two variables/values and a plus or minus symbol, respectively, between them [27] as demonstrated in the example below. The subtraction example provided also shows that when performing an operation between an integer and a float that Lua coerces the integer to a float, causing the difference to also be a float [28].

  Input:

```
a, b = 3, 2
print("Addition:"..(a+b))
b=2.0
print("Subtraction:"..(a-b))
```

  Output:

```
Addition:5
Subtraction:1.0
```

- **Multiplication**

  Much like addition and subtraction, there isn't anything really out of the ordinary when it comes to multiplying values in Lua. The user simply provides two operands that are separated by an asterisk (representing the

sign for multiplication) which computes the product [27], as shown in the following input.

Input:

```
a, b = 3, 2
print(a*b)
```

Output:

```
6
```

- **Float & Floor Division**

  As stated above, the Lua language provides two options for dividing values with float and floor division, akin to Python [27]. Float division, as the name suggests and as stated prior, computes the quotient as a float (including any decimal values therein) with the operator being denoted by a forward slash which is placed between two variables/values [27]; whereas, floor division is denoted by two forward slashes that are placed between two variables/values and computes the quotient rounded “towards negative infinity” (rounded downwards to the nearest integer) [27].

  Input:

```
a, b = 3, 2
print("Float Division:"..(a/b))
print("Floor Division:"..(a//b))
```

  Output:

```
Float Division:1.5
Floor Division:1
```

- **Modulo**

  The modulo operation was not originally included in Lua, only being added in the first revision of the fifth iteration of the language [30]. This

operation is now performed by placing a percent symbol between two operands, which calculates the remainder of their division with the quotient rounded “towards minus infinity” [27].

Input:

```
a, b = 3, 2
print(a%b)
```

Output:

```
1
```

- **Bitwise**

Lua did not have support for bitwise operators until rather recently with the 3rd revision of the fifth iteration of the language [17]. Bitwise operators in Lua function similarly to how they do in Python, by coercing their operands into integers (if they aren’t already) [28] which are then internally translated into their binary representations to have calculations performed on them in a bit by bit fashion before producing the result of the operation as an integer [31]. Lua has support for all of the same bitwise operators as Python, meaning it includes: bitwise AND; bitwise OR; bitwise XOR; bitwise left shift; bitwise right shift; and bitwise NOT [31].

  - **AND**

The bitwise AND operation, performed by placing an ampersand symbol between two variables/values, takes the binary representations of each and bit by bit checks them while returning a 1 for the position in binary if the bits of both variables/values are also 1s, otherwise it will return a 0 for the position [32]. After it compares each position between both numbers, it

will return the calculated binary result as an integer [31]. For example, as demonstrated below, the variable ‘a’ stores the value 5 (0101 in binary) and the variable ‘b’ stores the value 3 (0011 in binary) with the result of the operation being 1 (0001 in binary), due to both of the binary operands only having a 1 in the first position from the right.

Input:

```
a, b = 5, 3
print(a & b)
```

Output:

```
1
```

- **OR**

The bitwise OR operator, which is denoted by a bar symbol that is placed between two operands, takes the binary representations of its operands and bit by bit checks each while returning a 1 for that position in binary if the bits of either of the operands are 1s, otherwise it returns a 0 [32]. And, as previously stated, after each position is checked the calculated binary result will be returned as an integer [31]. In the following input, the variable ‘a’ stores the value 5 (0101 in binary) and the variable ‘b’ stores the value 3 (0011 in binary) with the result of the operation being 7 (0111 in binary), due to there being a 1 in either of the operands for the first three positions.

Input:

```
a, b = 5, 3
print(a | b)
```

Output:

```
7
```

- **XOR**

  Much like bitwise OR, the bitwise XOR operation, conducted by placing a tilde between two operands, takes the binary representations of them and bit by bit checks each; however, it will only return a 1 for the position in binary if only one of the operands has a 1 for that position and otherwise returns a 0 [32]. This operation also returns its calculated binary result as an integer [31]. Exhibited below, the variable ‘a’ stores the value 5 (0101 in binary) and the variable ‘b’ stores the value 3 (0011 in binary) with the result of the operation being 6 (0110 in binary), due to there only being a single 1 in the 3rd position of ‘a’ and a single 1 in the 2nd position of ‘b’.

