EBook Gratis

# APRENDIZAJE Lua

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#lua

# Tabla de contenido

# Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: lua

It is an unofficial and free Lua ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Lua.



Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

# Capítulo 1: Empezando con Lua

## Observaciones


Lua es un lenguaje de scripting minimalista, liviano e integrable. Está siendo diseñado, implementado y mantenido por un equipo de la PUC-Rio , la Pontificia Universidad Católica de Río de Janeiro en Brasil. La lista de correo está abierta para involucrarse.

Los casos de uso comunes para Lua incluyen scripts de videojuegos, extensión de aplicaciones con complementos y configuraciones, envolviendo un poco de lógica empresarial de alto nivel o simplemente incorporándolos en dispositivos como televisores, automóviles, etc.

Para las tareas de alto rendimiento, hay una implementación independiente utilizando el compilador Just -in-time disponible llamado LuaJIT .

## Versiones

| Versión | Notas | Fecha de lanzamiento |
|---|---|---|
| 1.0 | Lanzamiento inicial, no público. | 1993-07-28 |
| 1.1 | Primer lanzamiento público. Documento de conferencia describiéndolo. | 1994-07-08 |
| 2.1 | A partir de Lua 2.1, Lua estuvo disponible de forma gratuita para todos los fines, incluidos los usos comerciales. Diario que lo describe. | 1995-02-07 |
| 2.2 | Cadenas largas, la interfaz de depuración, mejores seguimientos de pila | 1995-11-28 |
| 2.4 | Compilador `luac` externo | 1996-05-14 |
| 2.5 | Patrón de coincidencia y funciones vararg. | 1996-11-19 |
| 3.0 | Se introdujo auxlib, una biblioteca para ayudar a escribir bibliotecas Lua | 1997-07-01 |

| Versión | Notas | Fecha de lanzamiento |
|---|---|---|
| 3.1 | Funciones anónimas y cierres de funciones mediante "upvalues". | 1998-07-11 |
| 3.2 | Biblioteca de depuración y nuevas funciones de tabla. | 1999-07-08 |
| 3.2.2 | | 2000-02-22 |
| 4.0 | Múltiples estados, declaraciones "for", renovación de API. | 2000-11-06 |
| 4.0.1 | | 2002-07-04 |
| 5.0 | Coroutines, metatables, alcance léxico completo, llamadas de cola, booleanos pasan a licencia MIT. | 2003-04-11 |
| 5.0.3 | | 2006-06-26 |
| 5.1 | Renovación del sistema de módulos, recolector de basura incremental, metatables para todos los tipos, actualización de luaconf.h, analizador completamente reentrante, argumentos variadic. | 2006-02-21 |
| 5.1.5 | | 2012-02-17 |
| 5.2 | Colector de basura de emergencia, goto, finalizadores para mesas. | 2011-12-16 |
| 5.2.4 | | 2015-03-07 |
| 5.3 | Compatibilidad básica con UTF-8, operaciones bitwise, enteros de 32 / 64bit. | 2015-01-12 |
| 5.3.4 | Ultima versión. | 2017-01-12 |

# Examples

## Instalación

### Binarios

Los binarios de Lua son proporcionados por la mayoría de las distribuciones de GNU / Linux como un paquete.

Por ejemplo, en Debian, Ubuntu y sus derivados se pueden adquirir ejecutando esto:

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

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

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

Hay algunas compilaciones semi oficiales proporcionadas para Windows, MacOS y algunos otros sistemas operativos alojados en SourceForge .

Los usuarios de Apple también pueden instalar Lua fácilmente usando Homebrew :

```
brew install lua
```

(Actualmente Homebrew tiene 5.2.4, para 5.3 ver Homebrew / versiones ).

**Fuente**

La fuente está disponible en la página oficial . La adquisición de fuentes y la construcción en sí debe ser trivial. En sistemas Linux lo siguiente debería ser suficiente:

```
$ 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
```

En el ejemplo anterior, básicamente estamos descargando un `tarball` fuente desde el sitio oficial, verificando su suma de comprobación y extrayendo y ejecutando `make` . (Revise la suma de verificación en la página oficial ).

Nota: debe especificar qué objetivo de compilación desea. En el ejemplo, especificamos `linux` . Otros objetivos de compilación disponibles incluyen `solaris` , `aix` , `bsd` , `freebsd` , `macosx` , `mingw` , etc. Consulte `doc/readme.html` , que se incluye en la fuente, para obtener más detalles. (También puede encontrar la última versión de README en línea ).

**Módulos**

Las bibliotecas estándar están limitadas a primitivas:

- `coroutine` - funcionalidad de gestión de coroutine
- `debug` - debug hooks y herramientas
- `io` - primitivas básicas de IO
- `package` - funcionalidad de gestión de módulos
- `string` - string y funcionalidad de coincidencia de patrones específicos de Lua
- `table` - primitivas para tratar con un tipo Lua esencial pero complejo - tablas
- `os` - operaciones básicas del sistema operativo
- `utf8` - `utf8` básicas de UTF-8 (desde Lua 5.3)

Todas estas bibliotecas pueden deshabilitarse para una compilación específica o cargarse en tiempo de ejecución.

Las bibliotecas y la infraestructura de Lua de terceros para distribuir módulos son escasas, pero

están mejorando. Proyectos como LuaRocks , Lua Toolbox y LuaDist están mejorando la situación. Se puede encontrar mucha información y muchas sugerencias en el antiguo Wiki de Lua , pero tenga en cuenta que parte de esta información es bastante antigua y está desactualizada.

## Comentarios

Los comentarios de una sola línea en Lua comienzan con `--` y continúan hasta el final de la línea:

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

Los comentarios del bloque comienzan con `--[[` y terminan con `]]` :

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

Los comentarios de bloque utilizan el mismo estilo de delimitadores que las cadenas largas; se puede agregar cualquier número de signos iguales entre los paréntesis para delimitar un comentario:

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

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

Un buen truco para comentar trozos de código es rodearlo con `--[[` y `--]]` :

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

Para reactivar el fragmento, simplemente añada un `-` a la secuencia de apertura de comentarios:

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

De esta manera, la secuencia `--` en la primera línea comienza un comentario de una sola línea, al igual que la última línea, y la declaración de `print` no se comenta.

Yendo un paso más allá, se pueden configurar dos bloques de código de tal manera que si el primer bloque se comenta, el segundo no lo estará, y viceversa:

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

Para activar el segundo fragmento mientras deshabilita el primer fragmento, elimine el encabezado `-` en la primera línea:

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

## Ejecución de programas lua.

Por lo general, Lua se envía con dos binarios:

- `lua` - intérprete independiente y shell interactivo
- `luac` - compilador bytecode

Digamos que tenemos un programa de ejemplo ( `bottles_of_mate.lua` ) como este:

```
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
```

El programa en sí puede ejecutarse ejecutando lo siguiente en su shell:

```
$ lua bottles_of_mate.lua
```

La salida debería verse así, ejecutándose en el bucle sin fin:

```
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.
...
```

Puedes compilar el programa en el código de bytes de Lua ejecutando lo siguiente en tu shell:

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

También está disponible la lista de bytecode ejecutando lo siguiente:

```
$ 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
```

### Empezando

# 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.
```

## tipos

```
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!"
```

## El tipo especial `nil`

Otro tipo en Lua es `nil` . El único valor en el `nil` tipo es `nil` . `nil` existe para ser diferente de todos los demás valores en Lua. Es un tipo de valor sin valor.

```
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!
```

## expresiones

```
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
```

## Definiendo funciones

```
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
```

## booleanos

Solo `false` y `nil` evalúan como falso, todo lo demás, incluyendo `0` y la cadena vacía evalúan como verdadero.

## recolección de basura

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

## mesas

```
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.
```

Estos son los conceptos básicos, pero hay una sección sobre tablas con más información.

## condiciones

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

# para bucles

Hay dos tipos de bucle `for` en Lua: un bucle numérico `for` bucle y un bucle genérico `for` .

- Un numérico `for` bucle tiene la siguiente forma:

```
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
```

La tercera expresión en un numérico `for` bucle es el paso por el cual el bucle se incrementará. Esto hace que sea fácil hacer bucles inversos:

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

Si se omite la expresión de paso, Lua supone un paso predeterminado de 1.

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

También tenga en cuenta que la variable de bucle es local al bucle `for` . No existirá después de que termine el bucle.

- Genérica `for` bucles de trabajo a través de todos los valores que devuelve una función de iterador:

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

Lua proporciona varios iteradores integrados (por ejemplo, `pairs` , `ipairs` ), y los usuarios pueden definir sus propios iteradores personalizados para usarlos con genéricos `for` bucles.

# hacer bloques

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

### Algunas cosas complicadas

A veces, Lua no se comporta como uno pensaría después de leer la documentación. Algunos de estos casos son:

# Nil y Nothing no son lo mismo (Common PITFALL!)

Como se esperaba, `table.insert(my_table, 20)` agrega el valor `20` a la tabla, y `table.insert(my_table, 5, 20)` agrega el valor 20 en la 5ª posición. ¿Qué hace, `table.insert(my_table, 5, nil)` embargo, `table.insert(my_table, 5, nil)` ? Uno podría esperar que trate a `nil` como ningún argumento en absoluto, e inserte el valor `5` al final de la tabla, pero en realidad agrega el valor `nil` en la quinta posición de la tabla. ¿Cuándo es esto un problema?

```
(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)
```

Algo similar sucede con `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
```

Esto también puede dar lugar a errores al utilizar un código de terceros. Si, por ejemplo, la documentación de algunos estados de función "devuelve donas si tiene suerte, nula", la implementación *podría ser* algo así.

