LUA : LE TUTORIEL

Par Claude URBAN

Date de publication : 5 juillet 2013

Dernière mise à jour : 5 juin 2016

Les informations et la liste des fonctions Lua, énumérées dans ce tutoriel sont extraites de l'ouvrage de MM Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes intitulé : **« Lua 5.2 Reference Manual »** que vous pouvez trouver dans sa version originale sur le site **http://www. lua.org**

Commentez

1 - Préambule........ 5
1-a - Introduction........ 5
1-b - De quoi aurez-vous besoin ?........ 5
1-c - Guide de style........ 5
2 - Les bases du langage........ 7
2-a - Qu'est-ce que Lua ?........ 7
2-b - Les conventions lexicales........ 8
2-c - Les types et les valeurs........ 8
2-d - La coercition........ 9
2-e - Les « chuncks » et les blocs........ 10
2-e-1 - Les chunks........ 10
2-e-2 - Les blocs........ 10
2-f - Les opérateurs arithmétiques........ 11
2-g - Les opérateurs relationnels........ 11
2-h - Les opérateurs logiques........ 12
2-i - L'opérateur « length »........ 12
2-j - L'ordre de priorité des opérateurs........ 13
2-k - La concaténation........ 13
3 - Les variables........ 14
3-a - Une variable, c'est quoi ?........ 14
3-b - Global ou local ?........ 15
3-c - Affectation........ 15
3-d - La portée lexicale........ 16
3-e - Le typage........ 17
3-f - À retenir........ 18
4 - Les structures de contrôle........ 18
4-a - Égal (=) ou bien égal (==) ?........ 18
4-b - Qu'est-ce qu'une « condition » d'exécution ?........ 18
4-c - if ... then ... else ... end........ 19
4-d - while ... do ... end........ 20
4-e - repeat ... until........ 21
4-f - return et break........ 21
4-g - À retenir........ 22
5 - La boucle « for »........ 22
5-a - for ... do ... end........ 22
5-b - La boucle « for numérique »........ 23
5-c - La boucle « for générique »........ 23
5-d - À retenir........ 24
6 - Les fonctions........ 24
6-a - Qu'est-ce qu'une fonction ?........ 24
6-b - Fonction, méthode, self, this 25
6-c - return, fausse variable et unpack........ 26
6-c-1 - La fausse variable........ 28
6-c-2 - La fonction « unpack »........ 28
6-d - Les arguments multiples........ 29
6-e - Les arguments nominatifs........ 29
6-f - Les fonctions anonymes........ 29
6-g - Les closures........ 31
6-h - La portée lexicale........ 32
6-i - Les variables non-globales........ 34
6-j - Les appels récursifs........ 36
6-k - À retenir........ 37
7 - Les itérateurs........ 37
7-a - Qu'est-ce qu'un itérateur ?........ 37
7-b - Les « for » génériques........ 38
7-c - La sémantique des « for » génériques........ 40
7-d - Stateless iterator........ 41
7-e - pairs(t) et ipairs(t)........ 41

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

7-f - À retenir....43
8 - Les tables....43
8-a - Qu'est-ce qu'une table ?....43
8-b - Les tables « made in Lua »....44
8-c - Les constructeurs....44
8-d - Les matrices et les tables....45
8-d-1 - Utiliser une table de table....45
8-d-2 - Rassembler les 2 indices en un seul....46
8-e - Les listes chaînées....46
8-e-1 - Qu'est-ce qu'une liste chaînée ?....46
8-e-2 - Différents types de listes....47
8-e-3 - Et avec Lua ?....47
8-f - Les files d'attente....48
8-g - Comment filtrer certains mots définis ?....49
8-h - Les tampons pour chaînes de caractères....50
8-h-1 - Et pour les gros fichiers ?....51
8-i - Afficher le contenu des champs d'une table....52
8-j - À retenir....53
9 - Les métatables....54
9-a - Qu'est-ce qu'une métatable ?....54
9-b - C'est bien tout ça, mais ... ça sert à quoi ?....54
9-c - Et ça fonctionne comment ?....55
9-d - Et concrètement, on fait comment ?....56
9-e - Les métaméthodes arithmétiques....58
9-f - Les métaméthodes relationnelles....60
9-g - Les autres métaméthodes....60
9-g-1 - __concat(t1, t2)....60
9-g-2 - __tostring(objet)....60
9-g-3 - __call(objet, ...)....61
9-g-4 - __metatable(objet)....61
9-g-5 - __index(table, clé)....61
9-g-6 - __newindex(objet, clé, valeur)....62
9-g-7 - __mode()....62
9-h - À retenir....62
10 - Les strings....63
10-a - Qu'est-ce qu'une chaîne de caractères ?....63
10-b - Déclaration, guillemets et crochets....63
10-c - Les séquences d'échappement....63
10-d - La concaténation....64
10-e - La coercition....65
10-f - Qu'est-ce qu'un « pattern » ?....65
10-g - Les captures....68
10-h - Exemples d'utilisation....68
10-h-1 - Les classes de caractères....69
10-h-2 - Les ensembles....69
10-h-3 - Les captures....69
10-i - À retenir....70
11 - Les coroutines....70
11-a - Qu'est-ce qu'une coroutine ?....70
11-b - Les bases de fonctionnement....70
11-c - Exemple d'utilisation....73
11-d - Les coroutines et les itérateurs....75
11-e - Le multithreading non-préemptif....76
11-f - À retenir....80
12 - Les tables d'environnement....81
12-a - Définition....81
12-b - Affectation....81
12-c - Environnement global....83

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

12-d - Environnement non-global....84
12-e - À retenir....85
13 - La bibliothèque Lua....85
13-a - Fonctions standards de base....85
13-b - Fonctions sur les coroutines....90
13-c - Fonctions d'exploitation....91
13-d - Fonctions d'entrée/sortie....93
13-e - Fonctions mathématiques....96
13-f - Fonctions sur les chaînes de caractères....98
13-g - Fonctions sur les tables....101
13-h - Table des codes de formatage....103
14 - Remerciements....104

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

1 - Préambule

1-a - Introduction

Ce tutoriel, qui **s'adresse plus spécialement aux débutants**, explique la façon d'écrire du code avec le langage de script Lua, dans le cadre d'une utilisation sous Windows.

J'ai repris pour l'essentiel, ma traduction précédemment effectuée du manuel de référence intitulé : *« Lua 5.2 Reference Manuel* », que j'ai reformatée sous forme d'un tutoriel afin d'en rendre la lecture plus agréable, plus fluide et moins rébarbative.
J'y ai aussi rajouté quelques exemples et commentaires, glanés principalement sur le net ou provenant de ma propre expérience.

Les interactions de Lua avec un *hôte* écrit en C ne seront pas abordées, ce qui signifie qu'il ne sera pas question ici des bibliothèques de l'API C (lua_xxx), ni des bibliothèques auxiliaires (luaL_Xxxx).

Lua étant un langage de script, il ne peut pas devenir *exécutable* au sens où on l'entend habituellement, comme avec les programmes écrit en C ou C++ et compilés pour tourner sous Windows (*.exe).

Mais il existe un « wrapper (1) » du nom de wxLuaFreeze.exe qui permet d'exécuter les programmes « *comme si »*, ils étaient vraiment Standalone (2) .

Ce logiciel est conçu pour faire tourner Lua dans le cadre d'une utilisation avec la bibliothèque graphique wxWidgets.
Voir à ce sujet : **wxLua et wxWidgets : Mode d'emploi.**

J'utiliserai donc cet environnement graphique pour écrire certains exemples que vous pourrez exécuter avec wxLuaFreeze.exe.

1-b - De quoi aurez-vous besoin ?

Pour suivre ce tutoriel vous aurez peut-être besoin de :

Charger wxLua, que vous trouverez **ici**.

Bien évidement, vous aurez besoin d'un bon éditeur de texte.
Si la langue anglaise ne vous rebute pas, vous trouverez dans ce que vous venez de charger un éditeur « maison » : wxLua.exe.
Sinon vous pouvez utiliser l'excellent Notepad++, que vous trouverez **ici**.

Vous devrez aussi renseigner les « variables d'environnement » de votre système d'exploitation, qui pour Windows se situent dans le panneau de configuration : **--> Système --> Avancé --> Variables d'environnement -->** ***Variables d'environnement***
Dans la fenêtre inférieure, « Variables système » : **--> Path --> Modifier***, rajouter le chemin où se situe le répertoire bin.* (Pour moi cela donne : ... D:\Lua\wxLua-2.8.12\bin.)
Puis, toujours dans la fenêtre inférieure, « Variables système » :**--> PATHEXT --> Modifier*****,*** rajouter : **;.WXLUA**

Plus tard, si vous souhaitez aller plus loin dans vos investigations, vous pourrez charger tout un ensemble de bibliothèques supplémentaires que vous trouverez **ici**.