  Input:

```
a, b = 5, 3
print(a ~ b)
```

  Output:

```
6
```

- **Left Shift**

  Bitwise left shift is utilized by placing two less than signs in the middle of two variables/values [31]. This operation shifts each bit in the binary representation of the left operand to the left by the number of bits specified by the right operand [32], with the shifted binary result being returned as an integer [31]. As shown in the following example, the variable ‘a’ stores 5 (0101 in binary) and each of its bits are to be shifted to the left by 3 (the value stored in the variable ‘b’). The result of this

operation is 40 (0010 1000 in binary) since the 1 in the first position of ‘a’ is shifted to the left by 3 and the 1 in the third position of ‘a’ is shifted to the left by 3.

Input:

```
a, b = 5, 3
print(a << b)
```

Output:

```
40
```

- **Right Shift**

  Bitwise right shift operations are performed with two greater than signs that are positioned between two operands and, similarly to the left shift operator, it shifts each bit in the binary representation of its left operand by the number of bits specified by its right operand; however, this time those bits are shifted to the right [32] while still returning the result as an integer [31]. This is demonstrated with the variable ‘a’, storing the value 5 (0101 in binary), having each of its bits shifted to the right by the value of the variable ‘b’, which is 3. The returned result is 0 because “displacements with absolute values equal to or higher than the number of bits in an integer result in zero,” since the result will end up having all of its bits shifted out as evidenced here [31].

  Input:

```
a, b = 5, 3
print(a >> b)
```

  Output:

```
0
```

- **NOT**

  The unary bitwise NOT operator is used by placing a tilde in front of a single variable/value which will return the number as a negative integer [31] with an additional bit added to it [32]. The input below showcases this with the NOT operator being placed before the variable ‘a’, representing the value of 5 (0101 in binary), and returning -6 which can be simply be calculated as -(0101 + 1) or -(0110).

  Input:

```
a = 5
print(~a)
```

  Output:

```
-6
```

- **Relational**

  Lua supports relational operations for making comparisons between multiple given values [33]. The included operators are all of the same ones which are offered in Python, namely: equality, inequality, less than, less than or equal to, greater than, and greater than or equal to [33].

  - **Equality**

    The equality operator, denoted by two consecutive equal signs, checks if the operands provided to its left and right are equivalent with each other and then returns the result as either true or false, but this comparison can only be evaluated by operands of the same type (comparisons between two different types always returns as false) [33]. The types that can be checked

for equality among two of themselves are numbers (checking if the numerical values are equivalent), strings (checking if they have the same byte content), objects (checking if they refer to the same object), and functions (checking for detectable differences between the two) [33].

Input:

```
a, b = 4,8
print(a==b)
print(a==print)
```

Output:

```
false
false
```

- **Inequality**

The inequality operator in Lua differs in notation from that of Python as it is written with a tilde, similarly to a bitwise NOT, that is then followed by an equal sign [33]. This evaluates as a direct negation to the equality operator, meaning it only returns the result as true if the operands are not equal to one another [33]. Therefore, for inequality, comparisons between two operands of different types will always return true, which is demonstrated within the example below through the comparison between the variable ‘a’ and the function ‘print’.

Input:

```
a, b = 4,8
print(a~=b)
print(a~=print)
```

Output:

```
true
true
```

- **Less than / or equal to**

  Unlike equality and inequality, a less than operation (denoted by a less than symbol between two values and returning either true or false based on if the first operand is smaller than the second) can only be performed on the types of number and string [34]. This limitation also extends to operations performed by the less than or equal operator (using a less than sign followed by an equal sign which is put between two values in order to determine if the first is equivalent to or smaller than the second) [34]. Examples containing both of these operators can be seen below.

  Input:

```
a, b = 4,8
print(a<b)
print(a<=b)
print(a<=a)
```

  Output:

```
true
true
true
```

- **Greater than / or equal to**

  Similar to both less than and less than or equal to, operations using the greater than operator (a greater than sign that is placed between two operands and returns true or false based on if the first is bigger than the

second) and the greater than or equal to operator (a greater than sign followed by an equal sign between two operands and returns true or false based on if the first is equivalent to or bigger that the second) can only be done on the types of number and string [34]. Examples containing both of these operators can be seen below.