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

esta implementación puede parecer razonable al principio; devuelve donas cuando tiene que hacerlo, y cuando escribe `result = func(false)` resultado contendrá el valor `nil` .

Sin embargo, si se escribiera `print(tostring(func(false)))` lua arrojaría un error que se parece a este `stdin:1: bad argument #1 to 'tostring' (value expected)`

¿Porqué es eso? `tostring` claramente obtiene una discusión, aunque sea `nil` . Incorrecto. func no devuelve nada en absoluto, por lo que `tostring(func(false))` es lo mismo que `tostring()` y NO lo mismo que `tostring(nil)` .

Los errores que dicen "valor esperado" son una clara indicación de que esta podría ser la fuente del problema.

# Dejando huecos en matrices

Este es un gran escollo si eres nuevo en lua, y hay mucha información en la categoría de tablas.

### Hola Mundo

Este es el código hola mundo:

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

¿Cómo funciona? ¡Es sencillo! Lua ejecuta `print()` función `print()` y usa `"Hello World"` cadena `"Hello World"` como argumento.

Lea Empezando con Lua en línea: https://riptutorial.com/es/lua/topic/659/empezando-con-lua

# Capítulo 2: Argumentos Variados

## Introducción

*Los Varargs* , como se conocen comúnmente, permiten que las funciones tomen un número arbitrario de argumentos sin especificación. Todos los argumentos dados a una función de este tipo se empaquetan en una estructura única conocida como la *lista vararg* ; que se escribe como `...` en lua. Existen métodos básicos para extraer el número de argumentos dados y el valor de esos argumentos utilizando la función `select()` , pero los patrones de uso más avanzados pueden aprovechar la estructura para su utilidad completa.

## Sintaxis

- *...* - Hace que la función cuyos argumentos en la lista en la que aparece aparezca sea una función variable.
- *seleccione (qué, ...)* - Si 'qué' es un número en el rango 1 al número de elementos en el vararg, devuelve el elemento 'qué' al último elemento en el vararg. La devolución será nula si el índice está fuera de límites. Si 'qué' es la cadena '#', devuelve el número de elementos en el vararg.

## Observaciones

### Eficiencia

La lista vararg se implementa como una lista vinculada en la implementación del lenguaje de la PUC-Rio, esto significa que los índices son O (n). Eso significa que iterar sobre los elementos en un vararg usando `select()` , como en el siguiente ejemplo, es una operación O (n ^ 2).

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

Si planea iterar sobre los elementos en una lista vararg, primero guarde la lista en una tabla. Los accesos de tabla son O (1), por lo que la iteración es O (n) en total. O, si lo desea, vea el ejemplo de `foldr()` en la sección de uso avanzado; utiliza la recursividad para iterar sobre una lista vararg en O (n).

### Definición de longitud de secuencia

El vararg es útil porque la longitud del vararg respeta cualquier nils explícitamente pasados (o computados). Por ejemplo.

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

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

Sin embargo, este comportamiento entra en conflicto con el comportamiento de las tablas, donde el operador de longitud `#` no funciona con 'agujeros' (nils incrustados) en secuencias. El cálculo de la longitud de una mesa con agujeros no está definido y no se puede confiar en él. Entonces, dependiendo de los valores en `...` , tomar la longitud de `{...}` puede no resultar en la respuesta *'correcta'* . En Lua 5.2+ se introdujo `table.pack()` para manejar esta deficiencia (hay una función en el ejemplo que implementa esta función en Lua puro).

**Uso idiomático**

Debido a que los varargs llevan su longitud, las personas los usan como secuencias para evitar el problema con los agujeros en las tablas. Este no fue su uso previsto y la implementación de referencia de Lua no se optimiza. Aunque tal uso se explora en los ejemplos, generalmente está mal visto.

# Examples

## Lo esencial

Las funciones Variadic se crean usando la sintaxis de `...` puntos suspensivos en la lista de argumentos de la definición de función.

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

Si llama a esta función como `id(1, 2, 3, 4, 5)` entonces `...` (También conocida como la lista vararg) contendría los valores `1, 2, 3, 4, 5` .

Las funciones pueden tomar argumentos requeridos, así como `...`

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

La forma más fácil de extraer elementos de la lista vararg es simplemente asignar variables de ella.

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

`select()` también se puede usar para encontrar el número de elementos y extraer elementos de `...` indirectamente.

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

`...` se puede empaquetar en una tabla para facilitar su uso, utilizando `{...}` . Esto coloca todos los argumentos en la parte secuencial de la tabla.

5.2

`table.pack(...)` también se puede usar para empaquetar la lista vararg en una tabla. La ventaja de `table.pack(...)` es que establece el campo `n` de la tabla devuelta en el valor de `select('#', ...)` . Esto es importante si su lista de argumentos puede contener nils (consulte la sección de comentarios a continuación).

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

La lista vararg también puede devolverse desde las funciones. El resultado es múltiples devoluciones.

```
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
```

## Uso avanzado

Como se indica en los ejemplos básicos, puede tener argumentos de límite variable y la lista de argumentos variable ( `...` ). Puede usar este hecho para separar una lista recursivamente como lo haría en otros idiomas (como Haskell). A continuación se muestra una implementación de `foldr()` que se aprovecha de eso. Cada llamada recursiva enlaza el encabezado de la lista vararg a `x` , y pasa el resto de la lista a una llamada recursiva. Esto destruye la lista hasta que solo hay un argumento ( `select('#', ...) == 0` ). Después de eso, cada valor se aplica al argumento de función `f` con el resultado calculado previamente.

```
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
```

Puede encontrar otras definiciones de funciones que aprovechan este estilo de programación aquí en el Número 3 a través del Número 8.

La única estructura de datos idiomática de Lua es la tabla. El operador de longitud de la tabla no está definido si no hay `nil` ubicados en ninguna parte de una secuencia. A diferencia de las tablas, la lista vararg respeta explícita `nil` s como se indica en los ejemplos básicos y la sección de observaciones (por favor leer esa sección, si no lo ha hecho todavía). Con poco trabajo, la lista de vararg puede realizar todas las operaciones que puede realizar una tabla además de la mutación. Esto hace que la lista vararg sea un buen candidato para implementar tuplas inmutables.

```
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
```

La principal diferencia entre lo que está arriba y las tablas es que las tablas son mutables y tienen una semántica de puntero, donde la tupla no tiene esas propiedades. Además, las tuplas pueden contener `nil` s explícitos y tener una operación de longitud nunca definida.

Lea Argumentos Variados en línea: https://riptutorial.com/es/lua/topic/4475/argumentos-variados

# Capítulo 3: Booleanos en Lua

## Observaciones

Booleanos, la verdad y la falsedad son directos en Lua. Para revisar:

1. Hay un tipo booleano con exactamente dos valores: `true` y `false` .
2. En un contexto condicional ( `if` , `elseif` , `while` , `until` ), no se requiere un valor booleano. Cualquier expresión puede ser utilizada.
3. En un contexto condicional, `false` y `nil` cuentan como falso, y todo lo demás cuenta como verdadero.
4. Aunque 3 ya implica esto: si viene de otros idiomas, recuerde que `0` y la cadena vacía cuentan como verdaderos en contextos condicionales en Lua.

## Examples

### El tipo booleano

# Booleanos y otros valores

Cuando se trata de lua, es importante diferenciar entre los valores booleanos `true` y `false` y valores que se evalúan como verdadero o falso.

Solo hay dos valores en lua que evalúan a falso: `nil` y `false` , mientras que todo lo demás, incluido el `0` numérico, se evalúa como verdadero.

Algunos ejemplos de lo que esto significa:

```
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.
```

# Operaciones logicas

Los operadores lógicos en lua no necesariamente devuelven valores booleanos:

`and` devolverá el segundo valor si el primer valor se evalúa como verdadero;

`or` devuelve el segundo valor si el primer valor se evalúa como falso;

Esto permite simular el operador ternario, al igual que en otros idiomas:

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

Esto también puede usarse para inicializar tablas si no existen

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

o para evitar el uso de sentencias if, facilitando la lectura del código

```
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)
```

# Comprobando si las variables están definidas

También se puede verificar fácilmente si existe una variable (si está definida), ya que las variables no existentes devuelven `nil` , lo que se evalúa como falso.

```
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
```

El único caso donde esto no se aplica es cuando una variable almacena el valor `false` , en cuyo caso técnicamente existe pero aún se evalúa como falso. Debido a esto, es un mal diseño crear funciones que devuelvan `false` y `nil` según el estado o la entrada. Sin embargo, podemos comprobar si tenemos un valor `nil` o `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
```

## Contextos condicionales

Los contextos condicionales en Lua ( `if` , `elseif` , `while` , `until` ) no requieren un valor booleano. Como muchos idiomas, cualquier valor de Lua puede aparecer en una condición. Las reglas para la evaluación son simples:

1. `false` y `nil` cuentan como falso.
2. Todo lo demás cuenta como verdad.

```
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
```

### Operadores logicos

En Lua, los booleanos pueden ser manipulados a través de *operadores lógicos* . Estos operadores incluyen `not` , `and` , y `or` .

En expresiones simples, los resultados son bastante directos:

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

---

## Orden de precedencia

El orden de prioridad es similar a los operadores matemáticos unarios `-` , `*` y `+` :

- `not`
- entonces `and`
- entonces `or`

Esto puede llevar a expresiones complejas:

```
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
```

---

## Evaluación abreviada

Los operadores `and` y `or` solamente podrían ser evaluados usando el primer operando, siempre que el segundo es innecesario:

```
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
```

---

## Operador condicional idiomático

Debido a la precedencia de los operadores lógicos, la capacidad de evaluación de atajos y la evaluación de valores no `false` y no `nil` como `true` , hay un operador condicional idiomático disponible en 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
```

Además, debido a la naturaleza de las estructuras `x and a or b` , `a` nunca se *devolverá* si se evalúa como `false` , este condicional siempre devolverá `b` sin importar lo que sea `x` .

