

# Lua

## Cheatsheet

P J

# Contents

<b>1</b>	<b>Lua Cheatsheet</b>	<b>3</b>
1.1	Hello, World! . . . . .	3
1.2	Comments . . . . .	3
1.3	Syntax . . . . .	4
1.4	Obtaining Lua . . . . .	4
<b>2</b>	<b>Expressions</b>	<b>5</b>
2.1	Types . . . . .	5
2.2	Literals . . . . .	11
2.3	Coercion . . . . .	11
<b>3</b>	<b>Statements</b>	<b>12</b>
3.1	Assignment . . . . .	12
3.2	Conditional statement . . . . .	16

# 1 Lua Cheatsheet

Lua is a powerful, fast, lightweight and embeddable programming language.

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

## 1.1 Hello, World!

We can use `print()` function:

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

or `io.write()` method:

```
1 io.write("Hello, World!\n")
```

## 1.2 Comments

A comment is a code annotation that is ignored by the programming language.

There are some way of creating comments in Lua:

```
1 -- this is a comment
2 print ("Hello") -- another comment
```

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

```
1 --[[
2     This is a multi-line comment
3 ]]
4 io.write("Just a simple test\n")
5 --[==[
6 This is a comment that contains
7 a closing long bracket of level 0 which is here:
8 ]]
```

```
9  However, the closing double bracket doesn't make the comment end,  
10 because the comment was opened with an opening long bracket of level 2,  
11 and only a closing long bracket of level 2 can close it.  
12 ]==]
```

## 1.3 Syntax

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

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

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

```
1  -- Different types  
2  local x = 10 -- number  
3  local name = "Neo" -- string  
4  local is_alive = false -- boolean  
5  local a = nil -- no value or invalid value
```

## 1.4 Obtaining Lua

Lua can be obtained on the official Lua website, on the [download page](#).

## 2 Expressions

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

```
1 3 + 5
2 -- The code above is erroneous because all it contains is an expression.
3 -- The computer cannot execute '3 + 5', since that does not make sense.
```

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

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

### 2.1 Types

Lua is a dynamically typed language, so the variables don't have types, only the values have types.

But we have types for the values. There is a function called 'type' that enables us to know the type of the variable.

```
1 print(type("What is my type")) --> string
2 t = 10
3
4 print(type(5.8 * t))           --> number
5 print(type(true))              --> boolean
6 print(type(print))             --> function
7 print(type(nil))               --> nil
8 print(type(type(ABC)))         --> 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.

The list of data types for values are given below.

Value Type	Description
<code>nil</code>	Used to differentiate the value from having some data or no( <code>nil</code> ) data.
<code>boolean</code>	Includes <code>true</code> and <code>false</code> as values.
<code>number</code>	Represents real(double precision floating point)
<code>string</code>	Represents array of characters.
<code>function</code>	Represents a method that is written in C or Lua.
<code>userdata</code>	Represents arbitrary C data.
<code>thread</code>	Represents independent threads of execution and
<code>table</code>	Represent ordinary arrays, symbol tables, sets, records, graphs, trees, etc., and implements associative arrays. It can hold any value (except <code>nil</code> ).

### 2.1.1 Numbers

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

Numbers can be constructed as integers, decimal numbers, decimal exponents or even in hexadecimal. Here are some valid numbers:

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

```

1  -- The Lua
2  local a = 1
3  local b = 2
4  local c = a + b

```

```

5  print(c) -- 3
6
7  local d = b - a
8  print(d)
9
10 local x = 1 * 3 * 4 -- 12
11 print(x)
12
13 local y = (1+3) * 2 -- 8
14 print(y)
15
16 print(10 / 2) -- 5
17 print(2 ^ 2) -- 4
18 print(5 % 2) -- 1
19 print(-b) -- -2
20
21 -- Incerment
22 local level = 1
23 level = level + 1
24 print(level)

```

## Arithmetic operations

The operators for numbers in Lua are the following:

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

## Integers

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

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

### 2.1.2 Nil

Nil is the type of the value nil, whose main property is to be different from any other value; it usually represents the absence of a useful value. A function that would return nil, for example, is a function that has nothing useful to return.

### 2.1.3 Boolean

A boolean value can be either true or false, but nothing else. This is literally written in Lua as true or false, which are reserved keywords. The following operators are often used with boolean values, but can also be used with values of any data type:

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

Essentially, the not operator just negates the boolean value (makes it false if it is true and



makes it true if it is false), the `and` operator returns true if both are true and false if not and the `or` operator returns true if either of arguments is true and false otherwise.

```
1 local is_alive = true
2 print(is_alive) -- true
3
4 local is_alive = false
5 print(is_alive) -- false
```

## 2.1.4 Strings

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

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

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

```
1 local phrase = [[My name is ]]
2 local name = 'P J'
3 print(phrase .. name) -- My name is P J
4
5 -- Strings and Numbers
6 local age = 21
7 local name = "Billy"
8 print(name .. " is " .. age .. " Years old")
```

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

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

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

Escape Sequence	Use
\a	Bell
\b	Backspace
\f	Formfeed
\n	New line
\r	Carriage return
\t	Tab
\v	Vertical tab
\\	Backslash
\"	Double quotes
\'	Single quotes
\[	Left square bracket
\]	Right square bracket

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

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

## Concatenation

In [formal language theory](#) and [computer programming](#), **string concatenation** is the operation of joining two character [strings](#) end-to-end. For example, the concatenation of “snow” and “ball” is “snowball”.

The string concatenation operator in Lua is denoted by two dots (..).

