eBook Gratuit

APPRENEZ Lua

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#lua

Table des matières

À propos

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

Chapitre 1: Démarrer avec Lua

Remarques

Lua est un langage de script minimaliste, léger et intégrable. Il est conçu, mis en œuvre et entretenu par une équipe de PUC-Rio , l'Université pontificale catholique de Rio de Janeiro au Brésil. La liste de diffusion est ouverte pour participer.

Les cas d'utilisation courants de Lua incluent les jeux vidéo de script, l'extension d'applications avec des plugins et des configs, l'intégration d'une logique métier de haut niveau ou l'intégration dans des périphériques tels que des téléviseurs, des voitures, etc.

Pour les tâches à hautes performances, il existe une implémentation indépendante utilisant un compilateur juste-à-temps, appelé LuaJIT .

Versions

Version	Remarques	Date de sortie
1.0	Version initiale, non publique.	1993-07-28
1.1	Première version publique. Document de conférence le décrivant.	1994-07-08
2.1	À partir de Lua 2.1, Lua est devenu librement disponible pour toutes les utilisations, y compris commerciales. Journal de journal le décrivant.	1995-02-07
2.2	Chaînes longues, interface de débogage, meilleure traçabilité des piles	1995-11-28
2.4	Compilateur `luac` externe	1996-05-14
2,5	Correspondance de modèle et fonctions vararg.	1996-11-19
3.0	Introduit auxlib, une bibliothèque d'aide à l'écriture des bibliothèques	1997-

Version	Remarques	Date de sortie
	Lua	07-01
3.1	Fonctions anonymes et fermetures de fonctions via "upvalues".	1998-07-11
3.2	Bibliothèque de débogage et nouvelles fonctions de table	1999-07-08
3.2.2		2000-02-22
4.0	États multiples, instructions "for", modification de l'API.	2000-11-06
4.0.1		2002-07-04
5.0	Les coroutines, les métabalises, la portée lexicale complète, les appels de queue, les booléens passent à la licence MIT.	2003-04-11
5.0.3		2006-06-26
5.1	Réorganisation du système de modules, récupération de place incrémentielle, métabalises pour tous les types, refonte luaconf.h, analyseur entièrement réentrant, arguments variadiques.	2006-02-21
5.1.5		2012-02-17
5.2	Ramasse-miettes d'urgence, goto, finaliseurs pour les tables.	2011-12-16
5.2.4		2015-03-07
5.3	Prise en charge de base UTF-8, opérations binaires, nombres entiers de 32/64 bits.	2015-01-12
5.3.4	Dernière version.	2017-01-12

Examples

Installation

Binaires

Les binaires Lua sont fournis par la plupart des distributions GNU / Linux en tant que package.

Par exemple, sur Debian, Ubuntu et leurs dérivés, il peut être acquis en exécutant ceci:

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

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

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

Il existe des versions semi-officielles pour Windows, MacOS et d'autres systèmes d'exploitation hébergés sur SourceForge .

Les utilisateurs d'Apple peuvent également installer Lua facilement en utilisant Homebrew :

```
brew install lua
```

(Actuellement, Homebrew a 5.2.4, pour 5.3, voir Homebrew / versions .)

La source

La source est disponible sur la page officielle . L'acquisition de sources et la création de celles-ci devraient être triviales. Sur les systèmes Linux, les éléments suivants devraient suffire:

```
$ 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
```

Dans l'exemple ci-dessus, nous téléchargeons essentiellement une `tarball` source depuis le site officiel, vérifions sa somme de contrôle et extrayons et exécutons `make` . (Vérifiez la somme de contrôle à la page officielle .)

Remarque: vous devez spécifier la cible de construction souhaitée. Dans l'exemple, nous avons spécifié `linux` . Les autres cibles de construction disponibles incluent `solaris` , `aix` , `bsd` , `freebsd` , `macosx` , `mingw` , etc. Consultez `doc/readme.html` , qui est inclus dans la source, pour plus de détails. (Vous pouvez également trouver la dernière version du fichier README en ligne .)

Modules

Les bibliothèques standard sont limitées aux primitives:

- `coroutine` - fonctionnalité de gestion de coroutine
- `debug` - hooks et outils de débogage
- `io` - Primitives IO de base
- `package` - fonctionnalité de gestion de module
- `string` - Chaîne et fonctionnalité de correspondance de modèle spécifique à Lua

- `table` - primitives pour traiter un type Lua essentiel mais complexe - tableaux
- `os` - opérations de base du système d'exploitation
- `utf8` - primitives UTF-8 de base (depuis Lua 5.3)

Toutes ces bibliothèques peuvent être désactivées pour une version spécifique ou chargées au moment de l'exécution.

Les bibliothèques et infrastructures tierces de Lua pour la distribution des modules sont rares, mais s'améliorent. Des projets comme LuaRocks , Lua Toolbox et LuaDist améliorent la situation. Beaucoup d'informations et de nombreuses suggestions peuvent être trouvées sur l'ancien Wiki Lua , mais sachez que certaines de ces informations sont assez anciennes et obsolètes.

commentaires

Les commentaires d'une seule ligne dans Lua commencent par `--` et continuent jusqu'à la fin de la ligne:

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

Les commentaires de bloc commencent par `--[[` et se terminent par `]]` :

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

Les commentaires de bloc utilisent le même style de délimiteur que les chaînes longues; n'importe quel nombre de signes égaux peut être ajouté entre les parenthèses pour délimiter un commentaire:

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

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

Une astuce pour commenter des morceaux de code est de l'entourer de `--[[` et `--]]` :

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

Pour réactiver le bloc, ajoutez simplement a `-` à la séquence d'ouverture des commentaires:

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

De cette façon, la séquence `--` dans la première ligne commence un commentaire sur une seule ligne, tout comme la dernière ligne, et l'instruction d' `print` n'est pas commentée.

En allant plus loin, deux blocs de code peuvent être configurés de telle manière que si le premier bloc est commenté, le second ne le sera pas et vice-versa:

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

Pour activer le deuxième bloc en désactivant le premier bloc, supprimez le premier `-` sur la première ligne:

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

Exécution de programmes Lua

En général, Lua est livré avec deux binaires:

- `lua` - interpréteur autonome et shell interactif
- `luac` - compilateur bytecode

Disons que nous avons un exemple de programme (`bottles_of_mate.lua`) comme ceci:

```
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
```

Le programme lui-même peut être exécuté en exécutant la suite sur Votre shell:

```
$ lua bottles_of_mate.lua
```

La sortie devrait ressembler à ceci, fonctionnant dans la boucle sans 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.
...
```

Vous pouvez compiler le programme dans le bytecode de Lua en exécutant ce qui suit sur votre shell:

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

La liste de bytecode est également disponible en exécutant ce qui suit:

```
$ 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
```

Commencer

les 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.
```

les types

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

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

Le type spécial `nil`

Un autre type dans Lua est `nil` . La seule valeur dans le `nil` type est `nil` . `nil` existe pour être différent de toutes les autres valeurs dans Lua. C'est une sorte de valeur sans valeur.

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

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

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

expressions

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

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

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

Définir des fonctions

```
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
```

booléens

Seulement `false` et `nil` considérés comme faux, tout le reste, y compris `0` et la chaîne vide, sont évalués comme vrais.

collecte des ordures

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

les tables

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

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

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

Ce sont les bases, mais il y a une section sur les tables avec plus d'informations.

conditions

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

pour les boucles

Il existe deux types de `for` boucle dans Lua: un numérique `for` la boucle et un générique `for` la boucle.

- Un numérique `for` boucle a la forme suivante:

  ```
  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 troisième expression en une valeur numérique `for` boucle est l'étape par laquelle la boucle est incrémenté. Cela rend facile de faire des boucles inverses:

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

  Si l'expression de l'étape est omise, Lua suppose une étape par défaut de 1.

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

  Notez également que la variable de boucle est locale à la boucle `for` . Il n'existera plus après la fin de la boucle.

- Générique `for` les boucles fonctionnent à travers toutes les valeurs qu'une fonction retourne iterator:

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

  Lua fournit plusieurs construit en itérateurs (par exemple, `pairs` , `ipairs`), et les utilisateurs peuvent définir leurs propres itérateurs personnalisés et à utiliser avec générique `for` les boucles.

faire des blocs

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

Des choses délicates

Parfois, Lua ne se comporte pas comme on le penserait après avoir lu la documentation. Certains de ces cas sont:

Nil et Rien ne sont pas pareils (PITFALL COMMUN!)

Comme prévu, `table.insert(my_table, 20)` ajoute la valeur `20` à la table et `table.insert(my_table, 5, 20)` ajoute la valeur 20 à la 5ème position. Qu'est-ce que `table.insert(my_table, 5, nil)` fait cependant? On pourrait s'attendre à ce qu'il traite `nil` comme aucun argument, et insère la valeur `5` à la fin de la table, mais il ajoute réellement la valeur `nil` à la 5ème position de la table. Quand est-ce un problème?

```
(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)
```

Une chose similaire se produit avec `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
```

Cela peut également entraîner des erreurs lors de l'utilisation de code tiers. Si, par exemple, la documentation de certaines fonctions indique "renvoie des beignets si chanceux, nul sinon", l'implémentation *peut* ressembler à ceci

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

cette mise en œuvre peut sembler raisonnable au début; il retourne des beignets quand il le faut, et quand vous tapez `result = func(false)` résultat contiendra la valeur `nil` .

Cependant, si l'on devait écrire `print(tostring(func(false)))` lua émettrait une erreur qui ressemble à celle-ci `stdin:1: bad argument #1 to 'tostring' (value expected)`

Pourquoi donc? `tostring` obtient clairement un argument, même s'il est `nil` . Faux. func ne retourne rien du tout, donc `tostring(func(false))` est identique à `tostring()` et PAS identique à `tostring(nil)` .