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

### Tablas de la verdad

Los operadores lógicos en Lua no "devuelven" el valor booleano, sino uno de sus argumentos. Usando `nil` para falso y los números para verdadero, así es como se comportan.

```
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
```

Como puede ver, Lua siempre devolverá el primer valor que hace que la verificación *falle* o tenga *éxito* . Aquí están las tablas de verdad que muestran eso.

```
  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
```

Para aquellos que lo necesitan, aquí hay dos funciones que representan estos operadores lógicos.

```
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
```

### Emulando al operador ternario con 'y' 'u' operadores lógicos.

En LUA, los operadores lógicos `and` y `or` devuelve uno de los operandos como el resultado en lugar de un resultado booleano. Como consecuencia, este mecanismo puede ser explotado para emular el comportamiento del operador ternario a pesar de que lua no tiene un operador ternario "real" en el idioma.

## Sintaxis

*condición* **y** *truthy_expr* **o** *falsey_expr*

# Uso en asignación / inicialización de variables

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

# Uso en constructor de tablas.

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

# Utilizar como argumento de función

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

# Utilizar en declaración de retorno

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

# Advertencia

Hay situaciones donde este mecanismo no tiene el comportamiento deseado. Considera este caso

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

En un operador ternario 'real', el valor esperado de `var` es `false` . En lua, sin embargo, la evaluación `and` 'falla' porque el segundo operando es falsey. Como resultado, `var should not happen` lugar.

Dos posibles soluciones a este problema, refactoriza esta expresión para que el operando del medio no sea falsey. p.ej.

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

o, alternativamente, utilizar la clásica `if then else` construir.

Lea Booleanos en Lua en línea: https://riptutorial.com/es/lua/topic/3101/booleanos-en-lua

# Capítulo 4: Conjuntos

## Examples

### Buscar un artículo en una lista

No hay una forma integrada de buscar una lista para un elemento en particular. Sin embargo, Programación en Lua muestra cómo puedes construir un conjunto que pueda ayudar:

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

Luego puedes poner tu lista en el Set y probar la membresía:

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

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

### Usando una mesa como un conjunto

## Crear un conjunto

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

Crea un conjunto con elementos estableciendo su valor en `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'
```

## Añadir un miembro al conjunto.

agrega un miembro estableciendo su valor en `true`

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

## Eliminar un miembro del conjunto

```
set.apple = nil
```

Es preferible utilizar `nil` lugar de `false` para eliminar 'apple' de la tabla porque hará que los elementos de iteración sean más simples. `nil` elimina la entrada de la tabla mientras se configura en `false` cambia su valor.

## Prueba de membresía

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

## Iterar sobre elementos en un conjunto.

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

Lea Conjuntos en línea: https://riptutorial.com/es/lua/topic/3875/conjuntos

# Capítulo 5: Coroutines

## Sintaxis

- coroutine.create (function) devuelve una coroutine (type (coroutine) == 'thread') que contiene la función.
- coroutine.resume (co, ...) reanudar, o iniciar la coroutine. Cualquier argumento adicional dado para reanudar se devuelve desde coroutine.yield () que previamente pausó la coroutine. Si no se ha iniciado la rutina, los argumentos adicionales se convierten en los argumentos de la función.
- coroutine.yield (...) produce la coroutine actualmente en ejecución. La ejecución se recupera después de la llamada a coroutine.resume () que inició esa coroutine. Cualquier argumento dado para ceder se devuelve del coroutine.resume () correspondiente que inició la coroutine.
- coroutine.status (co) devuelve el estado de la coroutine, que puede ser:
    - "muerto": la función en la coroutine ha llegado a su fin y la coroutine ya no se puede reanudar.
    - "running": el coroutine se ha reanudado y se está ejecutando
    - "normal": la coroutine ha reanudado otra coroutine.
    - "suspendido": la corutina ha cedido, y está esperando para ser reanudada
- coroutine.wrap (función) devuelve una función que, cuando se le llama, reanuda la coroutina que habría sido creada por coroutine.create (función).

## Observaciones

El sistema coroutine ha sido implementado en lua para emular multiproceso existente en otros idiomas. Funciona cambiando a una velocidad extremadamente alta entre diferentes funciones para que el usuario humano piense que se ejecutan al mismo tiempo.

## Examples

### Crea y utiliza una coroutina.

Todas las funciones para interactuar con coroutines están disponibles en la tabla de **coroutine** . Se crea una nueva coroutine usando la función **coroutine.create** con un solo argumento: una función con el código a ejecutar:

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

```
print(thread1)
-->> thread: 6b028b8c
```

Un objeto de coroutine devuelve el valor del tipo **thread** , que representa una nueva coroutine. Cuando se crea una nueva coroutine, su estado inicial se suspende:

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

Para reanudar o iniciar una rutina, se utiliza la función **coroutine.resume** , el primer argumento dado es el objeto de hilo:

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

Ahora el coroutine ejecuta el código y termina, cambiando su estado a **muerto** , que no se puede reanudar.

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

Coroutines puede suspender su ejecución y reanudarla más tarde gracias a la función **coroutine.yield** :

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

Como puede ver, **coroutine.yield ()** está presente dentro del bucle for, ahora, cuando reanudamos la coroutine, ejecutará el código hasta que alcance un coroutine.yield:

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

Después de terminar el bucle, el estado del hilo se convierte en **muerto** y no se puede reanudar. Coroutines también permite el intercambio entre datos:

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

Si la rutina se ejecuta de nuevo sin argumentos adicionales, el *complemento* seguirá siendo el

argumento de la primera reanudación, en este caso "stackoverflow":

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

Finalmente, cuando finaliza una rutina, cualquier valor devuelto por su función va al currículum correspondiente:

```
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
```

En esta función, se utilizan Coroutines para devolver los valores a un hilo de llamada desde lo profundo de una llamada recursiva.

```
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 también se puede utilizar para la evaluación perezosa.

```
-- 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 se puede usar para construcciones de tuberías como se describe en Programming In Lua . El autor de PiL, Roberto Ierusalimschy, también ha publicado un artículo sobre el uso de las coroutinas para implementar mecanismos de control de flujo más avanzados y generales, como las continuaciones.

Lea Coroutines en línea: https://riptutorial.com/es/lua/topic/3410/coroutines

# Capítulo 6: Escribir y utilizar módulos.

## Observaciones

El patrón básico para escribir un módulo es llenar una tabla con claves que son nombres de funciones y valores que son las funciones en sí mismas. El módulo luego devuelve esta función para que el código de llamada `require` y use. (Las funciones son valores de primera clase en Lua, por lo que almacenar una función en una tabla es fácil y común). La tabla también puede contener cualquier constante importante en forma de, digamos, cadenas o números.

## Examples

### Escribiendo el modulo

```
--- 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
}
```

Un enfoque alternativo al anterior es crear una tabla de nivel superior y luego almacenar las funciones directamente en ella. En ese idioma, nuestro módulo anterior se vería así:

```
-- 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
```

Desde el punto de vista de la persona que llama, hay poca diferencia entre los dos estilos. (Una diferencia que vale la pena mencionar es que el primer estilo hace que sea más difícil para los usuarios programar el módulo. Esto es un profesional o una estafa, según su punto de vista. Para obtener más detalles sobre esto, consulte esta publicación del blog de Enrique García Cota.

## Usando el modulo

```
-- 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
```

Lea Escribir y utilizar módulos. en línea: https://riptutorial.com/es/lua/topic/1148/escribir-y-utilizar-modulos-

# Capítulo 7: Funciones

## Sintaxis

- *funcname* = function (paramA, paramB, ...) body; return exprlist end - una función simple
- función *funcname* (paramA, paramB, ...) body; return exprlist end - taquigrafía de arriba
- local *funcname* = function (paramA, paramB, ...) body; return exprlist end - a lambda
- *funcname* local; *funcname* = function (paramA, paramB, ...) body; return exprlist end - lambda que puede hacer llamadas recursivas
- función local *funcname* (paramA, paramB, ...) body; return exprlist end - taquigrafía de arriba
- *funcname* (paramA, paramB, ...) - llamar a una función
- local *var* = *var* o "Predeterminado" - un parámetro predeterminado
- devuelve nil, "mensajes de error" - forma estándar de abortar con un error

## Observaciones

Las funciones generalmente se configuran con la `function a(b,c) ... end` y, rara vez, con el establecimiento de una variable en una función anónima ( `a = function(a,b) ... end` ). Lo contrario es cierto cuando se pasan las funciones como parámetros, la mayoría de las funciones anónimas se usan y las funciones normales no se usan con tanta frecuencia.

## Examples

### Definiendo una función

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

Veamos la sintaxis. En primer lugar, vemos una palabra clave de `function` . Bueno, eso es bastante descriptivo. A continuación vemos el identificador `add` ; el nombre. Luego vemos los argumentos `(a, b)` pueden ser cualquier cosa, y son locales. Solo dentro del cuerpo de la función podemos acceder a ellos. Vamos a saltar al final, vemos ... bueno, ¡el `end` ! Y todo lo que está en medio es la función del cuerpo; El código que se ejecuta cuando se llama. La palabra clave `return` es lo que hace que la función realmente dé un resultado útil. Sin ella, la función no devuelve nada, lo que equivale a devolver nil. Por supuesto, esto puede ser útil para cosas que interactúan con IO, por ejemplo:

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

En esa función, no usamos la declaración de retorno.

Las funciones también pueden devolver valores de forma condicional, lo que significa que una función tiene la opción de no devolver nada (nulo) o un valor. Esto se demuestra en el siguiente ejemplo.