Input:

```
a, b = 4,8
print(a>b)
print(a>=b)
print(a>=a)
```

Output:

```
false
false
true
```

- **Logical**

Lua also supports the same logical expressions as Python (and, or, not), all of which utilize truth tables for their evaluation and, also like in Python, mostly all return either the first or second argument of their expressions (instead of just true or false like in Java) based on their given result (this will be explained for each operator below) [35]. That said, Lua bears yet another similarity with Python in its use of short-circuit evaluation for the operations performed by both the AND operator and the OR operator [35]

- **AND**

  Logical AND operations (denoted by placing ‘and’ between two operands) use short-circuit evaluation, as mentioned above, which means that if the first argument is evaluated to false (or nil) then the second argument is not evaluated since the compound statement has already been determined to be false, and so the first argument simply gets returned [35]. This also means that the second argument is only ever returned if the first argument is true, as demonstrated in the following example with ‘1 and 2’.

  Input:

```
print(true and true)
print(false and (print("Short Circuit AND: Does not execute")))
print(1 and 2)
```

  Output:

```
true
false
2
```

- **OR**

  Similarly to logical ANDs, and as already stated prior, the logical OR operator (denoted by an ‘or’ which is positioned in between two operands) also uses short-circuit evaluation [35]. This causes the second argument of the operation to only ever be evaluated and returned if its first argument is determined to be false (or nil), meaning the first argument will simply be returned otherwise [35], which is displayed below with ‘1 or 2’.

  Input:

```
print(true or false)
```

```
print(true or (print("Short Circuit OR: Does not
execute")))
print(1 or 2)
```

Output:

```
true
true
1
```

- **NOT**

  Unlike the logical AND and OR operators, operations performed by the logical NOT operator (denoted with a ‘not’) simply return as true or false based on the negated result of the given expression [35]. This is showcased in the following input in which ‘not’ is placed before the expression ‘1 > 2,’ this expression is obviously false, however the logical NOT operator returns the negation which is true.

  Input:

```
print(not (1>2))
```

  Output:

```
true
```

There is also an order of precedence amongst all of these mentioned operators. In Lua 5.4, the highest priority is given to the [^] exponentiation operator, with second most priority being given to the unary operators ([not] logical NOT, [#] length, [-] negation, [~] bitwise NOT), which are followed by the multiplicative operators ([*] multiplication, [/] division, [//] floor division, [%] modulo), then the additive operators ([+] addition, [-] subtraction), next being the [..] concatenation operator, followed by both bitwise shifts ([<<] left and [>>] right), after that is the [&] bitwise AND operator, the [~] bitwise XOR operator, the [|] bitwise OR operator, all of the

relational operators ([<] less than, [>] greater than, [<=] less than or equal to, [>=] greater than or equal to, [~=] inequality, [==] equality), then the [and] logical AND operator, and with the lowest priority is the [or] logical OR operator [36]. However, parentheses take precedence over everything else and thus can be used to ensure an operation is evaluated first, and another thing to keep in mind when writing out expressions is that all of the binary operators are left associative except for exponentiation and concatenation [36].

-Control Structures

In terms of control structures, Lua allows for ‘if’ statements, ‘while’ loops, and ‘for’ loops (all of which function similarly to Python), as well as its somewhat uniquely named ‘repeat-until’ loops, but it does not support ‘switch’ statements (just like Python) [37]. Another similarity that Lua shares with Python, is that it allows programmers to optionally separate individual statements by using a semicolon (known as an “empty statement”) [38], however, unlike Python, Lua is not a language dictated by indentation [39] instead delimiting its code blocks via various keywords [38]. (This means that control structures can be written on one line, which I will briefly demonstrate in the ‘If’ subsection.) Another interesting aspect to the conditions of control structures in Lua is that all values aside from ‘false’ or ‘nil’ are considered to be ‘true’, this includes the number ‘0’ and empty string [40]. (An example of this will also be demonstrated in the ‘If’ subsection). Also of note, is that since the second revision of the fifth iteration of the language Lua has had support for ‘goto’ statements [41], which can be utilized to transfer the control of a program to a specified label within the same function it is used [40]. Alongside these ‘goto’ statements, Lua also has traditional ‘break’ statements (which allow for the ability to break out of loops when deemed necessary) [42]. (Examples of both of these will be demonstrated alongside the loops below.)