1-c - Guide de style

Sachant qu'un code est lu beaucoup plus souvent qu'il n'est écrit, il semble primordial de bien pouvoir se relire ... ce qui n'est pas toujours le cas.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Il n'existe pas de règle absolue en la matière, mais simplement des recommandations basées bien souvent sur l'expérience.

Ce petit guide de style, vise à améliorer la lisibilité du code par la cohérence de son écriture.

Il s'agit bien évidement d'une simple recommandation que vous êtes libre de suivre ou pas ... c'est vous qui voyez !

Le principal, me semble-t-il, est de choisir un style qui vous plaise et de ne plus en changer.

C'est à vous de pouvoir vous relire rapidement dans 15 jours, 3 semaines ou plus.

L'indentation se fait généralement avec 2 espaces :

```
for i, v in ipairs(t) do
  if type(v) == "string" then
    print(v)
  end
end
```

Pensez à définir des noms de variables le plus explicitement possible :
local x = ... ou **local y = ...** c'est pas terrible, mais ***local nbLignes*** ou ***local pxBanane*** par exemple, c'est déjà mieux, l'on sait tout de suite ce que cela veut dire.

La longueur du nom de la variable : Les noms de variables avec un plus grand champ d'application devrait être plus descriptif que ceux qui ont une plus petite portée.
Par exemple, la variable (i) est probablement un mauvais choix pour une variable globale dans un vaste programme, mais elle a parfaitement sa place en tant que compteur dans une petite boucle.

Le nom des variables est généralement écrit en lettres minuscules, par exemple : local maVariable.

Le nom des fonctions : Plusieurs façons de procéder ... à vous, là aussi de choisir !
nomdelafonction(),nom_de_la_fonction(),NomDeLaFonction(),Nom_De_La_Fonction(), vous n'avez que l'embarras du choix ! mais faites toujours la même chose.

Les constantes sont écrites en lettres CAPITALES séparées le cas échéant par un caractère de soulignement : **MA_VARIABLE_EST_UNE_CONSTANTE.**

Le trait de soulignement (_) est souvent utilisé comme espace réservé lorsque vous voulez ignorer une variable. Exemple : local _, x, _, _, y, z = NomDeVotreFonction().

Les variables i, k, v, t sont souvent utilisées comme suit :

```
for k, v in pairs(t) ... end
for i, v in ipairs(t) ... end
mt.__newindex = function(t, k, v) ... end
```

Pensez à utiliser autant que faire ce peut **les variables locales** qui prennent moins de place en mémoire et sont plus rapides ce qui rend le code plus clair et plus sûr.

```
local x = 0
local function count()
  x = x + 1
  print(x)
end
```

Les commentaires : Utiliser une espace après les deux tirets : *-- commentaires*.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

Il peut être intéressant aussi de renseigner à quoi correspondent les end de fin de boucle ou de fonctions, surtout pour les programmes importants.

```
for i, v in ipairs(t) do
  if type(v) == "string" then
    ... votre code ...
  end   -- if type(v)
end     -- ipairs(t)
```

Et pour finir, un petit mot sur le**« point virgule »** (;)
Lua n'impose pas sa présence à chaque fin de ligne.
Vous faites comme vous le sentez, avec ou sans c'est pareil ...

Par contre, plusieurs instructions sur la même ligne se doivent d'être séparées par un (;)
sinon, vous aurez droit à un beau plantage !

2 - Les bases du langage

2-a - Qu'est-ce que Lua ?

Lua est un puissant, rapide et léger, langage de script embarqué développé par une équipe à PUC-Rio, l'Université Pontificale Catholique de Rio de Janeiro, au Brésil.

Il est utilisé dans de nombreux programmes et projets à travers le monde comme : World of Warcraft, Far Cry et SimCity 4 pour ne citer qu'eux.

Lua est un langage d'extension de programme, conçu pour épauler une programmation procédurale générale, à l'aide d'équipements de description de données.

Lua est destiné à être utilisé comme un puissant et léger langage de script par n'importe quel programme qui en aurait besoin.

Lua est implémenté au travers d'une bibliothèque, écrite en C.

Être une extension de langage et non un programme principal, implique, de ne pouvoir fonctionner qu'en étant « embarqué » à un client d'accueil, appelé le programme d'intégration ou tout simplement l'hôte.

Comme dit précédemment, je ne parlerai ici que de Lua dans sa partie embarquée ou utilisée avec **wxLua** et la bibliothèque **wxWidgets**.

Je n'aborderai pas, par conséquent, ni les API C pour Lua, qui représentent l'ensemble des fonctions C mises à la disposition du programme hôte pour communiquer avec Lua (lua_XXX), ni la bibliothèque auxiliaire (LuaL_XXX).

Voir pour ceux que cela intéresse : **Manuel de référence Lua 5.2**

Et pour plus d'informations sur ce logiciel, les sources et la documentation, une seule adresse : **lua.org**

Ceci étant dit et avant de s'attaquer au dédale de la programmation, il est indispensable de connaître quelques règles propres à ce langage.

Aussi, allez-vous aborder dans les paragraphes suivants :

- - les conventions lexicales ;
- - les types et les valeurs ;

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

- - la coercition ;
- - les « chunks » ou morceaux de code ;
- - les opérateurs arithmétiques ;
- - les opérateurs relationnels ;
- - les opérateurs logiques ;
- - l'opérateur « *length* » ;
- - la concaténation ;
- - la priorité des opérateurs.

Ce sont, pour la plupart des sujets qui seront revus par la suite, mais il est parfois bon de taper plusieurs fois sur un clou, pour le faire rentrer dans le bois ...

Et n'oubliez pas que, la seule façon d'apprendre à « *monter à cheval* » est de ... monter à cheval ... alors, **à vos claviers** pour écrire, écrire, écrire encore et ré-écrire du code.

2-b - Les conventions lexicales

Les **noms** en Lua (également appelé identificateurs), peuvent être n'importe quelle chaîne de lettres, chiffres et caractères de soulignement ne commençant pas par un chiffre.

Les **mots-clés** mentionnés ci-dessous, sont réservés et ne peuvent donc pas être utilisés en tant que noms.

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

Lua est un langage sensible à la casse : and est un mot réservé, mais And et AND étant différents sont donc valides.
Par convention, les noms commençant par un caractère de soulignement suivi par des lettres capitales (comme _VERSION) sont réservés aux variables globales internes, utilisées par Lua.
Les chaînes littérales peuvent être délimitées par des apostrophes (') ou des guillemets (") et peuvent contenir les séquences d'échappement suivantes :

- "\a" (cloche) ;
- "\b" (backspace) ;
- "\f" (saut de page) ;
- "\n" (saut de ligne) ;
- "\r" (retour chariot) ;
- "\t" (tabulation horizontale) ;
- "\v" (onglet vertical) ;
- "\\" (barre oblique inverse) ;
- '\"' (guillemet anglais) ;
- "\'" (apostrophe).

Une constante numérique, peut être écrite avec une partie optionnelle décimale et un exposant optionnel décimal.
Lua accepte également les constantes entières hexadécimales, en les préfixant avec 0x.
Voici quelques exemples de constantes numériques valides : 3 - 3,0 - 3,1416 - 314.16e-2 - 0x56 - 0xff - 0.31416E1

Un petit commentaire commence par : -- *ici le petit commentaire.*
Un grand commentaire commence par : -- [[ici le grand commentaire]]

2-c - Les types et les valeurs

Lua est un langage à typage dynamique.
Ce qui signifie que les variables n'ont pas de type, contrairement aux valeurs qui elles sont typées.
Il n'existe pas de définition de type en Lua. Toutes les valeurs soutiennent leur propre type.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Avec Lua, toutes les valeurs sont des « *first-class values* ». (*valeurs de première classe*).
Ce qui signifie que toutes les valeurs peuvent être stockées dans des variables, passées comme arguments à d'autres fonctions, et retournées comme résultat.

Il existe huit types de base en Lua : nil, boolean, number, string, function, userdata, thread, table.