```
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
```

También es posible que una función devuelva múltiples valores separados por comas, como se muestra:

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

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

Las funciones también pueden ser declaradas locales.

```
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
```

También se pueden guardar en tablas:

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

## Llamando a una función.

Las funciones solo son útiles si podemos llamarlas. Para llamar a una función se usa la siguiente sintaxis:

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

Estamos llamando a la función de `print` . Usando el argumento `"Hello, World"` . Como es obvio, esto imprimirá `Hello, World` al flujo de salida. El valor devuelto es accesible, al igual que cualquier otra variable sería.

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

Las variables también se aceptan en los parámetros de una función.

```
local a = 10
```

```
local b = 60

local c = add(a, b)

print(c)
```

Las funciones que esperan una tabla o una cadena pueden invocarse con un azúcar sintáctico limpio: se pueden omitir los paréntesis que rodean la llamada.

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

### Funciones anonimas

## Creando funciones anonimas

Las funciones anónimas son como las funciones normales de Lua, excepto que no tienen un nombre.

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

Como puede ver, la función no está asignada a ningún nombre como `print` o `add` . Para crear una función anónima, todo lo que tiene que hacer es omitir el nombre. Estas funciones también pueden tomar argumentos.

## Entendiendo el azúcar sintáctico.

Es importante entender que el siguiente código

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

En realidad es sólo una taquigrafía para

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

Sin embargo, la función anterior **no** es anónima, ya que la función se asigna directamente a una variable.

## Las funciones son valores de primera clase.

Esto significa que una función es un valor con los mismos derechos que los valores convencionales como números y cadenas. Las funciones se pueden almacenar en variables, en

tablas, se pueden pasar como argumentos y otras funciones pueden devolverlas.

Para demostrar esto, también crearemos una función "media":

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

Entonces, ahora tenemos dos variables, `half` y `double` , ambas contienen una función como valor. ¿Qué pasaría si quisiéramos crear una función que alimentaría el número 4 en dos funciones dadas y calcularíamos la suma de ambos resultados?

Queremos llamar a esta función como `sumOfTwoFunctions(double, half, 4)` . Esto alimentará la función `double` , la `half` función y el entero `4` en nuestra propia función.

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

La función `sumOfTwoFunctions` anterior muestra cómo se pueden pasar las funciones dentro de los argumentos y se puede acceder a ellos con otro nombre.

## Parámetros por defecto

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

Esa función es una función simple, y funciona bien. Pero, ¿qué pasaría si acabamos de llamar `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 ?
```

Eso no es exactamente genial. Hay dos maneras de arreglar esto:

1. Inmediatamente regresas de la función:

```
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. Establece un parámetro por *defecto* .

Para hacer esto, simplemente usa esta expresión simple

```
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
```

El `name = name or "Jack"` idioma `name = name or "Jack"` funciona porque `or` en los cortocircuitos de Lua. Si el elemento en el lado izquierdo de un `or` es algo distinto de `nil` o `false` , el lado derecho nunca se evalúa. Por otro lado, si se llama a `sayHello` sin parámetro, entonces el `name` será `nil` , por lo que la cadena `"Jack"` se asignará al `name` . (Tenga en cuenta que este idioma, por lo tanto, no funcionará si el `false` booleano es un valor razonable para el parámetro en cuestión).

## Múltiples resultados

Las funciones en Lua pueden devolver múltiples resultados.

Por ejemplo:

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

Al llamar a una función, para guardar estos valores, debe usar la siguiente sintaxis:

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

Lo que resultará en `a = b = c = 5` en este caso. También es posible ignorar los valores devueltos utilizando la variable desechable _ en el lugar deseado en una lista de variables:

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

En este caso, el segundo valor devuelto será ignorado. También es posible ignorar los valores de retorno al no asignarlos a ninguna variable:

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

A la variable `a` se le asignará el primer valor de retorno y los dos restantes se descartarán.

Cuando una función devuelve una cantidad variable de resultados, uno puede almacenarlos todos en una tabla, ejecutando la función dentro de ella:

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

De esta manera, uno puede iterar sobre la tabla de `results` para ver qué función devolvió.

**Nota**

Esto puede ser una sorpresa en algunos casos, por ejemplo:

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

Esto sucede porque `string.gsub` devuelve 2 valores: la cadena dada, con las ocurrencias del patrón reemplazado y el número total de coincidencias que ocurrieron.

Para resolver esto, use una variable intermedia o ponga `()` alrededor de la llamada, así:

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

Esto toma solo el primer resultado de la llamada, e ignora el resto.

## Número variable de argumentos

### Argumentos Variados

## Argumentos con nombre

```
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
```

## Comprobando tipos de argumentos

Algunas funciones solo funcionan en un cierto tipo de argumento:

```
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
```

Esto tiene varias implicaciones:

```
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"
```

ahora podemos llamar a la función con cualquier parámetro que queramos, y no bloqueará el programa.

```
-- 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
```

Entonces, ¿qué pasa si tenemos una función que hace algo con una instancia de una clase específica? Esto es difícil, porque las clases y los objetos suelen ser tablas, por lo que la función `type` devolverá `'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
```

Solución: comparar los 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
```

### Cierres

```
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
```

## ejemplo de uso típico

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

## ejemplo de uso más avanzado

```
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
```

Lea Funciones en línea: https://riptutorial.com/es/lua/topic/1250/funciones

# Capítulo 8: Introducción a la API de Lua C

## Sintaxis

- lua_State * L = lua_open (); // Crear un nuevo estado de máquina virtual; Lua 5.0
- lua_State * L = luaL_newstate (); // Crear un nuevo estado de máquina virtual; Lua 5.1+
- int luaL_dofile (lua_State * L, const char * *filename* ); // Ejecutar un script de lua con el *nombre de archivo* dado usando el lua_State especificado
- void luaL_openlibs (lua_State * L); // Cargar todas las bibliotecas estándar en el lua_State especificado
- void lua_close (lua_State * L); // Cerrar el estado de VM y liberar cualquier recurso dentro
- void lua_call (lua_State * L, int nargs, int nresults); // Llame al luavalue al índice - (nargs + 1)

## Observaciones

Lua también proporciona una API de C adecuada para su máquina virtual. Al contrario de la máquina virtual en sí, la interfaz de la API C está basada en la pila. Por lo tanto, la mayoría de las funciones que se pretenden utilizar con los datos es agregar algunas cosas en la parte superior de la pila virtual o eliminarlas. Además, todas las llamadas a la API deben usarse con cuidado dentro de la pila y sus limitaciones.

En general, cualquier cosa disponible en el lenguaje Lua se puede hacer usando su API C. Además, hay algunas funciones adicionales como el acceso directo al registro interno, el cambio de comportamiento del asignador de memoria estándar o el recolector de basura.

Puede compilar los ejemplos proporcionados de la API de Lua C ejecutando lo siguiente en su terminal:

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

## Examples

### Creando la Máquina Virtual Lua

```
#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;
}
```

## Llamando a las funciones de Lua

```
#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;
 }
```

En el ejemplo anterior estamos haciendo estas cosas:

- creando y configurando Lua VM como se muestra en el primer ejemplo
- obteniendo y presionando una función Lua de la tabla global Lua en la pila virtual
- presionando la cadena `"Hello C API"` como un argumento de entrada en la pila virtual

- instruir a VM para que llame a una función con un argumento que ya está en la pila
- cerrando y limpiando

**NOTA:**

`lua_call()` en cuenta que `lua_call()` la función y sus argumentos de la pila dejando solo el resultado.

Además, sería más seguro usar la llamada protegida Lua - `lua_pcall()` lugar.

## Intérprete Lua integrado con API personalizada y personalización de Lua

Demuestre cómo incrustar un intérprete lua en el código C, exponer una función definida por C al script Lua, evaluar un script Lua, llamar a una función definida por C desde Lua y llamar a una función definida por Lua desde C (el host).

En este ejemplo, queremos que el estado de ánimo se establezca mediante un script Lua. Aquí está 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
```

Aviso, `mood()` no se llama en el script. Se acaba de definir para que la aplicación host llame. También `hostgetversion()` cuenta que el script llama a una función llamada `hostgetversion()` que no está definida en el script.

