LEARNING Lua

#lua

Table of Contents

About

Chapter 1: Getting started with Lua

Remarks

Lua is minimalistic, lightweight and embeddable scripting language. It's being designed, implemented, and maintained by a team at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil. The mailing list is open to get involved.

Common use-cases for Lua includes scripting video games, extending applications with plugins and configs, wrapping some high-level business logic or just embedding into devices like TVs, cars, etc.

For high performance tasks there is independent implementation using just-in-time-compiler available called LuaJIT.

Versions

Version	Notes	Release Date
1.0	Initial, non-public release.	1993-07-28
1.1	First public release. Conference paper describing it.	1994-07-08
2.1	Starting with Lua 2.1, Lua became freely available for all purposes, including commercial uses. Journal paper describing it.	1995-02-07
2.2	Long strings, the debug interface, better stack tracebacks	1995-11-28
2.4	External `luac` compiler	1996-05-14
2.5	Pattern matching and vararg functions.	1996-11-19
3.0	Introduced auxlib, a library for helping writing Lua libraries	1997-07-

Version	Notes	Release Date
		01
3.1	Anonymous functions and function closures via "upvalues".	1998-07-11
3.2	Debug library and new table functions	1999-07-08
3.2.2		2000-02-22
4.0	Multiple states, "for" statements, API revamp.	2000-11-06
4.0.1		2002-07-04
5.0	Coroutines, metatables, full lexical scoping, tail calls, booleans move to MIT license.	2003-04-11
5.0.3		2006-06-26
5.1	Module system revamp, incremental garbage collector, metatables for all types, luaconf.h revamp, fully reentrant parser, variadic arguments.	2006-02-21
5.1.5		2012-02-17
5.2	Emergency garbage collector, goto, finalizers for tables.	2011-12-16
5.2.4		2015-03-07
5.3	Basic UTF-8 support, bitwise ops, 32/64bit integers.	2015-01-12
5.3.4	Latest version.	2017-01-12

Examples

Installation

Binaries

Lua binaries are provided by most GNU/Linux distributions as a package.

For example, on Debian, Ubuntu, and their derivatives it can be acquired by executing this:

```
sudo apt-get install lua50
```

```
sudo apt-get install lua51
```

```
sudo apt-get install lua52
```

There are some semi-official builds provided for Windows, MacOS and some other operating systems hosted at SourceForge.

Apple users can also install Lua easily using Homebrew:

```
brew install lua
```

(Currently Homebrew has 5.2.4, for 5.3 see Homebrew/versions.)

Source

Source is available at the official page. Acquisition of sources and build itself should be trivial. On Linux systems the following should be sufficient:

```
$ wget http://lua.org/ftp/lua-5.3.3.tar.gz
$ echo "a0341bc3d1415b814cc738b2ec01ae56045d64ef ./lua-5.3.3.tar.gz" | sha1sum -c -
$ tar -xvf ./lua-5.3.3.tar.gz
$ make -C ./lua-5.3.3/ linux
```

In the example above we're basically downloading a source `tarball` from the official site, verifying its checksum, and extracting and executing `make`. (Double check the checksum at the official page .)

Note: you must specify what build target you want. In the example, we specified `linux`. Other available build targets include `solaris`, `aix`, `bsd`, `freebsd`, `macosx`, `mingw`, etc. Check out `doc/readme.html`, which is included in the source, for more details. (You can also find the latest version of the README online.)

Modules

Standard libraries are limited to primitives:

- `coroutine` - coroutine management functionality
- `debug` - debug hooks and tools
- `io` - basic IO primitives
- `package` - module management functionality
- `string` - string and Lua specific pattern matching functionality

- `table` - primitives for dealing with an essential but complex Lua type - tables
- `os` - basic OS operations
- `utf8` - basic UTF-8 primitives (since Lua 5.3)

All of those libraries can be disabled for a specific build or loaded at run-time.

Third-party Lua libraries and infrastructure for distributing modules is sparse, but improving. Projects like LuaRocks, Lua Toolbox, and LuaDist are improving the situation. A lot of information and many suggestions can be found on the older Lua Wiki, but be aware that some of this information is quite old and out of date.

Comments

Single-line comments in Lua start with `--` and continue until the end of line:

```
-- this is single line comment
-- need another line
-- huh?
```

Block comments start with `--[[` and end with `]]`:

```
--[[
    This is block comment.
    So, it can go on...
    and on...
    and on....
]]
```

Block comments use the same style of delimiters as long strings; any number of equal signs can be added between the brackets to delimit a comment:

```
--[=[
    This is also a block comment
    We can include "]]" inside this comment
--]=]

--[==[
    This is also a block comment
    We can include "]=]" inside this comment
--]==]
```

A neat trick to comment out chunks of code is to surround it with `--[[` and `--]]`:

```
--[[
    print'Lua is lovely'
--]]
```

To reactivate the chunk, simply append a `-` to the comment opening sequence:

```
---[[
    print'Lua is lovely'
--]]
```

This way, the sequence `--` in the first line starts a single-line comment, just like the last line, and the `print` statement is not commented out.

Taking this a step further, two blocks of code can be setup in such a way that if the first block is commented out the second won't be, and visa versa:

```
---[[
  print 'Lua is love'
--[=[]]
  print 'Lua is life'
--]=]
```

To active the second chunk while disabling the first chunk, delete the leading `-` on the first line:

```
--[[
  print 'Lua is love'
--[=[]]
  print 'Lua is life'
--]=]
```

Executing Lua programs

Usually, Lua is being shipped with two binaries:

- `lua` - standalone interpreter and interactive shell
- `luac` - bytecode compiler

Lets say we have an example program (`bottles_of_mate.lua`) like this:

```
local string = require "string"

function bottle_take(bottles_available)

    local count_str = "%d bottles of mate on the wall."
    local take_str = "Take one down, pass it around, " .. count_str
    local end_str = "Oh noes, " .. count_str
    local buy_str = "Get some from the store, " .. count_str
    local bottles_left = 0

    if bottles_available > 0 then
         print(string.format(count_str, bottles_available))
         bottles_left = bottles_available - 1
         print(string.format(take_str, bottles_left))
    else
        print(string.format(end_str, bottles_available))
        bottles_left = 99
        print(string.format(buy_str, bottles_left))
    end

    return bottles_left
end
```

```
local bottle_count = 99

while true do
    bottle_count = bottle_take(bottle_count)
end
```

The program itself can be ran by executing following on Your shell:

```
$ lua bottles_of_mate.lua
```

Output should look like this, running in the endless loop:

```
Get some from the store, 99 bottles of mate on the wall.
99 bottles of mate on the wall.
Take one down, pass it around, 98 bottles of mate on the wall.
98 bottles of mate on the wall.
Take one down, pass it around, 97 bottles of mate on the wall.
97 bottles of mate on the wall.
...
...
3 bottles of mate on the wall.
Take one down, pass it around, 2 bottles of mate on the wall.
2 bottles of mate on the wall.
Take one down, pass it around, 1 bottles of mate on the wall.
1 bottles of mate on the wall.
Take one down, pass it around, 0 bottles of mate on the wall.
Oh noes, 0 bottles of mate on the wall.
Get some from the store, 99 bottles of mate on the wall.
99 bottles of mate on the wall.
Take one down, pass it around, 98 bottles of mate on the wall.
...
```

You can compile the program into Lua's bytecode by executing following on Your shell:

```
$ luac bottles_of_mate.lua -o bottles_of_mate.luac
```

Also bytecode listing is available by executing following:

```
$ luac -l bottles_of_mate.lua


main <./bottles.lua:0,0> (13 instructions, 52 bytes at 0x101d530)
0+ params, 4 slots, 0 upvalues, 2 locals, 4 constants, 1 function
    1    [1]    GETGLOBAL    0 -1    ; require
    2    [1]    LOADK        1 -2    ; "string"
    3    [1]    CALL         0 2 2
    4    [22]    CLOSURE      1 0    ; 0x101d710
    5    [22]    MOVE         0 0
    6    [3]    SETGLOBAL    1 -3    ; bottle_take
    7    [24]    LOADK        1 -4    ; 99
    8    [27]    GETGLOBAL    2 -3    ; bottle_take
    9    [27]    MOVE         3 1
    10    [27]    CALL         2 2 2
    11    [27]    MOVE         1 2
    12    [27]    JMP          -5    ; to 8
    13    [28]    RETURN       0 1
```

```
function <./bottles.lua:3,22> (46 instructions, 184 bytes at 0x101d710)
1 param, 10 slots, 1 upvalue, 6 locals, 9 constants, 0 functions
    1    [5]    LOADK        1 -1    ; "%d bottles of mate on the wall."
    2    [6]    LOADK        2 -2    ; "Take one down, pass it around, "
    3    [6]    MOVE         3 1
    4    [6]    CONCAT       2 2 3
    5    [7]    LOADK        3 -3    ; "Oh noes, "
    6    [7]    MOVE         4 1
    7    [7]    CONCAT       3 3 4
    8    [8]    LOADK        4 -4    ; "Get some from the store, "
    9    [8]    MOVE         5 1
    10    [8]    CONCAT       4 4 5
    11    [9]    LOADK        5 -5    ; 0
    12    [11]    EQ           1 0 -5    ; - 0
    13    [11]    JMP          16    ; to 30
    14    [12]    GETGLOBAL    6 -6    ; print
    15    [12]    GETUPVAL     7 0    ; string
    16    [12]    GETTABLE     7 7 -7    ; "format"
    17    [12]    MOVE         8 1
    18    [12]    MOVE         9 0
    19    [12]    CALL         7 3 0
    20    [12]    CALL         6 0 1
    21    [13]    SUB          5 0 -8    ; - 1
    22    [14]    GETGLOBAL    6 -6    ; print
    23    [14]    GETUPVAL     7 0    ; string
    24    [14]    GETTABLE     7 7 -7    ; "format"
    25    [14]    MOVE         8 2
    26    [14]    MOVE         9 5
    27    [14]    CALL         7 3 0
    28    [14]    CALL         6 0 1
    29    [14]    JMP          15    ; to 45
    30    [16]    GETGLOBAL    6 -6    ; print
    31    [16]    GETUPVAL     7 0    ; string
    32    [16]    GETTABLE     7 7 -7    ; "format"
    33    [16]    MOVE         8 3
    34    [16]    MOVE         9 0
    35    [16]    CALL         7 3 0
    36    [16]    CALL         6 0 1
    37    [17]    LOADK        5 -9    ; 99
    38    [18]    GETGLOBAL    6 -6    ; print
    39    [18]    GETUPVAL     7 0    ; string
    40    [18]    GETTABLE     7 7 -7    ; "format"
    41    [18]    MOVE         8 4
    42    [18]    MOVE         9 5
    43    [18]    CALL         7 3 0
    44    [18]    CALL         6 0 1
    45    [21]    RETURN       5 2
    46    [22]    RETURN       0 1
```

Getting Started

variables

```
var = 50 -- a global variable
print(var) --> 50
do
  local var = 100 -- a local variable
  print(var) --> 100
end
print(var) --> 50
-- The global var (50) still exists
-- The local var (100) has gone out of scope and can't be accessed any longer.
```

types

```
num = 20 -- a number
num = 20.001 -- still a number
str = "zaldrizes buzdari iksos daor" -- a string
tab = {1, 2, 3} -- a table (these have their own category)
bool = true -- a boolean value
bool = false -- the only other boolean value
print(type(num)) --> 'number'
print(type(str)) --> 'string'
print(type(bool)) --> 'boolean'
type(type(num)) --> 'string'

-- Functions are a type too, and first-class values in Lua.
print(type(print)) --> prints 'function'
old_print = print
print = function (x) old_print "I'm ignoring the param you passed me!" end
old_print(type(print)) --> Still prints 'function' since it's still a function.
-- But we've (unhelpfully) redefined the behavior of print.
print("Hello, world!") --> prints "I'm ignoring the param you passed me!"
```

The special type `nil`

Another type in Lua is `nil`. The only value in the `nil` type is `nil`. `nil` exists to be different from all other values in Lua. It is a kind of non-value value.

```
print(foo) -- This prints nil since there's nothing stored in the variable 'foo'.
foo = 20
print(foo) -- Now this prints 20 since we've assigned 'foo' a value of 20.

-- We can also use `nil` to undefine a variable
foo = nil -- Here we set 'foo' to nil so that it can be garbage-collected.

if nil then print "nil" end --> (prints nothing)
-- Only false and nil are considered false; every other value is true.
if 0 then print "0" end --> 0
if "" then print "Empty string!" --> Empty string!
```

expressions

```
a = 3
b = a + 20 a = 2 print(b, a) -- hard to read, can also be written as
b = a + 20; a = 2; print(a, b) -- easier to read, ; are optional though
true and true --> returns true
true and 20 --> 20
false and 20 --> false
false or 20 --> 20
```

```
true or 20 --> true
tab or {}
  --> returns tab if it is defined
  --> returns {} if tab is undefined
  -- This is useful when we don't know if a variable exists
tab = tab or {} -- tab stays unchanged if it exists; tab becomes {} if it was previously nil.

a, b = 20, 30 -- this also works
a, b = b, a -- switches values
```

Defining functions

```
function name(parameter)
    return parameter
end
print(name(20)) --> 20
-- see function category for more information
name = function(parameter) return parameter end -- Same as above
```

booleans

Only `false` and `nil` evaluate as false, everything else, including `0` and the empty string evaluate as true.

garbage-collection

```
tab = {"lots", "of", "data"}
tab = nil; collectgarbage()
-- tab does no longer exist, and doesn't take up memory anymore.
```

tables

```
tab1 = {"a", "b", "c"}
tab2 = tab1
tab2[1] = "d"
print(tab1[1]) --> 'd' -- table values only store references.
--> assigning tables does not copy its content, only the reference.

tab2 = nil; collectgarbage()
print(tab1) --> (prints table address) -- tab1 still exists; it didn't get garbage-collected.

tab1 = nil; collectgarbage()
-- No more references. Now it should actually be gone from memory.
```

These are the basics, but there's a section about tables with more information.

conditions

```
if (condition) then
```

```
    -- do something
elseif (other_condition) then
    -- do something else
else
    -- do something
end
```

for loops

There are two types of `for` loop in Lua: a numeric `for` loop and a generic `for` loop.