```
1 print("snow" .. "ball") --> snowball
```

## 2.1.5 Other types

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

- *Functions* are pieces of code that can be called, receive values and return values back.
- *Tables* are data structures that can be used for data manipulation.
- *Userdata* are used internally by applications Lua is embedded in to allow Lua to communicate with that program through objects controlled by the application.

- *Threads* are used by coroutines, which allow many functions to run at the same time.

## 2.2 Literals

Literals are notations for representing fixed values in source code. All values can be represented as literals in Lua except **threads** and **userdata**.

String literals (literals that evaluate to strings), for example, consist of the text that the string must represent enclosed into single quotes, double quotes or long brackets.

Number literals, on the other hand, consist the number they represent expressed using decimal notation (ex: 12.43), scientific notation (ex: 3.1416e-2 and 0.31416E1) or hexadecimal notation (ex: 0xff).

## 2.3 Coercion

Coercion is the conversion of a value of one data type to a value of another data type. Lua provides automatic coercion between string and number values. Any arithmetic operation applied to a string will attempt to convert this string to a number.

Conversely, whenever a string is expected and a number is used instead, the number will be converted to a string.

This applies both to Lua operators and to default functions (functions that are provided with the language).

```
1 print("122" + 1) --> 123
2 print("The number is " .. 5 .. ".") --> The number is 5.
```

Coercion of numbers to strings and strings to numbers can also be done manually with the `tostring` and `tonumber` functions.

The former accepts a number as an argument and converts it to a string, while the second accepts a string as an argument and converts it to a number (a different base than the default decimal one can optionally be given in the second argument).

## 3 Statements

*Statements* are pieces of code that can be executed and that contain an instruction and expressions to use with it. Some statements will also contain code inside of themselves that may, for example, be run under certain conditions. Dissimilarly to expressions, they can be put directly in code and will execute.

Lua has few instructions, but these instructions, combined with other instructions and with complex expressions, give a good amount of control and flexibility to the user.

### 3.1 Assignment

Programmers frequently need to be able to store values in the memory to be able to use them later. This is done using variables.

*Variables* are references to a value which is stored in the computer's memory. They can be used to access a number later after storing it in the memory.

*Assignment* is the instruction that is used to assign a value to a variable. It consists of the name of the variable the value should be stored in, an equal sign, and the value that should be stored in the variable:

```
1 variable = 43
2 print(variable) --> 43
```

As demonstrated in the above code, the value of a variable can be accessed by putting the variable's name where the value should be accessed.

#### 3.1.1 Identifiers

**Identifiers**, in Lua, are also called names. They can be any text composed of letters, digits, and underscores and **not beginning with a digit**. They are used to name variables and table fields.

Here are some **valid** names:

- name
- hello
- \_
- \_tomatoes

- me41
- \_
- \_thisIs\_StillaValid23name

Here are some **invalid** names:

- 2hello : starts with a digit
- th\$i : contains a character that isn't a letter, a digit or an underscore
- hel!o : contains a character that isn't a letter, a digit or an underscore
- 563text : starts with a digit
- 82\_something : starts with a digit

Also, the following keywords are reserved by Lua and can not be used as names:

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

When naming a variable or a table field, you must choose a valid name for it. It must therefore start with a **letter** or an **underscore** and only contain **letters**, **underscores** and **digits**.

Note that Lua is case sensitive. This means that Hello and hello are two different names.

### 3.1.2 Scope

The **scope of a variable**, is the region of the code of the program where that variable is meaningful. The examples of variables you have seen before are all examples of global variables, variables which can be accessed from anywhere in the program.

Local variables, on the other hand, can only be used from the region of the program in which they were defined and in regions of the program that are located inside that region of the program. They are created exactly in the same way as global variables, but they must be prefixed with the `local` keyword.

The `do` statement will be used to describe them. The `do` statement is a statement that has no other purpose than to create a new block of code, and therefore a new scope. It ends with the `end` keyword:

```
1 local variable = 13 --[[ This defines a local variable that can be accessed from
2    anywhere in the script since it was defined in the main region. ]]
```