A continuación, definimos una aplicación host que usa 'mood.lua'. Aquí está el '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;
}
```

Y aquí está la salida:

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

¡Incluso después de que hayamos compilado 'hostlua.c', todavía podemos modificar 'mood.lua' para cambiar la salida de nuestro programa!

### Manipulación de la mesa

Para acceder o modificar un índice en una tabla, debe colocar la tabla en la pila de alguna manera.
Supongamos, para estos ejemplos, que su tabla es una variable global llamada tbl.

## Obteniendo el contenido en un índice particular:

```
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
}
```

Como hemos visto, todo lo que tiene que hacer es empujar la tabla en la pila, empujar el índice y llamar a lua_gettable. El argumento -2 significa que la tabla es el segundo elemento de la parte superior de la pila.
lua_gettable activa metamethods. Si no desea activar metamethods, use lua_rawget en su lugar. Utiliza los mismos argumentos.

## Configuración del contenido en un índice particular:

```
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;
}
```

El mismo ejercicio que el contenido. Tiene que empujar la pila, empujar el índice y luego empujar el valor en la pila. después de eso, llamas a lua_settable. El argumento -3 es la posición de la tabla en la pila. Para evitar la activación de metamethods, use lua_rawset en lugar de lua_settable. Utiliza los mismos argumentos.

## Transfiriendo el contenido de una tabla a otra:

```
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;
}
```

Ahora estamos reuniendo todo lo que aprendimos aquí. Pongo el contenido de la pila en los comentarios para que no te pierdas.

Colocamos ambas tablas en la pila, colocamos el índice de la tabla 1 en la pila y obtenemos el valor en `tbl1.index1` . Tenga en cuenta el argumento `-3` en gettable. Estoy mirando la primera mesa (tercera desde arriba) y no la segunda. Luego presionamos el índice de la segunda tabla, copiamos el `tbl1.index1` en la parte superior de la pila y luego llamamos a `lua_settable` , en el

cuarto elemento desde la parte superior.

Para limpiar la casa, purgué el elemento superior, por lo que solo quedan las dos mesas en la pila.

Lea Introducción a la API de Lua C en línea: https://riptutorial.com/es/lua/topic/671/introduccion-a-la-api-de-lua-c

# Capítulo 9: Iteradores

## Examples

### Genérico para bucle

Los iteradores utilizan una forma del bucle `for` conocido como el bucle for genérico .

La forma genérica del bucle `for` utiliza tres parámetros:

1. Una **función de iterador** que se llama cuando se necesita el siguiente valor. Recibe tanto el estado invariante como la variable de control como parámetros. Devolviendo `nil` señales de terminación.
2. El **estado invariante** es un valor que no cambia durante la iteración. Normalmente es el tema del iterador, como una tabla, cadena o datos de usuario.
3. La **variable de control** representa un valor inicial para la iteración.

Podemos escribir un bucle `for` para iterar todos los pares clave-valor en una tabla usando la siguiente función.

```
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
```

### Iteradores estándar

La biblioteca estándar de Lua proporciona dos funciones de iterador que pueden usarse con un bucle `for` para atravesar pares clave-valor dentro de las tablas.

Para iterar sobre una tabla de secuencias podemos usar la función de biblioteca 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
```

Para iterar sobre todas las claves y valores en cualquier tabla podemos usar los pares de funciones de la biblioteca.

```
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
```

### Iteradores sin estado

Ambos pares y ipairs representan iteratas sin estado. Un iterador sin estado usa solo el genérico para la variable de control del bucle y el estado invariante para calcular el valor de iteración.

## Pares iterador

Podemos implementar el iterador de `pairs` sin estado usando la `next` función.

```
-- 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

Podemos implementar el iterador `ipairs` sin `ipairs` en dos funciones separadas.

```
-- 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
```

## Iterador de caracteres

Podemos crear nuevos iteradores sin estado mediante el cumplimiento del contrato del bucle genérico `for` .

```
-- 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
```

# Iterador de números primos

Este es un ejemplo más simple de un iterador sin estado.

```
-- 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
```

### Iteradores de estado

Los iteradores de estado llevan información adicional sobre el estado actual del iterador.

# Usando tablas

El estado de adición se puede empaquetar en el genérico para el estado invariante del bucle .

```
  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
```

## Utilizando cierres

Estado adicional puede ser envuelto dentro de una función de cierre. Dado que el estado está completamente contenido en el alcance del cierre, el estado invariable y la variable de control no son necesarios.

```
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
```

## Utilizando coroutines

Se puede contener un estado adicional dentro de una rutina, de nuevo, el estado invariable y la variable de control no son necesarios.

```
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
```

Lea Iteradores en línea: https://riptutorial.com/es/lua/topic/4165/iteradores

# Capítulo 10: La coincidencia de patrones

## Sintaxis

- string.find (str, pattern [, init [, plain]]) - Devuelve el índice inicial y final de coincidencia en str
- string.match (str, patrón [, índice]) - Hace coincidir un patrón una vez (comenzando en el índice)
- string.gmatch (str, patrón): devuelve una función que recorre todas las coincidencias en str
- string.gsub (str, pattern, repl [, n]) - Reemplaza subcadenas (hasta un máximo de n veces)
- `.` representa a todos los personajes
- `%a` representa todas las letras
- `%l` representa todas las letras minúsculas
- `%u` representa todas las letras mayúsculas
- `%d` representa todos los dígitos
- `%x` representa todos los dígitos hexadecimales
- `%s` representa todos los caracteres de espacio en blanco
- `%p` representa todos los caracteres de puntuación
- `%g` representa todos los caracteres *imprimibles* excepto el espacio
- `%c` representa todos los caracteres de control
- `[set]` representa la clase que es la unión de todos los caracteres en set.
- `[^set]` representa el complemento de set
- `*` coincidencia codiciosa 0 o más ocurrencias de la clase de personaje anterior
- `+` coincidencia codiciosa 1 o más ocurrencias de la clase de personaje anterior
- `-` coincidencia perezosa 0 o más apariciones de la clase de carácter anterior
- `?` coincide exactamente con 0 o 1 ocurrencia de la clase de carácter anterior

## Observaciones

A lo largo de algunos ejemplos, se usa la notación `(<string literal>):function <string literal>`, que es equivalente a `string.function(<string literal>, <string literal>)` porque todas las

cadenas tienen un metatable con el conjunto de campos `__index` a la tabla de `string` .

# Examples

## Lua patrón a juego

En lugar de usar expresiones regulares, la biblioteca de cadenas Lua tiene un conjunto especial de caracteres que se usan en las coincidencias de sintaxis. Ambos pueden ser muy similares, pero la comparación de patrones de Lua es más limitada y tiene una sintaxis diferente. Por ejemplo, la secuencia de caracteres `%a` coincide con cualquier letra, mientras que su versión en mayúsculas representa *todos los caracteres que no son letras* , todas las clases de caracteres (una secuencia de caracteres que, como un patrón, puede coincidir con un conjunto de elementos) se enumeran a continuación.

| Clase de personaje | Sección correspondiente |
|---|---|
| %una | letras (AZ, az) |
| %do | caracteres de control (\ n, \ t, \ r, ...) |
| %re | dígitos (0-9) |
| % l | letra minúscula (az) |
| %pag | caracteres de puntuación (!,?, &, ...) |
| % s | personajes espaciales |
| % u | letras mayúsculas |
| % w | Caracteres alfanuméricos (AZ, az, 0-9) |
| %X | dígitos hexadecimales (\ 3, \ 4, ...) |
| % z | el personaje con representación 0 |
| . | Coincide con cualquier personaje |

Como se mencionó anteriormente, cualquier versión en mayúsculas de esas clases representa el complemento de la clase. Por ejemplo, `%D` coincidirá con cualquier secuencia de caracteres sin dígitos:

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

Además de las clases de caracteres, algunos personajes tienen funciones especiales como patrones:

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

El carácter `%` representa un carácter de escape, haciendo `%?` coincide con una interrogación y `%%` coincide con el símbolo de porcentaje. Puede usar el carácter `%` con cualquier otro carácter no alfanumérico, por lo tanto, si necesita escapar, por ejemplo, una cita, debe usar `\\` antes, que escapa cualquier carácter de una cadena lua.

Un conjunto de caracteres, representado entre corchetes ( `[]` ), le permite crear una clase de carácter especial, combinando diferentes clases y caracteres individuales:

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

Puede obtener el complemento del conjunto de caracteres comenzando con `^` :

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

En este ejemplo, `string.match` encontrará la primera aparición que no sea **b** , **a** o **r** .

Los patrones pueden ser más útiles con la ayuda de los modificadores de repetición / opcionales, los patrones en lua ofrecen estos cuatro caracteres:

| Personaje | Modificador |
|---|---|
| + | Una o mas repeticiones |
| * | Cero o más repeticiones. |
| - | También cero o más repeticiones. |
| ? | Opcional (cero o una ocurrencia) |

El carácter `+` representa uno o más caracteres coincidentes en la secuencia y siempre devolverá la secuencia coincidente más larga:

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

Como puede ver, `*` es similar a `+` , pero acepta cero apariciones de caracteres y se usa comúnmente para hacer coincidir espacios opcionales entre diferentes patrones.

El carácter `-` también es similar a `*` , pero en lugar de devolver la secuencia coincidente más larga, coincide con la más corta.

El modificador `?` coincide con un carácter opcional, lo que le permite hacer coincidir, por ejemplo, un dígito negativo:

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

El motor de coincidencia de patrones de Lua proporciona algunos elementos adicionales de coincidencia de patrones:

| Elemento del personaje | Descripción |
|---|---|
| `%n` | para n entre 1 y 9 coincide con una subcadena igual a la cadena n-th capturada |
| `%bxy` | hace coincidir la subcadena entre dos caracteres distintos (par equilibrado de `x` e `y` ) |
| `%f[set]` | patrón de frontera: coincide con una cadena vacía en cualquier posición, de modo que el siguiente carácter<br>pertenece al conjunto y el carácter anterior no pertenece al conjunto |

### string.find (Introducción)

# La funcion de `find`

Primero echemos un vistazo a la función `string.find` en general:

La función `string.find (s, substr [, init [, plain]])` devuelve el índice de inicio y fin de una subcadena si se encuentra, y nil de lo contrario, comenzando en el índice `init` si se proporciona (el valor predeterminado es 1).

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

# Introduciendo patrones

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

Todos **excepto** los siguientes caracteres se representan a sí mismos `^$()%.[]*+-?)` . Cualquiera de estos caracteres puede ser representado por un `%` siguiendo al propio personaje.