- **nil**, dont le type de valeur est rien, a la propriété principale d'être différent de toute autre valeur et représente généralement l'absence d'une valeur.
- **boolean**, dont les seules valeurs possibles sont *false* et *true*. Les deux types nil et false représentent ce qui est faux. Toute autre valeur est considérée comme vraie (true).
- **number**, représente un chiffre ou un nombre réel, en double précision à virgule flottante.
- **string**, est une table de caractères. Lua est « 8-bit clean » : les chaînes de caractères peuvent contenir n'importe quel caractère 8-bit, y compris "\0" (qui signifie la fin d'une chaîne).
- **function**, Lua peut appeler et manipuler des fonctions écrites en Lua et des fonctions écrites en C.
- **userdata**, est fourni pour permettre à des données C arbitraires, d'être stockées dans des variables Lua. Ce type correspond à un bloc de mémoire brut et n'a pas d'opération prédéfinie dans Lua, à l'exception d'affectation et de test d'identité. Toutefois, en utilisant les méta-tables, le programmeur peut définir certaines opérations pour les valeurs d'userdata. Les valeurs d'userdata ne peuvent pas être créés ou modifiées directement avec Lua, il faut passer par l'intermédiaire de l'API C. Cela garantit l'intégrité des données détenues par le programme hôte.
- **thread**, représente des « threads » indépendants de l'exécution qui sont utilisés pour mettre en œuvre des « coroutines ». Ne confondez pas les thread de Lua avec les « threads » du système d'exploitation. Lua soutient les « coroutines » sur tous les systèmes, même ceux qui ne supportent pas les « threads ».
- **table** , ce type met en œuvre des tables associatives, qui peuvent être indexées non seulement avec des nombres, mais aussi avec des valeurs quelconques, sauf nil. Les tables peuvent être hétérogènes, autrement dit, elles peuvent contenir des valeurs de tous types, à l'exception nil. Les tables sont les seules données mécaniquement structurées en Lua. Elles peuvent être utilisées pour représenter des tables ordinaires, des tables de symboles, de jeux, de livres, de graphiques, d'arbres, etc. Pour déclarer les enregistrements, Lua utilise le nom du champ comme index ; a.name est une façon plus simple d'écrire a["nom"] et a exactement la même signification. Il existe plusieurs méthodes pratiques pour créer des tables. (**Voir à ce sujet, le chapitre consacré aux tables.**)

Les tables, les fonctions, les threads, et les userdatas sont considérés comme des objets.

Les variables ne peuvent donc pas contenir ces valeurs en tant que telles, mais seulement des références à ces valeurs.

Les affectations, le passage de paramètres et les fonctions sont toujours manipulés par les références à ces valeurs et ces opérations n'impliquent aucune forme de copie.

La fonction *type* () retourne une chaîne, décrivant le type d'une valeur donnée.

2-d - La coercition

Lua permet une conversion automatique entre chaînes de caractères et valeurs numériques au moment de l'exécution.

Toute opération arithmétique appliquée à une chaîne de caractères tente de convertir cette chaîne en un chiffre, suivant les règles de conversion habituelles.

Inversement, chaque fois qu'un chiffre ou un nombre est utilisé lorsqu'une chaîne est prévue, ce dernier est converti en une chaîne dans un format raisonnable.

Pour un contrôle complet, sur la façon dont les chiffres et les nombres sont convertis en chaînes, utiliser la fonction *string.format*() de la bibliothèque sur les chaînes de caractères.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

2-e - Les « chuncks » et les blocs

2-e-1 - Les chunks

L'unité d'exécution de Lua est appelé « chunk ».
Et un « chunk », n'est ni plus ni moins qu'un morceau de code.

Un morceau de code, représente une séquence d'instructions qui sont exécutées séquentiellement.

Chaque séquence, peut être éventuellement suivie d'un point-virgule (;).
Ce n'est absolument pas une obligation, vous faites comme vous le sentez.
Avec ou sans, c'est pareil ... sauf sur une même ligne où le point virgule est OBLIGATOIRE pour séparer deux séquences.

Lua gère un morceau de code comme le corps d'une fonction anonyme avec un nombre variable d'arguments.

En tant que tel, des « chunks » peuvent recevoir des variables locales, des arguments et des valeurs de retour.

Un « chunk » peut être stocké dans un fichier ou dans une chaîne de caractères à l'intérieur du programme hôte.

Pour exécuter un « chunk », Lua pré-compile premièrement le « chunk » en instructions pour la machine virtuelle, puis exécute le code compilé avec un interpréteur.

2-e-2 - Les blocs

Un bloc est constitué d'une liste de déclarations.
Syntaxiquement, un bloc est équivalent à un « chunk ».
Un bloc peut être explicitement délimitée pour produire une seule déclaration, par exemple :
do ... bloc ... end

Les blocs explicites sont utiles pour contrôler la portée de déclaration des variables.
Ils sont aussi parfois utilisés pour ajouter un return ou un break dans le milieu d'un autre bloc.

La notion de blocest importante, car c'est elle qui détermine la portée des variables locales. Vous y reviendrez lorsque vous aborderez les variables. Et n'oubliez pas qu'un bloc peut aussi contenir un ou plusieurs autres blocs, qui peuvent aussi contenir d'autres blocs ...

Mais en attendant, et pour peut-être mieux fixer les esprits, voici quelques exemples :

```
do ... bloc ... end

function nomFonction() ... bloc ... end

for ... bloc ... end

if condition then ... bloc ... end

while ... bloc ... end

repeat  ... bloc ... until
```

Sans oublier, bien évidement, le fichier principal lui-même : « nomProgramme.lua » qui est le bloc principal renfermant tous les autres blocs ...

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

2-f - Les opérateurs arithmétiques

Lua soutient les opérateurs arithmétiques habituels :Les opérateurs logiques sont : and, or et not.

- Les opérateurs arithmétiques sont :
- les binaires :
    - **+** (addition),
    - **-** (soustraction),
    - ***** (multiplication),
    - **/** (division),
    - **%** (modulo),
    - **^** (élévation) ;
- et l'unaire :
- **-** (négation).

Si les opérandes sont des chiffres ou de nombres (ou des chaînes de caractères qui peuvent être convertis en nombres), alors toutes les opérations ont le sens usuel.

Élévation, fonctionne pour tout exposant. Par exemple, x^(-0,5) calcule l'inverse de la racine carrée de x.

Modulo est défini comme suit : a % b == a - *math.floor*(a/b)*b.
C'est le reste d'une division, qui arrondit le quotient vers moins l'infini.

2-g - Les opérateurs relationnels

Les opérateurs relationnels en Lua sont :

==	égal
~=	différent de
<	plus petit que
>	plus grand que
<=	plus petit que ou égal
>=	plus grand que ou égal

Le résultat de ces opérateurs est toujours true ou false.

L'égalité == (deux fois le signe égal) compare d'abord le type de ses opérandes.
Si les types sont différents, alors le résultat sera false.

Les valeurs nombres et chaînes sont comparées selon la méthode habituelle.

Les objets (tables, userdata, threads, et fonctions) sont comparés par référence.

Deux objets sont considérés comme égaux, uniquement s'ils ont le même objet.

Chaque fois que vous créez un nouvel objet (table, userdata, thread, ou fonction), il est automatiquement différent de n'importe quel autre objet existant.

Vous pouvez changer la façon dont Lua compare les tables et userdatas en utilisant la méta-méthode « eq ». (**Voir à ce sujet, le chapitre consacré aux métatables.**)

La coercition ne peut pas s'appliquer aux comparaisons d'égalité.
Ainsi, "0" == 0 s'évaluera à false, et t[0] et t["0"] représentent deux entrées différentes d'une table.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

L'opérateur ~=, (différent de) est exactement la négation de l'égalité ==.

Les opérateurs fonctionnent de la façon suivante :

- si les deux arguments sont des nombres, alors ils sont comparés en tant que tels ;
- dans le cas contraire, si les deux arguments sont des chaînes, alors leurs valeurs sont comparées en fonction de la localisation en cours ;
- sinon, Lua essaie d'appeler le « lt » ou « le » de la méta-méthode. (**Voir à ce sujet, le chapitre consacré aux métatables.**) ;
- une comparaison a > b est traduite en b < a et a >= b est traduite en b <= a.

2-h - Les opérateurs logiques

Les opérateurs logiques sont and, or, et not. (*Qui font aussi partie des mots réservés.*)

Comme les structures de contrôle (if, while, etc.), tous les opérateurs logiques considèrent à la fois false et nil comme faux et tout le reste comme vrai.

L'opérateur de négation not retourne toujours false ou true.

L'opérateur de conjonction and retourne son premier argument si cette valeur est false ou nil, ou son second argument dans le cas contraire.

L'opérateur de disjonction or retourne son premier argument si cette valeur est différente de nil et false, ou son second argument dans le cas contraire.

Les deux opérateurs and et or utilisent un raccourci d'évaluation.
C'est-à-dire que le second opérande est évalué uniquement si nécessaire.

Voici quelques exemples :

```
10 or 20  --> 10
10 or error()  --> 10
nil or "a"  --> "a"
nil and 10  --> nil
false and error()  --> false
false and nil  --> false
false or nil  --> nil
10 and 20  --> 20
```

2-i - L'opérateur « length »

L'opérateur **length** est représenté par l'opérateur unaire **#** (dièse).

La longueur d'une chaîne de caractères correspond à son nombre d'octets qui est, au sens habituel la longueur de la chaîne où chaque caractère est un octet.

La longueur d'une table t est définie comme étant tout indice entier n, tel que t[n] ne soit pas nil et que t[n+1] soit nil. En outre, si t[1] est nil, alors n est égal à zéro.

Pour une table régulière, où toutes les valeurs non-nil ont des clés de 1 à n, alors sa longueur sera exactement la valeur du dernier indice n.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

Si la table a des « trous » (c'est-à-dire des valeurs nil comprises entre d'autres valeurs non-nil), alors **#**t pourrait être l'un des indices qui précède directement une valeur nil. (Ce serait alors cette dernière valeur nil qui pourrait être considérée comme la fin du tableau.)