- **If**

If statements in Lua function the same as in most other programming languages, they accept boolean expressions as conditions and only the run code enclosed within the 'if' statement if their boolean expressions evaluate to true [43]. Upon first glance, their structure bears similarities to Python's 'if' statements as their conditions are not enclosed within parentheses (as they are in Java and C languages) but instead simply written after the 'if' keyword (however they still function properly if parentheses are included) [40]. That said, the similarities end there as Lua's conditions are not followed by a colon like in Python nor are they delimited by indentation. Lua instead uses the 'then' keyword to signal that everything that follows it (until an 'end' terminator, an 'elseif' word, or an 'else' keyword) is enclosed within the body of the if statement it is associated with [40].

Input:

```
--if (true)
flag = 0; --optional semicolon separator
if (flag) then print("flag is not false") end --all on one line
```

Output:

```
flag is true
```

As briefly mentioned, Lua allows for an optional 'else-part' to be added to an if statement through the 'else' keyword, which will execute if the condition of the 'if' evaluates to false [43].

Input:

```
--if (false)
flag = false
if (flag) then print("flag is not false") else print("flag is
false") end
```

Output:

```
flag is false
```

You can also nest other ‘if’ statements inside the ‘then-part’ of an ‘if’ statement or even its ‘else-part’, however each of these nested ‘if’ statements will require its own ‘end’ terminator [43].

Input:

```
--nested ifs
a, b = 1, 2
if a < 0 then
    print("a is less than 0")
    if b < a then
        print("b is less than a; b is also less than 0")
    end
else
    print("a is greater than 0")
    if b > a then
        print("b is greater a; b is also greater than 0")
    end
end
```

Output:

```
a is greater than 0
b is greater a; b is also greater than 0
```

As with other programming languages, Lua supports ‘elseif’ statements which will each only get evaluated (until one of them is true or an ‘else’ or ‘end’ is reached) if the condition of the ‘if’ statement they are associated with first evaluates to false [43].

Input:

```
--elseifs
b = 2
if b < 0 then
    print("b is less than 0")
elseif b > 1 then
```

```
    print("b is greater than 1")
else
    print("b is between 0 and 1")
end
```

Output:

```
b is greater than 1
```

- **While**

While loops in Lua, much like ‘if’ statements, delimit their bodies through the use of a keyword (in this case ‘do’) and an ‘end’ terminator [40]. Once the loop is entered its contents will be repeated until its associated boolean expression is evaluated to false [44]. The following example demonstrates a while loop associated with a variable that is initially assigned the value of ‘1’ and a loop with a condition that will only return false once the variable is greater than or equal to 10, with the body of the loop printing the current value of the variable and then incrementing its value by one. This should cause the loop to repeat 10 times, subsequently printing the numbers 1-10, but another ‘if’ has been nested inside this loop which contains a break statement (as mentioned earlier) that will end the loop at the start of its 5th iteration (causing only the numbers 1-4 to get printed).

Input:

```
counter = 1
while counter <= 10 do
    if counter == 5 then
        break
    end
    print(counter)
    counter = counter + 1
end
```

Output:

```
1
2
3
4
```

- **Repeat-until**

Python does not have ‘do while’ loops, nor does Lua, however the functionality of them is effectively the same as that of Lua’s ‘repeat-until’ loops [45]. Much like ‘do while’ loops, what separates ‘repeat-until’ loops from ‘while’ loops is the fact that they always execute at least once due to their boolean condition only being tested after the body of the loop [46]. However, unlike ‘while’ loops, ‘repeat-until’ loops are not delimited by ‘do’ or ‘end’ keywords but instead by the keywords of the loop itself, meaning ‘repeat’ and ‘until’ [40].

Input:

```
counter = 1
repeat
    print("loop executes at least one, even though condition is
already true")
until counter >= 1
```

Output:

```
loop executes at least one, even though condition is already true
```

- **For**

When it comes to ‘for’ loops, Lua has two kinds: the ‘numeric for’ loop, which functions very similarly to those of Java and C languages; and the ‘generic for’ loop which is much more like that of Python [47].