- A numeric `for` loop has the following form:

  ```
  for a=1, 10, 2 do -- for a starting at 1, ending at 10, in steps of 2
    print(a) --> 1, 3, 5, 7, 9
  end
  ```

  The third expression in a numeric `for` loop is the step by which the loop will increment. This makes it easy to do reverse loops:

  ```
   for a=10, 1, -1 do
     print(a) --> 10, 9, 8, 7, 6, etc.
   end
  ```

  If the step expression is left out, Lua assumes a default step of 1.

  ```
   for a=1, 10 do
     print(a) --> 1, 2, 3, 4, 5, etc.
   end
  ```

  Also note that the loop variable is local to the `for` loop. It will not exist after the loop is over.

- Generic `for` loops work through all values that an iterator function returns:

  ```
  for key, value in pairs({"some", "table"}) do
    print(key, value)
    --> 1 some
    --> 2 table
  end
  ```

  Lua provides several built in iterators (e.g., `pairs`, `ipairs`), and users can define their own custom iterators as well to use with generic `for` loops.

do blocks

```
local a = 10
do
    print(a) --> 10
    local a = 20
    print(a) --> 20
```

```
end
print(a) --> 10
```

Some tricky things

Sometimes Lua doesn't behave the way one would think after reading the documentation. Some of these cases are:

Nil and Nothing aren't the same (COMMON PITFALL!)

As expected, `table.insert(my_table, 20)` adds the value `20` to the table, and `table.insert(my_table, 5, 20)` adds the value 20 at the 5th position. What does `table.insert(my_table, 5, nil)` do though? One might expect it to treat `nil` as no argument at all, and insert the value `5` at the end of the table, but it actually adds the value `nil` at the 5th position of the table. When is this a problem?

```
(function(tab, value, position)
    table.insert(tab, position or value, position and value)
end)({}, 20)
-- This ends up calling table.insert({}, 20, nil)
-- and this doesn't do what it should (insert 20 at the end)
```

A similar thing happens with `tostring()`:

```
print (tostring(nil)) -- this prints "nil"
table.insert({}, 20) -- this returns nothing
-- (not nil, but actually nothing (yes, I know, in lua those two SHOULD
-- be the same thing, but they aren't))

-- wrong:
print (tostring( table.insert({}, 20) ))
-- throws error because nothing ~= nil

--right:
local _tmp = table.insert({}, 20) -- after this _tmp contains nil
print(tostring(_tmp)) -- prints "nil" because suddenly nothing == nil
```

This may also lead to errors when using third party code. If, for example, the documentation of some function states "returns donuts if lucky, nil otherwise", the implementation *might* looks somewhat like this

```
function func(lucky)
    if lucky then
        return "donuts"
    end
end
```

this implementation might seem reasonable at first; it returns donuts when it has to, and when you type `result = func(false)` result will contain the value `nil`.

However, if one were to write `print(tostring(func(false)))` lua would throw an error that looks somewhat like this one `stdin:1: bad argument #1 to 'tostring' (value expected)`

Why is that? `tostring` clearly gets an argument, even though it's `nil`. Wrong. func returns nothing at all, so `tostring(func(false))` is the same as `tostring()` and NOT the same as `tostring(nil)`.

Errors saying "value expected" are a strong indication that this might be the source of the problem.

Leaving gaps in arrays

This is a huge pitfall if you're new to lua, and there's a lot of information in the tables category

Hello World

This is hello world code:

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

How it works? It's simple! Lua executes `print()` function and uses `"Hello World"` string as argument.

Read Getting started with Lua online: https://riptutorial.com/lua/topic/659/getting-started-with-lua

Chapter 2: Booleans in Lua

Remarks

Booleans, truth, and falsity are straightforward in Lua. To review:

1. There is a boolean type with exactly two values: `true` and `false`.
2. In a conditional context (`if`, `elseif`, `while`, `until`), a boolean is not required. Any expression can be used.
3. In a conditional context, `false` and `nil` count as false, and everything else counts as true.
4. Although 3 already implies this: if you're coming from other languages, remember that `0` and the empty string count as true in conditional contexts in Lua.

Examples

The boolean type

Booleans and other values

When dealing with lua it is important to differentiate between the boolean values `true` and `false` and values that evaluate to true or false.

There are only two values in lua that evaluate to false: `nil` and `false`, while everything else, including the numerical `0` evaluate to true.

Some examples of what this means:

```
if 0 then print("0 is true") end --> this will print "true"
if (2 == 3) then print("true") else print("false") end --> this prints "false"
if (2 == 3) == false then print("true") end --> this prints "true"
if (2 == 3) == nil then else print("false") end
--> prints false, because even if nil and false both evaluate to false,
--> they are still different things.
```

Logical Operations

Logical operators in lua don't necessarily return boolean values:

`and` will return the second value if the first value evaluates to true;

`or` returns the second value if the first value evaluates to false;

This makes it possible to simulate the ternary operator, just like in other languages:

```
local var = false and 20 or 30 --> returns 30
local var = true and 20 or 30 --> returns 20
-- in C: false ? 20 : 30
```

This can also be used to initialize tables if they don't exist

```
tab = tab or {} -- if tab already exists, nothing happens
```

or to avoid using if statements, making the code easier to read

```
print(debug and "there has been an error") -- prints "false" line if debug is false
debug and print("there has been an error") -- does nothing if debug is false
-- as you can see, the second way is preferable, because it does not output
-- anything if the condition is not met, but it is still possible.
-- also, note that the second expression returns false if debug is false,
-- and whatever print() returns if debug is true (in this case, print returns nil)
```

Checking if variables are defined

One can also easily check if a variable exists (if it is defined), since non-existant variables return `nil`, which evaluates to false.

```
local tab_1, tab_2 = {}
if tab_1 then print("table 1 exists") end --> prints "table 1 exists"
if tab_2 then print("table 2 exists") end --> prints nothing
```

The only case where this does not apply is when a variable stores the value `false`, in which case it technically exists but still evaluates to false. Because of this, it is a bad design to create functions which return `false` and `nil` depending on the state or input. We can still check however whether we have a `nil` or a `false`:

```
if nil == nil then print("A nil is present") else print("A nil is not present") end
if false == nil then print("A nil is present") else print("A nil is not present") end
-- The output of these calls are:
-- A nil is present!
-- A nil is not present
```

Conditional contexts

Conditional contexts in Lua (`if`, `elseif`, `while`, `until`) do not require a boolean. Like many languages, any Lua value can appear in a condition. The rules for evaluation are simple:

1. `false` and `nil` count as false.

2. Everything else counts as true.

   ```
   if 1 then
     print("Numbers work.")
   end
   ```

```
if 0 then
  print("Even 0 is true")
end

if "strings work" then
  print("Strings work.")
end
if "" then
  print("Even the empty string is true.")
end
```

Logical Operators

In Lua, booleans can be manipulated through *logical operators*. These operators include `not`, `and`, and `or`.

In simple expressions, the results are fairly straightforward:

```
print(not true) --> false
print(not false) --> true
print(true or false) --> true
print(false and true) --> false
```

Order of Precedence

The order of precedence is similar to the math operators unary `-`, `*` and `+`:

- `not`
- then `and`
- then `or`

This can lead to complex expressions:

```
print(true and false or not false and not true)
print( (true and false) or ((not false) and (not true)) )
    --> these are equivalent, and both evaluate to false
```

Short-cut Evaluation

The operators `and` and `or` might only be evaluated using the first operand, provided the second is unnecessary:

```
function a()
    print("a() was called")
    return true
end

function b()
    print("b() was called")
    return false
end

print(a() or b())
    --> a() was called
    --> true
    --  nothing else
print(b() and a())
    --> b() was called
    --> false
    --  nothing else
print(a() and b())
    --> a() was called
    --> b() was called
    --> false
```

Idiomatic conditional operator

Due to the precedence of the logical operators, the ability for short-cut evaluation and the evaluation of non-`false` and non-`nil` values as `true`, an idiomatic conditional operator is available in Lua:

```
function a()
    print("a() was called")
    return false
end
function b()
    print("b() was called")
    return true
end
function c()
    print("c() was called")
    return 7
end

print(a() and b() or c())
    --> a() was called
    --> c() was called
    --> 7

print(b() and c() or a())
    --> b() was called
    --> c() was called
    --> 7
```

Also, due to the nature of the `x and a or b` structure, `a` will never be *returned* if it evaluates to `false`, this conditional will then always return `b` no matter what `x` is.

```
print(true and false or 1)  -- outputs 1
```

Truth tables

Logical operators in Lua don't "return" boolean, but one of their arguments. Using `nil` for false and numbers for true, here's how they behave.

```
print(nil and nil)       -- nil
print(nil and 2)         -- nil
print(1 and nil)         -- nil
print(1 and 2)           -- 2

print(nil or nil)        -- nil
print(nil or 2)          -- 2
print(1 or nil)          -- 1
print(1 or 2)            -- 1
```

As you can see, Lua will always return the first value that makes the check *fail* or *succeed*. Here's the truth tables showing that.

```
  x  |  y  || and            x  |  y  || or
------------------         ------------------
false|false||  x           false|false||  y
false|true ||  x           false|true ||  y
true |false||  y           true |false||  x
true |true ||  y           true |true ||  x
```

For those who need it, here's two function representing these logical operators.

```
function exampleAnd(value1, value2)
  if value1 then
    return value2
  end
  return value1
end

function exampleOr(value1, value2)
  if value1 then
    return value1
  end
  return value2
end
```

Emulating Ternary Operator with 'and' 'or' logical operators.

In lua, the logical operators `and` and `or` returns one of the operands as the result instead of a boolean result. As a consequence, this mechanism can be exploited to emulate the behavior of the ternary operator despite lua not having a 'real' ternary operator in the language.

Syntax

condition **and** *truthy_expr* **or** *falsey_expr*

Use in variable assignment/initialization

```
local drink = (fruit == "apple") and "apple juice" or "water"
```

Use in table constructor

```
local menu =
{
  meal  = vegan and "carrot" or "steak",
  drink = vegan and "tea"    or "chicken soup"
}
```

Use as function argument

```
print(age > 18 and "beer" or "fruit punch")
```

Use in return statement

```
function get_gradestring(student)
  return student.grade > 60 and "pass" or "fail"
end
```

Caveat

There are situations where this mechanism doesn't have the desired behavior. Consider this case

```
local var = true and false or "should not happen"
```

In a 'real' ternary operator, the expected value of `var` is `false`. In lua, however, the `and` evaluation 'falls through' because the second operand is falsey. As a result `var` ends up with `should not happen` instead.

Two possible workarounds to this problem, refactor this expression so the middle operand isn't falsey. eg.

```
local var = not true and "should not happen" or false
```

or alternatively, use the classical `if then else` construct.

Read Booleans in Lua online: https://riptutorial.com/lua/topic/3101/booleans-in-lua

Chapter 3: Coroutines

Syntax

- coroutine.create(function) returns a coroutine (type(coroutine) == 'thread') containing the function.
- coroutine.resume(co, ...) resume, or start the coroutine. Any additional arguments given to resume are returned from the coroutine.yield() that previously paused the coroutine. If the coroutine had not been started the additional arguments become the arguments of the function.
- coroutine.yield(...) yields the currently running coroutine. Execution picks back up after the call to coroutine.resume() that started that coroutine. Any arguments given to yield are returned from the corresponding coroutine.resume() that started the coroutine.
- coroutine.status(co) returns the status of the coroutine, which can be :
  - "dead" : the function in the coroutine has reached it's end and the coroutine cannot be resumed anymore
  - "running" : the coroutine has been resumed and is running
  - "normal" : the coroutine has resumed another coroutine
  - "suspended" : the coroutine has yielded, and is waiting to be resumed
- coroutine.wrap(function) returns a function that when called resumes the coroutine that would have been created by coroutine.create(function).

Remarks

The coroutine system has been implemented in lua to emulate multithreading existing in other languages. It works by switching at extremely high speed between different functions so that the human user think they are executed at the same time.