Les erreurs indiquant "valeur attendue" indiquent clairement que cela pourrait être la source du problème.

Laissant des lacunes dans les tableaux

C'est un énorme piège si vous êtes nouveau sur lua, et il y a beaucoup d' informations dans la catégorie des tables

Bonjour le monde

C'est bonjour le code du monde:

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

Comment ça marche? C'est simple! Lua exécute la fonction `print()` et utilise `"Hello World"` chaîne `"Hello World"` comme argument.

Lire Démarrer avec Lua en ligne: https://riptutorial.com/fr/lua/topic/659/demarrer-avec-lua

Chapitre 2: Arguments Variadic

Introduction

Les varargs , comme on les appelle communément, permettent aux fonctions de prendre un nombre arbitraire d'arguments sans spécification. Tous les arguments donnés à une telle fonction sont regroupés dans une structure unique appelée *liste vararg* ; qui est écrit comme `...` dans Lua. Il existe des méthodes de base pour extraire le nombre d'arguments donnés et la valeur de ces arguments à l'aide de la fonction `select()` , mais des modèles d'utilisation plus avancés peuvent tirer parti de la structure pour son utilitaire complet.

Syntaxe

- *...* - Fait la fonction dont la liste des arguments dans laquelle cela apparaît une fonction variadique
- *select (what, ...)* - Si 'what' est un nombre compris entre 1 et le nombre d'éléments du vararg, renvoie l'élément 'what'th au dernier élément de vararg. Le retour sera nul si l'index est hors limites. Si 'what' est la chaîne '#', retourne le nombre d'éléments dans vararg.

Remarques

Efficacité

La liste vararg est implémentée comme une liste liée dans l'implémentation PUC-Rio du langage, cela signifie que les index sont O (n). Cela signifie que l'itération sur les éléments d'un vararg en utilisant `select()` , comme l'exemple ci-dessous, est une opération O (n ^ 2).

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

Si vous prévoyez d'itérer sur les éléments d'une liste vararg, commencez par empaqueter la liste dans une table. Les accès à la table sont O (1), donc l'itération est O (n) au total. Ou, si vous le souhaitez, consultez l'exemple `foldr()` de la section d'utilisation avancée; il utilise la récursivité pour parcourir une liste vararg dans O (n).

Définition de la longueur de séquence

Le vararg est utile dans la mesure où la longueur du vararg respecte tous les nils explicitement passés (ou calculés). Par exemple.

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

test()              --> 0
```

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

Ce comportement est en conflit avec le comportement des tables, où l'opérateur de longueur `#` ne fonctionne pas avec des «trous» (nils intégrés) dans les séquences. Le calcul de la longueur d'une table avec des trous n'est pas défini et on ne peut pas s'y fier. Donc, en fonction des valeurs de `...` , prendre la longueur de `{...}` peut ne pas donner la réponse *"correcte"* . Dans Lua 5.2+, `table.pack()` été introduit pour gérer cette lacune (il y a une fonction dans l'exemple qui implémente cette fonction dans Lua pur).

Utilisation idiomatique

Parce que les varargs portent leur longueur, les gens les utilisent comme séquences pour éviter le problème avec des trous dans les tableaux. Ce n'était pas leur usage prévu et l'implémentation de référence de Lua n'optimise pas. Bien que cet usage soit exploré dans les exemples, il est généralement mal vu.

Examples

Les bases

Les fonctions variables sont créées à l'aide de la syntaxe `...` ellipses dans la liste d'arguments de la définition de fonction.

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

Si vous appeliez cette fonction comme `id(1, 2, 3, 4, 5)` alors `...` (AKA la liste vararg) contiendrait les valeurs `1, 2, 3, 4, 5` .

Les fonctions peuvent prendre les arguments requis ainsi que `...`

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

La méthode la plus simple pour extraire des éléments de la liste vararg consiste simplement à en attribuer des variables.

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

`select()` peut également être utilisé pour trouver le nombre d'éléments et extraire des éléments de `...` indirectement.

```
function my_print(...)
```

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

`...` peut être emballé dans un tableau pour faciliter son utilisation, en utilisant `{...}` . Cela place tous les arguments dans la partie séquentielle de la table.

5.2

`table.pack(...)` peut également être utilisé pour empaqueter la liste vararg dans une table. L'avantage de `table.pack(...)` est qu'il définit le champ `n` de la table renvoyée sur la valeur de `select('#', ...)` . Ceci est important si votre liste d'arguments peut contenir des nils (voir la section des remarques ci-dessous).

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

La liste vararg peut également être renvoyée par les fonctions. Le résultat est plusieurs retours.

```
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
```

Utilisation avancée

Comme indiqué dans les exemples de base, vous pouvez avoir des arguments liés aux variables et la liste des arguments variables (`...`). Vous pouvez utiliser ce fait pour séparer récursivement une liste comme vous le feriez dans d'autres langues (comme Haskell). Vous trouverez ci-dessous une implémentation de `foldr()` qui en tire parti. Chaque appel récursif lie la tête de la liste vararg à `x` et transmet le reste de la liste à un appel récursif. Cela déstructure la liste jusqu'à ce qu'il n'y ait qu'un seul argument (`select('#', ...) == 0`). Après cela, chaque valeur est appliquée à l'argument de fonction `f` avec le résultat calculé précédemment.

```
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
```

Vous pouvez trouver d' autres définitions de fonctions qui tirent parti de ce style de programmation ici dans le numéro 3 à numéro 8.

La seule structure de données idiomatique de Lua est la table. L'opérateur de longueur de la table n'est pas défini s'il y a des `nil` s situés n'importe où dans une séquence. Contrairement à des tables, la liste vararg respecte explicitement `nil` s comme indiqué dans les exemples de base et la section des remarques (s'il vous plaît lire cet article si vous n'avez pas encore). Avec peu de travail, la liste vararg peut effectuer toutes les opérations qu'une table peut avoir en plus de la mutation. Cela fait de la liste vararg un bon candidat pour l'implémentation de tuples immuables.

```
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 principale différence entre ce qui précède et les tableaux est que les tables sont mutables et ont une sémantique de pointeur, où le tuple ne possède pas ces propriétés. De plus, les n-uplets peuvent contenir des `nil` explicites et une opération de longueur jamais définie.

Lire Arguments Variadic en ligne: https://riptutorial.com/fr/lua/topic/4475/arguments-variadic

Chapitre 3: Booléens à Lua

Remarques

Les booléens, la vérité et la fausseté sont simples dans Lua. Réviser:

1. Il existe un type booléen avec exactement deux valeurs: `true` et `false` .
2. Dans un contexte conditionnel (`if` , `elseif` , `while` , `until`), un booléen n'est pas requis. Toute expression peut être utilisée.
3. Dans un contexte conditionnel, `false` et `nil` considérés comme faux et tout le reste compte comme vrai.
4. Bien que 3 implique déjà ceci: si vous venez d'autres langues, rappelez-vous que `0` et la chaîne vide comptent comme vrais dans les contextes conditionnels dans Lua.

Examples

Le type booléen

Booléens et autres valeurs

Lorsque vous traitez avec lua, il est important de différencier les valeurs booléennes `true` et `false` et les valeurs évaluées à true ou false.

Il n'y a que deux valeurs dans lua évaluées à false: `nil` et `false` , tandis que tout le reste, y compris le `0` numérique, est évalué à true.

Quelques exemples de ce que cela signifie:

```
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.
```

Opérations logiques

Les opérateurs logiques dans lua ne renvoient pas nécessairement des valeurs booléennes:

`and` retournera la deuxième valeur si la première valeur est évaluée à true;

`or` renvoie la deuxième valeur si la première valeur est fausse;

Cela permet de simuler l'opérateur ternaire, comme dans d'autres langages:

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

Cela peut également être utilisé pour initialiser des tables si elles n'existent pas

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

ou pour éviter d'utiliser des instructions if, rendant le code plus facile à lire

```
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)
```

Vérifier si les variables sont définies

On peut aussi facilement vérifier si une variable existe (si elle est définie), puisque les variables non-existantes renvoient `nil` , ce qui équivaut à false.

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

Le seul cas où cela ne s'applique pas est lorsqu'une variable stocke la valeur `false` , auquel cas elle existe techniquement mais est toujours évaluée à false. De ce fait, créer des fonctions qui renvoient `false` et `nil` fonction de l'état ou de l'entrée est une mauvaise conception. On peut quand même vérifier si on a un `nil` ou un `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
```

Contextes conditionnels

Les contextes conditionnels dans Lua (`if` , `elseif` , `while` , `until`) ne nécessitent pas de booléen. Comme beaucoup de langues, toute valeur Lua peut apparaître dans une condition. Les règles d'évaluation sont simples:

1. `false` et `nil` comptent comme faux.
2. Tout le reste compte comme étant vrai.

```
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
```

Opérateurs logiques

Dans Lua, les booléens peuvent être manipulés par *des opérateurs logiques* . Ces opérateurs n'incluent `not` , `and` et / `or` .

Dans les expressions simples, les résultats sont assez simples:

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

Ordre de préséance

L'ordre de priorité est similaire aux opérateurs mathématiques unaires `-` , `*` et `+` :

- `not`
- alors `and`
- alors `or`

Cela peut conduire à des expressions complexes:

```
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
```

Évaluation abrégée

Les opérateurs `and` / `or` peuvent uniquement être évalués à l'aide du premier opérande, à condition que le second ne soit pas nécessaire:

```
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
```

Opérateur conditionnel idiomatique

En raison de la priorité des opérateurs logiques, la capacité d'évaluation coupe courte et l'évaluation des non `false` et non `nil` valeurs `true` , un opérateur conditionnel idiomatiques est 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
```

En outre, en raison de la nature de la structure `x and a or b` , `a` ne sera jamais *renvoyé* si elle est évaluée à `false` , cette condition conditionnelle renverra toujours `b` quel que soit `x` .