```
("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

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

("hellllllo"):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
```

### La función `gmatch`

# Cómo funciona

La función `string.gmatch` tomará una cadena de entrada y un patrón. Este patrón describe en qué recuperar realmente. Esta función devolverá una función que en realidad es un iterador. El resultado de este iterador coincidirá con el patrón.

```
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
```

## Introduciendo capturas:

Esto es muy similar a la función normal, sin embargo, solo devolverá las capturas en lugar de la coincidencia completa.

```
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
```

### La función gsub

no confunda con la función string.sub, que devuelve una subcadena!

# Cómo funciona

## argumento de cadena

```
("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
```

## argumento de la función

```
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
```

## argumento de la mesa

```
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
```

Lea La coincidencia de patrones en línea: https://riptutorial.com/es/lua/topic/5829/la-coincidencia-de-patrones

# Capítulo 11: Manejo de errores

## Examples

### Usando pcall

`pcall` significa "llamada protegida". Se utiliza para agregar el manejo de errores a las funciones. `pcall` funciona de forma similar a `try-catch` en otros idiomas. La ventaja de `pcall` es que la ejecución completa de la secuencia de comandos no se interrumpe si se producen errores en las funciones llamadas con `pcall` . Si se produce un error dentro de una función llamada con `pcall` produce un error y el resto del código continúa su ejecución.

---

**Sintaxis:**

```
pcall( f , arg1,···)
```

---

**Valores de retorno:**

Devuelve dos valores

1. estado (booleano)

- Devuelve **true** si la función se ejecutó sin errores.
- Devuelve **falso** si ocurrió un error dentro de la función.

2. devuelva el valor de la función **o el** mensaje de error si se produjo un error dentro del bloque de funciones.

---

pcall se puede usar para varios casos, sin embargo, uno común es detectar errores de la función que se le ha asignado. Por ejemplo, digamos que tenemos esta función:

```
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
```

Cuando la función dada produce un error en la ejecución 3, el mensaje de error será claro para el usuario de que no proviene de su función, sino de la función que fue asignada a nuestra función. Además, con esto en mente, se puede hacer un BSoD de lujo notificando al usuario. Sin embargo, eso depende de la aplicación que implemente esta función, ya que lo más probable es que una API no lo haga.

**Ejemplo A** - *Ejecución sin 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
```

**Ejemplo B** - *Ejecución con 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"
```

**Ejemplo** - *Ejecución de código impecable.*

```
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"
```

## Manejo de errores en lua.

Suponiendo que tenemos la siguiente función:

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

Vamos a mejorarlo un poco

```
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
```

Si no queremos que una función bloquee un programa, incluso en caso de error, en lua es estándar hacer lo siguiente:

```
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
```

Ahora tenemos una función que se comporta así, podemos hacer cosas como esta:

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

Y si queremos que el programa se bloquee si algo sale mal, todavía podemos hacer esto:

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

Afortunadamente, ni siquiera tenemos que escribir todo eso cada vez; lua tiene una función que hace exactamente esto

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

Lea Manejo de errores en línea: https://riptutorial.com/es/lua/topic/4561/manejo-de-errores

# Capítulo 12: Mesas

## Sintaxis

- ipairs (numeric_table) - tabla Lua con índices numéricos iterador
- pares (input_table) - iterador genérico de tabla Lua
- key, value = next (input_table, input_key) - Selector de valores de tabla Lua
- table.insert (input_table, [posición], valor): inserte el valor especificado en la tabla de entrada
- removed_value = table.remove (input_table, [position]): resalta el último o elimina el valor especificado por position

## Observaciones

Las tablas son la única estructura de datos incorporada disponible en Lua. Esto puede ser elegante o confuso, dependiendo de cómo lo mires.

Una tabla Lua es una colección de pares clave-valor donde las claves son únicas y ni la clave ni el valor son `nil` . Como tal, una tabla Lua puede parecerse a un diccionario, un hashmap o una matriz asociativa de otros idiomas. Muchos patrones estructurales se pueden construir con tablas: pilas, colas, conjuntos, listas, gráficos, etc. Finalmente, las tablas se pueden usar para construir *clases* en Lua y para crear un sistema de *módulos* .

Lua no impone ninguna regla particular sobre cómo se usan las tablas. Los elementos contenidos en una tabla pueden ser una mezcla de tipos de Lua. Así, por ejemplo, una tabla podría contener cadenas, funciones, valores booleanos, números *e incluso otras tablas* como valores o claves.

Se dice que una tabla Lua con teclas enteras positivas consecutivas que comienzan con 1 tiene una secuencia. Los pares clave-valor con claves enteras positivas son los elementos de la secuencia. Otros lenguajes llaman a esto una matriz basada en 1. Ciertas operaciones y funciones estándar solo funcionan en la secuencia de una tabla y algunas tienen un comportamiento no determinista cuando se aplican a una tabla sin una secuencia.

Establecer un valor en una tabla en `nil` elimina de la tabla. Los iteradores ya no verían la clave relacionada. Al codificar una tabla con una secuencia, es importante evitar romper la secuencia; Solo elimine el último elemento o use una función, como la `table.remove` estándar.eliminar, que desplaza los elementos hacia abajo para cerrar la brecha.

## Examples

### Creando tablas

Crear una tabla vacía es tan simple como esto:

```
local empty_table = {}
```

También puede crear una tabla en forma de una matriz simple:

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

Tenga en cuenta que, de forma predeterminada, la indexación de tablas comienza en 1.

También es posible crear una tabla con elementos asociativos:

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

El uso anterior es sintaxis de azúcar para lo que está debajo. Las claves en este caso son del tipo cadena. La sintaxis anterior se agregó para que las tablas aparezcan como registros. Esta sintaxis de estilo de registro es paralela a la sintaxis para indexar tablas con claves de cadena, como se ve en el tutorial de "uso básico".

Como se explica en la sección de comentarios, la sintaxis de estilo de grabación no funciona para todas las claves posibles. Además, una clave puede ser cualquier valor de cualquier tipo, y los ejemplos anteriores solo cubrían cadenas y números secuenciales. En otros casos necesitarás usar la sintaxis explícita:

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

## Mesas iterantes

La biblioteca estándar de Lua proporciona una función de `pairs` que itera sobre las claves y los valores de una tabla. Cuando se itera con `pairs` no hay un orden específico para el recorrido, *incluso si las claves de la tabla son numéricas* .

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

Para tablas que usan **teclas numéricas** , Lua proporciona una función `ipairs` . La función `ipairs` siempre se repetirá desde la `table[1]` , la `table[2]` , etc. hasta que se encuentre el primer valor `nil` .

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

Tenga en cuenta que la iteración que usa `ipairs()` no funcionará como desearía en algunas ocasiones:

- `input_table` tiene "agujeros" en él. (Para obtener más información, consulte la sección "Cómo evitar los huecos en las tablas utilizadas como matrices"). Por ejemplo:

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

- Las teclas no eran todas numéricas. Por ejemplo:

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

Por supuesto, lo siguiente también funciona para una tabla que es una secuencia adecuada:

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

Iterar una tabla numérica en orden inverso es fácil:

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

Una forma final de iterar sobre tablas es usar el `next` selector en un bucle genérico `for` . Al igual que los `pairs` no hay un orden especificado para el recorrido. (El método de `pairs` usa `next` internamente. Por lo tanto, usar `next` es esencialmente una versión más manual de `pairs` . Consulte los `pairs` en el manual de referencia de Lua y `next` en el manual de referencia de Lua para obtener más detalles).

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

## Uso básico

El uso básico de la tabla incluye acceder y asignar elementos de la tabla, agregar contenido de la tabla y eliminar el contenido de la tabla. Estos ejemplos asumen que sabes cómo crear tablas.

### Elementos de acceso

Dada la siguiente tabla,

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

Uno puede indexar la parte secuencial de la tabla utilizando la sintaxis del índice, siendo el argumento de la sintaxis del índice la clave del par clave-valor deseado. Como se explica en el tutorial de creación, la mayor parte de la sintaxis de la declaración es azúcar sintáctica para declarar pares clave-valor. Los elementos incluidos secuencialmente, como los primeros cinco valores en `example_table` , utilizan valores enteros crecientes como claves; la sintaxis de registro utiliza el nombre del campo como una cadena.

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

Para las claves de cadena, hay una sintaxis de azúcar para establecer un paralelo con la sintaxis de estilo de registro para las claves de cadena en la creación de tablas. Las siguientes dos líneas son equivalentes.

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

Puede acceder a las tablas utilizando las claves que no ha usado antes, eso no es un error como en otros idiomas. Al hacerlo, devuelve el valor predeterminado `nil` .

**Asignando elementos**

Puede modificar los elementos de tabla existentes asignándolos a una tabla mediante la sintaxis de índice. Además, la sintaxis de indexación de estilo de registro también está disponible para establecer valores