Examples

Create and use a coroutine

All functions to interact with coroutines are avaliable in the **coroutine** table. A new coroutine is created by using the **coroutine.create** function with a single argument: a function with the code to be executed:

```
thread1 = coroutine.create(function()
            print("honk")
        end)

print(thread1)
```

```
-->> thread: 6b028b8c
```

A coroutine object returns value of type **thread**, representing a new coroutine. When a new coroutine is created, its initial state is suspended:

```
print(coroutine.status(thread1))
-->> suspended
```

To resume or start a coroutine, the function **coroutine.resume** is used, the first argument given is the thread object:

```
coroutine.resume(thread1)
-->> honk
```

Now the coroutine executes the code and terminates, changing its state to **dead**, wich cannot be resumed.

```
print(coroutine.status(thread1))
-->> dead
```

Coroutines can suspend its execution and resume it later thanks to the **coroutine.yield** function:

```
thread2 = coroutine.create(function()
        for n = 1, 5 do
            print("honk "..n)
            coroutine.yield()
        end
    end)
```

As you can see, **coroutine.yield()** is present inside the for loop, now when we resume the coroutine, it will execute the code until it reachs a coroutine.yield:

```
coroutine.resume(thread2)
-->> honk 1
coroutine.resume(thread2)
-->> honk 2
```

After finishing the loop, the thread status becomes **dead** and cannot be resumed. Coroutines also allows the exchange between data:

```
thread3 = coroutine.create(function(complement)
    print("honk "..complement)
    coroutine.yield()
    print("honk again "..complement)
end)
coroutine.resume(thread3, "stackoverflow")
-->> honk stackoverflow
```

If the coroutine is executed again with no extra arguments, the *complement* will still the argument from the first resume, in this case "stackoverflow":

```
coroutine.resume(thread3)
-->> honk again stackoverflow
```

Finally, when a coroutine ends, any values returned by its function go to the corresponding resume:

```
thread4 = coroutine.create(function(a, b)
    local c = a+b
    coroutine.yield()
    return c
end)
coroutine.resume(thread4, 1, 2)
print(coroutine.resume(thread4))
-->> true, 3
```

Coroutines are used in this function to pass values back to a calling thread from deep within a recursive call.

```
local function Combinations(l, r)
    local ll = #l
    r = r or ll
    local sel = {}
    local function rhelper(depth, last)
        depth = depth or 1
        last = last or 1
        if depth > r then
            coroutine.yield(sel)
        else
            for i = last, ll - (r - depth) do
                sel[depth] = l[i]
                rhelper(depth+1, i+1)
            end
        end
    end
    return coroutine.wrap(rhelper)
end

for v in Combinations({1, 2, 3}, 2) do
    print("{"..table.concat(v, ", ").."}")
end
--> {1, 2}
--> {1, 3}
--> {2, 3}
```

Coroutines can also be used for lazy evaluation.

```
-- slices a generator 'c' taking every 'step'th output from the generator
-- starting at the 'start'th output to the 'stop'th output
function slice(c, start, step, stop)
    local _
    return coroutine.wrap(function()
        for i = 1, start-1 do
            _ = c()
        end
        for i = start, stop do
            if (i - start) % step == 0 then
                coroutine.yield(c())
            else
                _ = c()
            end
        end
    end)
end


local alphabet = {}
for c = string.byte('a'), string.byte('z') do
    alphabet[#alphabet+1] = string.char(c)
end
-- only yields combinations 100 through 102
-- requires evaluating the first 100 combinations, but not the next 5311633
local s = slice(Combinations(alphabet, 10), 100, 1, 102)
for i in s do
    print(table.concat(i))
end
--> abcdefghpr
--> abcdefghps
--> abcdefghpt
```

Coroutines can be used for piping constructs as described in Programming In Lua. The author of PiL, Roberto Ierusalimschy, has also published a paper on using coroutines to implement more advanced and general flow control mechanics like continuations.

Read Coroutines online: https://riptutorial.com/lua/topic/3410/coroutines

Chapter 4: Error Handling

Examples

Using pcall

`pcall` stands for "protected call". It is used to add error handling to functions. `pcall` works similar as `try-catch` in other languages. The advantage of `pcall` is that the whole execution of the script is not being interrupted if errors occur in functions called with `pcall`. If an error inside a function called with `pcall` occurs an error is thrown and the rest of the code continues execution.

Syntax:

```
pcall( f , arg1,···)
```

Return Values:

Returns two values

1. status (boolean)

- Returns **true** if the function was executed with no errors.
- Returns **false** if an error occured inside the function.

2. return value of the function **or** error message if an error occurred inside the function block.

---

pcall may be used for various cases, however a common one is to catch errors from the function which has been given to your function. For instance, lets say we have this function:

```
local function executeFunction(funcArg, times) then
    for i = 1, times do
        local ran, errorMsg = pcall( funcArg )
        if not ran then
            error("Function errored on run " .. tostring(i) .. "\n" .. errorMsg)
        end
    end
end
```

When the given function errors on run 3, the error message will be clear to the user that it is not coming from your function, but from the function which was given to our function. Also, with this in mind a fancy BSoD can be made notifying the user. However, that is up to the application which implements this function, as an API most likely won't be doing that.

Example A - *Execution without pcall*

```
function square(a)
```

```
  return a * "a"    --This will stop the execution of the code and throws an error, because of
the attempt to perform arithmetic on a string value
end

square(10);

print ("Hello World")    -- This is not being executed because the script was interrupted due
to the error
```

Example B - *Execution with pcall*

```
function square(a)
  return a * "a"
end

local status, retval = pcall(square,10);

print ("Status: ", status)        -- will print "false" because an error was thrown.
print ("Return Value: ", retval)  -- will print "input:2: attempt to perform arithmetic on a
string value"
print ("Hello World")    -- Prints "Hello World"
```

Example - *Execution of flawless code*

```
function square(a)
  return a * a
end

local status, retval = pcall(square,10);

print ("Status: ", status)        -- will print "true" because no errors were thrown
print ("Return Value: ", retval)  -- will print "100"
print ("Hello World")    -- Prints "Hello World"
```

Handling errors in Lua

Assuming we have the following function:

```
function foo(tab)
  return tab.a
end -- Script execution errors out w/ a stacktrace when tab is not a table
```

Let's improve it a bit

```
function foo(tab)
  if type(tab) ~= "table" then
    error("Argument 1 is not a table!", 2)
  end
  return tab.a
end -- This gives us more information, but script will still error out
```

If we don't want a function to crash a program even in case of an error, it is standard in lua to do the following:

```
function foo(tab)
    if type(tab) ~= "table" then return nil, "Argument 1 is not a table!" end
    return tab.a
end -- This never crashes the program, but simply returns nil and an error message
```

Now we have a function that behaves like that, we can do things like this:

```
if foo(20) then print(foo(20)) end -- prints nothing
result, error = foo(20)
if result then print(result) else log(error) end
```

And if we DO want the program to crash if something goes wrong, we can still do this:

```
result, error = foo(20)
if not result then error(error) end
```

Fortunately we don't even have to write all that every time; lua has a function that does exactly this

```
result = assert(foo(20))
```

Read Error Handling online: https://riptutorial.com/lua/topic/4561/error-handling

Chapter 5: Functions

Syntax

- *funcname* = function(paramA, paramB, ...) body; return exprlist end -- a simple function
- function *funcname*(paramA, paramB, ...) body; return exprlist end -- shorthand for above
- local *funcname* = function(paramA, paramB, ...) body; return exprlist end -- a lambda
- local *funcname*; *funcname* = function(paramA, paramB, ...) body; return exprlist end -- lambda that can do recursive calls
- local function *funcname*(paramA, paramB, ...) body; return exprlist end -- shorthand for above
- *funcname*(paramA, paramB, ...) -- call a function
- local *var* = *var* or "Default" -- a default parameter
- return nil, "error messages" -- standard way to abort with an error

Remarks

Functions are usually set with `function a(b,c) ... end` and rarely with setting a variable to an anonymous function (`a = function(a,b) ... end`). The opposite is true when passing functions as parameters, anonymous functions are mostly used, and normal functions aren't used as often.

Examples

Defining a function

```
function add(a, b)
    return a + b
end
-- creates a function called add, which returns the sum of it's two arguments
```

Let's look at the syntax. First, we see a `function` keyword. Well, that's pretty descriptive. Next we see the `add` identifier; the name. We then see the arguments `(a, b)` these can be anything, and they are local. Only inside the function body can we access them. Let's skip to the end, we see... well, the `end`! And all that's in between is the function body; the code that's ran when it is called. The `return` keyword is what makes the function actually give some useful output. Without it, the function returns nothing, which is equivalent to returning nil. This can of course be useful for things that interact with IO, for example:

```
function printHello(name)
    print("Hello, " .. name .. "!");
end
```

In that function, we did not use the return statement.

Functions can also return values conditionally, meaning that a function has the choice of returning nothing (nil) or a value. This is demonstrated in the following example.

```
function add(a, b)
    if (a + b <= 100) then
        return a + b -- Returns a value
    else
        print("This function doesn't return values over 100!") -- Returns nil
    end
end
```

It is also possible for a function to return multiple values seperated by commas, as shown:

```
function doOperations(a, b)
    return a+b, a-b, a*b
end

added, subbed, multiplied = doOperations(4,2)
```

Functions can also be declared local

```
do
    local function add(a, b) return a+b end
    print(add(1,2)) --> prints 3
end
print(add(2, 2)) --> exits with error, because 'add' is not defined here
```

They can be saved in tables too:

```
tab = {function(a,b) return a+b end}
(tab[1])(1, 2) --> returns 3
```

Calling a function.

Functions are only useful if we can call them. To call a function the following syntax is used:

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

We're calling the `print` function. Using the argument `"Hello, World"`. As is obvious, this will print `Hello, World` to the output stream. The returned value is accessible, just like any other variable would be.

```
local added = add(10, 50) -- 60
```

Variables are also accepted in a function's parameters.

```
local a = 10
local b = 60

local c = add(a, b)

print(c)
```

Functions expecting a table or a string can be called with a neat syntactic sugar: parentheses surrounding the call can be omitted.

```
print"Hello, world!"
for k, v in pairs{"Hello, world!"} do print(k, v) end
```

Anonymous functions

Creating anonymous functions

Anonymous functions are just like regular Lua functions, except they do not have a name.

```
doThrice(function()
    print("Hello!")
end)
```

As you can see, the function is not assigned to any name like `print` or `add`. To create an anonymous function, all you have to do is omit the name. These functions can also take arguments.

Understanding the syntactic sugar

It is important to understand that the following code

```
function double(x)
    return x * 2
end
```

is actually just a shorthand for

```
double = function(x)
    return x * 2
end
```

However, the above function is **not** anonymous as the function is directly assigned to a variable!

Functions are first class values

This means that a function is a value with the same rights as conventional values like numbers and strings. Functions can be stored in variables, in tables, can be passed as arguments, and can be returned by other functions.

To demonstrate this, we'll also create a "half" function:

```
half = function(x)
    return x / 2
end
```

So, now we have two variables, `half` and `double`, both containing a function as a value. What if we wanted to create a function that would feed the number 4 into two given functions, and compute the sum of both results?

We'll want to call this function like `sumOfTwoFunctions(double, half, 4)`. This will feed the `double` function, the `half` function, and the integer `4` into our own function.

```
function sumOfTwoFunctions(firstFunction, secondFunction, input)
    return firstFunction(input) + secondFunction(input)
end
```

The above `sumOfTwoFunctions` function shows how functions can be passed around within arguments, and accessed by another name.

Default parameters

```
function sayHello(name)
    print("Hello, " .. name .. "!")
end
```

That function is a simple function, and it works well. But what would happen if we just called `sayHello()`?

```
stdin:2: attempt to concatenate local 'name' (a nil value)
stack traceback:
    stdin:2: in function 'sayHello'
    stdin:1: in main chunk
    [C]: in ?
```

That's not exactly great. There are two ways of fixing this:

1. You immediately return from the function:

   ```
   function sayHello(name)
     if not (type(name) == "string") then
       return nil, "argument #1: expected string, got " .. type(name)
     end -- Bail out if there's no name.
     -- in lua it is a convention to return nil followed by an error message on error

     print("Hello, " .. name .. "!") -- Normal behavior if name exists.
   end
   ```

2. You set a *default* parameter.

   To do this, simply use this simple expression

```
function sayHello(name)
    name = name or "Jack" -- Jack is the default,
                          -- but if the parameter name is given,
                          -- name will be used instead
    print("Hello, " .. name .. "!")
end
```

The idiom `name = name or "Jack"` works because `or` in Lua short circuits. If the item on the left side of an `or` is anything other than `nil` or `false`, then the right side is never evaluated. On the other hand, if `sayHello` is called with no parameter, then `name` will be `nil`, and so the string `"Jack"` will be assigned to `name`. (Note that this idiom, therefore, will not work if the boolean `false` is a reasonable value for the parameter in question.)

Multiple results

Functions in Lua can return multiple results.

For example:

```
function triple(x)
    return x, x, x
end
```

When calling a function, to save these values, you must use the following syntax:

```
local a, b, c = triple(5)
```

Which will result in `a = b = c = 5` in this case. It is also possible to ignore returned values by using the throwaway variable _ in the desired place in a list of variables:

```
local a, _, c = triple(5)
```

In this case, the second returned value will be ignored. It's also possible to ignore return values by not assigning them to any variable:

```
local a = triple(5)
```

Variable `a` will be assigned the first return value and the remaining two will be discarded.

When a variable amount of results are returned by a function, one can store them all in a table, by executing the function inside it:

```
local results = {triple(5)}
```

This way, one can iterate over the `results` table to see what the function returned.

Note

This can be a surprise in some cases, for example:

```
local t = {}
table.insert(t, string.gsub("  hi", "^%s*(.*)$", "%1")) --> bad argument #2 to 'insert'
(number expected, got string)
```

This happens because `string.gsub` returns 2 values: the given string, with occurrences of the pattern replaced, and the total number of matches that occurred.

To solve this, either use an intermediate variable or put `()` around the call, like so:

```
table.insert(t, (string.gsub("  hi", "^%s*(.*)$", "%1"))) --> works. t = {"hi"}
```

This grabs only the first result of the call, and ignores the rest.