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

Tables de vérité

Les opérateurs logiques dans Lua ne "renvoient" pas un booléen, mais l'un de leurs arguments. En utilisant `nil` pour les faux et les nombres pour vrai, voici comment ils se comportent.

```
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
```

Comme vous pouvez le voir, Lua retournera toujours la première valeur qui fait *échouer* ou *réussir* la vérification. Voici les tableaux de vérité montrant cela.

```
  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
```

Pour ceux qui en ont besoin, voici deux fonctions représentant ces opérateurs logiques.

```
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
```

Émulation d'un opérateur ternaire avec les opérateurs logiques 'et' ou '.

Dans lua, les opérateurs logiques `and` / `or` retournent l'un des opérandes comme résultat au lieu d'un résultat booléen. En conséquence, ce mécanisme peut être exploité pour émuler le comportement de l'opérateur ternaire même si lua ne possède pas d'opérateur ternaire «réel» dans le langage.

Syntaxe

condition **et** *truthy_expr* **ou** *falsey_expr*

Utilisation en affectation / initialisation de variables

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

Utiliser dans un constructeur de table

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

Utiliser comme argument de fonction

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

Utiliser dans la déclaration de retour

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

Caveat

Il existe des situations où ce mécanisme n'a pas le comportement souhaité. Considérez ce cas

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

Dans un opérateur ternaire «réel», la valeur attendue de `var` est `false` . En lua, cependant, la `and` évaluation « tombe à travers » parce que le second opérande est Falsey. Par conséquent, `var` se termine avec `should not happen` place.

Deux solutions possibles à ce problème, refactorisez cette expression pour que l'opérande du milieu ne soit pas faux. par exemple.

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

ou bien, utilisez la construction classique `if then else` .

Lire Booléens à Lua en ligne: https://riptutorial.com/fr/lua/topic/3101/booleens-a-lua

Chapitre 4: Coroutines

Syntaxe

- coroutine.create (function) renvoie une coroutine (type (coroutine) == 'thread') contenant la fonction.
- coroutine.resume (co, ...) reprend ou démarre la coroutine. Tous les arguments supplémentaires donnés pour reprendre sont renvoyés par le coroutine.yield () qui a précédemment suspendu la coroutine. Si la coroutine n'a pas été démarrée, les arguments supplémentaires deviennent les arguments de la fonction.
- coroutine.yield (...) donne la coroutine en cours d'exécution. L'exécution reprend après l'appel à coroutine.resume () qui a démarré cette coroutine. Tous les arguments donnés pour céder sont renvoyés par la coroutine.resume () correspondante qui a démarré la coroutine.
- coroutine.status (co) renvoie le statut de la coroutine, qui peut être:
    - "mort": la fonction dans la coroutine a atteint sa fin et la coroutine ne peut plus être reprise
    - "running": la coroutine a repris et est en cours d'exécution
    - "normal": la coroutine a repris une autre coroutine
    - "suspendu": la coroutine a cédé et attend d'être reprise
- coroutine.wrap (function) renvoie une fonction qui, lorsqu'elle est appelée, reprend la coroutine qui aurait été créée par coroutine.create (function).

Remarques

Le système de coroutine a été implémenté en lua pour émuler le multithreading existant dans d'autres langages. Cela fonctionne en commutant à une vitesse extrêmement élevée entre les différentes fonctions pour que l'utilisateur humain pense qu'elles sont exécutées en même temps.

Examples

Créer et utiliser une coroutine

Toutes les fonctions d'interaction avec les coroutines sont disponibles dans la table de **coroutine** . Une nouvelle coroutine est créée en utilisant la fonction **coroutine.create** avec un seul argument: une fonction avec le code à exécuter:

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

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

Un objet coroutine renvoie une valeur de type **thread** , représentant une nouvelle coroutine. Lorsqu'une nouvelle coroutine est créée, son état initial est suspendu:

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

Pour reprendre ou démarrer une coroutine, la fonction **coroutine.resume** est utilisée, le premier argument donné est l'objet thread:

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

Maintenant, la coroutine exécute le code et se termine, changeant son état en **mort** , ce qui ne peut pas être repris.

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

Coroutines peut suspendre son exécution et la reprendre plus tard grâce à la fonction **coroutine.yield** :

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

Comme vous pouvez le voir, **coroutine.yield ()** est présent dans la boucle for, maintenant, lorsque nous reprendrons la coroutine, il exécutera le code jusqu'à ce qu'il atteigne un coroutine.yield:

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

Une fois la boucle terminée, l'état du fil devient **mort** et ne peut plus être repris. Coroutines permet également l'échange entre les données:

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

Si la coroutine est à nouveau exécutée sans arguments supplémentaires, le *complément* sera toujours l'argument du premier résumé, dans ce cas "stackoverflow":

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

Enfin, quand une coroutine se termine, toutes les valeurs renvoyées par sa fonction vont au résumé correspondant:

```
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
```

Les coroutines sont utilisées dans cette fonction pour renvoyer des valeurs à un thread appelant depuis un appel récursif.

```
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}
```

Les coroutines peuvent également être utilisées pour une évaluation paresseuse.

```
-- 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
```

Les coroutines peuvent être utilisées pour les constructions de tuyauterie comme décrit dans Programmation en Lua . L'auteur de PiL, Roberto Ierusalimschy, a également publié un article sur l'utilisation de coroutines pour mettre en œuvre des mécanismes de contrôle de flux plus avancés et généraux, comme les continuations.

Lire Coroutines en ligne: https://riptutorial.com/fr/lua/topic/3410/coroutines

Chapitre 5: Correspondance de motif

Syntaxe

- string.find (str, pattern [, init [, plain]]) - Retourne l'index de début et de fin de la correspondance dans str
- string.match (str, pattern [, index]) - Correspond à un modèle une fois (à partir de l'index)
- string.gmatch (str, pattern) - Retourne une fonction qui parcourt toutes les correspondances dans str
- string.gsub (str, pattern, repl [, n]) - Remplace les sous-chaînes (jusqu'à un maximum de n fois)
- `.` représente tous les caractères
- `%a` représente toutes les lettres
- `%l` représente toutes les lettres minuscules
- `%u` représente toutes les lettres majuscules
- `%d` représente tous les chiffres
- `%x` représente tous les chiffres hexadécimaux
- `%s` représente tous les caractères d'espacement
- `%p` représente tous les caractères de ponctuation
- `%g` représente tous les *caractères imprimables à l'* exception de l'espace
- `%c` représente tous les caractères de contrôle
- `[set]` représente la classe qui est l'union de tous les caractères de l'ensemble.
- `[^set]` représente le complément de l'ensemble
- `*` greedy correspond à 0 ou plusieurs occurrences de la classe de caractères précédente
- `+` greedy match 1 ou plusieurs occurrences de la classe de caractères précédente
- `-` lazy match 0 ou plusieurs occurrences de la classe de caractères précédente
- `?` correspond exactement à 0 ou 1 occurrence de la classe de caractères précédente

Remarques

Dans certains exemples, la notation `(<string literal>):function <string literal>` est utilisée, ce qui équivaut à `string.function(<string literal>, <string literal>)` car toutes les chaînes peuvent être `__index` ensemble de champs `__index` . à la table de `string` .

Examples

Lua correspondance des modèles

Au lieu d'utiliser l'expression rationnelle, la bibliothèque de chaînes Lua possède un ensemble spécial de caractères utilisé dans les correspondances de syntaxe. Les deux peuvent être très similaires, mais la correspondance de modèle Lua est plus limitée et a une syntaxe différente. Par exemple, la séquence de caractères `%a` correspond à n'importe quelle lettre, tandis que sa version majuscule représente *tous les caractères autres que des lettres* , toutes les classes de caractères (une séquence de caractères pouvant correspondre à un ensemble d'éléments).

Classe de personnage	Section correspondant
%une	lettres (AZ, az)
% c	caractères de contrôle (\ n, \ t, \ r, ...)
%ré	chiffres (0-9)
% l	lettre minuscule (az)
% p	caractères de ponctuation (!,?, &, ...)
% s	caractères d'espace
% u	lettres capitales
% w	caractères alphanumériques (AZ, az, 0-9)
%X	chiffres hexadécimaux (\ 3, \ 4, ...)
% z	le personnage avec la représentation 0
.	Correspond à n'importe quel personnage

Comme mentionné ci-dessus, toute version majuscule de ces classes représente le complément de la classe. Par exemple, `%D` correspondra à toute séquence de caractères autre que des chiffres:

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

En plus des classes de caractères, certains caractères ont des fonctions spéciales comme motifs:

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

Le caractère `%` représente un caractère échappé, faisant `%?` correspond à une interrogation et `%%` correspond au symbole de pourcentage. Vous pouvez utiliser le caractère `%` avec tout autre caractère non alphanumérique. Par conséquent, si vous devez échapper, par exemple, à une citation, vous devez utiliser `\\` avant celle-ci, qui échappe à tout caractère d'une chaîne lua.

Un jeu de caractères, représenté entre crochets (`[]`), vous permet de créer une classe de caractères spéciaux, combinant différentes classes et caractères uniques:

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

Vous pouvez obtenir le complément du jeu de caractères en le démarrant avec `^` :

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

Dans cet exemple, `string.match` trouvera la première occurrence qui n'est pas **b** , **a** ou **r** .

Les patterns peuvent être plus utiles à l'aide de modificateurs de répétition / facultatifs, les patterns de lua offrent ces quatre caractères:

Personnage	Modificateur
+	Une ou plusieurs répétitions
*	Zéro ou plus de répétitions
-	Aussi zéro ou plusieurs répétitions
?	Facultatif (zéro ou une occurrence)

Le caractère `+` représente un ou plusieurs caractères correspondants dans la séquence et restituera toujours la séquence correspondante la plus longue:

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

Comme vous pouvez le voir, `*` est similaire à `+` , mais il accepte les occurrences de caractères et est généralement utilisé pour faire correspondre des espaces facultatifs entre différents modèles.

