We Don't Have ChaptersThe Crash Course to Lua

CRASHING INTO LUA

We will start off this tutorial with a complete, working program that will most likely make absolutely no sense to you. Don't fret, we will go through the code line by line to quickly get you acquainted with the feel of Lua.

What Drives the Universe?

Let's begin with a grandiose task: to define the factors of the Universe.

What we type in:

```lua
function get_all_factors(number)
    --[[--
    Gets all of the factors of a given number

    @Parameter: number
        The number to find the factors of

    @Returns: A table of factors of the number
    --]]--
    local factors = {}
    for possible_factor=1, math.sqrt(number), 1 do
        local remainder = number%possible_factor

        if remainder == 0 then
            local factor, factor_pair = possible_factor, number/possible_factor
            table.insert(factors, factor)

            if factor ~= factor_pair then
                table.insert(factors, factor_pair)
            end
        end
    end

    table.sort(factors)
    return factors
end

--The Meaning of the Universe is 42. Let's find all of the factors driving the Universe.

the_universe = 42
factors_of_the_universe = get_all_factors(the_universe)

--Print out each factor

print("Count",  "The Factors of Life, the Universe, and Everything")
table.foreach(factors_of_the_universe, print)
```

What comes out:

```
Count   The Factors of Life, the Universe, and Everything
1       1
2       2
3       3
4       6
5       7
6       14
7       21
8       42
```

Whoa, what just happened?

We've just successfully calculated the factors of the universe. More specifically, we've just factored the answer to **Life, the Universe, and Everything**. If you've ever taken a basic mathematics course in school, you should remember that factors of a number will divide wholly into the number without leaving a remainder. For example, the factors of 4 are **1, 2, and 4**.

Now let's dissect the code. You may not even know where to begin, but worry not, that was never the expectation. Let's look at the first line

Function Definition The First Line

```lua
function get_all_factors(number)
    ...
end
```

As you may recall from Algebra, a function takes in a number and outputs another. In Lua, this notion is extended in that ANY object can be passed into a function and ANY object type can be outputted. A function is created by writing `function name`. In our case, our function is called **get_all_factors**. The object that is passed into a function is called a **parameter**. When you create a function, you have to know what you need to pass into the function, therefore you need to explicitly state what the parameters are. You define the parameters by enclosing them between parenthesis after the name of the function. In this case, our parameter is an object called **number**.

You can also create a function with zero or multiple parameters, thus the following statement is also validly Lua.

```lua
function someFunction(parameter, anotherParameter)
    --Valid
end
```

We also have to close off a function with a trailing **end** keyword. Everything in between the first **)** and the **end** keyword is called a **chunk**. The chunk consists of the actual code of the function.

We'll now look at a few **statements** that are not valid function declarations.

You cannot create a function with a name that has a space in it.

```lua
function some Function(parameter, anotherParameter)
    -- Not Valid
end
```

You cannot create a function with Function or some other form of function (Lua is case-sensitive)

```lua
Function someFunction(parameter, anotherParameter)
    -- Not Valid
end
```

Not using commas in the parameter list.

```lua
function someFunction(parameter anotherParameter)
    -- Not Valid
end
```

Forgetting to close off the parameter list parenthesis set.

```lua
function someFunction(parameter, anotherParameter
    -- Not Valid (missing a ')')
end
```

The following is a valid statement, however the style is not recommended.

```lua
function
    someFunction
        (
            parameter
                ,
                    anotherParameter
                        )
                            -- Valid
                             end
```

Block Comments The Next Line... and then some

```lua
--[[--
Gets all of the factors of a given number

@Parameter: number
    The number to find the factors of

@Returns: A table of factors of the number
--]]--
```

In case that you haven't figured it out yet, **--** denotes the beginning of a single line**comment**. Comments are not parsed, and they exist for the sole purpose of annotating the source code. For example:

What we type in:

```lua
print("Break me off a piece of that... Football Cream?") -- jk, it's KitKat Bar
```