```
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
```

También puede agregar nuevos elementos a una tabla existente usando la asignación.

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

*Observación especial:* algunas cadenas no son compatibles con la sintaxis de registro. Vea la sección de comentarios para más detalles.

**Removiendo elementos**

Como se indicó anteriormente, el valor predeterminado para una clave sin valor asignado es `nil` . Eliminar un elemento de una tabla es tan simple como restablecer el valor de una clave al valor predeterminado.

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

Los elementos ahora son indistinguibles de un elemento no establecido.

### Longitud de la mesa

Las tablas son simplemente matrices asociativas (ver comentarios), pero cuando se usan claves enteras contiguas comenzando desde 1, se dice que la tabla tiene una *secuencia* .

La búsqueda de la longitud de la parte de la secuencia de una tabla se realiza utilizando `#` :

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

Puede usar la operación de longitud para agregar fácilmente elementos a una tabla de secuencia.

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

En el ejemplo anterior, el valor anterior de `#example_table` es `8` , al agregar `1` obtiene la siguiente clave de entero válida en la secuencia, `9` , así que ... `example_table[9] = 'a'` . Esto funciona para cualquier longitud de tabla.

*Observación especial: el* uso de teclas de enteros que no son contiguas y que comienzan desde 1 rompe la secuencia de la tabla en una *tabla dispersa* . El resultado de la operación de longitud no está definido en ese caso. Vea la sección de comentarios.

### Uso de las funciones de la biblioteca de tablas para agregar / eliminar elementos

Otra forma de agregar elementos a una tabla es la función `table.insert()` . La función de inserción solo funciona en tablas de secuencia. Hay dos formas de llamar a la función. El primer ejemplo muestra el primer uso, donde se especifica el índice para insertar el elemento (el segundo argumento). Esto empuja todos los elementos del índice dado a `#table` una posición. El segundo ejemplo muestra el otro uso de `table.insert()` , donde el índice no se especifica y el valor dado se agrega al final de la tabla (índice `#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"}
```

En paralelo a `table.insert()` para eliminar elementos está `table.remove()` . De manera similar, tiene dos semánticas de llamada: una para eliminar elementos en una posición dada y otra para eliminar desde el final de la secuencia. Al eliminar desde la mitad de una secuencia, todos los elementos siguientes se desplazan hacia abajo un índice.

```
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"
```

Estas dos funciones mutan la tabla dada. Como podría indicar el segundo método para llamar a

`table.insert()` y `table.remove()` proporciona la semántica de la pila a las tablas. Aprovechando eso, puede escribir código como el siguiente ejemplo.

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

Implementa el Shuffle de Fisher-Yates, quizás de manera ineficiente. Utiliza `table.insert()` para agregar el elemento extraído aleatoriamente al final de la misma tabla, y `table.remove()` para extraer aleatoriamente un elemento de la parte restante de la tabla sin modificar.

### Evitar los huecos en las tablas utilizadas como matrices.

## Definiendo nuestros términos

Por *matriz* aquí nos referimos a una tabla Lua utilizada como secuencia. Por ejemplo:

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

Estamos utilizando esta tabla como una secuencia: un grupo de elementos codificados por enteros. Muchos idiomas llaman a esto una matriz, y nosotros también. Pero estrictamente hablando, no hay tal cosa como una matriz en Lua. Solo hay tablas, algunas de las cuales son de tipo matriz, algunas de las cuales son de tipo hash (o de tipo diccionario, si lo prefiere), y algunas de las cuales están mezcladas.

Un punto importante sobre nuestra variedad de `pets` es que no tiene huecos. El primer artículo, `pets[1]` , es la cadena "perros", el segundo artículo, `pets[2]` , es la cadena "gatos", y el último artículo, `pets[3]` , es "pájaros". La biblioteca estándar de Lua y la mayoría de los módulos escritos para Lua asumen que 1 es el primer índice de secuencias. Por lo tanto, una matriz sin `1..n` tiene elementos de `1..n` sin perder ningún número en la secuencia. (En el caso límite, `n = 1` , y la matriz solo contiene un elemento).

Lua proporciona la función `ipairs` para iterar sobre dichas tablas.

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

Esto imprimiría "El artículo en la posición 1 es perros", "El artículo en la posición 2 es gatos", "El artículo en la posición 3 es pájaros".

Pero, ¿qué pasa si hacemos lo siguiente?

```
local pets = {"dogs", "cats", "birds"}
pets[12] = "goldfish"
```

```
for idx, pet in ipairs(pets) do
  print("Item at position " .. idx .. " is " .. pet .. ".")
end
```

Una matriz como este segundo ejemplo es una matriz dispersa. Hay lagunas en la secuencia. Esta matriz se ve así:

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

Los valores nulos no ocupan memoria adicional; internamente lua solo guarda los valores `[1] = "dogs"` , `[2] = "cats"` , `[3] = "birtds"` y `[12] = "goldfish"`

Para responder a la pregunta inmediata, `ipairs` se detendrá después de las aves; "goldfish" en `pets[12]` nunca se alcanzará a menos que ajustemos nuestro código. Esto se debe a que `ipairs` itera desde `1..n-1` donde `n` es la posición del primer `nil` encontrado. Lua define la `table[length-of-table + 1]` como `nil` . Entonces, en una secuencia apropiada, la iteración se detiene cuando Lua intenta obtener, por ejemplo, el cuarto elemento en una matriz de tres elementos.

# ¿Cuando?

Los dos lugares más comunes para que surjan problemas con matrices dispersas son (i) cuando se trata de determinar la longitud de la matriz y (ii) cuando se intenta iterar sobre la matriz. En particular:

- Cuando se usa el operador de longitud `#` ya que el operador de longitud deja de contar en el primer `nil` encontrado.
- Cuando se utiliza la función `ipairs()` ya que como se mencionó anteriormente, deja de iterar en el primer `nil` encontrado.
- Cuando se utiliza la función `table.unpack()` ya que este método detiene el desempaquetado en el primer `nil` encontrado.
- Cuando se utilizan otras funciones que (directa o indirectamente) acceden a cualquiera de las anteriores.

Para evitar este problema, es importante escribir su código de modo que si espera que una tabla sea una matriz, no introduzca espacios. Las brechas se pueden introducir de varias maneras:

- Si agrega algo a una matriz en la posición incorrecta.
- Si inserta un valor `nil` en una matriz.
- Si eliminas valores de una matriz.

Podrías pensar: "Pero nunca haría ninguna de esas cosas". Bueno, no intencionalmente, pero aquí hay un ejemplo concreto de cómo las cosas pueden salir mal. Imagina que deseas escribir un método de filtro para Lua como Ruby `select` y Perl's `grep` . El método aceptará una función de prueba y una matriz. Se itera sobre la matriz, llamando al método de prueba en cada elemento a su vez. Si el elemento pasa, entonces ese elemento se agrega a una matriz de resultados que se devuelve al final del método. La siguiente es una implementación con errores:

```
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
```

El problema es que cuando la función devuelve `false` , omitimos un número en la secuencia. Imagine el `filter(isodd, {1,2,3,4,5,6,7,8,9,10})` llamadas `filter(isodd, {1,2,3,4,5,6,7,8,9,10})` : habrá espacios en la tabla devuelta cada vez que haya un número par en la matriz que se pasa al `filter` .

Aquí hay una implementación fija:

```
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
```

# Consejos

1. Use las funciones estándar: `table.insert(<table>, <value>)` siempre se agrega al final de la matriz. `table[#table + 1] = value` es una mano corta para esto. `table.remove(<table>, <index>)` moverá todos los valores siguientes para llenar el espacio (lo que también puede hacerlo lento).
2. Compruebe si hay valores `nil` **antes de** insertar, evitando cosas como `table.pack(function_call())` , que podría deslizar valores `nil` en nuestra tabla.
3. Verifique los valores `nil` **después de la** inserción y, si es necesario, llene el espacio desplazando todos los valores consecutivos.
4. Si es posible, utilice valores de marcador de posición. Por ejemplo, cambie `nil` por `0` o algún otro valor de marcador de posición.
5. Si es inevitable dejar brechas, esto debe documentarse (comentar).
6. Escriba un `__len()` y use el operador `#` .

Ejemplo para 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'
```

Otra alternativa es usar la función `pairs()` y filtrar los índices no enteros:

```
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
```

Lea Mesas en línea: https://riptutorial.com/es/lua/topic/676/mesas

# Capítulo 13: Metatables

## Sintaxis

- [[local] *mt* =] getmetatable ( *t* ) -> recupera metatable asociado para ' *t* '
- [[local] *t* =] setmetatable ( *t* , *mt* ) -> establece la metatable para ' *t* ' en ' *mt* ' y devuelve ' *t* '

## Parámetros

| Parámetro | Detalles |
|---|---|
| t | Variable referente a una tabla lua; También puede ser una tabla literal. |
| monte | Tabla para usar como metatable; puede tener cero o más campos metamétodo establecidos. |

## Observaciones

Hay algunos metamétodos no mencionados aquí. Para ver la lista completa y su uso, consulte la entrada correspondiente en el manual de lua .

## Examples

### Creación y uso de metatables.

Un metatable define un conjunto de operaciones que alteran el comportamiento de un objeto lua. Un metatable es solo una tabla ordinaria, que se usa de una manera especial.