Un programme peut modifier le comportement de l'opérateur **length** par n'importe quelle valeur, par le biais des chaînes de méta-méthodes. (**Voir à ce sujet, le chapitre consacré aux métatables.**)

```
local string = "Lua: Mode d'emploi."
local long = #string
print(long)    --> 19
```

Y compris les espaces, les deux points (:), l'apostrophe (') et le point (.).

2-j - L'ordre de priorité des opérateurs

```
or
and
<    >    <=    >=    ~=    ==
..
+    -
*    /    %
not    #    - (unary)
^
```

La priorité des opérateurs suit l'ordre établi ci-dessus, de la plus petite à la plus grande priorité.

Bien évidement, vous pouvez utiliser des parenthèses pour modifier la préséance d'une expression.

La concaténation (**..**) et l'élévation (**^**), sont des opérateurs associatifs à droite.
Tous les autres opérateurs binaires, sont associatifs à gauche.

2-k - La concaténation

L'opérateur de concaténation de chaînes de caractères dans Lua, est représenté par deux points (**..**).

Si les deux opérandes sont des nombres, alors ils sont convertis en chaînes de caractères.

Si un des opérandes est un nombre, il est converti en chaîne de caractères.

Et si les deux opérandes sont des chaînes, alors ils n'ont pas besoin de conversion.

Sinon, le « concat » de la méta-méthode est appelé. (**Voir à ce sujet, le chapitre consacré aux métatables.**)

```
string = "Lua:"
string2 = "Mode"
string3 = "d'emploi."
nb = 2011
stringA = string1.." "..string2
stringB = " "..string3.." Novembre "..nb
print(stringA..stringB)

--> Lua: Mode d'emploi. Novembre 2011
```

Bien, ceci étant dit, vous allez enfin pouvoir entrer dans le vif du sujet et vous pencher sérieusement sur la façon d'utiliser ce langage.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

La première ligne de code que tape sur son ordinateur un programmeur est très certainement celle qui va permettre d'initialiser une ou plusieurs variables.

Car aucun programme ne peut tourner, s'il n'a pas au moins une variable déclarée !

Mais une variable, c'est quoi ?

3 - Les variables

3-a - Une variable, c'est quoi ?

Une variable est un emplacement mémoire dans lequel on va ranger des données.
(Par exemple, les tiroirs d'une commode, empilés les uns sur les autres.)

Chaque emplacement mémoire (tiroir donc) possède une adresse qui lui est propre et dans lequel vous ne pouvez ranger que 8 bits.

Adresses sur 32 bits	Valeurs sur 8 bits
0000 ... 0000	01011001
0000 ... 0001	11010111
0000 ... 0010	00101110

Pour pouvoir ranger plus que 8 bits, il faudra utiliser plusieurs emplacements mémoires, sachant que le nombre d'emplacements est fonction du nombre d'adresses disponibles qui dépend du CPU et de l'OS.

Pour valoriser 16 bits, on utilisera 2 cases mémoires sur la même adresse, 3 pour 24 bits et 4 cases mémoires pour mémoriser par exemple, une adresse de 32 bits ...

Adresses sur 32 bits	Valeurs sur 8 bits
0000 ... 0000	01011001
	11010111
0000 ... 0001	00101110
	11100101
	00111011
FFFF ... FFFF	10101010

Afin de permettre un suivi et une reconnaissance de ces variables, vous allez leur donner un nom.
(Comme un étiquetage de chaque tiroir.)

maVariable = x

Lua utilise la non-valeur nil , pour représenter une absence de valeur. (Un tiroir vide sera égal à nil .)

Dans l'exemple suivant, la variable *maVariable n'* est pas encore définie. Elle n'a donc aucune valeur et l'emplacement mémoire pourra ainsi être récupérée par le « *garbage collector* (3) » .

maVariable = nil

En principe vous n'aurez pas à le faire, mais si toutefois vous devez annuler une variable, il vous suffira de la *valoriser* à nil . (Ou plus exactement de la non-valoriser !)

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

3-b - Global ou local ?

En Lua, il existe deux sortes de variables :
-- La **variable globale**, qui n'a pas besoin d'être déclarée.
nomVariable = *la valeur que l'on veut lui affecter.*
-- La **variable locale**, qui se déclare de la façon suivante :
local nomVariable = *la valeur que l'on veut lui affecter.*

Toute variable est supposée être globale, sauf si explicitement elle a été déclarée en tant que locale.

La variable globale existe pour **l' ensemble** du programme, alors que la variable locale n'existe qu'à **l' intérieur** de l'environnement dans lequel elle a été créée.

Avant sa première affectation, la valeur d'une variable, est égale à nil .

Toujours utiliser, autant que faire se peut, les variables locales. Ces dernières occupent moins de place en mémoire et sont plus rapides.

3-c - Affectation

Comment affecter une valeur à une variable ?

Vous choisissez une lettre ou plusieurs, écrit en CAPITALES ou en minuscules, peu importe, du moment qu'il ne commence pas par un chiffre.

local b1 sera valable, alors que local 1b sera rejeté, tout comme sera rejeté toute variable utilisant un « mot-clé », tel que définis ci-dessous :

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

Et rappelez-vous aussi que **Lua est sensible à la *casse***, ce qui signifie que si and est un mot réservé et donc interdit, rien ne vous empêche d'utiliser And ou AND ou aNd, qui sont eux différents et donc valides.

Sachez aussi que, par convention, les noms commençant par un caractère de soulignement suivi par des lettres capitales (comme _VERSION) sont réservés aux variables globales internes, utilisées par Lua.

Et que généralement, les lettres CAPITALES sont réservées pour les valeurs constantes. (*Qui sont des « variables » qui ne varient pas !*)

Affectation simple :

local x = 25 ;
local y = "bananes" ;

Affectation multiple :

local x, y = 25, "banane";
local x, y, z, w = 25, "banane", "2012", longueur ;

x aura pour valeur le nombre 25 ;
y aura pour valeur la chaîne "banane" ;
z aura pour valeur la chaîne "2012" ;

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

et w aura pour valeur la variable *longueur.*

Tout peut être déclaré comme variable : Un nom, une fonction, une table ...

Déclaration d'une table : local nomTable = {} ;

Déclaration d'une fonction : local function NomFonction() ... end ;

Déclaration d'un chiffre : local x = 235 ;

Déclaration d'un nom : local x = "poubelle" ;

Déclaration d'une chaîne de caractères : local x = "La cigale ayant chanté tout l'été ... " ;

Déclaration d'une constante : local CECI_EST_UNE_VARIABLE_FIXE = 123456 ;

etc.

3-d - La portée lexicale

Lua est un langage à portée lexicale.

Il est important d'avoir toujours à l'esprit cette notion de portée lexicale et de bien comprendre où commence et où se termine la portée d'une variable, faute de quoi ... c' *est le plantage assuré* !

La portée d'une variable, commence juste après la première déclaration de la variable à l'intérieur d'un bloc et dure jusqu'à la fin de ce bloc.

Un bloc, est un ensemble d'instructions sur plusieurs lignes.

Une fonction, une boucle for, while, do, etc. constitue un bloc et toute déclaration d'une variable à l'intérieure d'un bloc, appartient à ce bloc.

Un petit exemple :

```
x = 10   -- ceci est une variable globale
do -- déclaration d'un nouveau bloc
  local x = x    -- initialisation d'une nouvelle variable 'x'
               -- MAIS cette fois ci "local" en reprenant la
               -- première valeur de x global. (10)
  print("1: "..x)    --> 1: 10
  x = x + 1
  do    -- déclaration d'un autre bloc à l'intérieure du premier
   local x = x + 1    -- encore une autre variable 'x' ...
                      -- mais locale au deuxième bloc
   print("2: "..x)    --> 2: 12
  end
  print("3: "..x)    --> 3: 11
end
print("4: "..x)    --> 4: 10
```

La variable x est maintenant sortie des deux blocs, (les variables locales n'existent plus) , elle retrouve donc sa première valeur : x = 10

Et il ne faut pas oublier que chaque exécution d'une déclaration qui redéfinit une variable locale crée automatiquement « une nouvelle variable locale ».

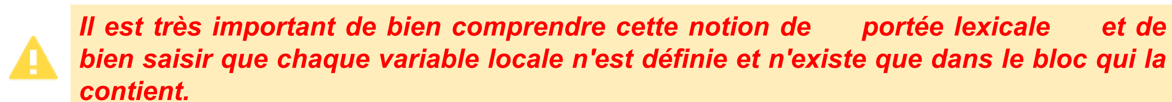
Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Prenez l'exemple suivant :

```
a = {}    -- déclaration d'une table 'a'
local x = 20    -- déclaration d'une variable locale 'x'
for i = 1, 10 do
  local y = 0 -- déclaration d'une nouvelle variable locale
              -- et interne à la boucle for ... end
  a[i] = function() y = y + 1; return x + y end
     print("i = "..i.."    (x + y) = "..x + y)
end
```

Les **tables** et les **fonctions** seront expliquées plus loin.