- **Numeric For**

  'Numeric for' loops are made up of three components separated by commas, an 'initial value', a 'limit', and a 'step' (which can optionally be left out and defaults to 1) [47]. The initial value is a variable which gets set before the start of the loop and then acts as the loop's counter, being incremented or decremented by whatever is chosen as the loop's step (be it a negative or positive value), and this causes the loop to end once it has reached the loop's limit [47]. This means that there is no true "statement 2" condition to the numeric for loop as there is in Java or C languages, however the limit could be viewed like a 'greater than or equal to' condition when decrementing the initial value and as a 'less than or equal to' condition when incrementing the initial value [47]. After the three components have been set, the body of the loop is then delimited by the keyword "do" and the terminator "end" [47]. One important thing to note, however, is that if the initial value and step of the loop are both not set to integers, then all three of the values will be converted to floats (thus being subject to floating-point accuracy) [47]. An example of a numeric for loop is provided below, along with a demonstration of a 'goto' statement's ability to jump to another segment of a program. In the input, the numerical for loop has been given an 'initial value' of 2, a 'limit' of 20, and a 'step' of positive 2, so under normal circumstance the initial value would increase by 2 on each loop until it is no longer less than or equal to 20; but, due to the 'goto' statement enclosed in the loop's nested 'if' statement, control of the program will instead jump to the label "skip" (denoted in Lua by two colons, a

string, and then two more colons [40]) upon the initial value reaching 10, so the remainder of the loop as well as the print statement after it will be skipped.

Input:

```
for i=2,20,2 do
    print(i)
    if(i == 10) then
        goto skip
    end
end
print("*loop end*")

::skip::

print("*program end*")
```

Output:

```
2
4
6
8
10
*program end*
```

- **Generic For**

‘Generic for’ loops utilize two variables which represent parts of an individual element in a sequence that is being iterated through via an “iterator function” (continuing to iterate until an index/key of ‘nil’ is reached in that sequence) [47]. There are two of these iterator functions provided for tables to be used by generic for loops: one is the ‘ipairs’ function, which takes a table (array) and returns upon each iteration of the loop the index of an element and its value; and the other is the ‘pairs’ function, which takes a table (associative array, also known as a dictionary in Python) and returns upon each iteration of the loop the key of the

element and its value [48]. An example of a generic for loop being used to iterate through an array and return the index and value of each element is showcased below.

Input:

```
table = {"dog", "cat", "monkey", "lion", "bear"}

for i, v in ipairs(table) do
    print(i, v)
end
```

Output:

```
1       dog
2       cat
3       monkey
4       lion
5       bear
```

-Functions

Functions in Lua are defined by the keyword ‘function’, followed by the name of the function, its arguments delimited by parentheses (which can be given as ‘...’ to indicate a variable number of arguments like with the standard print function) [49], the body of the function (containing whatever task it is to perform), and then an ‘end’ terminator [50]. As in Python, functions allow for code blocks to be modularized into subroutines [51] whose segments of code are only run when called to (by using the function’s name followed by its specified arguments enclosed within parentheses) [49]. However, as evidenced by their given definition in Lua and the example provided below, they differ in syntax from Python’s functions as there is no ‘def’ keyword, no colon to indicate where its body begins, and again there is no needed indentation (though the example is indented for readability). This following input demonstrates a simple

function which receives a table (array) and loops through its elements while calculating their sum, before returning it to where it was called from (in this case the ‘print’ function).

Input:

```
function getSum(array)
    sum = 0
    for i,v in ipairs(array) do
        sum = sum + v
    end
    return sum
end

odds = {1, 3, 5, 7, 9}
print(getSum(odds))
```

Output:

```
25
```

As just displayed, and much like in Python and other programming languages, functions created within Lua utilize ‘return’ statements for retrieving results from within functions [42] (and if the end of a function is reached with no ‘return’ statement having been found then the function simply returns with no results) [49]. And, like stated earlier in this report, Lua happens to treat functions as first class values which can be stored in variables [25], as well as allowing for multiple values to be assigned to multiple variables on one line as a list [29]. This all ties into a rather unique aspect of programming in Lua and that is the fact that multiple values can be returned from a given function [52]; in fact, the limit on the number of values that a function may return in Lua is “guaranteed to be greater than 1000” [49]. (Lua will also discard any returned values that do not match the given number of variables) [52]. The following example showcases this capability by returning five strings all from the same ‘multiReturn’ function to the variables ‘a’, ‘b’, ‘c’, and ‘d’.