Variable number of arguments

Variadic Arguments

Named Arguments

```
local function A(name, age, hobby)
    print(name .. "is " .. age .. " years old and likes " .. hobby)
end
A("john", "eating", 23) --> prints 'john is eating years old and likes 23'
-- oops, seems we got the order of the arguments wrong...
-- this happens a lot, specially with long functions that take a lot of arguments
-- and where the order doesn't follow any particular logic

local function B(tab)
    print(tab.name .. "is " .. tab.age .. " years old and likes " .. tab.hobby)
end
local john = {name="john", hobby="golf", age="over 9000", comment="plays too much golf"}
B(john)
--> will print 'John is over 9000 years old and likes golf'
-- I also added a 'comment' argument just to show that excess arguments are ignored by the
function

B({name = "tim"}) -- can also be written as
B{name = "tim"} -- to avoid cluttering the code
--> both will print 'tim is nil years old and likes nil'
-- remember to check for missing arguments and deal with them

function C(tab)
    if not tab.age then return nil, "age not defined" end
    tab.hobby = tab.hobby or "nothing"
    -- print stuff
end

-- note that if we later decide to do a 'person' class
-- we just need to make sure that this class has the three fields
-- age, hobby and name, and it will be compatible with these functions

-- example:
local john = ClassPerson.new("John", 20, "golf") -- some sort of constructor
john.address = "some place" -- modify the object
john:do_something("information") -- call some function of the object
C(john) -- this works because objects are *usually* implemented as tables
```

Checking argument types

Some functions only work on a certain type of argument:

```
function foo(tab)
    return tab.bar
end
--> returns nil if tab has no field bar, which is acceptable
--> returns 'attempt to index a number value' if tab is, for example, 3
--> which is unacceptable

function kungfoo(tab)
    if type(tab) ~= "table" then
        return nil, "take your useless " .. type(tab) .." somewhere else!"
    end

    return tab.bar
end
```

this has several implications:

```
print(kungfoo(20)) --> prints 'nil, take your useless number somewhere else!'

if kungfoo(20) then print "good" else print "bad" end --> prints bad

foo = kungfoo(20) or "bar" --> sets foo to "bar"
```

now we can call the function with whatever parameter we want, and it won't crash the program.

```
-- if we actually WANT to abort execution on error, we can still do
result = assert(kungfoo({bar=20})) --> this will return 20
result = assert(kungfoo(20)) --> this will throw an error
```

So, what if we have a function that does something with an instance of a specific class? This is difficult, because classes and objects are usually tables, so the `type` function will return `'table'`.

```
local Class = {data="important"}
local meta = {__index=Class}

function Class.new()
    return setmetatable({}, meta)
end
-- this is just a very basic implementation of an object class in lua

object = Class.new()
fake = {}

print(type(object)), print(type(fake)) --> prints 'table' twice
```

Solution: compare the metatables

```
-- continuation of previous code snippet
Class.is_instance(tab)
    return getmetatable(tab) == meta
end

Class.is_instance(object) --> returns true
Class.is_instance(fake) --> returns false
Class.is_instance(Class) --> returns false
Class.is_instance("a string") --> returns false, doesn't crash the program
Class.is_instance(nil) --> also returns false, doesn't crash either
```

Closures

```
do
    local tab = {1, 2, 3}
    function closure()
        for key, value in ipairs(tab) do
            print(key, "I can still see you")
        end
    end
    closure()
    --> 1 I can still see you
    --> 2 I can still see you
    --> 3 I can still see you
end

print(tab) --> nil
-- tab is out of scope

closure()
--> 1 I can still see you
--> 2 I can still see you
--> 3 I can still see you
-- the function can still see tab
```

typical usage example

```
function new_adder(number)
    return function(input)
        return input + number
    end
end
add_3 = new_adder(3)
print(add_3(2)) --> prints 5
```

more advanced usage example

```
function base64.newDecoder(str) -- Decoder factory
    if #str ~= 64 then return nil, "string must be 64 characters long!" end

    local tab = {}
    local counter = 0
    for c in str:gmatch"." do
        tab[string.byte(c)] = counter
        counter = counter + 1
    end

    return function(str)
        local result = ""

        for abcd in str:gmatch"..?.?.?" do
            local a, b, c, d = string.byte(abcd,1,-1)
            a, b, c, d = tab[a], tab[b] or 0, tab[c] or 0, tab[d] or 0
            result = result .. (
                string.char( ((a<<2)+(b>>4))%256 ) ..
                string.char( ((b<<4)+(c>>2))%256 ) ..
                string.char( ((c<<6)+d)%256 )
            )
        end
        return result
    end
end
```

Read Functions online: https://riptutorial.com/lua/topic/1250/functions

Chapter 6: Garbage collector and weak tables

Syntax

1. collectgarbage(gcrule [, gcdata]) -- collect garbage using gcrule
2. setmetatable(tab, {__mode = weakmode}) -- set weak mode of tab to weakmode

Parameters

parameter	details
gcrule & gcdata	Action to gc (garbage collector): `"stop"` (stop collecting), `"restart"` (start collecting again), `"collect"` or `nil` (collect all garbage), `"step"` (do one collecting step), `"count"` (return count of used memory in KBs), `"setpause"` and data is number from `0`% to `100`% (set pause parameter of gc), `"setstepmul"` and data is number from `0`% to `100` (set `"stepmul"` for gc).
weakmode	Type of weak table: `"k"` (only weak keys), `"v"` (only weak values), `"vk"` (weak keys and values)

Examples

Weak tables

```
local t1, t2, t3, t4 = {}, {}, {}, {} -- Create 4 tables
local maintab = {t1, t2} -- Regular table, strong references to t1 and t2
local weaktab = setmetatable({t1, t2, t3, t4}, {__mode = 'v'}) -- table with weak references.

t1, t2, t3, t4 = nil, nil, nil, nil -- No more "strong" references to t3 and t4
print(#maintab, #weaktab) --> 2 4

collectgarbage() -- Destroy t3 and t4 and delete weak links to them.
print(#maintab, #weaktab) --> 2 2
```

Read Garbage collector and weak tables online: https://riptutorial.com/lua/topic/5769/garbage-collector-and-weak-tables

Chapter 7: Introduction to Lua C API

Syntax

- lua_State *L = lua_open(); // Create a new VM state; Lua 5.0
- lua_State *L = luaL_newstate(); // Create a new VM state; Lua 5.1+
- int luaL_dofile(lua_State *L, const char **filename*); // Run a lua script with the given *filename* using the specified lua_State
- void luaL_openlibs(lua_State *L); // Load all standard libraries into the specified lua_State
- void lua_close(lua_State *L); // Close VM state and release any resources inside
- void lua_call(lua_State *L, int nargs, int nresults); // Call the luavalue at index -(nargs + 1)

Remarks

Lua as well provides a proper C API to it's Virtual Machine. In contrary to VM itself, C API interface is stack based. So, most of the functions intended to be used with data is either adding some stuff on-top of virtual stack, or removing from it. Also, all the API calls must be used carefully within stack and it's limitations.

In general, anything available on Lua language can be done using it's C API. Also, there is some addition functionality like direct access to internal registry, change behavior of standard memory allocator or garbage collector.

You can compile provided Lua C API examples by executing following on Your terminal:

```
$ gcc -Wall ./example.c -llua -ldl -lm
```

Examples

Creating Lua Virtual Machine

```
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>

int main(void)
{
```

5.1

```
  /* Start by creating a new VM state */
  lua_State *L = luaL_newstate();

  /* Load standard Lua libraries: */
  luaL_openlibs(L);
```

5.1

```
  /* For older version of Lua use lua_open instead */
  lua_State *L = lua_open();

  /* Load standard libraries*/
  luaopen_base(L);
  luaopen_io(L);
  luaopen_math(L);
  luaopen_string(L);
  luaopen_table(L);
```

```
    /* do stuff with Lua VM. In this case just load and execute a file: */
    luaL_dofile(L, "some_input_file.lua");

    /* done? Close it then and exit. */
    lua_close(L);

    return EXIT_SUCCESS;
}
```

Calling Lua functions

```
#include <stdlib.h>

#include <lauxlib.h>
#include <lua.h>
#include <lualib.h>

int main(void)
{
    lua_State *lvm_hnd = lua_open();
    luaL_openlibs(lvm_hnd);

    /* Load a standard Lua function from global table: */
    lua_getglobal(lvm_hnd, "print");

    /* Push an argument onto Lua C API stack: */
    lua_pushstring(lvm_hnd, "Hello C API!");

    /* Call Lua function with 1 argument and 0 results: */
    lua_call(lvm_hnd, 1, 0);

    lua_close(lvm_hnd);

    return EXIT_SUCCESS;
 }
```

In the example above we're doing these things:

- creating and setting up Lua VM as shown on the first example
- getting and pushing a Lua function from global Lua table onto virtual stack
- pushing string `"Hello C API"` as an input argument onto the virtual stack
- instructing VM to call a function with one argument which is already on the stack
- closing and cleaning up

NOTE:

Bare in mind, that `lua_call()` pops the function and it's arguments from the stack leaving only the result.

Also, it would be safer using Lua protected call - `lua_pcall()` instead.

Embedded Lua Interpreter with Custom API and Lua Customization

Demonstrate how to embed a lua interpreter in C code, expose a C-defined function to Lua script, evaluate a Lua script, call a C-defined function from Lua, and call a Lua-defined function from C (the host).

In this example, we want the mood to be set by a Lua script. Here is mood.lua:

```
-- Get version information from host
major, minor, build = hostgetversion()
print( "The host version is ", major, minor, build)
print("The Lua interpreter version is ", _VERSION)

-- Define a function for host to call
function mood( b )

    -- return a mood conditional on parameter
    if (b and major > 0) then
        return 'mood-happy'
    elseif (major == 0) then
        return 'mood-confused'
    else
        return 'mood-sad'
    end
end
```

Notice, `mood()` is not called in the script. It is just defined for the host application to call. Also notice that the script calls a function called `hostgetversion()` that is not defined in the script.

Next, we define a host application that uses 'mood.lua'. Here is the 'hostlua.c':

```
#include <stdio.h>
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>

/*
 * define a function that returns version information to lua scripts
 */
static int hostgetversion(lua_State *l)
{
    /* Push the return values */
    lua_pushnumber(l, 0);
    lua_pushnumber(l, 99);
    lua_pushnumber(l, 32);

    /* Return the count of return values */
    return 3;
}

int main (void)
```

```
{
    lua_State *l = luaL_newstate();
    luaL_openlibs(l);

    /* register host API for script */
    lua_register(l, "hostgetversion", hostgetversion);

    /* load script */
    luaL_dofile(l, "mood.lua");

    /* call mood() provided by script */
    lua_getglobal(l, "mood");
    lua_pushboolean(l, 1);
    lua_call(l, 1, 1);

    /* print the mood */
    printf("The mood is %s\n", lua_tostring(l, -1));
    lua_pop(l, 1);

    lua_close(l);
    return 0;
}
```

And here is the output:

```
The host version is     0    99    32
Lua interpreter version is     Lua 5.2
The mood is mood-confused
```

Even after we have compile 'hostlua.c', we are still free to modify 'mood.lua' to change the output of our program!

Table manipulation

In order to access or alter an index on a table, you need to somehow place the table into the stack.
Let's assume, for this examples that your table is a global variable named tbl.

Getting the content at a particular index:

```
int getkey_index(lua_State *L)
{
  lua_getglobal(L, "tbl");    // this put the table in the stack
  lua_pushstring(L, "index"); // push the key to access
  lua_gettable(L, -2);        // retrieve the corresponding value; eg. tbl["index"]

  return 1;                   // return value to caller
}
```

As we have seen, all you have to do is to push the table into the stack, push the index and call lua_gettable. the -2 argument means that the table is the second element from the top of the stack.
lua_gettable triggers metamethods. If you do not want to trigger metamethods, use lua_rawget

instead. It uses the same arguments.

Setting the content at a particular index:

```
int setkey_index(lua_State *L)
{
  // setup the stack
  lua_getglobal(L, "tbl");
  lua_pushstring(L, "index");
  lua_pushstring(L, "value");
  // finally assign the value to table; eg. tbl.index = "value"
  lua_settable(L, -3);

  return 0;
}
```

The same drill as getting the content. You have to push the stack, push the index and then push the value into the stack. after that, you call lua_settable. the -3 argument is the position of the table in the stack. To avoid triggering metamethods, use lua_rawset instead of lua_settable. It uses the same arguments.

Transferring the content from a table to another:

```
int copy_tableindex(lua_State *L)
{
    lua_getglobal(L, "tbl1"); // (tbl1)
    lua_getglobal(L, "tbl2");// (tbl1)(tbl2)
    lua_pushstring(L, "index1");// (tbl1)(tbl2)("index1")
    lua_gettable(L, -3);// (tbl1)(tbl2)(tbl1.index1)
    lua_pushstring(L, "index2");// (tbl1)(tbl2)(tbl1.index1)("index2")
    lua_pushvalue(L, -2); // (tbl1)(tbl2)(tbl1.index1)("index2")(tbl1.index1)
    lua_settable(L, -4);// (tbl1)(tbl2)(tbl1.index1)
    lua_pop(L, 1);

    return 0;
}
```

Now we are putting together all we learned here. I put the stack content on the comments so you do not get lost.

We put both tables into the stack, push the index of table 1 into the stack, and get the value at `tbl1.index1`. Note the `-3` argument on gettable. I am looking at the first table (third from the top) and not the second. Then we push the index of the second table, copy the `tbl1.index1` to the top of the stack and then call `lua_settable`, on the 4th item from the top.