Mais pour le moment, il faut simplement comprendre que la boucle tourne 10 fois, ce qui représente dix instances de la **fonction anonyme** .
Chaque boucle, va utiliser une nouvelle variable **y** , tandis que toutes partagent le même **x** .
La variable **i** prendra + 1 à chaque tour, normal, mais **x + y** sera toujours égal à 20 du fait que **y** redevient local à chaque tour et donc = 0.


Il est très important de bien comprendre cette notion de portée lexicale et de bien saisir que chaque variable locale n'est définie et n'existe que dans le bloc qui la contient.

3-e - Le typage

Un aspect important avec les variables de Lua, concerne le typage ... ou plus exactement l'absence de typage.

Néanmoins, il faut se souvenir qu'il existe 8 types de base : nil, boolean, number, string, userdata, function, thread, et table.

Dans beaucoup de langages vous devez définir le type de variable que le programme devra utiliser : int, short, long, double, etc.

Avec Lua il n' en est RIEN. Ce n'est pas la variable qui est typée, **mais la valeur** .

Lua est un langage à **typage dynamique** . Ce qui signifie que vous ne devez pas indiquer le type de valeur que devra contenir la variable.
La variable connaîtra son type en fonction de la valeur ou de l'objet qui lui aura été assigné.

```
local x = 123      -- x est un nombre
local x = "123"    -- x est une chaîne de caractères
local x = {}       -- x est une table
```

Chaque type de variable est défini en fonction de ce que vous lui mettez dedans ...

- si vous y mettez une fonction le type de la variable seraune *function* ;
- si vous y mettez un boolean (true ou false) le type de la variable sera boolean ;
- si vous y mettez nil , le type de la variable sera *nil ;*
- etc.

Vous ne pourrez bien sûr comparer ou additionner éventuellement que des variables de types identiques.

Il peut donc être nécessaire de vérifier le type d'une variable avant de procéder à une quelconque manipulation.

Et pour ce faire, il existe une fonction de base intitulée *type* ().

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
print(type(nomVariable))   -- retourne le type de nomVariable
```

3-f - À retenir

1. *une variable est un emplacement mémoire ;*
2. *on donne un nom à une variable et une valeur : maVariable = 0 ;*
3. *on la définit en tant que globale ou locale ;*
4. *nil indique une variable vide ;*
5. *pour une affectation simple : x = 25 ;*
6. *pour une affectation multiple : x, y, z = 10, "abc", 322 ;*
7. *la portée lexicale est à l'intérieur du bloc qui a créé la variable ;*
8. *il existe huit « types » de base : nil, boolean, number, string, userdata, function, thread, et table ;*
9. *Lua est un langage à **typage dynamique**. Il n'y a pas de type à l'initialisation d'une variable et la variable adopte le type de la valeur qu'elle détient ;*
10. *type(maVariable) --> donne le type de, maVariable.*

4 - Les structures de contrôle

4-a - Égal (=) ou bien égal (==) ?

Avant de voir (*ou revoir ?*) le sujet sur les boucles et les structures de contrôle, il me semble indispensable d'effectuer un petit rappel sur la signification exacte du signe *égal*, afin d'éviter de nombreux plantages.

x=2 --> indique que vous transférez la valeur 2 dans la variable x.

x==25 --> signifie que la variable x est égale à 25.

```
if x == 25 then   y = 2   end
```

Ceci peut paraître simpliste, mais croyez-moi, ce n'est pas anodin.

4-b - Qu'est-ce qu'une « condition » d'exécution ?

Pour expliquer ce que sont « les structures de contrôle », le mot **condition** sera plusieurs fois utilisé, comme dans les exemples suivants :

```
if  condition  then ... on exécute ...  end
while  condition  do ... on exécute ... end
repeat  instruction  until  condition
```

Ce mot **condition**, représente un opérateur relationnel, tel que défini ci-dessous :

Conditions	Significations
==	Égal
~=	Différent de
<	Plus petit que
>	Plus grand que
<=	Plus petit que ou égal
>=	Plus grand que ou égal

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Le résultat de ces opérateurs est toujours true ou false.

Bien évidemment vous ne pouvez comparer que ce qui est comparable, et : "0" == 0 sera évalué à false.
("0" est de type string, alors que 0 est de type nombre.)

Vous pouvez aussi, utiliser l'opérateur logique de négation not, qui retourne toujours true ou false.

Les autres opérateurs arithmétiques et logiques sont eux aussi utilisés, mais conjointement avec les opérateurs relationnels, par exemple :

```
if (x == 25) and (y >= 10) then ici votre code end
if (z ~= "salut") or (z ~= "bye") then ici votre code end
```

Petit rappel sur les autres opérateurs :

Les opérateurs logiques sont : and, or et not.

Les opérateurs arithmétiques sont :

- les binaires :
  - **+** (addition),
  - **-** (soustraction),
  - ***** (multiplication),
  - **/** (division),
  - **%** (modulo),
  - **^** (élévation) ;
- et l'unaire :
  - **-** (négation).

Il existe encore deux autres opérateurs dont on reparlera plus loin, lorsque vous aborderez le chapitre sur les « *strings* ».

Il s'agit de **length** qui détermine la longueur d'une chaîne de caractères et qui est représenté par l'opérateur unaire **#** (dièse) et l'opérateur de concaténation (..).

4-c - if ... then ... else ... end

Il s'agit d'un test conditionnel, qui doit se lire de la façon suivante :

si ... *la condition est remplie* ... **alors**
... *on effectue le travail suivant* ...
sinon ... *la condition n'est pas remplie.*
... *on effectue ce travail ci* ...
fin ... *fin .*

Mais revenez un instant, à la notion de « portée lexicale ».

À l'intérieur de la structure de contrôle suivante, regardez où se situent les différents « blocs » ?

```
bloc0 : {
  if condition1 then bloc1 : (ici votre code 1)
  elseif condition2 then bloc2 : ((ici votre code 2))
  else bloc3 : (((ici votre code 3)))
  end
}
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

Les variables locales définies dans chaque bloc, ne seront accessibles qu'à l'intérieur du bloc qui les contient.

Les variables (locales ou globales) définies à l'extérieur du bloc0 if **...** end, seront accessibles dans n'importe quel autre bloc, le 1, 2 ou 3.

elseif permet de faire plusieurs tests les uns à la suite des autres dans une même boucle if **...** end. C'est la même chose que plusieurs boucles successives, mais c'est plus compact, plus clair et ça limite le nombre de end.

Et pour terminer ce sous-chapitre, un petit extrait d'un programme quelconque, simplement en tant qu'exemple et pour illustrer ces propos :

```
if document.index < index then
  -- Si condition inférieure on fait ça

elseif document.index == index then
  -- Si condition égale on fait cela

elseif document.index > index then
  -- Si condition plus grand on fait ceci

else
  -- Et en dernier ressort, (certainement un « bug » ?)
  -- alors on fait ça

end
```

4-d - while ... do ... end

La boucle while exécute les instructions, tant que la condition est vérifiée.

```
while ... condition ...  do
  -- ... ici votre code ...
end
```

Lua teste en premier la *condition* :

- si la condition est true, Lua exécute de nouveau la boucle ;
- si la condition est false la boucle prend fin ;
- un break peut être utilisé pour sortir plus tôt de la boucle.

ATTENTION à bien coder cette boucle, car on a vite fait de se retrouver dans une boucle qui ne se termine jamais ... surtout avec une condition booléenne.

Un premier exemple de boucle arithmétique :

```
local i  = 0
while i <= 20 do  -- tant que i reste inférieur ou égal à 20
  Affiche(i)  -- on fait ce que l'on a à faire.
  -- (ici, on appelle la fonction Affiche() et on lui passe le paramètre i)
  i = i + 1  -- on incrémente i
  -- et on reboucle tant que i reste < ou = à 20
end
```

Dans cet exemple, dès que la variable i est supérieure à 20 (i = 21) la condition devient false et la boucle s'arrête.

Un deuxième exemple de boucle avec une condition booléenne :

```
function Recherche()
  local i = 1
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
  while true do
    local name = auraFunction(unit, i) -- cette fonction retourne un nom
    if (not name) then
      return false  -- s'il n'y en a pas on sort et on retourne false
    end
    if (auraName == name) then
      return true  -- si le nom est celui que l'on cherche on sort et on retourne true
    end
    i = i + 1  -- et on incrémente i tant qu'on a pas trouvé ce que l'on cherche
  end
end
```

Dans cet exemple, la condition reste true en permanence, mais l'on sort dès que l'on a trouvé ce que l'on cherche.

Il s'agit d'une fonction extraite d'un programme, que vous ne pouvez pas tester ici. À vous d'imaginer quelque chose de similaire ... histoire de voir si vous avez bien assimilé ce qui précède.