Le caractère `-` est également similaire à `*` , mais au lieu de renvoyer la plus longue séquence correspondante, elle correspond à la plus courte.

Le modificateur `?` correspond à un caractère facultatif, ce qui vous permet de faire correspondre, par exemple, un chiffre négatif:

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

Le moteur d'appariement de motifs Lua fournit quelques éléments de correspondance de motifs supplémentaires:

Objet de caractère	La description
`%n`	pour n entre 1 et 9 correspond une sous-chaîne égale à la n-ème chaîne capturée
`%bxy`	correspond à la sous-chaîne entre deux caractères distincts (paire équilibrée de `x` et `y`)
`%f[set]`	modèle de frontière: correspond à une chaîne vide à n'importe quelle position de sorte que le caractère suivant appartient à définir et le caractère précédent n'appartient pas à définir

string.find (Introduction)

La fonction de `find`

Tout d'abord, regardons la fonction `string.find` en général:

La fonction `string.find (s, substr [, init [, plain]])` renvoie l'index de début et de fin d'une sous-chaîne si elle est trouvée et nulle sinon, à partir de l'index `init` si elle est fournie (la valeur par défaut est 1).

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

Introduire des Patterns

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

Tous **sauf** les caractères suivants représentent eux-mêmes `^$()%.[]*+-?)` . N'importe lequel de ces caractères peut être représenté par un `%` suivant le caractère lui-même.

```
("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 fonction `gmatch`

Comment ça marche

La fonction `string.gmatch` prendra une chaîne d'entrée et un motif. Ce modèle décrit ce qu'il faut réellement récupérer. Cette fonction renverra une fonction qui est en réalité un itérateur. Le résultat de cet itérateur correspondra au motif.

```
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
```

Présentation des captures:

Ceci est très similaire à la fonction normale, mais elle ne renverra que les captures au lieu de la correspondance complète.

```
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 fonction gsub

ne confondez pas avec la fonction string.sub, qui retourne une sous-chaîne!

Comment ça marche

argument de chaîne

```
("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
```

argument de fonction

```
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
```

argument de table

```
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
```

Lire Correspondance de motif en ligne: https://riptutorial.com/fr/lua/topic/5829/correspondance-de-motif

Chapitre 6: Écrire et utiliser des modules

Remarques

Le schéma de base pour écrire un module est de remplir une table avec des clés qui sont des noms de fonctions et des valeurs qui sont les fonctions elles-mêmes. Le module renvoie alors cette fonction pour que le code appelant soit `require` et utilisé. (Les fonctions sont des valeurs de première classe dans Lua. Le stockage d'une fonction dans une table est donc simple et courant.) La table peut également contenir des constantes importantes sous forme de chaînes ou de nombres.

Examples

Ecrire le module

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

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

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

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

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

Une approche alternative à celle ci-dessus consiste à créer une table de niveau supérieur puis à stocker les fonctions directement dans celle-ci. Dans cet idiome, notre module ci-dessus ressemblerait à ceci:

```
-- 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
```

Du point de vue de l'appelant, il y a peu de différence entre les deux styles. (Une différence mérite d'être mentionnée, c'est que le premier style rend plus difficile le monkeypatch du module. Ceci est un pro ou un con, selon votre point de vue. Pour plus de détails à ce sujet, voir ce blog par Enrique García Cota.)

Utiliser le module

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

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

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

Lire Écrire et utiliser des modules en ligne: https://riptutorial.com/fr/lua/topic/1148/ecrire-et-utiliser-des-modules

Chapitre 7: Ensembles

Examples

Rechercher un article dans une liste

Il n'y a pas de moyen intégré pour rechercher une liste pour un élément particulier. Cependant, la programmation en Lua montre comment vous pouvez créer un ensemble pouvant vous aider:

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

Ensuite, vous pouvez inscrire votre liste dans l'ensemble et tester votre adhésion:

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

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

Utiliser une table comme un ensemble

Créer un ensemble

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

Créez un ensemble avec des éléments en définissant leur valeur sur `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'
```

Ajouter un membre à l'ensemble

ajouter un membre en définissant sa valeur sur `true`

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

Supprimer un membre de l'ensemble

```
set.apple = nil
```

Utiliser `nil` au lieu de `false` pour supprimer 'apple' de la table est préférable car cela rendra les éléments itératifs plus simples. `nil` supprime l'entrée de la table tandis que la valeur `false` change sa valeur.

Test d'adhésion

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

Itérer sur des éléments dans un ensemble

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

Lire Ensembles en ligne: https://riptutorial.com/fr/lua/topic/3875/ensembles

Chapitre 8: Garbage Collector et tables faibles

Syntaxe

1. collectgarbage (gcrule [, gcdata]) - Collecte les ordures en utilisant gcrule
2. setmetatable (tab, {__mode = strongmode}) - Définit le mode faible de l'onglet sur faiblesse

Paramètres

paramètre	détails
gcrule & gcdata	Action vers gc (garbage collector): `"stop"` (arrête la collecte), `"restart"` (recommence à collecter), `"collect"` ou `nil` (collecte de tous les déchets), `"step"` (fait une étape de collecte), `"count"` (retourne le nombre de mémoires utilisées en Ko), `"setpause"` et les données sont `"setpause"` entre `0` % et `100` % (paramètre de pause de gc), `"setstepmul"` et les données sont `"setstepmul"` entre `0` % et `100` (définissez `"stepmul"` pour gc) .
mode faible	Type de tableau faible: `"k"` (uniquement les clés faibles), `"v"` (uniquement les valeurs faibles), `"vk"` (clés et valeurs faibles)

Examples

Tables faibles

```
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
```

Lire Garbage Collector et tables faibles en ligne: https://riptutorial.com/fr/lua/topic/5769/garbage-collector-et-tables-faibles

Chapitre 9: Introduction à l'API Lua C

Syntaxe

- lua_State * L = lua_open (); // Crée un nouvel état de VM; Lua 5.0
- lua_State * L = luaL_newstate (); // Crée un nouvel état de VM; Lua 5.1+
- int luaL_dofile (lua_State * L, const char * *filename*); // Exécuter un script lua avec le *nom de fichier* donné en utilisant le lua_State spécifié
- annuler luaL_openlibs (lua_State * L); // Charge toutes les bibliothèques standard dans le lua_State spécifié
- annuler lua_close (lua_State * L); // Ferme l'état de la VM et libère les ressources à l'intérieur
- void lua_call (lua_State * L, int nargs, int nresults); // Appelle la luavalue à l'index - (nargs + 1)

Remarques

Lua fournit également une API C correcte à sa machine virtuelle. Contrairement à la VM elle-même, l'interface de l'API C est basée sur la pile. Ainsi, la plupart des fonctions destinées à être utilisées avec des données consistent à ajouter des éléments sur la pile virtuelle ou à en retirer. De plus, tous les appels d'API doivent être utilisés avec soin dans la pile et ses limites.

En général, tout ce qui est disponible sur le langage Lua peut être fait en utilisant son API C. En outre, il existe des fonctionnalités supplémentaires telles que l'accès direct au registre interne, le changement de comportement de l'allocateur de mémoire standard ou du garbage collector.

Vous pouvez compiler les exemples d'API Lua C fournis en exécutant les opérations suivantes sur votre terminal:

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

Examples

Création d'une machine virtuelle 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;
}
```

Fonctions d'appel 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;
 }
```

Dans l'exemple ci-dessus, nous faisons ces choses:

- créer et configurer Lua VM comme indiqué sur le premier exemple
- obtenir et pousser une fonction Lua de la table Lua globale sur une pile virtuelle
- pousser la chaîne `"Hello C API"` comme argument d'entrée sur la pile virtuelle

- demander à VM d'appeler une fonction avec un argument qui est déjà sur la pile
- fermeture et nettoyage

REMARQUE:

N'oubliez pas que `lua_call()` la fonction et ses arguments dans la pile, ne laissant que le résultat.

En outre, il serait plus sûr d'utiliser Lua protected call - `lua_pcall()` place.

Interpréteur Lua intégré avec personnalisation de l'API et de Lua

Démontrer comment intégrer un interpréteur lua en code C, exposer une fonction C-defined au script Lua, évaluer un script Lua, appeler une fonction C-defined depuis Lua et appeler une fonction définie par Lua à partir de C (l'hôte).

Dans cet exemple, nous voulons que l'humeur soit définie par un script Lua. Voici 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
```

Notez que `mood()` n'est pas appelé dans le script. Il est juste défini pour l'application hôte à appeler. Notez également que le script appelle une fonction appelée `hostgetversion()` qui n'est pas définie dans le script.