For housecleaning sake, I have purged the top element, so only the two tables remains at the stack.

Read Introduction to Lua C API online: https://riptutorial.com/lua/topic/671/introduction-to-lua-c-api

Chapter 8: Iterators

Examples

Generic For Loop

Iterators utilize a form of the `for` loop known as the generic for loop.

The generic form of the `for` loop uses three parameters:

1. An **iterator function** that gets called when the next value is needed. It receives both the invariant state and control variable as parameters. Returning `nil` signals termination.
2. The **invariant state** is a value that doesn't change during the iteration. It is typically the subject of the iterator, such as a table, string, or userdata.
3. The **control variable** represents an initial value for iteration.

We can write a `for` loop to iterate all key-value pairs in a table using the next function.

```
local t = {a=1, b=2, c=3, d=4, e=5}

-- next is the iterator function
-- t is the invariant state
-- nil is the control variable (calling next with a nil gets the first key)
for key, value in next, t, nil do
  -- key is the new value for the control variable
  print(key, value)
  -- Lua calls: next(t, key)
end
```

Standard Iterators

The Lua standard library provides two iterator functions that can be used with a `for` loop to traverse key-value pairs within tables.

To iterate over a sequence table we can use the library function ipairs.

```
for index, value in ipairs {'a', 'b', 'c', 'd', 'e'} do
  print(index, value)  --> 1 a, 2 b, 3 c, 4 d, 5 e
end
```

To iterator over all keys and values in any table we can use the library function pairs.

```
for key, value in pairs {a=1, b=2, c=3, d=4, e=5} do
  print(key, value)  --> e 5, c 3, a 1, b 2, d 4  (order not specified)
end
```

Stateless Iterators

Both pairs and ipairs represent stateless iterators. A stateless iterator uses only the generic for loop's control variable and invariant state to compute the iteration value.

Pairs Iterator

We can implement the stateless `pairs` iterator using the `next` function.

```
-- generator function which initializes the generic for loop
local function pairs(t)
  -- next is the iterator function
  -- t is the invariant state
  -- control variable is nil
  return next, t, nil
end
```

Ipairs Iterator

We can implement the stateless `ipairs` iterator in two separate functions.

```
-- function which performs the actual iteration
local function ipairs_iter(t, i)
  local i = i + 1  -- next index in the sequence (i is the control variable)
  local v = t[i]   -- next value (t is the invariant state)
  if v ~= nil then
    return i, v    -- index, value
  end
  return nil       -- no more values (termination)
end

-- generator function which initializes the generic for loop
local function ipairs(t)
  -- ipairs_iter is the iterator function
  -- t is the invariant state (table to be iterated)
  -- 0 is the control variable (first index)
  return ipairs_iter, t, 0
end
```

Character Iterator

We can create new stateless iterators by fulfilling the contract of the generic `for` loop.

```
-- function which performs the actual iteration
local function chars_iter(s, i)
  if i < #s then
    i = i + 1
    return i, s:sub(i, i)
  end
end

-- generator function which initializes the generic for loop
local function chars(s)
  return chars_iter, s, 0
end
```

```
-- used like pairs and ipairs
for i, c in chars 'abcde' do
    print(i, c) --> 1 a, 2 b, 3 c, 4 f, 5 e
end
```

Prime Numbers Iterator

This is one more simple example of a stateless iterator.

```
-- prime numbers iterator
local incr = {4, 1, 2, 0, 2}
function primes(s, p, d)
   s, p, d = s or math.huge, p and p + incr[p % 6] or 2, 1
   while p <= s do
      repeat
         d = d + incr[d % 6]
         if d*d > p then return p end
      until p % d == 0
      p, d = p + incr[p % 6], 1
   end
end

-- print all prime numbers <= 100
for p in primes, 100 do  -- passing in the iterator (do not call the iterator here)
   print(p)  -->  2  3  5  7  11 ... 97
end

-- print all primes in endless loop
for p in primes do  -- please note: "in primes", not "in primes()"
   print(p)
end
```

Stateful Iterators

Stateful iterators carry some additional information about the current state of the iterator.

Using Tables

The addition state can be packed into the generic for loop's invariant state.

```
  local function chars_iter(t, i)
    local i = i + 1
    if i <= t.len then
      return i, t.s:sub(i, i)
    end
  end

  local function chars(s)
    -- the iterators state
    local t = {
      s = s,    -- the subject
      len = #s  -- cached length
    }
    return chars_iter, t, 0
  end

  for i, c in chars 'abcde' do
    print(i, c) --> 1 a, 2 b, 3 c, 4 d, 5 e
  end
```

Using Closures

Additional state can be wrapped within a function closure. Since the state is fully contained in the scope of the closure the invariant state and control variable are not needed.

```
local function chars(s)
  local i, len = 0, #s
  return function() -- iterator function
    i = i + 1
    if i <= len then
      return i, s:sub(i, i)
    end
  end
end

for i, c in chars 'abcde' do
  print(i, c) --> 1 a, 2 b, 3 c, 4 d, 5 e
end
```

Using Coroutines

Additional state can be contained within a coroutine, again the invariant state and control variable are not needed.

```
local function chars(s)
  return coroutine.wrap(function()
    for i = 1, #s do
      coroutine.yield(s:sub(i, i))
    end
  end)
end

for c in chars 'abcde' do
  print(c) --> a, b, c, d, e
end
```

Read Iterators online: https://riptutorial.com/lua/topic/4165/iterators

Chapter 9: Metatables

Syntax

- [[local] *mt* =]getmetatable(*t*) --> retrieve associated metatable for '*t*'
- [[local] *t* =]setmetatable(*t*, *mt*) --> set the metatable for '*t*' to '*mt*' and returns '*t*'

Parameters

Parameter	Details
t	Variable referring to a lua table; can also be a table literal.
mt	Table to use as a metatable; can have zero or more metamethod fields set.

Remarks

There are some metamethods not mentioned here. For the full list and their usage, see the corresponding entry in the lua manual.

Examples

Creation and usage of metatables

A metatable defines a set of operations which alter the behaviour of a lua object. A metatable is just an ordinary table, which is used in a special way.

```
local meta = { } -- create a table for use as metatable

-- a metatable can change the behaviour of many things
-- here we modify the 'tostring' operation:
-- this fields should be a function with one argument.
-- it gets called with the respective object and should return a string
meta.__tostring = function (object)
    return string.format("{ %d, %d }", object.x, object.y)
end

-- create an object
local point = { x = 13, y = -2 }
-- set the metatable
setmetatable(point, meta)

-- since 'print' calls 'tostring', we can use it directly:
print(point) -- prints '{ 13, -2 }'
```

Using tables as metamethods

Some metamethods don't have to be functions. To most important example for this is the `__index` metamethod. It can also be a table, which is then used as lookup. This is quite commonly used in the creation of classes in lua. Here, a table (often the metatable itself) is used to hold all the operations (methods) of the class:

```
local meta = {}
-- set the __index method to the metatable.
-- Note that this can't be done in the constructor!
meta.__index = meta

function create_new(name)
    local self = { name = name }
    setmetatable(self, meta)
    return self
end

-- define a print function, which is stored in the metatable
function meta.print(self)
    print(self.name)
end

local obj = create_new("Hello from object")
obj:print()
```

Garbage collector - the __gc metamethod

5.2

Objects in lua are garbage collected. Sometimes, you need to free some resource, want to print a message or do something else when an object is destroyed (collected). For this, you can use the `__gc` metamethod, which gets called with the object as argument when the object is destroyed. You could see this metamethod as a sort of destructor.

This example shows the `__gc` metamethod in action. When the inner table assigned to `t` gets garbage collected, it prints a message prior to being collected. Likewise for the outer table when reaching the end of script:

```
local meta =
{
    __gc = function(self)
        print("destroying self: " .. self.name)
    end
}

local t = setmetatable({ name = "outer" }, meta)
do
    local t = { name = "inner" }
    setmetatable(t, meta)
end
```

More metamethods

There are many more metamethods, some of them are arithmetic (e.g. addition, subtraction, multiplication), there are bitwise operations (and, or, xor, shift), comparison (<, >) and also basic type operations like == and # (equality and length). Lets build a class which supports many of these operations: a call for rational arithmetic. While this is very basic, it shows the idea.

```
local meta = {
    -- string representation
    __tostring = function(self)
        return string.format("%s/%s", self.num, self.den)
    end,
    -- addition of two rationals
    __add = function(self, rhs)
        local num = self.num * rhs.den + rhs.num * self.den
        local den = self.den * rhs.den
        return new_rational(num, den)
    end,
    -- equality
    __eq = function(self, rhs)
        return self.num == rhs.num and self.den == rhs.den
    end
}

-- a function for the creation of new rationals
function new_rational(num, den)
    local self = { num = num, den = den }
    setmetatable(self, meta)

    return self
end

local r1 = new_rational(1, 2)
print(r1) -- 1/2

local r2 = new_rational(1, 3)
print(r1 + r2) -- 5/6

local r3 = new_rational(1, 2)
print(r1 == r3) -- true
-- this would be the behaviour if we hadn't implemented the __eq metamethod.
-- this compares the actual tables, which are different
print(rawequal(r1, r3)) -- false
```

Make tables callable

There is a metamethod called `__call`, which defines the bevahiour of the object upon being used as a function, e.g. `object()`. This can be used to create function objects:

```
-- create the metatable with a __call metamethod
local meta = {
    __call = function(self)
        self.i = self.i + 1
    end,
    -- to view the results
    __tostring = function(self)
        return tostring(self.i)
    end
}
```

```
function new_counter(start)
    local self = { i = start }
    setmetatable(self, meta)
    return self
end

-- create a counter
local c = new_counter(1)
print(c) --> 1
-- call -> count up
c()
print(c) --> 2
```

The metamethod is called with the corresponding object, all remaining arguments are passed to the function after that:

```
local meta = {
    __call = function(self, ...)
        print(self.prepend, ...)
    end
}

local self = { prepend = "printer:" }
setmetatable(self, meta)

self("foo", "bar", "baz")
```

Indexing of tables

Perhaps the most important use of metatables is the possibility to change the indexing of tables. For this, there are two actions to consider: *reading* the content and *writing* the content of the table. Note that both actions are only triggered if the corresponding key is not present in the table.

Reading

```
local meta = {}

-- to change the reading action, we need to set the '__index' method
-- it gets called with the corresponding table and the used key
-- this means that table[key] translates into meta.__index(table, key)
meta.__index = function(object, index)
    -- print a warning and return a dummy object
    print(string.format("the key '%s' is not present in object '%s'", index, object))
    return -1
end

-- create a testobject
local t = {}

-- set the metatable
setmetatable(t, meta)

print(t["foo"]) -- read a non-existent key, prints the message and returns -1
```

This could be used to raising an error while reading a non-existent key:

```
-- raise an error upon reading a non-existent key
meta.__index = function(object, index)
    error(string.format("the key '%s' is not present in object '%s'", index, object))
end
```

Writing

```
local meta = {}

-- to change the writing action, we need to set the '__newindex' method
-- it gets called with the corresponding table, the used key and the value
-- this means that table[key] = value translates into meta.__newindex(table, key, value)
meta.__newindex = function(object, index, value)
    print(string.format("writing the value '%s' to the object '%s' at the key '%s'",
                         value, object, index))
    --object[index] = value -- we can't do this, see below
end

-- create a testobject
local t = { }

-- set the metatable
setmetatable(t, meta)

-- write a key (this triggers the method)
t.foo = 42
```

You may now ask yourself how the actual value is written in the table. In this case, it isn't. The problem here is that metamethods can trigger metamethods, which would result in an infinitive loop, or more precisely, a stack overflow. So how can we solve this? The solution for this is called *raw table access*.

Raw table access

Sometimes, you don't want to trigger metamethods, but really write or read exactly the given key, without some clever functions wrapped around the access. For this, lua provides you with raw table access methods:

```
-- first, set up a metatable that allows no read/write access
local meta = {
    __index = function(object, index)
        -- raise an error
        error(string.format("the key '%s' is not present in object '%s'", index, object))
    end,
    __newindex = function(object, index, value)
        -- raise an error, this prevents any write access to the table
        error(string.format("you are not allowed to write the object '%s'", object))
    end
}

local t = { foo = "bar" }
setmetatable(t, meta)

-- both lines raise an error:
--print(t[1])
```

```
--t[1] = 42

-- we can now circumvent this problem by using raw access:
print(rawget(t, 1)) -- prints nil
rawset(t, 1, 42) -- ok

-- since the key 1 is now valid, we can use it in a normal manner:
print(t[1])
```

With this, we can now rewrite ower former `__newindex` method to actually write the value to the table:

```
meta.__newindex = function(object, index, value)
    print(string.format("writing the value '%s' to the object '%s' at the key '%s'",
                         value, object, index))
    rawset(object, index, value)
end
```

Simulating OOP

```
local Class = {} -- objects and classes will be tables
local __meta = {__index = Class}
-- ^ if an instance doesn't have a field, try indexing the class
function Class.new()
    -- return setmetatable({}, __meta) -- this is shorter and equivalent to:
    local new_instance = {}
    setmetatable(new_instance, __meta)
    return new_instance
end
function Class.print()
    print "I am an instance of 'class'"
end

local object = Class.new()
object.print() --> will print "I am an instance of 'class'"
```

Instance methods can be written by passing the object as the first argument.

```
-- append to the above example
function Class.sayhello(self)
    print("hello, I am ", self)
end
object.sayhello(object) --> will print "hello, I am <table ID>"
object.sayhello() --> will print "hello, I am nil"
```

There is some syntactic sugar for this.

```
function Class:saybye(phrase)
    print("I am " .. self .. "\n" .. phrase)
end
object:saybye("c ya") --> will print "I am <table ID>
                      -->             c ya"
```

We can also add default fields to a class.

```
local Class = {health = 100}
local __meta = {__index = Class}

function Class.new() return setmetatable({}, __meta) end
local object = Class.new()
print(object.health) --> prints 100
Class.health = 50; print(object.health) --> prints 50
-- this should not be done, but it illustrates lua indexes "Class"
-- when "object" doesn't have a certain field
object.health = 200 -- This does NOT index Class
print(object.health) --> prints 200
```

Read Metatables online: https://riptutorial.com/lua/topic/2444/metatables

Chapter 10: Object-Orientation

Introduction

Lua itself offers no class system. It is, however possible to implement classes and objects as tables with just a few tricks.