Input:

```
function multiReturn()
    s1="This "
    s2="is "
    s3="all "
    s4="one "
    s5="return."
    return s1, s2, s3, s4, s5
end

a,b,c,d = multiReturn()
print(a..b..c..d)
```

Output:

```
This is all one
```

- **Coroutines**

As briefly discussed much earlier in the "Data Types and Variables" section of this report, Lua has a data type known as a 'thread' that makes the use of its 'coroutines' possible [15]. Coroutines, also known as "collaborative multithreading" since they do not run concurrently, are separate threads of execution (containing functions) which can be yielded and resumed on command [53]. To create a coroutine one must first create a new variable (a thread) that will store the function 'coroutine.create(),' and then define that given coroutine's function as the parameter to 'create()' [53]. Within that coroutine's function the user can specify whatever task they wish the coroutine to encompass, as well as dictate at which point they want the coroutine to yield its execution for other threads [53]. Once the coroutine has been properly defined it can be called to and started via 'coroutine.resume()', which is also how a coroutine that was previously suspended can be restarted and picked right back up where it had previously suspended; and, once this coroutine has finished its entire operation the thread will be marked 'dead' [53]. (A

basic example of this is shown below, along with some simple information that can be retrieved about coroutines.)

Input:

```
thread_1 = coroutine.create(
    function()
        print("Coroutine is executing. (x1)")
        coroutine.yield() --coroutine yields for other threads
        print("Coroutine is executing. (x2)")
        coroutine.yield() --coroutine again yields for other threads
        print("Coroutine is executing. (x3)")
    end
)

--Thread Information
print("data type: "..type(thread_1))
print(thread_1) --prints the thread's address
print("thread's coroutine status: "..coroutine.status(thread_1).."\n")

coroutine.resume(thread_1) --starts the thread's coroutine
coroutine.resume(thread_1) --restarts the thread's coroutine
coroutine.resume(thread_1) --restarts the thread's coroutine
print("thread's coroutine status: "..coroutine.status(thread_1)) --now
that the coroutine has completed
```

Output:

```
data type: thread
thread: 000000000142e2f8
thread's coroutine status: suspended

Coroutine is executing. (x1)
Coroutine is executing. (x2)
Coroutine is executing. (x3)
thread's coroutine status: dead
```

The memory that's still allocated by a thread after its coroutine has been marked dead will eventually be automatically freed up, since "Lua manages memory automatically by running a

garbage collector to collect all dead objects" [54]. (This functionality is very similar to the automatic garbage collection provided within Java.)

Conclusion

In conclusion, and as has hopefully been indicated thus far, there are many unique aspects to Lua such as with its expressive tables, its threads and coroutines, its ability for functions to return multiple values, its two types of for loops, and so much more (as I've only begun to scratch the surface of everything this language can do and still this report is as long as it is). Lua truly is exceedingly user-friendly in its simplicity with regards to read and writability, and it is all very easy to learn thanks to the extensive and freely accessible documentation that is provided through the languages' official website. Personally, I found this language to be really intriguing due to its scripting uses within the field of video games and I'm glad that I took the time to learn about it, so if anyone else is interested in learning even more about the language then they should be sure to check out the "Lua 5.4 Reference Manual" and the "Programming in Lua" text book (made up of approximately 30 chapters, but it's slightly outdated since its based on Lua 5.0) which are both available in an online format on lua.org.

References

[1] "About," *Lua*. [Online]. Available: https://www.lua.org/about.html. [Accessed: 03-Apr-2022].

[2] R. Ierusalimschy, "24 – an overview of the C API," *Programming in Lua : 24*, Dec-2003. [Online]. Available: https://www.lua.org/pil/24.html. [Accessed: 03-Apr-2022].

[3] S. Wickramasinghe, "The lua programming language beginner's guide," *BMC Blogs*, 04-Jun-2021. [Online]. Available: https://www.bmc.com/blogs/lua-programming-language/. [Accessed: 03-Apr-2022].

[4] "Showcase," *Lua*. [Online]. Available: https://www.lua.org/showcase.html. [Accessed: 03-Apr-2022].

[5] "TIOBE programming community index definition," *TIOBE*. [Online]. Available: https://www.tiobe.com/tiobe-index/programming-languages-definition/. [Accessed: 03-Apr-2022].