Ensuite, nous définissons une application hôte utilisant 'mood.lua'. Voici le '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;
}
```

Et voici la sortie:

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

Même après avoir compilé 'hostlua.c', nous sommes toujours libres de modifier 'mood.lua' pour modifier la sortie de notre programme!

Manipulation de table

Pour accéder ou modifier un index sur une table, vous devez en quelque sorte placer la table dans la pile.
Supposons, pour cet exemple, que votre table est une variable globale nommée tbl.

Obtenir le contenu à un index particulier:

```
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
}
```

Comme nous l'avons vu, il suffit de pousser la table dans la pile, de pousser l'index et d'appeler lua_gettable. L'argument -2 signifie que la table est le deuxième élément du haut de la pile. lua_gettable déclenche les méthodes de métamorphose. Si vous ne souhaitez pas déclencher de métaméthode, utilisez plutôt lua_rawget. Il utilise les mêmes arguments.

Définition du contenu à un index particulier:

```
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;
}
```

Le même exercice que l'obtention du contenu. Vous devez pousser la pile, appuyer sur l'index puis pousser la valeur dans la pile. après cela, vous appelez lua_settable. l'argument -3 est la position de la table dans la pile. Pour éviter de déclencher des méthodes, utilisez lua_rawset au lieu de lua_settable. Il utilise les mêmes arguments.

Transférer le contenu d'une table à une autre:

```
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;
}
```

Nous rassemblons maintenant tout ce que nous avons appris ici. Je mets le contenu de la pile sur les commentaires afin que vous ne vous perdiez pas.

Nous mettons les deux tables dans la pile, poussez l'index de la table 1 dans la pile et obtenez la valeur à `tbl1.index1` . Notez l'argument `-3` sur gettable. Je regarde la première table (la troisième en partant du haut) et non la seconde. Ensuite, nous poussons l'index de la deuxième table, copions le `tbl1.index1` en haut de la pile, puis appelons `lua_settable` , sur le 4ème élément depuis le haut.

Pour l'entretien ménager, j'ai purgé l'élément supérieur, de sorte que seules les deux tables restent sur la pile.

Lire Introduction à l'API Lua C en ligne: https://riptutorial.com/fr/lua/topic/671/introduction-a-l-api-lua-c

Chapitre 10: La gestion des erreurs

Examples

En utilisant pcall

`pcall` signifie "appel protégé". Il est utilisé pour ajouter la gestion des erreurs aux fonctions. `pcall` fonctionne comme un `try-catch` dans d'autres langues. L'avantage de `pcall` est que toute l'exécution du script n'est pas interrompue si des erreurs surviennent dans les fonctions appelées avec `pcall` . Si une erreur dans une fonction appelée avec `pcall` se produit, une erreur est `pcall` et le reste du code continue son exécution.

Syntaxe:

```
pcall( f , arg1,···)
```

Valeurs de retour:

Renvoie deux valeurs

1. statut (booléen)

- Renvoie **true** si la fonction a été exécutée sans erreur.
- Renvoie **false** si une erreur est survenue dans la fonction.

2. Renvoie la valeur de la fonction **ou du** message d'erreur si une erreur s'est produite dans le bloc fonction.

---

pcall peut être utilisé pour divers cas, cependant, il est courant de détecter les erreurs de la fonction donnée à votre fonction. Par exemple, disons que nous avons cette fonction:

```
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
```

Lorsque la fonction donnée commet des erreurs lors de l'exécution 3, le message d'erreur sera clair pour l'utilisateur qu'il ne provient pas de votre fonction, mais de la fonction qui a été donnée à notre fonction. Dans cette optique, un BSoD sophistiqué peut être envoyé à l'utilisateur. Cependant, cela dépend de l'application qui implémente cette fonction, car une API ne le fera probablement pas.

Exemple A - *Exécution sans 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
```

Exemple B - *Exécution avec 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"
```

Exemple - *Exécution d'un code sans faille*

```
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"
```

Gestion des erreurs dans Lua

En supposant que nous avons la fonction suivante:

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

Améliorons un peu

```
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 nous ne voulons pas qu'une fonction plante un programme même en cas d'erreur, il est standard de faire ce qui suit:

```
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
```

Maintenant, nous avons une fonction qui se comporte comme ça, nous pouvons faire des choses comme ceci:

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

Et si nous voulons que le programme plante si quelque chose ne va pas, nous pouvons toujours le faire:

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

Heureusement, nous n'avons même pas à écrire tout cela à chaque fois. Lua a une fonction qui fait exactement cela

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

Lire La gestion des erreurs en ligne: https://riptutorial.com/fr/lua/topic/4561/la-gestion-des-erreurs

Chapitre 11: Les fonctions

Syntaxe

- *funcname* = function (paramA, paramB, ...) body; return exprlist end - une fonction simple
- fonction *funcname* (paramA, paramB, ...) body; return exprlist end - raccourci pour ci-dessus
- *funcname* local = fonction (paramA, paramB, ...) body; return exprlist end - un lambda
- *funcname* local; *funcname* = function (paramA, paramB, ...) body; return exprlist end - lambda qui peut faire des appels récursifs
- fonction locale *funcname* (paramA, paramB, ...) body; return exprlist end - raccourci pour ci-dessus
- *funcname* (paramA, paramB, ...) - appelle une fonction
- local *var* = *var* ou "Default" - un paramètre par défaut
- retourne nil, "messages d'erreur" - façon standard d'abandonner avec une erreur

Remarques

Les fonctions sont généralement définies avec la `function a(b,c) ... end` et rarement avec la définition d'une variable à une fonction anonyme (`a = function(a,b) ... end`). Le contraire est vrai lors du passage de fonctions en tant que paramètres, les fonctions anonymes sont principalement utilisées et les fonctions normales ne sont pas utilisées aussi souvent.

Examples

Définir une fonction

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

Regardons la syntaxe. Tout d'abord, nous voyons un mot-clé de `function` . Eh bien, c'est assez descriptif. Ensuite, nous voyons l'identifiant `add` ; le nom. Nous voyons alors les arguments `(a, b)` qui peuvent être n'importe quoi, et ils sont locaux. Nous ne pouvons y accéder que dans le corps de la fonction. Passons à la fin, on voit ... enfin la `end` ! Et tout ce qui est entre les deux est le corps de la fonction; le code qui est exécuté quand il est appelé. Le mot-clé `return` est ce qui fait que la fonction donne des résultats utiles. Sans elle, la fonction ne renvoie rien, ce qui équivaut à renvoyer zéro. Cela peut bien sûr être utile pour les choses qui interagissent avec IO, par exemple:

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

Dans cette fonction, nous n'avons pas utilisé l'instruction return.

Les fonctions peuvent également renvoyer des valeurs de manière conditionnelle, ce qui signifie qu'une fonction a le choix de ne rien renvoyer (aucune). Ceci est démontré dans l'exemple suivant.

```
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
```

Il est également possible pour une fonction de renvoyer plusieurs valeurs séparées par des virgules, comme indiqué:

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

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

Les fonctions peuvent également être déclarées 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
```

Ils peuvent également être enregistrés dans des tableaux:

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

Appeler une fonction

Les fonctions ne sont utiles que si nous pouvons les appeler. Pour appeler une fonction, la syntaxe suivante est utilisée:

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

Nous appelons la fonction d' `print` . En utilisant l'argument `"Hello, World"` . Comme il est évident, cela affichera `Hello, World` dans le flux de sortie. La valeur renvoyée est accessible, comme toute autre variable serait.

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

Les variables sont également acceptées dans les paramètres d'une fonction.

```
local a = 10
```

```
local b = 60

local c = add(a, b)

print(c)
```

Les fonctions qui attendent une table ou une chaîne peuvent être appelées avec un sucre syntaxique net: les parenthèses entourant l'appel peuvent être omises.

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

Fonctions anonymes

Création de fonctions anonymes

Les fonctions anonymes sont comme les fonctions Lua ordinaires, sauf qu'elles n'ont pas de nom.

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

Comme vous pouvez le voir, la fonction n’est assignée à aucun nom comme `print` ou `add` . Pour créer une fonction anonyme, il vous suffit d'omettre le nom. Ces fonctions peuvent également prendre des arguments.

Comprendre le sucre syntaxique

Il est important de comprendre que le code suivant

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

est en fait juste un raccourci pour

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

Cependant, la fonction ci-dessus n'est **pas** anonyme car la fonction est directement affectée à une variable!

Les fonctions sont des valeurs de première classe

Cela signifie qu'une fonction est une valeur avec les mêmes droits que les valeurs conventionnelles telles que les nombres et les chaînes. Les fonctions peuvent être stockées dans des variables, dans des tables, peuvent être passées en arguments et peuvent être renvoyées par

d'autres fonctions.

Pour le démontrer, nous allons également créer une fonction "half":

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

Donc, maintenant nous avons deux variables, `half` et `double` , contenant toutes deux une fonction en tant que valeur. Que faire si nous voulions créer une fonction qui alimenterait le nombre 4 en deux fonctions données et calculer la somme des deux résultats?

Nous voudrons appeler cette fonction comme `sumOfTwoFunctions(double, half, 4)` . Cela alimentera la fonction `double` , la `half` fonction et le nombre entier `4` dans notre propre fonction.

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

La fonction `sumOfTwoFunctions` ci-dessus montre comment les fonctions peuvent être transmises aux arguments et accédées par un autre nom.

Paramètres par défaut

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

Cette fonction est une fonction simple et fonctionne bien. Mais que se passerait-il si on appelait seulement `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 ?
```

Ce n'est pas vraiment génial. Il y a deux manières de corriger cela:

1. Vous revenez immédiatement de la fonction:

```
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. Vous définissez un paramètre *par défaut* .

Pour ce faire, utilisez simplement cette expression 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
```

Le `name = name or "Jack"` idiome `name = name or "Jack"` fonctionne parce que `or` dans les courts-circuits Lua. Si l'élément à gauche de `or` est différent de `nil` ou `false` , le côté droit n'est jamais évalué. D'un autre côté, si `sayHello` est appelé sans paramètre, le `name` sera `nil` et la chaîne `"Jack"` sera assignée à son `name` . (Notez donc que cet idiome ne fonctionnera pas si le booléen `false` est une valeur raisonnable pour le paramètre en question.)

Résultats multiples

Les fonctions dans Lua peuvent renvoyer plusieurs résultats.

Par exemple:

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

Lors de l'appel d'une fonction, pour enregistrer ces valeurs, vous devez utiliser la syntaxe suivante:

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

Ce qui se traduira par `a = b = c = 5` dans ce cas. Il est également possible d'ignorer les valeurs renvoyées en utilisant la variable jetable _ à l'endroit souhaité dans une liste de variables:

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

Dans ce cas, la deuxième valeur renvoyée sera ignorée. Il est également possible d'ignorer les valeurs de retour en ne les affectant à aucune variable:

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

La variable `a` verra attribuer la première valeur de retour et les deux autres seront rejetées.