Syntax

- `function <class>.new() return setmetatable({}, {__index=<class>}) end`

Examples

Simple Object Orientation

Here's a basic example of how to do a very simple class system

```
Class = {}
local __instance = {__index=Class} -- Metatable for instances
function Class.new()
    local instance = {}
    setmetatable(instance, __instance)
    return instance
-- equivalent to: return setmetatable({}, __instance)
end
```

To add variables and/or methods, just add them to the class. Both can be overridden for every instance.

```
Class.x = 0
Class.y = 0
Class:getPosition()
    return {self.x, self.y}
end
```

And to create an instance of the class:

```
object = Class.new()
```

or

```
setmetatable(Class, {__call = Class.new}
    -- Allow the class itself to be called like a function
object = Class()
```

And to use it:

```
object.x = 20
-- This adds the variable x to the object without changing the x of
-- the class or any other instance. Now that the object has an x, it
-- will override the x that is inherited from the class
print(object.x)
-- This prints 20 as one would expect.
print(object.y)
-- Object has no member y, therefore the metatable redirects to the
-- class table, which has y=0; therefore this prints 0
object:getPosition() -- returns {20, 0}
```

Changing metamethods of an object

Having

```
local Class = {}
Class.__meta = {__index=Class}
function Class.new() return setmetatable({}, Class.__meta)
```

Assuming we want to change the behavior of a single instance `object = Class.new()` using a metatable,

there are a few mistakes to avoid:

```
setmetatable(object, {__call = table.concat}) -- WRONG
```

This exchanges the old metatable with the new one, therefore breaking the class inheritance

```
getmetatable(object).__call = table.concat -- WRONG AGAIN
```

Keep in mind that table "values" are only reference; there is, in fact, only one actual table for all the instances of an object unless the constructor is defined as in [1], so by doing this we modify the behavior of *all* instances of the class.

One correct way of doing this:

Without changing the class:

```
setmetatable(
    object,
    setmetatable(
        {__call=table.concat},
        {__index=getmetatable(object)}
    )
)
```

How does this work? - We create a new metatable as in mistake #1, but instead of leaving it empty, we create a soft copy to the original metatable. One could say the new metatable "inherits" from the original one as if it was a class instance itself. We can now override values of the original metatable without modifying them.

Changing the class:

1st (recommended):

```
local __instance_meta = {__index = Class.__meta}
-- metatable for the metatable
-- As you can see, lua can get very meta very fast
function Class.new()
    return setmetatable({}, setmetatable({}, __instance_meta))
end
```

2nd (less recommended): see [1]

---

[1] `function Class.new() return setmetatable({}, {__index=Class}) end`

Read Object-Orientation online: https://riptutorial.com/lua/topic/8908/object-orientation

Chapter 11: Pattern matching

Syntax

- string.find(str, pattern [, init [, plain]]) -- Returns start and end index of match in str
- string.match(str, pattern [, index]) -- Matches a pattern once (starting at index)
- string.gmatch(str, pattern) -- Returns a function that iterates through all matches in str
- string.gsub(str, pattern, repl [, n]) -- Replaces substrings (up to a max of n times)
- `.` represents all characters
- `%a` represents all letters
- `%l` represents all lowercase letters
- `%u` represents all uppercase letters
- `%d` represents all digits
- `%x` represents all hexadecimal digits
- `%s` represents all whitespace characters
- `%p` represents all punctuation characters
- `%g` represents all *printable* characters except space
- `%c` represents all control characters
- `[set]` represents the class which is the union of all characters in set.
- `[^set]` represents the complement of set
- `*` greedy match 0 or more occurrences of previous character class
- `+` greedy match 1 or more occurrences of previous character class
- `-` lazy match 0 or more occurrences of previous character class
- `?` match exactly 0 or 1 occurrence of previous character class

Remarks

Throughout some examples, the notation `(<string literal>):function <string literal>` is used, which is equivalent to `string.function(<string literal>, <string literal>)` because all strings have a metatable with the `__index` field set to the `string` table.

Examples

Lua pattern matching

Instead of using regex, the Lua string library has a special set of characters used in syntax matches. Both can be very similar, but Lua pattern matching is more limited and has a different syntax. For instance, the character sequence `%a` matches any letter, while its upper-case version represents *all non-letters characters*, all characters classes (a character sequence that, as a pattern, can match a set of items) are listed below.

Character class	Matching section
%a	letters (A-Z, a-z)
%c	control characters (\n, \t, \r, ...)
%d	digits (0-9)
%l	lower-case letter (a-z)
%p	punctuation characters (!, ?, &, ...)
%s	space characters
%u	upper-case letters
%w	alphanumeric characters (A-Z, a-z, 0-9)
%x	hexadecimal digits (\3, \4, ...)
%z	the character with representation 0
.	Matches any character

As mentioned above, any upper-case version of those classes represents the complement of the class. For instance, `%D` will match any non-digit character sequence:

```
string.match("f123", "%D")          --> f
```

In addition to character classes, some characters have special functions as patterns:

```
( ) % . + - * [ ? ^ $
```

The character `%` represents a character escape, making `%?` match an interrogation and `%%` match the percentage symbol. You can use the `%` character with any other non-alphanumeric character, therefore, if you need to escape, for instance, a quote, you must use `\\` before it, which escapes any character from a lua string.

A character set, represented inside square brackets (`[]`), allows you to create a special character class, combining different classes and single characters:

```
local foo = "bar123bar2341"
print(foo:match "[arb]")            --> b
```

You can get the complement of the character set by starting it with `^`:

```
local foo = "bar123bar2341"
print(string.match(foo, "[^bar]"))  --> 1
```

In this example, `string.match` will find the first occurrence that isn't **b**, **a** or **r**.

Patterns can be more useful with the help of repetition/optional modifiers, patterns in lua offer these four characters:

Character	Modifier
+	One or more repetitions
*	Zero or more repetitions
-	Also zero or more repetitions
?	Optional (zero or one occurrence)

The character `+` represents one or more matched characters in the sequence and it will always return the longest matched sequence:

```
local foo = "12345678bar123"
print(foo:match "%d+")  --> 12345678
```

As you can see, `*` is similar to `+`, but it accepts zero occurrences of characters and is commonly used to match optional spaces between different patterns.

The character `-` is also similar to `*`, but instead of returning the longest matched sequence, it matches the shortest one.

The modifier `?` matches an optional character, allowing you to match, for example, a negative digit:

```
local foo = "-20"
print(foo:match "[+-]?%d+")
```

Lua pattern matching engine provides a few additional pattern matching items:

Character item	Description
`%n`	for n between 1 and 9 matches a substring equal to the n-th captured string

Character item	Description
`%bxy`	matches substring between two distinct characters (balanced pair of `x` and `y`)
`%f[set]`	frontier pattern: matches an empty string at any position such that the next character belongs to set and the previous character does not belong to set

string.find (Introduction)

The `find` function

First let's take a look at the `string.find` function in general:

The function `string.find (s, substr [, init [, plain]])` returns the start and end index of a substring if found, and nil otherwise, starting at the index `init` if it is provided (defaults to 1).

```
("Hello, I am a string"):find "am" --> returns 10 11
-- equivalent to string.find("Hello, I am a string", "am") -- see remarks
```

Introducing Patterns

```
("hello world"):find ".- " -- will match characters until it finds a space
    --> so it will return 1, 6
```

All **except** the following characters represent themselves `^$()%.[]*+-?)`. Any of these characters can be represented by a `%` following the character itself.

```
("137'5 m47ch s0m3 d1g175"):find "m%d%d" -- will match an m followed by 2 digit
    --> this will match m47 and return 7, 9

("stack overflow"):find "[abc]" -- will match an 'a', a 'b' or a 'c'
    --> this will return 3 (the A in stAck)

("stack overflow"):find "[^stack ]"
    -- will match all EXCEPT the letters s, t, a, c and k and the space character
    --> this will match the o in overflow

("hello"):find "o%d?" --> matches o, returns 5, 5
("hello20"):find "o%d?" --> matches o2, returns 5, 6
    -- the ? means the character is optional

("helllllo"):find "el+" --> will match elllll
("heo"):find "el+" --> won't match anything

("helllllo"):find "el*" --> will match elllll
("heo"):find "el*" --> will match e
```

```
("helelo"):find "h.+l" -- + will match as much as it gets
    --> this matches "helel"
("helelo"):find "h.-l" -- - will match as few as it can
    --> this wil only match "hel"

("hello"):match "o%d*"
    --> like ?, this matches the "o", because %d is optional
("hello20"):match "o%d*"
    --> unlike ?, it maches as many %d as it gets, "o20"
("hello"):match "o%d"
    --> wouldn't find anything, because + looks for 1 or more characters
```

The `gmatch` function

How it works

The `string.gmatch` function will take an input string and a pattern. This pattern describes on what to actually get back. This function will return a function which is actually an iterator. The result of this iterator will match to the pattern.

```
type(("abc"):gmatch ".") --> returns "function"

for char in ("abc"):gmatch "." do
    print char -- this prints:
    --> a
    --> b
    --> c
end

for match in ("#afdde6"):gmatch "%x%x" do
    print("#" .. match) -- prints:
    --> #af
    --> #dd
    --> #e6
end
```

Introducing captures:

This is very similair to the regular function, however it will return only the captures instead the full match.

```
for key, value in ("foo = bar, bar=foo"):gmatch "(%w+)%s*=%s*(%w+)" do
    print("key: " .. key .. ", value: " .. value)
    --> key: foo, value: bar
    --> key: bar, value: foo
end
```

The gsub function

do not confuse with the string.sub function, which returns a substring!

How it works

string argument

```
("hello world"):gsub("o", "0")
    --> returns "hell0 w0rld", 2
    -- the 2 means that 2 substrings have been replaced (the 2 Os)

("hello world, how are you?"):gsub("[^%s]+", "word")
    --> returns "word word, word word word?", 5

("hello world"):gsub("([^%s])([^%s]*)", "%2%1")
    --> returns "elloh orldw", 2
```

function argument

```
local word = "[^%s]+"

function func(str)
    if str:sub(1,1):lower()=="h" then
        return str
    else
        return "no_h"
    end
end
("hello world"):gsub(word, func)
    --> returns "hello no_h", 2
```

table argument

```
local word = "[^%s]+"

sub = {}
sub["hello"] = "g'day"
sub["world"] = "m8"

("hello world"):gsub(word, sub)
    --> returns "g'day m8"

("hello world, how are you?"):gsub(word, sub)
    --> returns "g'day m8, how are you?"
    -- words that are not in the table are simply ignored
```

Read Pattern matching online: https://riptutorial.com/lua/topic/5829/pattern-matching

Chapter 12: PICO-8

Introduction

The PICO-8 is a fantasy console programmed in embedded Lua. It already has good documentation. Use this topic to demonstrate undocumented or under-documented features.