What comes out:

```
Break me off a piece of that... Football Cream?
```

At the same time, we can create **multilined comments** by enclosing the lines within **--[[ ]]**. Note that bracketed comments do not extend towards the end of the line, so the following will just as well work

```
print("Break me off a piece of that... Football Cream?"--[[jk,
it's KitKat Bar]])
```

Creating Variables Look ma, I can count

There would be no point in making computers if the computer can't remember anything. Every piece of data that is stored within the computer is stored in an identifiable location. In Lua, data are stored within **variables**, which are identified (or named) by a single word. Below are some examples of variable declarations:

Give me a number

```
CanIHazANomNom = 1000
```

Give me some text

```
GIMMEH = "HAI WORLD"
```

Give me a table. A table is a list of objects enclosed between {}

```
BunchOfText = {"I Do Not Like Them Sam I Am", "I Do Not Like Green
Eggs and Ham", "I Would Not Like Them Here or There", "I Would Not
Like Them Anywhere"}
```

Give me a boolean

```
NoYouMoron = false
```

Copy a variable

```
AreWeGonnaDieIn2012 = NoYouMoron
```

Switching a value when creating the variable. If the first value is false or nil, then the variable switches to the second value.

```
Apocalypse = AreWeGonnaDieIn2012 or "2012's a joke."
```

Don't give me anything - Note, assigning the value nil to a variable will destroy it.

```
Nothing = nil
```

I can even have a function

```
someFunction = function(parameter, anotherParameter) end
```

Assigning a value of a completely different type to an existing value

```
Nothing = GIMMEH
```

Assigning multiple variables

```
A, B, C, D = "a", "b", "oops I took both c and d" -- A is "a", B is
"b", C is "oops I took both c and d", and D is nil
```

We can also create a **local** variable by adding the keyword **local** before the variable declaration, such as

```lua
local factors = {}
```

A local variable can only be used within the chunk/block where it is declared. This may seem confusing at first, so consider the following example:

What we type in:

```lua
function GreenEggsAndHam()
    local Sam = "I Am Sam"
end
print(Sam)
```

What comes out:

```
nil
```

Since the variable **Sam** is declared as a local variable inside the function**GreenEggsAndHam** only, we can't use it in the **Global Scope**, or the scoping level of the running program. Seems simple now doesn't it? However, what happens if a local variable is declared with the same name as a global variable? For example

What we type in:

```lua
Sam = "Sam I Am"
function GreenEggsAndHam()
    local Sam = "I Am Sam"
end
print(Sam)
```

What comes out:

```
Sam I Am
```

Well, it seems as if changes made to the local variable do not effect the global variable**Sam**. Therefore, you can think of local variables as isolated variables.

For Loops What goes around doesn't always come around.

```lua
for possible_factor=1, math.sqrt(number), 1 do
    local remainder = number%possible_factor

    if remainder == 0 then
        local factor, factor_pair
= possible_factor, number/possible_factor
        table.insert(factors, factor)

        if factor ~= factor_pair then
            table.insert(factors, factor_pair)
        end
    end
end
```

This might seems a bit complicated at this time. Let's take a look at a simpler **for-loop**.

What we type in:

```
MAXIMUM = 10
STEP = 1
for the_number = 1, MAXIMUM, STEP do
    print("The Number is now ",the_number)
end
```

What comes out:

```
The Number is now       1
The Number is now       2
The Number is now       3
The Number is now       4
The Number is now       5
The Number is now       6
The Number is now       7
The Number is now       8
The Number is now       9
The Number is now       10
```

The syntax of a **counting for-loop** is `for variable=start, maximum (variable < maximum), increment do`. Every time that the loop executes the block of code, the variable is automatically incremented by the **increment** value. Let's look at a few other examples of counting for-loops.

The Maximum is now 5, so the script will only print 5 times

What we type in:

```
for the_number = 1, 5, 1 do
    print("The Number is now ",the_number)
end
```

What comes out:

```
The Number is now       1
The Number is now       2
The Number is now       3
The Number is now       4
The Number is now       5
```

The incremental step size is now 2, so the script will only print the every other time.