Lorsqu'une fonction renvoie une quantité variable de résultats, on peut tous les stocker dans une table, en exécutant la fonction à l'intérieur:

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

De cette façon, on peut parcourir le tableau des `results` pour voir ce que la fonction a renvoyé.

Remarque

Cela peut être une surprise dans certains cas, par exemple:

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

Cela se produit parce que `string.gsub` renvoie 2 valeurs: la chaîne donnée, les occurrences du motif remplacé et le nombre total de correspondances.

Pour résoudre ce problème, utilisez une variable intermédiaire ou mettez `()` autour de l'appel, comme ceci:

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

Ceci ne saisit que le premier résultat de l'appel et ignore le reste.

Nombre variable d'arguments

Arguments Variadic

Arguments nommés

```
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
```

Vérification des types d'argument

Certaines fonctions ne fonctionnent que sur un certain type d'argument:

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

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

    return tab.bar
end
```

cela a plusieurs implications:

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

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

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

maintenant nous pouvons appeler la fonction avec n'importe quel paramètre que nous voulons, et cela ne plantera pas le programme.

```
-- 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
```

Alors, que faire si nous avons une fonction qui fait quelque chose avec une instance d'une classe spécifique? C'est difficile, car les classes et les objets sont généralement des tables, de sorte que la fonction `type` renvoie `'table'` .

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

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

object = Class.new()
```

```
fake = {}

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

Solution: comparez les métabalises

```
-- 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
```

Fermetures

```
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
```

exemple d'utilisation typique

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

exemple d'utilisation plus avancé

```
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
```

Lire Les fonctions en ligne: https://riptutorial.com/fr/lua/topic/1250/les-fonctions

Chapitre 12: Les itérateurs

Examples

Générique pour boucle

Les itérateurs utilisent une forme de la boucle `for` connue sous le nom de boucle générique .

La forme générique de la boucle `for` utilise trois paramètres:

1. Une **fonction itérateur** appelée lorsque la valeur suivante est requise. Il reçoit à la fois l'état invariant et la variable de contrôle en tant que paramètres. De retour `nil` terminaison de signaux.
2. L' **état invariant** est une valeur qui ne change pas pendant l'itération. Il s'agit généralement du sujet de l'itérateur, tel qu'une table, une chaîne ou des données utilisateur.
3. La **variable de contrôle** représente une valeur initiale pour l'itération.

Nous pouvons écrire une boucle `for` pour parcourir toutes les paires clé-valeur dans une table en utilisant la fonction suivante .

```
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
```

Itérateurs standard

La bibliothèque standard Lua fournit deux fonctions d'itérateur qui peuvent être utilisées avec une boucle `for` pour parcourir des paires clé-valeur dans des tables.

Pour parcourir une table de séquence, nous pouvons utiliser la fonction de bibliothèque 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
```

Pour itérer toutes les clés et valeurs de n'importe quelle table, nous pouvons utiliser les paires de fonctions de bibliothèque.

```
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
```

Itérateurs sans état

Les deux paires et ipairs représentent des itérateurs sans état. Un itérateur sans état utilise uniquement la variable de contrôle et l'état invariant génériques de la boucle pour calculer la valeur d'itération.

Itérateur de paires

Nous pouvons implémenter l'itérateur de `pairs` sans état en utilisant la fonction `next` .

```
-- 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

Nous pouvons implémenter l'itérateur `ipairs` sans `ipairs` dans deux fonctions distinctes.

```
-- 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
```

Itérateur de personnage

Nous pouvons créer de nouveaux itérateurs apatrides en remplissant le contrat du générique `for` la boucle.

```
-- 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
```

Itérateur de nombres premiers

C'est un exemple plus simple d'itérateur sans état.

```
-- 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
```

Les itérateurs

Les itérateurs à états contiennent des informations supplémentaires sur l'état actuel de l'itérateur.

Utiliser des tables

L'état d'addition peut être compressé dans le générique pour l' état invariant de la boucle .

```
  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
```

Utiliser des fermetures

L'état supplémentaire peut être enveloppé dans une fermeture de fonction. Puisque l'état est entièrement contenu dans la portée de la fermeture, l'état invariant et la variable de contrôle ne sont pas nécessaires.

```
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
```

Utiliser Coroutines

Un état supplémentaire peut être contenu dans une coroutine, là encore, l'état invariant et la variable de contrôle ne sont pas nécessaires.

```
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
```

Lire Les itérateurs en ligne: https://riptutorial.com/fr/lua/topic/4165/les-iterateurs

Chapitre 13: les tables

Syntaxe

- ipairs (numeric_table) - Table Lua avec itérateur d'indices numériques
- pairs (input_table) - itérateur de table Lua générique
- key, value = next (input_table, input_key) - Sélecteur de valeur de table Lua
- table.insert (input_table, [position], value) - Insère la valeur spécifiée dans la table d'entrée
- removed_value = table.remove (input_table, [position]) - pop ou supprimer la valeur spécifiée par la position

Remarques

Les tableaux sont la seule structure de données intégrée disponible dans Lua. Ceci est soit une simplicité élégante, soit déroutante, en fonction de la façon dont vous la regardez.

Une table Lua est une collection de paires clé-valeur où les clés sont uniques et ni la clé ni la valeur ne sont `nil` . En tant que tel, une table Lua peut ressembler à un dictionnaire, un hashmap ou un tableau associatif provenant d'autres langues. De nombreux motifs structurels peuvent être construits avec des tables: piles, files d'attente, ensembles, listes, graphiques, etc. Enfin, les tables peuvent être utilisées pour construire des *classes* dans Lua et créer un système de *modules* .

Lua n'applique aucune règle particulière sur la façon dont les tables sont utilisées. Les éléments contenus dans une table peuvent être un mélange de types Lua. Ainsi, par exemple, une table peut contenir des chaînes, des fonctions, des valeurs booléennes, des nombres *et même d'autres tables en* tant que valeurs ou clés.

Une table Lua avec des clés entières positives consécutives commençant par 1 est dite avoir une séquence. Les paires clé-valeur avec des clés entières positives sont les éléments de la séquence. D'autres langages appellent cela un tableau à 1 base. Certaines opérations et fonctions standard ne fonctionnent que sur la séquence d'une table et certaines ont un comportement non déterministe lorsqu'elles sont appliquées à une table sans séquence.

Définir une valeur dans une table à `nil` supprime de la table. Les itérateurs ne verraient plus la clé associée. Lors du codage d'une table avec une séquence, il est important d'éviter de casser la séquence; Supprimez uniquement le dernier élément ou utilisez une fonction, comme la `table.remove` standard.remove, qui déplace les éléments vers le bas pour combler l'écart.

Examples

Créer des tables

Créer une table vide est aussi simple que cela:

```
local empty_table = {}
```

Vous pouvez également créer une table sous la forme d'un tableau simple:

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

Gardez à l'esprit que par défaut, l'indexation de la table commence à 1.

Il est également possible de créer une table avec des éléments associatifs:

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

L'utilisation ci-dessus est du sucre syntaxique pour ce qui est ci-dessous. Les clés dans cette instance sont du type string. La syntaxe ci-dessus a été ajoutée pour faire apparaître les tables sous forme d'enregistrements. Cette syntaxe de type enregistrement est mise en parallèle avec la syntaxe d'indexation des tables avec des clés de chaîne, comme indiqué dans le didacticiel «Utilisation de base».

Comme expliqué dans la section Remarques, la syntaxe de type enregistrement ne fonctionne pas pour toutes les clés possibles. De plus, une clé peut être n'importe quelle valeur, et les exemples précédents ne couvrent que les chaînes et les numéros séquentiels. Dans d'autres cas, vous devrez utiliser la syntaxe explicite:

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

Tables itératives

La bibliothèque standard Lua fournit une fonction de `pairs` qui parcourt les clés et les valeurs d'une table. Lors de l'itération par `pairs` il n'y a pas d'ordre de traversée spécifié, *même si les clés de la table sont numériques* .

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

Pour les tableaux utilisant **des touches numériques** , Lua fournit une fonction `ipairs` . La fonction

`ipairs` toujours une itération depuis la `table[1]` , la `table[2]` , etc. jusqu'à ce que la première valeur `nil` soit trouvée.

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

Soyez averti que l'itération utilisant `ipairs()` ne fonctionnera pas comme vous le souhaitez à quelques occasions:

- `input_table` contient des "trous". (Reportez-vous à la section "Éviter les lacunes dans les tableaux utilisés comme tableaux" pour plus d'informations.) Par exemple:

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

- les clés n'étaient pas toutes numériques. Par exemple:

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

Bien sûr, ce qui suit fonctionne également pour une table qui est une séquence correcte:

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

Itérer une table numérique dans l'ordre inverse est facile:

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

Une dernière façon d'itérer sur des tables est d'utiliser le `next` sélecteur dans un générique `for` la boucle . Comme les `pairs` il n'y a pas d'ordre spécifié pour la traversée. (La méthode des `pairs` utilise `next` interne. Utiliser `next` est donc une version plus manuelle des `pairs` . Voir les `pairs` dans le manuel de référence de Lua et `next` dans le manuel de référence de Lua pour plus de détails.)

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

Utilisation de base

L'utilisation de table de base inclut l'accès et l'affectation d'éléments de table, l'ajout de contenu de table et la suppression du contenu de tableau. Ces exemples supposent que vous savez créer des tableaux.

Accéder aux éléments

Vu le tableau suivant,

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

On peut indexer la partie séquentielle de la table en utilisant la syntaxe d'index, l'argument de la syntaxe d'index étant la clé de la paire clé-valeur souhaitée. Comme expliqué dans le tutoriel de création, la plupart des syntaxes de déclaration sont du sucre syntaxique pour déclarer des paires clé-valeur. Les éléments inclus séquentiellement, comme les cinq premières valeurs dans `example_table` , utilisent des valeurs entières croissantes comme clés; la syntaxe de l'enregistrement utilise le nom du champ sous forme de chaîne.

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

Pour les clés de chaîne, il existe un sucre de syntaxe pour mettre en parallèle la syntaxe de style d'enregistrement pour les clés de chaîne dans la création de table. Les deux lignes suivantes sont équivalentes.

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

Vous pouvez accéder aux tables à l'aide de clés que vous n'avez jamais utilisées auparavant, ce qui n'est pas une erreur, comme c'est le cas dans d'autres langues. Cela renvoie la valeur par défaut `nil` .

Affectation d'éléments

Vous pouvez modifier des éléments de table existants en les affectant à une table à l'aide de la syntaxe d'index. En outre, la syntaxe d’indexation de style d’enregistrement est également disponible pour définir des valeurs