Examples

Game loop

It's entirely possible to use PICO-8 as an interactive shell, but you probably want to tap into the game loop. In order to do that, you must create at least one of these callback functions:

- `_update()`
- `_update60()` (after v0.1.8)
- `_draw()`

A minimal "game" might simply draw something on the screen:

```
function _draw()
  cls()
  print("a winner is you")
end
```

If you define `_update60()`, the game loop tries to run at 60fps and ignores `update()` (which runs at 30fps). Either update function is called before `_draw()`. If the system detects dropped frames, it'll skip the draw function every other frame, so it's best to keep game logic and player input in the update function:

```
function _init()
  x = 63
  y = 63

  cls()
end

function _update()
  local dx = 0 dy = 0

  if (btn(0)) dx-=1
  if (btn(1)) dx+=1
  if (btn(2)) dy-=1
  if (btn(3)) dy+=1

  x+=dx
  y+=dy
  x%=128
  y%=128
end
```

```
function _draw()
  pset(x,y)
end
```

The `_init()` function is, strictly speaking, optional as commands outside of any function are run at startup. But it's a handy way to reset the game to initial conditions without rebooting the cartridge:

```
if (btn(4)) _init()
```

Mouse input

Although it's not officially supported, you can use mouse input in your games:

```
function _update60()
  x = stat(32)
  y = stat(33)

  if (x>0 and x<=128 and
      y>0 and y<=128)
  then

    -- left button
    if (band(stat(34),1)==1) then
      ball_x=x
      ball_y=y
    end
  end

  -- right button
  if (band(stat(34),2)==2) then
    ball_c+=1
    ball_c%=16
  end

  -- middle button
  if (band(stat(34),4)==4) then
    ball_r+=1
    ball_r%=64
  end
end

function _init()
  ball_x=63
  ball_y=63
  ball_c=10
  ball_r=1
end

function _draw()
  cls()
  print(stat(34),1,1)
  circ(ball_x,ball_y,ball_r,ball_c)
  pset(x,y,7) -- white
end
```

Game modes

If you want a title screen or an endgame screen, consider setting up a mode switching mechanism:

```
function _init()
  mode = 1
end

function _update()
  if (mode == 1) then
    if (btnp(5)) mode = 2
  elseif (mode == 2) then
    if (btnp(5)) mode = 3
  end
end

function _draw()
  cls()
  if (mode == 1) then
    title()
  elseif (mode == 2) then
    print("press 'x' to win")
  else
    end_screen()
  end
end

function title()
  print("press 'x' to start game")
end

function end_screen()
  print("a winner is you")
end
```

Read PICO-8 online: https://riptutorial.com/lua/topic/8715/pico-8

Chapter 13: Sets

Examples

Search for an item in a list

There's no built in way to search a list for a particular item. However Programming in Lua shows how you might build a set that can help:

```
function Set (list)
  local set = {}
  for _, l in ipairs(list) do set[l] = true end
  return set
end
```

Then you can put your list in the Set and test for membership:

```
local items = Set { "apple", "orange", "pear", "banana" }

if items["orange"] then
  -- do something
end
```

Using a Table as a Set

Create a set

```
 local set = {} -- empty set
```

Create a set with elements by setting their value to `true`:

```
 local set = {pear=true, plum=true}

 -- or initialize by adding the value of a variable:
 local fruit = 'orange'
 local other_set = {[fruit] = true} -- adds 'orange'
```

Add a member to the set

add a member by setting its value to `true`

```
  set.peach = true
  set.apple = true
  -- alternatively
  set['banana'] = true
  set['strawberry'] = true
```

Remove a member from the set

```
set.apple = nil
```

Using `nil` instead of `false` to remove 'apple' from the table is preferable because it will make iterating elements simpler. `nil` deletes the entry from the table while setting to `false` changes its value.

Membership Test

```
if set.strawberry then
    print "We've got strawberries"
end
```

Iterate over elements in a set

```
for element in pairs(set) do
    print(element)
end
```

Read Sets online: https://riptutorial.com/lua/topic/3875/sets

Chapter 14: Tables

Syntax

- ipairs(numeric_table) -- Lua table with numeric indices iterator
- pairs(input_table) -- generic Lua table iterator
- key, value = next(input_table, input_key) -- Lua table value selector
- table.insert(input_table, [position], value) -- insert specified value into the input table
- removed_value = table.remove(input_table, [position]) -- pop last or remove value specified by position

Remarks

Tables are the only built-in data structure available in Lua. This is either elegant simplicity or confusing, depending on how you look at it.

A Lua table is a collection of key-value pairs where the keys are unique and neither the key nor the value is `nil`. As such, a Lua table can resemble a dictionary, hashmap or associative array from other languages. Many structural patterns can be built with tables: stacks, queues, sets, lists, graphs, etc. Finally, tables can be used to build *classes* in Lua and to create a *module* system.

Lua does not enforce any particular rules on how tables are used. The items contained in a table can be a mixture of Lua types. So, for example, one table could contain strings, functions, booleans, numbers, *and even other tables* as values or keys.

A Lua table with consecutive positive integer keys beginning with 1 is said to have a sequence. The key-value pairs with positive integer keys are the elements of the sequence. Other languages call this a 1-based array. Certain standard operations and functions only work on the sequence of a table and some have non-deterministic behavior when applied to a table without a sequence.

Setting a value in a table to `nil` removes it from the table. Iterators would no longer see the related key. When coding for a table with a sequence, it is important to avoid breaking the sequence; Only remove the last element or use a function, like the standard `table.remove`, that shifts elements down to close the gap.

Examples

Creating tables

Creating an empty table is as simple as this:

```
local empty_table = {}
```

You can also create a table in the form of a simple array:

```
local numeric_table = {
    "Eve", "Jim", "Peter"
}
-- numeric_table[1] is automatically "Eve", numeric_table[2] is "Jim", etc.
```

Bear in mind that by default, table indexing starts at 1.

Also possible is creating a table with associative elements:

```
local conf_table = {
    hostname = "localhost",
    port     = 22,
    flags    = "-Wall -Wextra"
    clients  = {                -- nested table
        "Eve", "Jim", "Peter"
    }
}
```

The usage above is syntax sugar for what's below. The keys in this instance are of the type, string. The above syntax was added to make tables appear as records. This record-style syntax is paralleled by the syntax for indexing tables with string keys, as seen in 'basic usage' tutorial.

As explained in the remarks section, the record-style syntax doesn't work for every possible key. Additionally a key can be any value of any type, and the previous examples only covered strings and sequential numbers. In other cases you'll need to use the explicit syntax:

```
local unique_key = {}
local ops_table = {
    [unique_key] = "I'm unique!"
    ["^"]  = "power",
    [true] = true
}
```

Iterating tables

The Lua standard library provides a `pairs` function which iterates over the keys and values of a table. When iterating with `pairs` there is no specified order for traversal, *even if the keys of the table are numeric*.

```
for key, value in pairs(input_table) do
    print(key, " -- ", value)
end
```

For tables using **numeric keys**, Lua provides an `ipairs` function. The `ipairs` function will always iterate from `table[1]`, `table[2]`, etc. until the first `nil` value is found.

```
for index, value in ipairs(numeric_table) do
    print(index, ". ", value)
end
```

Be warned that iteration using `ipairs()` will not work as you might want on few occasions:

- `input_table` has "holes" in it. (See the section on "Avoiding gaps in tables used as arrays" for more information.) For example:

  ```
  table_with_holes = {[1] = "value_1", [3] = "value_3"}
  ```

- keys weren't all numeric. For example:

  ```
  mixed_table = {[1] = "value_1", ["not_numeric_index"] = "value_2"}
  ```

Of course, the following also works for a table that is a proper sequence:

```
for i = 1, #numeric_table do
    print(i, ". ", numeric_table[i])
end
```

Iterating a numeric table in reverse order is easy:

```
for i = #numeric_table, 1, -1 do
    print(i, ". ", numeric_table[i])
end
```

A final way to iterate over tables is to use the `next` selector in a generic `for` loop. Like `pairs` there is no specified order for traversal. (The `pairs` method uses `next` internally. So using `next` is essentially a more manual version of `pairs`. See `pairs` in Lua's reference manual and `next` in Lua's reference manual for more details.)

```
for key, value in next, input_table do
    print(key, value)
end
```

Basic Usage

Basic table usage includes accessing and assigning table elements, adding table content, and removing table content. These examples assume you know how to create tables.

Accessing Elements

Given the following table,

```
local example_table = {"Nausea", "Heartburn", "Indigestion", "Upset Stomach",
                       "Diarrhea", cure = "Pepto Bismol"}
```

One can index the sequential part of the table by using the index syntax, the argument to the index syntax being the key of the desired key-value pair. As explained in the creation tutorial, most of the declaration syntax is syntactic sugar for declaring key-value pairs. Sequentially included elements, like the first five values in `example_table`, use increasing integer values as keys; the record syntax uses the name of the field as a string.

```
print(example_table[2])        --> Heartburn
print(example_table["cure"])   --> Pepto Bismol
```

For string keys there is syntax sugar to parallel the record-style syntax for string keys in table creation. The following two lines are equivalent.

```
print(example_table.cure)      --> Pepto Bismol
print(example_table["cure"])   --> Pepto Bismol
```

You can access tables using keys that you haven't used before, that is not an error as it in other languages. Doing so returns the default value `nil`.

Assigning Elements

You can modify existing table elements by assigning to a table using the index syntax. Additionally, the record-style indexing syntax is available for setting values as well

```
example_table.cure = "Lots of water, the toilet, and time"
print(example_table.cure)    --> Lots of water, the toilet, and time

example_table[2] = "Constipation"
print(example_table[2])      --> Constipation
```

You can also add new elements to an existing table using assignment.

```
example_table.copyright_holder = "Procter & Gamble"
example_table[100] = "Emergency source of water"
```

Special Remark: Some strings are not supported with the record-syntax. See the remarks section for details.

Removing Elements

As stated previously, the default value for a key with no assigned value is `nil`. Removing an element from a table is as simple as resetting the value of a key back to the default value.

```
example_table[100] = "Face Mask"
```

The elements is now indistinguishable from an unset element.

Table Length

Tables are simply associative arrays (see remarks), but when contiguous integer keys are used starting from 1 the table is said to have a *sequence*.

Finding the length of the sequence part of a table is done using `#`:

```
local example_table = {'a', 'l', 'p', 'h', 'a', 'b', 'e', 't'}
print(#example_table)    --> 8
```

You can use the length operation to easily append items to a sequence table.

```
example_table[#example_table+1] = 'a'
print(#example_table)    --> 9
```

In the above example, the previous value of `#example_table` is `8`, adding `1` gives you the next valid integer key in the sequence, `9`, so... `example_table[9] = 'a'`. This works for any length of table.

Special Remark: Using integer keys that aren't contiguous and starting from 1 breaks the sequence making the table into a *sparse table*. The result of the length operation is undefined in that case. See the remarks section.

Using Table Library Functions to Add/Remove Elements

Another way to add elements to a table is the `table.insert()` function. The insert function only works on sequence tables. There are two ways to call the function. The first example shows the first usage, where one specifies the index to insert the element (the second argument). This pushes all elements from the given index to `#table` up one position. The second example shows the other usage of `table.insert()`, where the index isn't specified and the given value is appended to the end of the table (index `#table + 1`).

```
local t = {"a", "b", "d", "e"}
table.insert(t, 3, "c")        --> t = {"a", "b", "c", "d", "e"}

t = {"a", "b", "c", "d"}
table.insert(t, "e")           --> t = {"a", "b", "c", "d", "e"}
```

To parallel `table.insert()` for removing elements is `table.remove()`. Similarly it has two calling semantics: one for removing elements at a given position, and another for removing from the end of the sequence. When removing from the middle of a sequence, all following elements are shifted down one index.

```
local t = {"a", "b", "c", "d", "e"}
local r = table.remove(t, 3)       --> t = {"a", "b", "d", "e"}, r = "c"

t = {"a", "b", "c", "d", "e"}
r = table.remove(t)                --> t = {"a", "b", "c", "d"}, r = "e"
```

These two functions mutate the given table. As you might be able to tell the second method of calling `table.insert()` and `table.remove()` provides stack semantics to tables. Leveraging that, you can write code like the example below.

```
function shuffle(t)
    for i = 0, #t-1 do
        table.insert(t, table.remove(t, math.random(#t-i)))
    end
end
```

It implements the Fisher-Yates Shuffle, perhaps inefficiently. It uses the `table.insert()` to append the randomly extracted element onto the end of same table, and the `table.remove()` to randomly extract an element from the remaining unshuffled portion of the table.

Avoiding gaps in tables used as arrays

Defining our terms

By *array* here we mean a Lua table used as a sequence. For example:

```
-- Create a table to store the types of pets we like.
local pets = {"dogs", "cats", "birds"}
```

We're using this table as a sequence: a group of items keyed by integers. Many languages call this an array, and so will we. But strictly speaking, there's no such thing as an array in Lua. There are just tables, some of which are array-like, some of which are hash-like (or dictionary-like, if you prefer), and some of which are mixed.

An important point about our `pets` array is that is has no gaps. The first item, `pets[1]`, is the string "dogs", the second item, `pets[2]`, is the string "cats", and the last item, `pets[3]`, is "birds". Lua's standard library and most modules written for Lua assume 1 as the first index for sequences. A gapless array therefore has items from `1..n` without missing any numbers in the sequence. (In the limiting case, `n = 1`, and the array has only one item in it.)

Lua provides the built-in function `ipairs` to iterate over such tables.

```
-- Iterate over our pet types.
for idx, pet in ipairs(pets) do
  print("Item at position " .. idx .. " is " .. pet .. ".")
end
```

This would print "Item at position 1 is dogs.", "Item at position 2 is cats.", "Item at position 3 is birds."

But what happens if we do the following?

```
local pets = {"dogs", "cats", "birds"}
pets[12] = "goldfish"
for idx, pet in ipairs(pets) do
  print("Item at position " .. idx .. " is " .. pet .. ".")
end
```

An array such as this second example is a sparse array. There are gaps in the sequence. This array looks like this:

```
{"dogs", "cats", "birds", nil, nil, nil, nil, nil, nil, nil, nil, "goldfish"}
-- 1        2       3      4    5    6    7    8    9    10   11       12
```

The nil values do not take up any aditional memory; internally lua only saves the values `[1] = "dogs"`, `[2] = "cats"`, `[3] = "birtds"` and `[12] = "goldfish"`

To answer the immediate question, `ipairs` will stop after birds; "goldfish" at `pets[12]` will never be reached unless we adjust our code. This is because `ipairs` iterates from `1..n-1` where `n` is the position of the first `nil` found. Lua defines `table[length-of-table + 1]` to be `nil`. So in a proper sequence, iteration stops when Lua tries to get, say, the fourth item in a three-item array.

When?

The two most common places for issues to arise with sparse arrays are (i) when trying to determine the length of the array and (ii) when trying to iterate over the array. In particular:

- When using the `#` length operator since the length operator stops counting at the first `nil` found.
- When using the `ipairs()` function since as mentioned above it stops iterating at the first `nil` found.
- When using the `table.unpack()` function since this method stops unpacking at the first `nil` found.
- When using other functions that (directly or indirectly) access any of the above.