```
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 }'
```

## Usando tablas como metamétodos

Algunos metamétodos no tienen que ser funciones. El ejemplo más importante para esto es el `__index` metamethod. También puede ser una tabla, que luego se utiliza como búsqueda. Esto es bastante usado en la creación de clases en lua. Aquí, una tabla (a menudo, la propia metatable) se usa para mantener todas las operaciones (métodos) de la clase:

```
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()
```

## Recolector de basura - el metamétodo __gc

5.2

Los objetos en lua son recolectados basura. A veces, necesita liberar algún recurso, desea imprimir un mensaje o hacer otra cosa cuando un objeto es destruido (recolectado). Para esto, puedes usar el `__gc` __gc, que se llama con el objeto como argumento cuando el objeto es destruido. Podrías ver este método como una especie de destructor.

Este ejemplo muestra el `__gc` __gc en acción. Cuando la tabla interna asignada a `t` obtiene la basura recolectada, imprime un mensaje antes de ser recolectada. Del mismo modo para la tabla exterior al llegar al final del 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
```

## Más metamétodos

Hay muchos más metamétodos, algunos de ellos son aritméticos (por ejemplo, suma, resta, multiplicación), hay operaciones a nivel de bits (y, o, xor, cambio), comparación (<,>) y también operaciones de tipo básico como == y # (Igualdad y longitud). Permite construir una clase que admita muchas de estas operaciones: una llamada a la aritmética racional. Si bien esto es muy básico, muestra la 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
```

## Hacer mesas invocables.

Hay un método llamado `__call` , que define el comportamiento del objeto cuando se usa como una función, por ejemplo, `object()` . Esto se puede utilizar para crear objetos de función:

```
-- 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
```

El metamétodo se llama con el objeto correspondiente, todos los argumentos restantes se pasan a la función después de eso:

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

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

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

### Indexación de tablas

Quizás el uso más importante de los metatables es la posibilidad de cambiar la indexación de tablas. Para esto, hay dos acciones a considerar: *leer* el contenido y *escribir* el contenido de la tabla. Tenga en cuenta que ambas acciones solo se activan si la clave correspondiente no está presente en la tabla.

## Leyendo

```
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
```

Esto podría usarse para generar un error al leer una clave que no existe:

```
-- 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
```

# Escritura

```
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
```

Ahora puede preguntarse cómo se escribe el valor real en la tabla. En este caso, no lo es. El problema aquí es que los metamétodos pueden desencadenar metamétodos, lo que daría lugar a un bucle infinitivo, o más precisamente, a un desbordamiento de pila. Entonces, ¿cómo podemos resolver esto? La solución para esto se llama *acceso de tabla sin formato* .

## Acceso a la tabla sin procesar

A veces, no desea activar metamétodos, sino escribir o leer exactamente la clave dada, sin algunas funciones inteligentes envueltas alrededor del acceso. Para esto, lua le proporciona métodos de acceso a la tabla en bruto:

```
-- 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])
```

Con esto, ahora podemos reescribir el antiguo método `__newindex` para escribir el valor en la tabla:

```
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
```

## Simulando 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'"
```

Los métodos de instancia pueden escribirse pasando el objeto como primer argumento.

```
-- 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"
```

Hay algo de azúcar sintáctica para esto.

```
function Class:saybye(phrase)
    print("I am " .. self .. "\n" .. phrase)
end
```

```
object:saybye("c ya") --> will print "I am <table ID>
                      -->             c ya"
```

También podemos agregar campos predeterminados a una clase.

```
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
```

Lea Metatables en línea: https://riptutorial.com/es/lua/topic/2444/metatables

# Capítulo 14: Orientación a objetos

## Introducción

Lua en sí no ofrece ningún sistema de clase. Sin embargo, es posible implementar clases y objetos como tablas con solo algunos trucos.

## Sintaxis

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

## Examples

### Orientación a objetos simples

Aquí hay un ejemplo básico de cómo hacer un sistema de clases muy simple.

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

Para agregar variables y / o métodos, simplemente agréguelos a la clase. Ambos pueden ser anulados para cada instancia.

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

Y para crear una instancia de la clase:

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

o

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

Y para usarlo:

```
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}
```

## Cambio de metamétodos de un objeto.

Teniendo

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

Suponiendo que queremos cambiar el comportamiento de un solo `object = Class.new()` instancia `object = Class.new()` usando un metatable,

Hay algunos errores para evitar:

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

Esto intercambia el antiguo metatable con el nuevo, rompiendo así la herencia de clase.

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

Tenga en cuenta que los "valores" de la tabla son solo una referencia; de hecho, solo hay una tabla real para todas las instancias de un objeto, a menos que el constructor se defina como en [1] , por lo que al modificar el comportamiento de *todas las* instancias de la clase.

---

Una forma correcta de hacer esto:

Sin cambiar la clase:

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

¿Como funciona esto? - Creamos un nuevo metatable como en el error # 1, pero en lugar de dejarlo vacío, creamos una copia en papel del metatable original. Se podría decir que el nuevo metatable "hereda" del original como si fuera una instancia de clase. Ahora podemos anular los valores del metatable original sin modificarlos.

Cambiando la clase:

1º (recomendado):

```
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
```

2º (menos recomendado): ver [1]

---

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

Lea Orientación a objetos en línea: https://riptutorial.com/es/lua/topic/8908/orientacion-a-objetos

# Capítulo 15: PICO-8

## Introducción

El PICO-8 es una consola de fantasía programada en Lua embebido. Ya tiene buena documentación . Utilice este tema para demostrar características no documentadas o insuficientemente documentadas.

## Examples

### Bucle de juego

Es completamente posible usar PICO-8 como una cáscara interactiva , pero probablemente quieras aprovechar el bucle del juego. Para hacer eso, debes crear al menos una de estas funciones de devolución de llamada:

- `_update()`
- `_update60()` (después de v0.1.8 )
- `_draw()`

Un "juego" mínimo podría simplemente dibujar algo en la pantalla:

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

Si define `_update60()` , el bucle de juego intenta ejecutarse a 60 fps e ignora `update()` (que se ejecuta a 30fps). Se llama a cualquiera de las funciones de actualización antes de `_draw()` . Si el sistema detecta cuadros perdidos, omitirá la función de dibujo cada otro cuadro, por lo que es mejor mantener la lógica del juego y la entrada del jugador en la función de actualización:

```
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
```

La función `_init()` es, estrictamente hablando, opcional, ya que los comandos fuera de cualquier función se ejecutan al inicio. Pero es una forma práctica de restablecer el juego a las condiciones iniciales sin reiniciar el cartucho:

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

## Entrada del mouse

Aunque no es oficialmente compatible, puedes usar la entrada del mouse en tus juegos:

```
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
```

## Modos de juego

Si desea una pantalla de título o una pantalla de final de juego, considere configurar un mecanismo de cambio de modo:

```
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
```

Lea PICO-8 en línea: https://riptutorial.com/es/lua/topic/8715/pico-8

# Capítulo 16: Recolector de basura y mesas débiles.

## Sintaxis

1. collectgarbage (gcrule [, gcdata]) - recolecta basura usando gcrule
2. setmetatable (tab, {__mode = weakmode}) - establece el modo débil de tabulación en modo débil

## Parámetros

| parámetro | detalles |
|---|---|
| gcrule & gcdata | Acción a gc (recolector de basura): `"stop"` (dejar de recolectar), `"restart"` (comenzar a recolectar nuevamente), `"collect"` o `nil` (recolectar toda la basura), `"step"` (hacer un paso de recolección), `"count"` ( devuelva el recuento de la memoria utilizada en KBs), `"setpause"` y los datos son números del `0` % al `100` % (establecer el parámetro de pausa de gc), `"setstepmul"` y los datos son números del `0` % al `100` (establecer `"stepmul"` para gc) . |
| modo débil | Tipo de tabla débil: `"k"` (solo claves débiles), `"v"` (solo valores débiles), `"vk"` (claves débiles y valores) |

## Examples

### Mesas débiles

```
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
```

Lea Recolector de basura y mesas débiles. en línea: https://riptutorial.com/es/lua/topic/5769/recolector-de-basura-y-mesas-debiles-

# Creditos

| S. No | Capítulos | Contributors |
|---|---|---|
| 1 | Empezando con Lua | 1971chevycamaro, Allan Burleson, Community, DarkWiiPlayer, Darryl L Johnson, elektron, greatwolf, Guilherme Salazar, hjpotter92, hugomg, Kamiccolo, lhf, Nikola Geneshki, SoniEx2, Telemachus |
| 2 | Argumentos Variados | greatwolf, Kamiccolo, ktb, RamenChef, SoniEx2 |
| 3 | Booleanos en Lua | DarkWiiPlayer, engineercoding, greatwolf, Kamiccolo, Katenkyo , Samuel McKay, Telemachus |
| 4 | Conjuntos | Egor Skriptunoff, Jon Ericson, ryanpattison |
| 5 | Coroutines | 010110110101, Bjornir, Eshkation, Kamiccolo, ktb, SoniEx2 |
| 6 | Escribir y utilizar módulos. | SoniEx2, Telemachus |
| 7 | Funciones | Art C, Basilio German, DarkWiiPlayer, Firas Moalla, greatwolf, Guilherme Salazar, Jon Ericson, Katenkyo, ktb, MBorsch, Necktrox, qaisjp, RBerteig, Romário, SoniEx2, Telemachus, Unheilig, WolfgangTS |
| 8 | Introducción a la API de Lua C | greatwolf, Jeremy Thien, Kamiccolo, Luiz Menezes, RBerteig, tversteeg |
| 9 | Iteradores | Adam, Egor Skriptunoff, greatwolf |
| 10 | La coincidencia de patrones | DarkWiiPlayer, engineercoding, Eshkation, greatwolf, Kamiccolo , Stephen Leppik |
| 11 | Manejo de errores | Black, DarkWiiPlayer, engineercoding, greatwolf |
| 12 | Mesas | DarkWiiPlayer, greatwolf, Hastumer, Kamiccolo, ktb, mjanicek, SoniEx2, Telemachus, Tom Blodget |
| 13 | Metatables | DarkWiiPlayer, greatwolf, Kamiccolo, pschulz, Telemachus |
| 14 | Orientación a objetos | DarkWiiPlayer, Kamiccolo |
| 15 | PICO-8 | Jon Ericson |
| 16 | Recolector de | greatwolf, Kamiccolo, val |

| | |
|---|---|
| | basura y mesas<br>débiles. |