```
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
```

Vous pouvez également ajouter de nouveaux éléments à une table existante à l'aide de l'affectation.

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

Remarque spéciale: certaines chaînes ne sont pas prises en charge avec la syntaxe d'enregistrement. Voir la section des remarques pour plus de détails.

Suppression d'éléments

Comme indiqué précédemment, la valeur par défaut pour une clé sans valeur assignée est `nil` . Supprimer un élément d'une table est aussi simple que de redéfinir la valeur d'une clé sur sa

valeur par défaut.

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

Les éléments ne peuvent plus être distingués d'un élément non défini.

Longueur de la table

Les tableaux sont simplement des tableaux associatifs (voir remarques), mais lorsque des clés entières contiguës sont utilisées à partir de 1, la table est dite avoir une *séquence* .

La longueur de la partie séquence d'une table se fait en utilisant `#` :

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

Vous pouvez utiliser l'opération length pour ajouter facilement des éléments à une table de séquence.

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

Dans l'exemple ci-dessus, la valeur précédente de `#example_table` est `8` , l'ajout de `1` vous donne la prochaine clé entière valide dans la séquence, `9` , donc ... `example_table[9] = 'a'` . Cela fonctionne pour n'importe quelle longueur de table.

Remarque spéciale: L' utilisation de clés entières qui ne sont pas contiguës et qui partent de 1 rompt la séquence, ce qui fait de la table une *table fragmentée* . Le résultat de l'opération de longueur est indéfini dans ce cas. Voir la section des remarques.

Utilisation des fonctions de la bibliothèque de tables pour ajouter / supprimer des éléments

Une autre façon d'ajouter des éléments à une table est la fonction `table.insert()` . La fonction d'insertion ne fonctionne que sur les tables de séquence. Il y a deux façons d'appeler la fonction. Le premier exemple montre le premier usage, où on spécifie l'index pour insérer l'élément (le second argument). Cela pousse tous les éléments de l'index donné à `#table` jusqu'à une position. Le deuxième exemple montre l'autre utilisation de `table.insert()` , où l'index n'est pas spécifié et la valeur donnée est ajoutée à la fin de la table (index `#table + 1`).

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

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

La `table.insert()` parallèle à `table.insert()` pour supprimer des éléments est `table.remove()` . De même, il a deux sémantiques d'appel: une pour supprimer des éléments à une position donnée et une autre pour supprimer de la fin de la séquence. Lors de la suppression du milieu d'une séquence, tous les éléments suivants sont décalés d'un index vers le bas.

```
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"
```

Ces deux fonctions mutent la table donnée. Comme vous pourriez être en mesure de dire la deuxième méthode d'appeler `table.insert()` et `table.remove()` fournit la sémantique de la pile aux tables. En tirant parti de cela, vous pouvez écrire du code comme l'exemple ci-dessous.

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

Il implémente le Fisher-Yates Shuffle, peut-être de manière inefficace. Il utilise la `table.insert()` pour ajouter l'élément extrait de manière aléatoire à la fin de la même table, et la `table.remove()` pour extraire aléatoirement un élément de la partie restante sans mélange de la table.

Éviter les lacunes dans les tableaux utilisés comme tableaux

Définir nos termes

Par *tableau* , nous entendons ici une table Lua utilisée comme séquence. Par exemple:

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

Nous utilisons cette table comme une séquence: un groupe d'éléments indexés par des entiers. De nombreuses langues appellent cela un tableau, et nous aussi. Mais à proprement parler, il n'ya pas de tableau dans Lua. Il n'y a que des tables, dont certaines ressemblent à des tableaux, d'autres à la forme de hachage (ou de dictionnaire, si vous préférez), et d'autres sont mélangées.

Un point important à propos de notre tableau pour `pets` est qu'il n'y a pas de lacunes. Le premier élément, `pets[1]` , est la chaîne "dogs", le deuxième élément, `pets[2]` , est la chaîne "cats", et le dernier élément, `pets[3]` , est "birds". La bibliothèque standard de Lua et la plupart des modules écrits pour Lua supposent 1 comme premier index pour les séquences. Un tableau sans gabarit a donc des éléments de `1..n` sans manquer aucun nombre dans la séquence. (Dans le cas limite, `n = 1` , et le tableau ne contient qu'un élément.)

Lua fournit la fonction `ipairs` pour parcourir ces tables.

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

Ceci imprimerait "les objets à la position 1 sont les chiens", "les objets à la position 2 sont les

chats", "les objets à la position 3 sont les oiseaux".

Mais que se passe-t-il si nous faisons ce qui suit?

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

Un tableau tel que ce deuxième exemple est un tableau fragmenté. Il y a des lacunes dans la séquence. Ce tableau ressemble à ceci:

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

Les valeurs nulles ne prennent aucune mémoire supplémentaire; lua en interne ne sauvegarde que les valeurs `[1] = "dogs"` , `[2] = "cats"` , `[3] = "birtds"` et `[12] = "goldfish"`

Pour répondre à la question immédiate, `ipairs` s'arrêtera après les oiseaux; "goldfish" sur les `pets[12]` ne sera jamais atteint à moins que nous ajustions notre code. Ceci est dû au fait que `ipairs` itération à partir de `1..n-1` où `n` est la position du premier `nil` trouvé. Lua définit la `table[length-of-table + 1]` comme étant `nil` . Donc, dans une séquence correcte, l'itération s'arrête lorsque Lua essaie, par exemple, d'obtenir le quatrième élément d'un tableau à trois éléments.

Quand?

Les deux endroits les plus courants pour les problèmes qui surviennent avec les baies éparses sont (i) lorsque vous essayez de déterminer la longueur du tableau et (ii) lorsque vous essayez de parcourir le tableau. En particulier:

- Lorsque vous utilisez l'opérateur de longueur `#` , l'opérateur de longueur cesse de compter au premier `nil` trouvé.
- Lorsque vous utilisez la fonction `ipairs()` , comme mentionné ci-dessus, elle cesse d'itérer au premier `nil` trouvé.
- Lorsque vous utilisez la fonction `table.unpack()` car cette méthode arrête la décompression au premier `nil` trouvé.
- Lorsque vous utilisez d'autres fonctions qui accèdent (directement ou indirectement) à l'une des fonctions ci-dessus.

Pour éviter ce problème, il est important d'écrire votre code afin que si vous vous attendez à ce qu'une table soit un tableau, vous n'introduisez pas de lacunes. Les lacunes peuvent être introduites de plusieurs manières:

- Si vous ajoutez quelque chose à un tableau au mauvais endroit.
- Si vous insérez une valeur `nil` dans un tableau.
- Si vous supprimez des valeurs d'un tableau.

Vous pourriez penser: "Mais je ne ferais jamais aucune de ces choses." Eh bien, pas intentionnellement, mais voici un exemple concret de la façon dont les choses pourraient mal tourner. Imaginez que vous vouliez écrire une méthode de filtrage pour Lua comme `select` de Ruby et `grep` de Perl. La méthode acceptera une fonction de test et un tableau. Il itère sur le tableau, appelant la méthode de test sur chaque élément à son tour. Si l'élément passe, cet élément est ajouté à un tableau de résultats renvoyé à la fin de la méthode. Ce qui suit est une mise en œuvre boguée:

```
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
```

Le problème est que lorsque la fonction retourne `false` , nous sautons un nombre dans la séquence. Imaginez un `filter(isodd, {1,2,3,4,5,6,7,8,9,10})` appel `filter(isodd, {1,2,3,4,5,6,7,8,9,10})` : il y aura des lacunes dans la table renvoyée chaque fois qu'un nombre pair sera passé dans le tableau à `filter` .

Voici une implémentation fixe:

```
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
```

Conseils

1. Utilisez les fonctions standard: `table.insert(<table>, <value>)` ajoute toujours à la fin du tableau. `table[#table + 1] = value` est un raccourci pour cela. `table.remove(<table>, <index>)` déplacera toutes les valeurs suivantes pour combler le vide (ce qui peut également le ralentir).
2. Vérifiez les valeurs `nil` **avant d'** insérer, en évitant des choses comme `table.pack(function_call())` , qui peuvent introduire des valeurs `nil` dans notre table.
3. Vérifier les valeurs `nil` **après l'** insertion et, si nécessaire, combler le vide en déplaçant toutes les valeurs consécutives.
4. Si possible, utilisez des valeurs d'espace réservé. Par exemple, changez la valeur `nil` pour `0` ou une autre valeur d'espace réservé.
5. Si des lacunes sont inévitables, cela devrait être correctement documenté (commenté).

6. Ecrivez une `__len()` et utilisez l'opérateur `#` .

Exemple pour 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'
```

Une autre alternative consiste à utiliser la fonction `pairs()` et à filtrer les indices non entiers:

```
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
```

Lire les tables en ligne: https://riptutorial.com/fr/lua/topic/676/les-tables

Chapitre 14: Métatables

Syntaxe

- [[local] *mt* =] getmetatable (*t*) -> récupère la métatable associée pour ' *t* '
- [[local] *t* =] setmetatable (*t* , *mt*) -> définit la métatable pour ' *t* ' en ' *mt* ' et renvoie ' *t* '

Paramètres

Paramètre	Détails
t	Variable faisant référence à une table lua; peut aussi être un littéral de table.
mt	Tableau à utiliser comme métatable; peut avoir zéro ou plusieurs champs de métaméthode définis.