```

3  do
4      -- This statement creates a new block and also a new scope.
5      variable = variable + 5 -- This adds 5 to the variable, which now equals 18.
6      local variable = 17 --[[ This creates a variable with the same name as the
7          previous variable, but this one is local to the scope created by
8          the do statement. ]]
9      variable = variable - 1 --[[ This subtracts 1 from the local variable, which
10         now equals 16. ]]
11     print(variable) --> 16
12 end
13 print(variable) --> 18

```

When a scope ends, all the variables in it are **gotten rid of**. Regions of code can use variables defined in regions of code they are included in, but if they “overwrite” them by defining a local variable with the same name, that local variable will be used instead of the one defined in the other region of code. This is why the first call to the print function prints 16 while the second, which is outside the scope created by the do statement, prints 18.

In practice, only local variables should be used because they can be defined and accessed faster than global variables, since they are stored in registers instead of being stored in the environment of the current function, like global variables.

Registers are areas that Lua uses to store local variables to access them quickly, and can only usually contain up to 200 local variables. The processor, an important component of all computers, also has registers, but these are not related to Lua’s registers. Each function (including the main thread, the core of the program, which is also a function) also has its own environment, which is a table that uses indices for the variable names and stores the values of these variables in the values that correspond to these indices.

### 3.1.3 Forms of assignment

Some assignment patterns are sufficiently common for syntactic sugar to have been introduced to make their use simpler.

#### Augmented assignment

*Augmented assignment*, which is also called *compound assignment*, is a type of assignment that gives a variable a value that is relative to its previous value. It is used when it is

necessary to change the value of a variable in a way that is relative to its previous value, such as when that variable's value must be incremented.

In C, JavaScript, Ruby, Python and some other languages, the code `a += 8` will increment the value of `a` by 8. Lua does not have syntactic sugar for augmented assignment, which means that it is necessary to write `a = a + 8`.

## Chained assignment

*Chained assignment* is a type of assignment that gives a single value to many variables. The code `a = b = c = d = 0`, for example, would set the values of `a`, `b`, `c` and `d` to 0 in C and Python. In Lua, this code will raise an error because Lua does not have syntactic sugar for chained assignment, so it is necessary to write the previous example like this:

```
1 d = 0
2 c = d -- or c = 0
3 b = c -- or b = 0
4 a = b -- or a = 0
```

## Parallel assignment

*Parallel assignment*, which is also called *simultaneous assignment* and *multiple assignment*, is a type of assignment that simultaneously assigns different values (they can also be the same value) to different variables. Unlike chained assignment and augmented assignment, parallel assignment is available in Lua.

```
1 a, b, c, d = 0, 0, 0, 0
```

- If you provide more variables than values, some variables will not be assigned any value.
- If you provide more values than variables, the extra values will be ignored.

More technically, the list of values is adjusted to the length of list of variables before the assignment takes place, which means that excess values are removed and that extra `nil` values are added at its end to make it have the same length as the list of variables.

- If a function call is present *at the end of the values list*, the values it returns will be added at the end of that list, unless the function call is put between parentheses.

Moreover, unlike most programming languages Lua enables reassignment of variables' values through *permutation*. For example:

```

1 first_variable, second_variable = 54, 87
2 first_variable, second_variable = second_variable, first_variable
3 print(first_variable, second_variable) --> 87 54

```

This works because the assignment statement evaluates all the variables and values before assigning anything. Assignments are performed as if they were really simultaneous, which means you can assign at the same time a value to a variable and to a table field indexed with that variable's value before it is assigned a new value. In other words, the following code will set `dictionary[2]`, and not `dictionary[1]`, to 12.

```

1 dictionary = {}
2 index = 2
3 index, dictionary[index] = index - 1, 12

```

## 3.2 Conditional statement

Conditional statements are instructions that check whether an expression is true and execute a certain piece of code if it is. If the expression is not true, they just skip over that piece of code and the program continues. In Lua, the only conditional statement uses the `if` instruction. `false` and `nil` are both considered as false, while everything else is considered as true.

```

1 local number = 6
2 if number < 10 then
3     print("The number " .. number .. " is smaller than ten.")
4 end
5 --[[ Other code can be here and it will execute regardless of whether the code in
6 the conditional statement executed. ]]

```

In the code above, the variable `number` is assigned the number 6 with an assignment statement. Then, a conditional statement checks if the value stored in the variable `number` is smaller than ten, which is the case here. If it is, it prints "The number 6 is smaller than ten."

It is also possible to execute a certain piece of code only if the expression was not true by using the `else` keyword and to chain conditional statements with the `elseif` keyword:



```

1  local number = 15
2  if number < 10 then
3      print("The number is smaller than ten.")
4  elseif number < 100 then
5      print("The number is bigger than or equal to ten, but smaller than one hundred.")
6  elseif number ~= 1000 and number < 3000 then
7      print("number is bigger or equal to 100, smaller than 3000 is not exactly 1000.")
8  else
9      print("number is either 1000 or bigger than 2999.")
10 end

```

Note that the `else` block must always be the last one. There cannot be an `elseif` block after the `else` block. The `elseif` blocks are only meaningful if none of the blocks that preceded them was executed.

Operators used to compare two values, some of which are used in the code above, are called relational operators. If the relation is true, they return the boolean value `true`. Otherwise, they return the boolean value `false`.

operator	action
<code>==</code>	equal to
<code>~=</code>	not equal to
<code>&lt;=</code>	less than or equal to
<code>&gt;=</code>	more than or equal to
<code>&lt;</code>	less than
<code>&gt;</code>	more than

The code above also demonstrates how the `and` keyword can be used to combine many boolean expressions in a conditional expression.