4-e - repeat ... until

La boucle repeat exécute les instructions, tant que la condition n'est pas satisfaite.

```
repeat
  -- ... ici votre code ...
until  condition
```

Dans la boucle repeat, ce n'est pas le mot-clé until qui met fin au bloc, mais ce dernier prend fin après la *condition :* repeat ... [bloc : ... until ... condition].

Ainsi, la condition peut se référer à des variables locales déclarées à l'intérieur du bloc.

```
local i = 0
repeat
  i = i + 1
  Affiche(i)
until i == 21
```

Explication :

Tant que **i** n'est pas égal à 21, l'opération se répète. La fonction **Affiche(i)** affiche donc 21 numéros, puis s'arrête.

L'on voit bien que dans while, la condition est à l'intérieur de la boucle, alors que dans repeat, la condition est à l'extérieur.

Là aussi, un break peut être utilisé pour sortir plus tôt de la boucle.

4-f - return et break

Les instructions break et return autorisent un saut en dehors du bloc dans lequel elles se trouvent.
Elles ne peuvent donc être écrites, que comme étant la dernière déclaration d'un bloc.

Vous utiliserez un break pour finir prématurément une boucle.
Cette instruction doit impérativement se trouver à l'intérieur de la boucle. (*for, repeat ou while*).

Après cette instruction, le programme continue immédiatement après la fin de la boucle.

L'instruction return, retourne occasionnellement le résultat d'une fonction ou la termine simplement.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Chaque fonction dispose d'un return implicite qui vous dispense d'en rajouter un.

Un dernier petit exemple pour terminer ce sous-chapitre :

```
-- Pour rechercher un caractère par son code ASCII
if (car == 40) or (car == 41) then
  -- on fait ici ce que l'on a à faire
  return car
else
  break
end
```

Ce qui peut se traduit par :

si ... on a trouvé ce que l'on cherche ... alors
... on fait ce que l'on a à faire et on sort du test en retournant la valeur de **car**, que l'on a trouvé (40 ou 41).
sinon ... on sort (break) du test conditionnel sans rien faire et le programme continue directement après la fin du test.
fin

4-g - À retenir

1. *ne pas confondre : **=** qui signifie transfert et **==** qui signifie égal ;*
2. *les opérateurs relationnels servent de conditions aux boucles ;*
3. *faire très attention à la portée des variables ;*
4. *while ... condition ... do ... faire ... end : la condition est à l'intérieur de la boucle ;*
5. *repeat ... faire ... until ... condition : la condition est à l'extérieur de la boucle ;*
6. *break : permet de sortir d'une boucle, avant la fin ;*
7. *return : retourne le résultat d'une fonction.*

5 - La boucle « for »

5-a - for ... do ... end

La boucle for j = 1, 5 do ... end, se lit de la façon suivante :

pour ... chaque valeur de j, de 1 à 5 ... faire
... *ici votre code* ...
fin

Souvenez-vous du sens du signe **=** qui signifie : « transfert ». La variable locale (à la boucle) **j**, prendra successivement toutes les valeurs de 1 à 5 et votre code sera exécuté pour chaque valeur de **j**.

La boucle prendra fin, lorsque **j** aura atteint sa valeur max.

*La déclaration for possède deux formes : Une forme **numérique** et une forme **générique**.*
*La boucle **for numérique**, répète un bloc de code, suivant une progression arithmétique de sa variable de contrôle.*
*La boucle **for générique** est utilisée pour effectuer certains travaux sur les fonctions, appelées itérateurs et expliqué dans **un chapitre qui leur est consacré**.*

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

5-b - La boucle « for numérique »

La boucle « for numérique » à la syntaxe suivante :

```
for valeur = e1, e2, e3  do
  -- bloc (ici vous traitez l'information)
end
```

- e1 représente la variable qui sera incrémentée à chaque tour ;
- e2 représente la limite de fin de boucle ;
- e3 facultatif, représente le « pas » de boucle. Si e3 est omis le pas sera égal à 1.

Les trois expressions de contrôle sont évaluées une fois, avant que la boucle ne commence. Elles doivent être des chiffres ou des nombres.

Vous pouvez aussi utiliser break pour sortir d'une boucle for.

*La variable de boucle est **locale à la boucle**.*
Il n'est donc pas nécessaire de la déclarer local.
Vous ne pourrez plus utiliser sa valeur après le « end » de la boucle, un break ou un return.
*Si vous avez besoin de cette valeur, il vous faudra l'affecter à une autre variable. Sous forme d'une variable **globale** à l'intérieur de la boucle et avant d'en sortir ou **locale** avant d'y entrer.*

Exemple d'une simple boucle :

```
for i = 1, 10 do
  Affiche(i)
end
```

Exemple d'une boucle inverse, qui décompte par « pas » de -1 :

```
for i = 32, 0, -1 do
  local b = 2^i  -- 2 à la puissance i
  Affiche(i, b)
end
```

5-c - La boucle « for générique »

La boucle for générique est utilisée pour effectuer certains travaux sur les fonctions, appelées **itérateurs**.

Elle a la syntaxe suivante : recherche de variables dans une table.

```
for clef, valeur in pairs(table) do ... bloc ... end
ou
for index, valeur in ipairs(table) do ... bloc ... end
```

Mais en attendant de voir le **chapitre consacré à ces itérateurs**, souvenez-vous qu'un itérateur est une construction qui permet de répéter (itérer) une même instruction sur les différents éléments d'un ensemble d'articles.

Cela permet, par exemple de lire tous les éléments d'une table ou d'une liste, très RAPIDEMENT.

À chaque itération, la fonction est appelée à produire une nouvelle valeur.

Elle s'arrêtera lorsque cette nouvelle valeur sera égale à nil.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

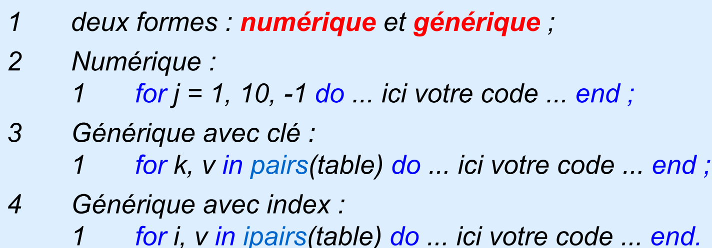
Ci-dessous un exemple : comment afficher les éléments d'une table.

```
local maTable = {1, 25, "trois", 255, "banane", "fraise", 12, "ceci est un exemple", 34586}

for k, v in pairs(maTable) do
  Affiche(k, v)
end
```

Tout comme précédemment, *clef* ou *index* et *valeur* sont locales à la boucle et vous pouvez aussi utiliser un break ou un return pour sortir plus tôt de la boucle.

5-d - À retenir



1. *deux formes : **numérique** et **générique** ;*
2. *Numérique :*
   1. *for j = 1, 10, -1 do ... ici votre code ... end ;*
3. *Générique avec clé :*
   1. *for k, v in pairs(table) do ... ici votre code ... end ;*
4. *Générique avec index :*
   1. *for i, v in ipairs(table) do ... ici votre code ... end.*

6 - Les fonctions

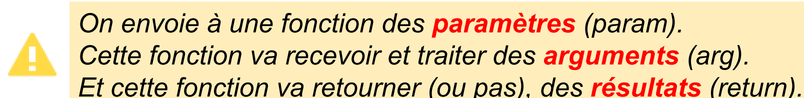
6-a - Qu'est-ce qu'une fonction ?

Une fonction est un terme générique qui désigne une partie d'un programme indépendant qui peut être appelée par un autre programme ou par elle-même, dans ce cas on dit qu'il y a récursivité.

Une fonction (*aussi appelé « routine »*), est une portion de code représentant un bloc et contenant un sous programme.

Suivant le langage utilisé et plus généralement en POO, l'on parlera aussi de méthode.

Mais pour ce qui nous intéresse ici, le langage Lua, nous parlerons tout simplement de fonction.



*On envoie à une fonction des **paramètres** (param).*
*Cette fonction va recevoir et traiter des **arguments** (arg).*
*Et cette fonction va retourner (ou pas), des **résultats** (return).*

La syntaxe pour définir une fonction est :

```
1/ Définition :
    function Nom_de_votre_fonction(arg1, arg2, ...)
      -- ici le corps de la fonction
    end

2/ Appel :
    Nom_de_votre_fonction(param1, param2, param3, param4)
```

Une fonction est une expression exécutable, dont la valeur est de type fonction.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Les déclarations suivantes...	...peuvent aussi s'écrire.
function f() ... *corps* ... end	f = function() ... *corps* ... end
function t.a.b.c.f() ... *corps* ... end	t.a.b.c.f = function () ... *corps* ... end
local function f() ... *corps* ... end	local f=nil ; f = function() ... *corps* ... end

***ATTENTION :** Vous devez d'abord définir la variable f(local f = nil), ensuite vous lui affectez la fonction. local f = nil ; f = function() ... end*
***Mais surtout, ne faites pas :** local f = function() ... corps ... end.*

Trois petits points dans la liste de arguments (**...**), informent *votre fonction* qu'elle est susceptible de recevoir d'autres arguments dont vous ne connaissez pas encore le nombre. (**Voir à ce sujet, le paragraphe sur les arguments multiples.**)

Les paramètres agissent en tant que variables locales et sont initialisés avec les valeurs de l'argument.