Remarques

Il y a quelques méthodes non mentionnées ici. Pour la liste complète et leur utilisation, voir l'entrée correspondante dans le manuel de lua .

Examples

Création et utilisation de métabalises

Un métatable définit un ensemble d'opérations qui modifient le comportement d'un objet lua. Un métatable est juste une table ordinaire, qui est utilisée d'une manière spéciale.

```
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 }'
```

Utiliser des tables comme méthodes

Certaines méthodes ne doivent pas nécessairement être des fonctions. L'exemple le plus important pour cela est la `__index` métadonnées __index. Il peut également s'agir d'une table, qui est ensuite utilisée comme référence. Ceci est assez couramment utilisé dans la création de classes en lua. Ici, une table (souvent la métatable elle-même) est utilisée pour contenir toutes les opérations (méthodes) de la classe:

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

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

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

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

Garbage Collector - la metamethod __gc

5.2

Les objets en lua sont des ordures collectées. Parfois, vous devez libérer des ressources, imprimer un message ou faire autre chose lorsqu'un objet est détruit (collecté). Pour cela, vous pouvez utiliser la `__gc` __gc, qui est appelée avec l'objet comme argument lorsque l'objet est détruit. Vous pouvez voir cette méthode comme une sorte de destructeur.

Cet exemple montre la `__gc` __gc en action. Lorsque la table interne affectée à `t` reçoit des ordures, elle imprime un message avant d'être collecté. De même pour la table externe à la fin du 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
```

Plus de méthodes

Il y a beaucoup plus de méta-méthodes, certaines sont arithmétiques (par exemple addition, soustraction, multiplication), il y a des opérations binaires (et / ou xor, shift), des comparaisons (<,>) et des opérations de base comme == et # (égalité et longueur). Permet de construire une classe qui supporte plusieurs de ces opérations: un appel à l'arithmétique rationnelle. Bien que ce soit très simple, cela montre l'idée.

```
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
```

Rendre les tables appelables

Il existe une méthode appelée `__call` , qui définit le bevahiour de l'objet lorsqu'il est utilisé comme une fonction, par exemple `object()` . Cela peut être utilisé pour créer des objets de fonction:

```
-- 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
```

La metamethod est appelée avec l'objet correspondant, tous les arguments restants sont transmis à la fonction après cela:

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

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

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

Indexation des tables

L'utilisation la plus importante des métabalises est peut-être la possibilité de modifier l'indexation des tables. Pour cela, il y a deux actions à prendre en compte: *lire* le contenu et *écrire* le contenu de la table. Notez que les deux actions ne sont déclenchées que si la clé correspondante n'est pas présente dans la table.

En train de lire

```
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
```

Cela pourrait être utilisé pour générer une erreur en lisant une clé inexistante:

```
-- 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
```

L'écriture

```
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
```

Vous pouvez maintenant vous demander comment la valeur réelle est écrite dans le tableau. Dans ce cas, ce n'est pas le cas. Le problème ici est que les méthodes peuvent déclencher des méthodes, ce qui entraînerait une boucle infinitive, ou plus précisément un débordement de pile. Alors, comment pouvons-nous résoudre ce problème? La solution pour cela est appelée *accès table brut* .

Accès à la table brute

Parfois, vous ne voulez pas déclencher de métaméthode, mais vous écrivez ou lisez vraiment exactement la clé donnée, sans certaines fonctions astucieuses entourant l'accès. Pour cela, lua vous fournit des méthodes d'accès aux tables brutes:

```
-- 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])
```

Avec ceci, nous pouvons maintenant réécrire la méthode précédente de `__newindex` pour écrire réellement la valeur dans la table:

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

Simulation de la POO

```
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'"
```

Les méthodes d'instance peuvent être écrites en passant l'objet comme premier argument.

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

Il y a du sucre syntaxique pour cela.

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

Nous pouvons également ajouter des champs par défaut à une classe.

```
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
```

Lire Métatables en ligne: https://riptutorial.com/fr/lua/topic/2444/metatables

Chapitre 15: Orientation Objet

Introduction

Lua lui-même n'offre aucun système de classe. Il est cependant possible d'implémenter des classes et des objets sous forme de tableaux avec juste quelques astuces.

Syntaxe

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

Examples

Orientation d'objet simple

Voici un exemple de base de la façon de faire un système de classe très 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
```

Pour ajouter des variables et / ou des méthodes, ajoutez-les simplement à la classe. Les deux peuvent être remplacés pour chaque instance.

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

Et pour créer une instance de la classe:

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

ou

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

Et pour l'utiliser:

```
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}
```

Changer les méthodes d'un objet

Ayant

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

En supposant que nous voulions changer le comportement d'un seul `object = Class.new()` instance `object = Class.new()` utilisant une métatable,

il y a quelques erreurs à éviter:

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

Cela échange l'ancien avec le nouveau, brisant ainsi l'héritage de classe

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

Gardez à l'esprit que les "valeurs" de la table ne sont que des références. il n'y a en fait qu'une seule table réelle pour toutes les instances d'un objet, sauf si le constructeur est défini comme dans [1] , ce qui modifie le comportement de *toutes les* instances de la classe.

Une façon correcte de le faire:

Sans changer de classe:

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

Comment cela marche-t-il? - Nous créons une nouvelle métifiable comme dans l'erreur n ° 1, mais au lieu de la laisser vide, nous créons une copie électronique sur la métifiable originale. On pourrait dire que la nouvelle métatable "hérite" de l'original comme s'il s'agissait d'une instance de classe elle-même. Nous pouvons maintenant remplacer les valeurs de la métatable d'origine sans

les modifier.

Changer la classe:

1er (recommandé):

```
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
```

2ème (moins recommandé): voir [1]

---

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

Lire Orientation Objet en ligne: https://riptutorial.com/fr/lua/topic/8908/orientation-objet

Chapitre 16: PICO-8

Introduction

Le PICO-8 est une console fantastique programmée en Lua intégré. Il a déjà une bonne documentation . Utilisez cette rubrique pour illustrer les fonctionnalités non documentées ou sous-documentées.

Examples

Boucle de jeu

Il est tout à fait possible d'utiliser PICO-8 en tant que shell interactif , mais vous souhaiterez probablement puiser dans la boucle du jeu. Pour ce faire, vous devez créer au moins une de ces fonctions de rappel:

- `_update()`
- `_update60()` (après la v0.1.8)
- `_draw()`

Un "jeu" minimal peut simplement dessiner quelque chose à l'écran:

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

Si vous définissez `_update60()` , la boucle du jeu tente de fonctionner à 60 images par seconde et ignore `update()` (qui tourne à 30 images par seconde). La fonction de mise à jour est appelée avant `_draw()` . Si le système détecte des images perdues, il ignore la fonction de dessin toutes les deux images, il est donc préférable de conserver la logique du jeu et les entrées du lecteur dans la fonction de mise à jour:

```
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 fonction `_init()` est, à proprement parler, facultative car les commandes extérieures à toute fonction sont exécutées au démarrage. Mais c'est un moyen pratique de réinitialiser le jeu aux conditions initiales sans redémarrer la cartouche:

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

Entrée de la souris

Bien que ce ne soit pas officiellement supporté, vous pouvez utiliser la souris dans vos jeux:

```
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
```

Modes de jeu

Si vous voulez un écran de titre ou un écran de fin de partie, envisagez de configurer un mécanisme de changement de mode:

```
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
```

Lire PICO-8 en ligne: https://riptutorial.com/fr/lua/topic/8715/pico-8

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Lua	1971chevycamaro, Allan Burleson, Community, DarkWiiPlayer, Darryl L Johnson, elektron, greatwolf, Guilherme Salazar, hjpotter92, hugomg, Kamiccolo, lhf, Nikola Geneshki, SoniEx2, Telemachus
2	Arguments Variadic	greatwolf, Kamiccolo, ktb, RamenChef, SoniEx2
3	Booléens à Lua	DarkWiiPlayer, engineercoding, greatwolf, Kamiccolo, Katenkyo , Samuel McKay, Telemachus
4	Coroutines	010110110101, Bjornir, Eshkation, Kamiccolo, ktb, SoniEx2
5	Correspondance de motif	DarkWiiPlayer, engineercoding, Eshkation, greatwolf, Kamiccolo , Stephen Leppik
6	Écrire et utiliser des modules	SoniEx2, Telemachus
7	Ensembles	Egor Skriptunoff, Jon Ericson, ryanpattison
8	Garbage Collector et tables faibles	greatwolf, Kamiccolo, val
9	Introduction à l'API Lua C	greatwolf, Jeremy Thien, Kamiccolo, Luiz Menezes, RBerteig, tversteeg
10	La gestion des erreurs	Black, DarkWiiPlayer, engineercoding, greatwolf
11	Les fonctions	Art C, Basilio German, DarkWiiPlayer, Firas Moalla, greatwolf, Guilherme Salazar, Jon Ericson, Katenkyo, ktb, MBorsch, Necktrox, qaisjp, RBerteig, Romário, SoniEx2, Telemachus, Unheilig, WolfgangTS
12	Les itérateurs	Adam, Egor Skriptunoff, greatwolf
13	les tables	DarkWiiPlayer, greatwolf, Hastumer, Kamiccolo, ktb, mjanicek, SoniEx2, Telemachus, Tom Blodget
14	Métatables	DarkWiiPlayer, greatwolf, Kamiccolo, pschulz, Telemachus
15	Orientation Objet	DarkWiiPlayer, Kamiccolo

16	PICO-8	Jon Ericson