[6] "The Lua programming language," *TIOBE*. [Online]. Available: https://www.tiobe.com/tiobe-index/lua/. [Accessed: 03-Apr-2022].

[7] N. Kolakowski, "Lua enjoys programming language revival, powered by gaming," *Dice Insights*, 09-Mar-2022. [Online]. Available: https://insights.dice.com/2022/03/09/lua-enjoys-programming-language-revival-powered-by-gaming/. [Accessed: 03-Apr-2022].

[8] "3.1 – lexical conventions," *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#3.1. [Accessed: 04-Apr-2022].

[9] R. Ierusalimschy, "2 – types and values," *Programming in Lua : 2*, Dec-2003. [Online]. Available: https://www.lua.org/pil/2.html. [Accessed: 04-Apr-2022].

[10] R. Ierusalimschy, “1.2 – global variables,” *Programming in Lua : 1.2*, Dec-2003. [Online]. Available: https://www.lua.org/pil/1.2.html. [Accessed: 04-Apr-2022].

[11] R. Ierusalimschy, “4.2 – local variables and blocks,” *Programming in Lua : 4.2*, Dec-2003. [Online]. Available: https://www.lua.org/pil/4.2.html. [Accessed: 04-Apr-2022].

[12] “3.3.7 – local declarations,” *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: https://www.lua.org/manual/5.4/manual.html#3.3.7. [Accessed: 04-Apr-2022].

[13] “3.3.8 – to-be-closed variables,” *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: https://www.lua.org/manual/5.4/manual.html#3.3.8. [Accessed: 04-Apr-2022].

[14] R. Ierusalimschy, “2.1 – nil,” *Programming in Lua : 2.1*, Dec-2003. [Online]. Available: https://www.lua.org/pil/2.1.html. [Accessed: 04-Apr-2022].

[15] “2.1 – values and types,” *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#2.1. [Accessed: 04-Apr-2022].

[16] R. Ierusalimschy, “2.3 – numbers,” *Programming in Lua : 2.3*, Dec-2003. [Online]. Available: https://www.lua.org/pil/2.3.html. [Accessed: 03-Apr-2022].

[17] “Changes since lua 5.2,” *Lua 5.3 readme*. [Online]. Available: https://www.lua.org/manual/5.3/readme.html#changes. [Accessed: 03-Apr-2022].

[18] “6.7 – mathematical functions,” *Lua 5.3 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.3/manual.html#6.7. [Accessed: 04-Apr-2022].

[19] R. Ierusalimschy, “2.4 – strings,” *Programming in Lua : 2.4*, Dec-2003. [Online]. Available: https://www.lua.org/pil/2.4.html. [Accessed: 04-Apr-2022].

[20] “3.4.6 – concatenation,” *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#3.4.6. [Accessed: 04-Apr-2022].

[21] “3.4.7 – the length operator,” *Lua 5.4 reference manual*, Jun-2020. [Online]. Available:

http://www.lua.org/manual/5.4/manual.html#3.4.7. [Accessed: 25-Apr-2022].

[22] R. Ierusalimschy, "2.5 – tables," *Programming in Lua : 2.5*, Dec-2003. [Online]. Available: https://www.lua.org/pil/2.5.html. [Accessed: 04-Apr-2022].

[23] R. Ierusalimschy, "11.1 – arrays," *Programming in Lua : 11.1*, Dec-2003. [Online]. Available: https://www.lua.org/pil/11.1.html. [Accessed: 03-Apr-2022].

[24] "6.6 – table manipulation," *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#6.6. [Accessed: 27-Apr-2022].

[25] R. Ierusalimschy, "2.6 – functions," *Programming in Lua : 2.6*, Dec-2003. [Online]. Available: https://www.lua.org/pil/2.6.html. [Accessed: 04-Apr-2022].

[26] "Lua - coroutines," *tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/lua/lua_coroutines.htm. [Accessed: 04-Apr-2022].

[27] "3.4.1 – arithmetic operators," *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#3.4.1. [Accessed: 04-Apr-2022].

[28] "3.4.3 – coercions and conversions," *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#3.4.3. [Accessed: 04-Apr-2022].

[29] R. Ierusalimschy, "4.1 – assignment," *Programming in Lua : 4.1*, Dec-2003. [Online]. Available: https://www.lua.org/pil/4.1.html. [Accessed: 04-Apr-2022].