What we type in:

```
for the_number = 1, 10, 2 do
    print("The Number is now ",the_number)
end
```

What comes out:

```
The Number is now       1
The Number is now       3
The Number is now       5
The Number is now       7
The Number is now       9
```

You can also simplify the expression by omitting the increment, which is automatically assumed to be 1

What we type in:

```lua
for the_number = 1, 5 do
    print("The Number is now ",the_number)
end
```

What comes out:

```
The Number is now       1
The Number is now       2
The Number is now       3
The Number is now       4
The Number is now       5
```

Now going back to the original example

```lua
for possible_factor=1, math.sqrt(number), 1 do
    ...
end
```

Therefore, if we call **get_all_factors** with **9** as number (`get_all_factors(9)`), then we can simplify the loop down to:

```lua
for possible_factor = 1, 3, 1 do
```

Conditional Statements true or false

Conditional statements are simple if switches. For example

```lua
if true then print("Hi Mom") end
```

The blocks within these conditional statements will only run if the if-statement **DOES NOT** evaluate to **false** or **nil** Below are some examples:

if true then

```lua
its_2010 = true
if its_2010 then
    print("It's 2010 now")
end
```

if the variable exists then - This is useful to make sure that variables exist since undeclared variables evaluates to nil.

```lua
function SayHiTo(name)
    if name then
        print("Hello",name)
    end
end

SayHiTo("Jack")
SayHiTo()
```

```
Hello   Jack
```

if ... **else** ... end - We can make our previous example more useful.

```
function SayHiTo(name)
    if name then
        print("Hello",name)
    else
        print("Hello Stranger")
    end
end

SayHiTo("Jack")
SayHiTo()
```

```
Hello   Jack
Hello Stranger
```

not - Negates the statement

```
function SayHiTo(name)
    if not name then
        print("Hello Stranger")
    else
        print("Hello", name)
    end
end

SayHiTo("Jack")
SayHiTo()
```

```
Hello   Jack
Hello Stranger
```

Simple **equality**, checking if two values are equal

```
function I_want_to_find(the_number)
    for a_number = 1, 100 do
        if the_number == a_number then
            print ("I found him!")
        else
            print ("Sorry, he's not here.")
        end
    end
end
I_want_to_find(50)
I_want_to_find(0)
```

```
I found him!
Sorry, he's not here.
```

inequality, checking if two values are not equal

```
print (3 ~= 4)
```

```
true
```

Operators Boo Math

The Modulus Operator (%)

The Modulus operator is usually the most obscure integer operator, therefore we'll start off with it. It basically gets the remainder between the division of two numbers. For example:

```
print(14 % 5) -- Outputs 4
```

Remainder division is very useful to find multiples of a number. For example, if we want to find out if a number is even, all you have to do is to write the following

```
function is_it_even(some_number)
    return some_number%2 == 0
end
```

Whenever some_number/2 is a whole number, or when the remainder between some_number and 2 is 0, then the function will return true.

Let's take a look at line 12 of our example program:

```
12  local remainder = number%possible_factor
```

Following what we've just discussed, the variable remainder will equal zero (**0**) whenever the variable **number** is a multiple of **possible_factor**. Since the goal of the function is to find all factors of a number, by looping through all of the numbers between 1 and the number and checking if number is a multiple of possible_factor will in turn reveal all of the factors of the number. Indeed, the next line directly checks to see if the remainder of the two numbers is 0.

The Other Common Operators (+ - * / ^)

If you've ever used the Microsoft Windows Calculator, you probably have an idea of what each of these symbols represent. **+** stands for plus, **-** stands for minus, ***** stands for multiplication, **/** stands for division, and **^** stands for **to the power of**. These operators will only work on numbers and numbers only. Therefore, the statement

```
print("Hello"+2010)
```

is not valid and will raise an error.

Concatenation (..)