Les résultats d'une fonction seront retournés à l'aide de la déclaration return.
Si le contrôle atteint la fin de la fonction, sans rencontrer de déclaration return, alors la fonction se termine, mais sans renvoyer de résultat.

Lua peut appeler des fonctions écrites en Lua et des fonctions écrites en C.

Toutes les bibliothèques standards Lua sont écrites en C et comprennent des fonctions pour la manipulation des chaînes de caractères (string), la manipulation des tables, la manipulation des entrées/sorties (In/Out), le système d'exploitation (OS), les fonctions mathématiques et le débogage.

Il y a aucune différence, entre appeler une fonction Lua, ou une fonction C.

Un simple exemple :

```lua
-- la fonction Calcul reçoit 2 arguments (x et y)
local function Calcul(x, y)
  local total = x * y

  print("total = "..total)
end

-- Affectation multiple
local param1, param2 = 25, 13

-- On appelle la fonction Calcul en lui passant 2 paramètres
Calcul(param1, param2)
```

Les variables *param1* et *param2*, existent dans la fonction sous la forme des 2 arguments **x** et **y** qui disparaissent à la sortie de la fonction, tout comme la variable locale **total**.

6-b - Fonction, méthode, self, this ...

Comme vu précédemment, suivant le langage utilisé et plus particulièrement en *Programmation Orientée Objet*, vous utiliserez le terme de « méthode » pour désigner un sous-programme qui travaille sur l'objet d'une classe donnée ou de ses descendants.

En dehors d'une POO, vous utiliserez le terme de « fonction ».

Une fonction peut retourner un résultat alors qu' une méthode ne retourne rien.

Comment faire pour appeler une méthode ?

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

En utilisant l'opérateur deux points (:).

Considérez que bouton est un objet et que SetLabel() est une méthode attachée à cet objet.

Pour appeler SetLabel() vous procéderez de la façon suivante : bouton:SetLabel("nomBouton").

Comment faire pour appeler une fonction ?

Tout simplement en la nommant : NomFonction(param1, param2, param3).

« self », c' est quoi ?

En Programmation Orienté Objet, Lua considère chaque objet défini comme une table.

self , est par convention, le nom de la table en utilisation.

self est le nom donné à une variable locale, utilisée en POO, pour représenter l'objet défini.

C'est une variable locale, cela signifie qu'elle est extrêmement rapide d'accès et qu'elle n'existe plus en dehors de la méthode.

self peut être utilisé en tant que convention établie, pour simplifier les écritures.

Lors de l'appel d'une méthode, (*de l' appel d' un sous-programme sur un objet*), l'objet précédemment défini peut être représenté par « self ».

Pourquoi ne faut-il pas utilisez « this » ?

this n'est pas un terme Lua , c'est une variable globale, utilisée dans certains jeux vidéo tel que *World of Warcraft* , pour distribuer les « events » (événements).

Pour ceux qui seraient intéressés par le sujet et qui souhaitent approfondir leurs connaissances, rendez-vous sur le site ***wxLua et wxWidgets : Mode d'emploi*** *pour étudier des exemples d'utilisation des classes et des méthodes de wxWidgets avec Lua.*

6-c - return, fausse variable et unpack

Les fonctions écrites en Lua peuvent retourner plusieurs résultats les uns à la suite des autres. Il suffit de tous les énumérer après le mot clé return.

Encore faut-il ne pas oublier de spécifier à la fonction, où elle doit retourner ce qu'on lui demande de retourner.

Généralement cela se passe de la façon suivante :

```
function NomFonction(arg1, arg2)
  -- Corps de la fonction qui calcule x et y
  return x, y
end
local var1, var2 = 0, 0  -- Affectation multiple

var1, var2 = NomFonction(param1, param2)  -- Appel de la fonction
```

Les variables *var1* et *var2* attendent chacune une valeur en retour de la fonction. Elles recevront respectivement **x** et **y**. (var1 = x et var2 = y).

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

RÈGLE N° 1 :
Lua ajuste le nombre de résultats retournés, en fonction des circonstances de l'appel.

```
function Fonction1() end  --> ne retourne rien
function Fonction2() return R1 end  --> retourne 1 résultat
function Fonction3() return R1, R2 end  --> retourne 2 résultats
```

RÈGLE N° 2 :
La fonction produit autant de résultats que nécessaire, afin de correspondre aux variables.

```
local x, y, z = 0, 0, 0
x, y = Fonction3()          --> x = R1 et y = R2
x = Fonction3()          --> x = R1 et R2 sera oublié
x, y, z = 10, Fonction3()     --> x = 10, y = R1 et z = R2
```

RÈGLE N° 3 :
Une fonction qui n'a pas de résultat, retourne nil.

```
local x, y, z = 0, 0, 0
x, y = Fonction1()    --> x = nil, y = nil (la fonction n'a pas de return)
x, y = Fonction2()    --> x = R1, y = nil (la fonction a 1 seul return)
x, y, z = Fonction3() --> x = R1, y = R2, z = nil (il n'y a pas de 3e return)
```

RÈGLE N° 4 :
Si le résultat retourné n'a pas de variable, il sera oublié.

```
local x, y = 0, 0
x, y = Fonction2(), 20        --> x = R1, y = 20
x, y = Fonction1(), 20, 30  --> x = nil, y = 20 et 30 est oublié
```

RÈGLE N° 5 :
Quand une fonction appelée est le dernier argument, (ou le seul argument) d'une autre fonction (print), tous les résultats de cette fonction appelée seront considérés comme arguments.

```
print(Fonction1())              -->
print(Fonction2())          --> R1
print(Fonction3())              --> R1    R2

print(Fonction3(), 1)         --> R1    1
print(Fonction3(),.. "x")     --> R1x     (c'est la concaténation de Fonction3 et de "x")
```

Quand la fonction (ici Fonction3()) apparaît à l'intérieur d'une expression, (les 2 derniers exemples) Lua ajuste le nombre de résultat à 1. Ainsi dans la dernière ligne, seul R1 sera utilisé pour la concaténation.

RÈGLE N° 6 :
Un constructeur de table, peut lui aussi recevoir tous les résultats d'un appel à fonction sans aucun ajustement.

```
tableA = {Fonction1()}     --> tableA = {} (tableau vide)
tableA = {Fonction2()}     --> tableA = {R1}
tableA = {Fonction3()}     --> tableA = {R1, R2}
```

RÈGLE N° 7 :
Et pour terminer, la déclaration return f() qui retourne toutes les valeurs fournies par f.

```
function Fonction(arg)
  if arg == 0 then return Fonction1()
  elseif arg == 1 then return Fonction2()
  elseif arg == 2 then return Fonction3()
  end
end
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
print(Fonction(1))      --> R1
print(Fonction(2))      --> R1  R2
print(Fonction(0))      -- (aucun résultat)
print(Fonction(3))      -- (aucun résultat)
```

6-c-1 - La fausse variable

Lorsqu'une fonction retourne plusieurs valeurs et que vous ne souhaitez en conserver qu'un nombre particulier, la solution consiste à utiliser ce qu'on appelle une « fausse variable » définie par un trait de soulignement. (_)

```
local _, x, _, _, y, z = NomDeVotreFonction()
```

Manifestement, cette fonction est prévue pour retourner 6 variables, mais vous ne souhaitez conserver que la 2e, 5e et 6e.

6-c-2 - La fonction « unpack »

Cette fonction spéciale, écrite en C, reçoit un tableau et retourne comme résultat tous les éléments de ce tableau à partir de l'index de départ, l'index 1.

La fonction *unpack*permet d'appeler n'importe quelle fonction avec n'importe quel argument.
Si vous voulez appeler la variable function nomFonction() avec comme argument, les éléments situés dans un tableau, il vous suffit de faire : nomFonction(*unpack*(nomTableau))

L'appel à *unpack* retourne toutes les valeurs du tableau *nomTableau*, valeurs qui deviendront les arguments de *nomFonction()*.

```
local Recherche = nil
Recherche = string.find
tableA = {"hello", "ll"}

print(Recherche(unpack(tableA)))     --> 3 et 4
```

Un dernier exemple, pour clore ce chapitre :

```
-- la fonction Calcul reçoit 2 arguments
local function Calcul(x, y)
  local a = x * y
  local b = x + y
  local c = x - y
  local d = x / y
  return a, b, c, d
end

-- Affectation multiple
 local param1, param2 = 25, 13
 local multi, add, sous, div = 0, 0, 0, 0

-- On appelle la fonction Calcul en lui passant 2 paramètres
-- Et on attend que la fonction retourne 4 résultats
 multi, add, sous, div = Calcul(param1, param2)

 print(multi, add, sous, div)
```

La fonction calcul, attend en retour les résultats « *multi, add, sous et div* ».