[30] "Version history," *Lua*. [Online]. Available: http://www.lua.org/versions.html#5.1. [Accessed: 12-Apr-2022].

[31] "3.4.2 – bitwise operators," *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#3.4.2. [Accessed: 13-Apr-2022].

[32] "Python bitwise operators," *GeeksforGeeks*, 25-Mar-2022. [Online]. Available: https://www.geeksforgeeks.org/python-bitwise-operators/. [Accessed: 13-Apr-2022].

[33] “3.4.4 – relational operators,” *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#3.4.4. [Accessed: 19-Apr-2022].

[34] R. Ierusalimschy, “3.2 – Relational Operators,” *Programming in Lua : 3.2*, Dec-2003. [Online]. Available: https://www.lua.org/pil/3.2.html. [Accessed: 25-Apr-2022].

[35] “3.4.5 – logical operators,” *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#3.4.5. [Accessed: 25-Apr-2022].

[36] “3.4.8 – precedence,” *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#3.4.8. [Accessed: 25-Apr-2022].

[37] R. Ierusalimschy, “4.3 – Control Structures,” *Programming in Lua : 4.3*, Dec-2003. [Online]. Available: https://www.lua.org/pil/4.3.html. [Accessed: 03-Apr-2022].

[38] “3.3.1 – blocks,” *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#3.3.1. [Accessed: 27-Apr-2022].

[39] “Code indentation in lua programming,” *tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/code-indentation-in-lua-programming. [Accessed: 04-Apr-2022].

[40] “3.3.4 – control structures,” *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#3.3.4. [Accessed: 27-Apr-2022].

[41] “Changes since lua 5.1,” *Lua 5.2 readme*. [Online]. Available: https://www.lua.org/manual/5.2/readme.html#changes. [Accessed: 28-Apr-2022].

[42] R. Ierusalimschy, “4.4 – break and return,” *Programming in Lua : 4.4*, Dec-2003. [Online]. Available: https://www.lua.org/pil/4.4.html. [Accessed: 27-Apr-2022].

[43] R. Ierusalimschy, “4.3.1 – if then else,” *Programming in Lua : 4.3.1*, Dec-2003. [Online]. Available: https://www.lua.org/pil/4.3.1.html. [Accessed: 03-Apr-2022].

[44] R. Ierusalimschy, “4.3.2 – while,” *Programming in Lua : 4.3.2*, Dec-2003. [Online]. Available: https://www.lua.org/pil/4.3.2.html. [Accessed: 03-Apr-2022].

[45] “Lua - repeat...until loop,” *tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/lua/lua_repeat_until_loop.htm. [Accessed: 04-Apr-2022].

[46] R. Ierusalimschy, “4.3.3 – repeat,” *Programming in Lua : 4.3.3*, Dec-2003. [Online]. Available: https://www.lua.org/pil/4.3.3.html. [Accessed: 03-Apr-2022].

[47] “3.3.5 – For Statement,” *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#3.3.5. [Accessed: 30-Apr-2022].

[48] R. Ierusalimschy, “7.3 – stateless iterators,” *Programming in Lua : 7.3*, Dec-2003. [Online]. Available: https://www.lua.org/pil/7.3.html. [Accessed: 30-Apr-2022].

[49] “3.4.11 – function definitions,” *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#3.4.11. [Accessed: 02-May-2022].

[50] “3.4.10 – function calls,” *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#3.4.10. [Accessed: 02-May-2022].

[51] R. Ierusalimschy, “5 – functions,” *Programming in Lua : 5*, Dec-2003. [Online]. Available: https://www.lua.org/pil/5.html. [Accessed: 02-May-2022].

[52] R. Ierusalimschy, “5.1 – multiple results,” *Programming in Lua : 5.1*, Dec-2003. [Online]. Available: https://www.lua.org/pil/5.1.html. [Accessed: 02-May-2022].

[53] “2.6 – coroutines,” *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: http://www.lua.org/manual/5.4/manual.html#2.6. [Accessed: 02-May-2022].

[54] “2.5 – garbage collection,” *Lua 5.4 reference manual*, Jun-2020. [Online]. Available: https://www.lua.org/manual/5.4/manual.html#2.5. [Accessed: 02-May-2022].