Unlike most other languages, Lua do not allow **string concatenation** (joining two strings together) via the **+** operator. Instead, you will have to use **..** (two periods) to concatenate two strings. For example:

```
print("I".."am".."Sam")
```

Will print out **IamSam**

Length (#)

The length operator **#** (hash-mark, sharp) is unique in Lua in that most other languages have length as either a builtin function or an object method. Lua on the other hand uses # as a prefix operator to determine the length of either a **string** or a **table**.

```lua
bag_of_stuff = {"do", "re", "me", "fa", "so", "la", "si"}
print("I haz "..#bag_of_stuff.." things")
```

Note that using the # operator on anything besides a string or a table will result in an error.

The Assignment Operator (=)

As you may have guessed, the statement `me = "me"` assigns the value "me" to the variable **me**. The **=** operator however does not query whether two expressions are equal to each other (this is done through the **==** operator) so take care to not confuse the two.

We will spend very little time discussing the uses of these operators as they are, at their very worst, trivial. If you still find yourself having trouble with these operators, then I would suggest that you take a look at http://lua.lickert.net/operators/index_en.html

Back to the Example

We now have most of the required materials that are needed to fully interpret the example code. There are only three functions that we have not discussed yet, and I will briefly discuss them right now.

math.sqrt(number) returns the square root of the number.
table.insert(table, element) inserts the object element into the table table
table.sort(table) sorts a table of numbers in ascending order.

Now we're ready to delve back into the example. Let's start back at the for loop. Whenever we attempt to disect a loop, we should always consider single iterations, that is, what the Lua interpreter executes within a single step in the for loop. In this case, we will let number equal 10 and possible_factor equal 2

```lua
local remainder = number%possible_factor

if remainder == 0 then
    local factor, factor_pair = possible_factor, number/possible_factor
    table.insert(factors, factor)

    if factor ~= factor_pair then
        table.insert(factors, factor_pair)
    end
end
```

12 The variable remainder equals 10%2 or 0, as 2 divides wholly into 10.
14 Since remainder does equal 0, we will be able to execute lines 15 through 20.
15 We declare factor to equal to possible_factor, or 2, and factor_pair to equal to number/possible_factor, or 10/2 = 5. Think about it, if 2 is a factor of 10, and 2*5 equals 10, then would that also mean 5 is a factor of 10 as well?
16 Insert 2 into the table factors.
18 Because the square root of a number is also its factor, there are certain cases when factor and factor_pair are the same number. In these cases, we want to avoid inserting the same number into the table twice.
possible_factor is now 3, start over at line 12 again

If we are to continue this loop, then we would have to follow the following logic:

12 **remainder** equals 10%3, or 1 (10-3*3=1)
14 Our remainder is not 0, therefore we skip lines 15 through 20.
possible_factor is now 4, which is larger than math.sqrt(10), so we leave the for-loop

And by the time that possible_factor reaches 4, we have already scanned through all of the possible factors of the number 10. Therefore, we now have stored all of the factors of the number. However, before we hand those factors over, we want to sort them in ascending order.

```
table.sort(factors)
```

After which we can finally hand over, or return, the value back and exit the function.

```
return factors
```

Outputting the Table

The rest of the code concerns with printing out the results of the table. You will need to be acquainted with the **table.foreach(table, func)** function which does nothing more than go through a table and calling the function in the second parameter field**func**, in this case print, with the element of the tables as the parameter.

```
--The Meaning of the Universe is 42. Let's find all of the factors driving the Universe.

the_universe = 42
factors_of_the_universe = get_all_factors(the_universe)

--Print out each factor

print("Count",  "The Factors of Life, the Universe, and Everything")
table.foreach(factors_of_the_universe, print)
```

In which case we will find the following output:

```
Count   The Factors of Life, the Universe, and Everything
1       1
```

```
2        2
3        3
4        6
5        7
6        14
7        21
8        42
```

That's it for our crash course, the crash course was meant to be a guide to get you quickly acquainted with the language style of Lua. It will do nothing more or less that just that. I will see you in a future chapter of this tutorial.