To avoid this problem, it is important to write your code so that if you expect a table to be an array, you don't introduce gaps. Gaps can be introduced in several ways:

- If you add something to an array at the wrong position.
- If you insert a `nil` value into an array.
- If you remove values from an array.

You might think, "But I would never do any of those things." Well, not intentionally, but here's a concrete example of how things could go wrong. Imagine that you want to write a filter method for Lua like Ruby's `select` and Perl's `grep`. The method will accept a test function and an array. It iterates over the array, calling the test method on each item in turn. If the item passes, then that item gets added to a results array which is returned at the end of the method. The following is a buggy implementation:

```
local filter = function (fun, t)
  local res = {}
  for idx, item in ipairs(t) do
    if fun(item) then
      res[idx] = item
    end
  end

  return res
end
```

The problem is that when the function returns `false`, we skip a number in the sequence. Imagine calling `filter(isodd, {1,2,3,4,5,6,7,8,9,10})`: there will be gaps in the returned table every time there's an even number in the array passed to `filter`.

Here's a fixed implementation:

```
local filter = function (fun, t)
  local res = {}
  for _, item in ipairs(t) do
    if fun(item) then
      res[#res + 1] = item
    end
  end

  return res
end
```

Tips

1. Use standard functions: `table.insert(<table>, <value>)` always appends to the end of the array. `table[#table + 1] = value` is a short hand for this. `table.remove(<table>, <index>)` will move all following values back to fill the gap (which can also make it slow).
2. Check for `nil` values **before** inserting, avoiding things like `table.pack(function_call())`, which might sneak `nil` values into our table.
3. Check for `nil` values **after** inserting, and if necessary filling the gap by shifting all consecutive values.
4. If possible, use placeholder values. For example, change `nil` for `0` or some other placeholder value.
5. If leaving gaps is unavoidable, this should be propperly documented (commented).
6. Write a `__len()` metamethod and use the `#` operator.

Example for 6.:

```
tab = {"john", "sansa", "daenerys", [10] = "the imp"}
print(#tab) --> prints 3
setmetatable(tab, {__len = function() return 10 end})
-- __len needs to be a function, otherwise it could just be 10
print(#tab) --> prints 10
for i=1, #tab do print(i, tab[i]) end
--> prints:
-- 1 john
-- 2 sansa
-- 3 daenerys
-- 4 nil
-- ...
-- 10 the imp

for key, value in ipairs(tab) do print(key, value) end
--> this only prints '1 john \n 2 sansa \n 3 daenerys'
```

Another alternative is to use the `pairs()` function and filter out the non-integer indices:

```
for key in pairs(tab) do
    if type(key) == "number" then
        print(key, tab[key]
    end
end
-- note: this does not remove float indices
-- does not iterate in order
```

Read Tables online: https://riptutorial.com/lua/topic/676/tables

Chapter 15: Variadic Arguments

Introduction

Varargs, as they are commonly known, allow functions to take an arbitrary number of arguments without specification. All arguments given to such a function are packaged into a single structure known as the *vararg list*; which is written as `...` in Lua. There are basic methods for extracting the number of given arguments and the value of those arguments using the `select()` function, but more advanced usage patterns can leverage the structure to it's full utility.

Syntax

- *...* -- Makes the function whose arguments list in which this appears a variadic function
- *select(what, ...)* -- If 'what' is a number in range 1 to the number of elements in the vararg, returns the 'what'th element to the last element in the vararg. The return will be nil if the index is out of bounds. If 'what' is the string '#', returns the number of elements in the vararg.

Remarks

Efficiency

The vararg list is implemented as a linked list in the PUC-Rio implementation of the language, this means that indexes are O(n). That means that iterating over the elements in a vararg using `select()`, like the example below, is an O(n^2) operation.

```
for i = 1, select('#', ...) do
    print(select(i, ...))
end
```

If you plan on iterating over the elements in a vararg list, first pack the list into a table. Table accesses are O(1), so iterating is O(n) in total. Or, if you are so inclined, see the `foldr()` example from the advanced usage section; it uses recursion to iterate over a vararg list in O(n).

Sequence Length Definition

The vararg is useful in that the length of the vararg respects any explicitly passed (or computed) nils. For example.

```
function test(...)
    return select('#', ...)
end

test()              --> 0
test(nil, 1, nil)   --> 3
```

This behavior conflicts with the behavior of tables however, where the length operator `#` does not work with 'holes' (embedded nils) in sequences. Computing the length of a table with holes is undefined and cannot be relied on. So, depending upon the values in `...`, taking the length of `{...}` may not result in the *'correct'* answer. In Lua 5.2+ `table.pack()` was introduced to handle this deficiency (there is a function in the example that implements this function in pure Lua).

Idiomatic Use

Because varargs carry around their length people use them as sequences to avoid the issue with holes in tables. This was not their intended usage and one the reference implementation of Lua does not optimize for. Although such usage is explored in the examples, it is generally frowned upon.

Examples

Basics

Variadic functions are created using the `...` ellipses syntax in the argument list of the function definition.

```
function id(...)
    return
end
```

If you called this function as `id(1, 2, 3, 4, 5)` then `...` (AKA the vararg list) would contain the values `1, 2, 3, 4, 5`.

Functions can take required arguments as well as `...`.

```
function head(x, ...)
    return x
end
```

The easiest way to pull elements from the vararg list is to simply assign variables from it.

```
function head3(...)
    local a, b, c = ...
    return a, b, c
end
```

`select()` can also be used to find the number of elements and extract elements from `...` indirectly.

```
function my_print(...)
    for i = 1, select('#', ...) do
        io.write(tostring(select(i, ...)) .. '\t')
    end
    io.write '\n'
end
```

`...` can be packed into a table for ease of use, by using `{...}`. This places all the arguments in the sequential part of the table.

5.2

`table.pack(...)` can also be used to pack the vararg list into a table. The advantage of `table.pack(...)` is that it sets the `n` field of the returned table to the value of `select('#', ...)`. This is important if your argument list may contain nils (see remarks section below).

```
function my_tablepack(...)
    local t = {...}
    t.n = select('#', ...)
    return t
end
```

The vararg list may also be returned from functions. The result is multiple returns.

```
function all_or_none(...)
    local t = table.pack(...)
    for i = 1, t.n do
        if not t[i] then
            return    -- return none
        end
    end
    return ...    -- return all
end
```

Advanced Usage

As stated in the basic examples, you can have variable bound arguments and the variable argument list (`...`). You can use this fact to recursively pull apart a list as you would in other languages (like Haskell). Below is an implementation of `foldr()` that takes advantage of that. Each recursive call binds the head of the vararg list to `x`, and passes the rest of the list to a recursive call. This destructures the list until there is only one argument (`select('#', ...) == 0`). After that, each value is applied to the function argument `f` with the previously computed result.

```
function foldr(f, ...)
    if select('#', ...) < 2 then return ... end
    local function helper(x, ...)
        if select('#', ...) == 0 then
          return x
        end
        return f(x, helper(...))
    end
    return helper(...)
end

function sum(a, b)
    return a + b
end

foldr(sum, 1, 2, 3, 4)
--> 10
```

You can find other function definitions that leverage this programming style here in Issue #3 through Issue #8.

Lua's sole idiomatic data structure is the table. The table length operator is undefined if there are `nil`s located anywhere in a sequence. Unlike tables, the vararg list respects explicit `nil`s as stated in the basic examples and the remarks section (please read that section if you haven't yet). With little work the vararg list can perform every operation a table can besides mutation. This makes the vararg list a good candidate for implementing immutable tuples.

```
function tuple(...)
    -- packages a vararg list into an easily passable value
    local co = coroutine.wrap(function(...)
        coroutine.yield()
        while true do
            coroutine.yield(...)
        end
    end)
    co(...)
    return co
end

local t = tuple((function() return 1, 2, nil, 4, 5 end)())

print(t())                 --> 1    2    nil    4    5    | easily unpack for multiple args
local a, b, d = t()        --> a = 1, b = 2, c = nil      | destructure the tuple
print((select(4, t())))    --> 4                          | index the tuple
print(select('#', t()))    --> 5                          | find the tuple arity (nil
respecting)

local function change_index(tpl, i, v)
    -- sets a value at an index in a tuple (non-mutating)
    local function helper(n, x, ...)
        if select('#', ...) == 0 then
            if n == i then
                return v
            else
                return x
            end
        else
            if n == i then
                return v, helper(n+1, ...)
            else
                return x, helper(n+1, ...)
            end
        end
    end
    return tuple(helper(1, tpl()))
end

local n = change_index(t, 3, 3)
print(t())                 --> 1    2    nil    4    5
print(n())                 --> 1    2    3    4    5
```

The main difference between what's above and tables is that tables are mutable and have pointer semantics, where the tuple does not have those properties. Additionally, tuples can hold explicit `nil`s and have a never-undefined length operation.

Read Variadic Arguments online: https://riptutorial.com/lua/topic/4475/variadic-arguments

Chapter 16: Writing and using modules

Remarks

The basic pattern for writing a module is to fill a table with keys that are function names and values that are the functions themselves. The module then returns this function for calling code to `require` and use. (Functions are first-class values in Lua, so storing a function in a table is easy and common.) The table can also contain any important constants in the form of, say, strings or numbers.

Examples

Writing the module

```
--- trim: a string-trimming module for Lua
-- Author, date, perhaps a nice license too
--
-- The code here is taken or adapted from  material in
-- Programming in Lua, 3rd ed., Roberto Ierusalimschy

-- trim_all(string) => return string with white space trimmed on both sides
local trim_all = function (s)
  return (string.gsub(s, "^%s*(.-)%s*$", "%1"))
end

-- trim_left(string) => return string with white space trimmed on left side only
local trim_left = function (s)
  return (string.gsub(s, "^%s*(.*)$", "%1"))
end

-- trim_right(string) => return string with white space trimmed on right side only
local trim_right = function (s)
  return (string.gsub(s, "^(.-)%s*$", "%1"))
end

-- Return a table containing the functions created by this module
return {
  trim_all = trim_all,
  trim_left = trim_left,
  trim_right = trim_right
}
```

An alternative approach to the one above is to create a top-level table and then store the functions directly in it. In that idiom, our module above would look like this:

```
-- A conventional name for the table that will hold our functions
local M = {}

-- M.trim_all(string) => return string with white space trimmed on both sides
function M.trim_all(s)
  return (string.gsub(s, "^%s*(.-)%s*$", "%1"))
end
```

```
-- M.trim_left(string) => return string with white space trimmed on left side only
function M.trim_left(s)
  return (string.gsub(s, "^%s*(.*)$", "%1"))
end

-- trim_right(string) => return string with white space trimmed on right side only
function M.trim_right(s)
  return (string.gsub(s, "^(.-)%s*$", "%1"))
end


return M
```

From the point of view of the caller, there is little difference between the two styles. (One difference worth mentioning is that the first style makes it more difficult for users to monkeypatch the module. This is either a pro or a con, depending on your point of view. For more detail about this, see this blog post by Enrique García Cota.)

Using the module

```
-- The following assumes that trim module is installed or in the caller's package.path,
-- which is a built-in variable that Lua uses to determine where to look for modules.
local trim = require "trim"

local msg = "    Hello, world!      "
local cleaned = trim.trim_all(msg)
local cleaned_right = trim.trim_right(msg)
local cleaned_left = trim.trim_left(msg)

-- It's also easy to alias functions to shorter names.
local trimr = trim.trim_right
local triml = trim.trim_left
```

Read Writing and using modules online: https://riptutorial.com/lua/topic/1148/writing-and-using-modules

Credits

S. No	Chapters	Contributors
1	Getting started with Lua	1971chevycamaro, Allan Burleson, Community, DarkWiiPlayer, Darryl L Johnson, elektron, greatwolf, Guilherme Salazar, hjpotter92, hugomg, Kamiccolo, lhf, Nikola Geneshki, SoniEx2, Telemachus
2	Booleans in Lua	DarkWiiPlayer, engineercoding, greatwolf, Kamiccolo, Katenkyo , Samuel McKay, Telemachus
3	Coroutines	010110110101, Bjornir, Eshkation, Kamiccolo, ktb, SoniEx2
4	Error Handling	Black, DarkWiiPlayer, engineercoding, greatwolf
5	Functions	Art C, Basilio German, DarkWiiPlayer, Firas Moalla, greatwolf, Guilherme Salazar, Jon Ericson, Katenkyo, ktb, MBorsch, Necktrox, qaisjp, RBerteig, Romário, SoniEx2, Telemachus, Unheilig, WolfgangTS
6	Garbage collector and weak tables	greatwolf, Kamiccolo, val
7	Introduction to Lua C API	greatwolf, Jeremy Thien, Kamiccolo, Luiz Menezes, RBerteig, tversteeg
8	Iterators	Adam, Egor Skriptunoff, greatwolf
9	Metatables	DarkWiiPlayer, greatwolf, Kamiccolo, pschulz, Telemachus
10	Object-Orientation	DarkWiiPlayer, Kamiccolo
11	Pattern matching	DarkWiiPlayer, engineercoding, Eshkation, greatwolf, Kamiccolo , Stephen Leppik
12	PICO-8	Jon Ericson
13	Sets	Egor Skriptunoff, Jon Ericson, ryanpattison
14	Tables	DarkWiiPlayer, greatwolf, Hastumer, Kamiccolo, ktb, mjanicek, SoniEx2, Telemachus, Tom Blodget
15	Variadic Arguments	greatwolf, Kamiccolo, ktb, RamenChef, SoniEx2
16	Writing and using modules	SoniEx2, Telemachus