Tout ce qui est à l'intérieur de Calcul() n'existe plus et l'on récupère grâce au « *return a, b, c, d* » les variables locales précédemment définies. (*multi, add, sous, div).*

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

6-d - Les arguments multiples

Plusieurs « fonctions types » Lua, reçoivent un nombre variable d'arguments.
Par exemple, vous avez souvent appelé la fonction print, avec un ou deux arguments, voire plus :
print(x, y, z) ou print("x "..y.." z").

Quand une fonction est appelée, les arguments sont collectés sous forme d'un tableau auquel la fonction aura accès au travers du paramètre caché (arg).

La table **arg** possède un champ particulier, **n** qui indique le nombre d'arguments collectés.

```
function nom(a, b) end

nom(3)          -- le param(3) sera traité par la fonction
nom(3, 4)       -- les param(3 et 4) seront traités par la fonction
nom(3, 4, 5) -- les param(3 et 4) seront traités par la fonction
                -- mais pas le 3e param qui est égal à 5

function nom01(a, b, ...)
  print(a, b, ...)
end

nom01(3, 4, 5, 6) -- tous les param seront traités par la fonction
                  -- print affichera: 3 4 5 6
```

6-e - Les arguments nominatifs

La passation des arguments en Lua est **positionnelle**.

Le 1er argument prend la valeur du 1er paramètre, le 2e du 2e ...

Le 1er entré sera le 1er traité, donc le 1er sorti.

Cette passation de paramètres, peut être une aide précieuse, dans le cas d'une longue liste de paramètres à passer, comme pour la création d'une fenêtre « Windows » qui nécessite beaucoup de paramètres.

```
w = Window{x = 0, y = 0, width = 300, height = 200, title = "Lua", background = "blue", border
 = true}
```

*Dans le cas d'arguments **nominatifs** comme mentionné ci-dessus, la codification n'est pas :*
*w = Window**(**x = 0, y = 0, width = 300, height = 200, title = "Lua", background = "blue", border = true**)***
mais bien :
*w = Window**{**x = 0, y = 0, width = 300, height = 200, title = "Lua", background = "blue", border = true**}***
***Pas de parenthèses (), mais des accolades { }**.*

6-f - Les fonctions anonymes

*Avec Lua, **une fonction est une valeur** , au même titre que les nombres et les chaînes de caractères.*

À ce titre, elle peut être stockée comme une variable locale ou globale, une table, peut être passée comme argument et elle peut aussi être retournée par une autre fonction.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

Une fonction n'a pas de nom. Lorsque vous parlez de la fonction print(), vous parlez en fait *du nom de la variable qui contient la fonction*.

La définition d'une fonction, est une déclaration qui assigne une valeur de type function à une variable.

L'expression function(x) ... end est un constructeur de fonction, tout comme **{}** est un **constructeur de table**.

Le résultat d'un constructeur de fonction est appelé : « fonction anonyme ». (C'est-à-dire qui n'a pas de nom !)

Mais avec Lua, toute fonction est avant tout UNE FONCTION ANONYME ...

Rappelez-vous, dans le paragraphe définition, vous avez vu que :

```
function f() corps de la fonction  end  -- fonction nominative

  -- est équivalent à :

f = function() corps de la fonction  end  -- fonction anonyme
```

Une fonction anonyme est constituée d'un bloc de code réutilisable comme une fonction, appelable comme une fonction, mais sans nom ...

Mais, me diriez-vous, puisqu'elle n'a pas de nom, comment faire pour l'appeler ?

Bonne question. Eh bien c'est très simple, pour déclarer une fonction anonyme, il vous suffit de faire comme pour une fonction classique, mais sans indiquer de nom.

```
-- En l'affectant à une variable
local var1 = function() corps de la fonction end

-- Ou en la plaçant à l'intérieur d'une autre fonction
function MaFonction()
  FonctionLua(function(arg))
    -- ici se situe la fonction anonyme
  end)
end
```

C'est bien beau tout ça, me diriez-vous, mais à quoi ça sert ?

Concrètement, à rien. Il n'existe aucune opération que l'on fasse avec une fonction anonyme qu'on ne puisse faire avec une fonction dite « normale ». On utilise les fonctions anonymes par goût ou pour des raisons de style.

Les fonctions anonymes sont très pratiques pour créer des fonctions dites « *jetables » :* lorsque l'on a besoin d'utiliser une fonction qu'une seule fois.

On ne compte pas la réutiliser, donc inutile de la mettre à part : on la définit et on la passe en callback (4) tout de suite.

Si vous développez avec la bibliothèque graphique « wxWidgets », vous vous apercevrez que les fonctions anonymes sont très utilisées pour effectuer les connexions aux événements.

```
-- Connecte l'événement fermeture de la fenêtre.
frame:Connect(wx.wxEVT_CLOSE_WINDOW, function(event)
                frame:Destroy(); event:Skip()
              end)
```

L'exemple qui va suivre, utilise une fonction de tri, extraite de la bibliothèque Lua sur **les**.

```
-- Considérez le tableau suivant:
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
local maTable = {2, 6, 7, 9, 0, 3, 5, 1, 4, 8}

-- Pour ranger ces chiffres dans un ordre croissant, il suffit d'utiliser la
-- fonction suivante:
table.sort(maTable)
-- Cette fonction trie le contenu d'une table dans un ordre croissant et
-- renvoie le résultat directement dans la table d'origine.

-- Maintenant, afin de vérifier le nouveau classement de la table, vous
-- pouvez afficher son contenu.
-- Mais, vous ne pouvez pas passer directement par la fonction print(), car
-- elle renverrait le N° de la table et non son contenu.

-- Dans notre exemple, vous pouvez utiliser la fonction suivante :
print(table.concat(maTable))    --> 123456789

-- Les chiffres sont dans le bon ordre, mais difficilement lisible.
-- Pour améliorer cela:
print(table.concat(maTable, ", "))    --> 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Et si vous préférez voir s'afficher les chiffres dans un ordre décroissant et bien faite intervenir la « fonction anonyme ».

```
table.sort(maTable, function(t1, t2)
             return (t1 > t2)
           end)

print(table.concat(maTable, ", "))    --> 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
```

Quelques explications :

La fonction *table.sort*(table, comparaison) possède 2 arguments :

Le premier est le nom de la table.

Le deuxième argument est une fonction de comparaison.

Cette fonction de comparaison doit renvoyer une valeur booléenne indiquant si le premier argument doit être avant le deuxième argument dans l'ordre de tri. Et en absence du deuxième argument, le tri se fait dans un ordre croissant.

Une fonction qui possède une autre fonction comme argument, ici le tri, est appelée **fonction d'ordre supérieur**.

Une **fonction d'ordre supérieur** est un puissant mécanisme de programmation, tout comme l'utilisation des **fonctions anonymes** en tant qu'argument.

Rappelez-vous que les **fonctionsd'ordre supérieur** n'ont pas de droit particulier, elles sont une simple conséquence de la capacité que possède Lua pour gérer ses fonctions et que les fonctions écrites en Lua, peuvent être stockées dans des variables globales, des variables locales et dans les champs d'une table.

6-g - Les closures

Avant de continuer sur les fonctions, il me semble important de clarifier le terme de **closure** qui sera plusieurs fois utilisé dans les chapitres et exemples suivants.

Il est important, avant de poursuivre, de réviser les notions de variables, de portée lexicale, de bloc contenant et contenu ...

Définition :
Une **closure** (*en français, fermeture ou clôture*) est une fonction qui capture les références de variables libres (5) dans l'environnement lexical.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

À quoi ça sert ! À conserver la valeur d'une variable à un moment donné.

Explication :

Une variable possède deux composantes : un nom et une valeur.

Le principe d'une variable est que sa valeur peut varier d'un instant à l'autre.

Mais il arrive que l'on ait besoin de conserver cette valeur telle qu'elle est à un moment donné pour l'utiliser plus tard.

Pour ce faire, on va utiliser une **closure**.

```
function Calcul(nombre)
  function Ajouter(valeur)
    return nombre + valeur
  end
  return Ajouter
 end
```

Cette fonction, comme son nom l'indique, est prévue pour effectuer un « calcul ».

Elle va recevoir comme argument : *nombre*

À l'intérieur de la *fonction Calcul()*, on définit une *fonction Ajouter()*.

Vous remarquez que la variable *nombre* y est accessible, car elle a été définie en dehors de la *fonction Ajouter()*.

La *fonction Ajouter()* étant une variable, vous avez le droit de la retourner en tant que résultat de la fonction. (return Ajouter).

```
local x = Calcul(10)

print(x(2))         --> 12
print(x(5))         --> 15
print(x(9))         --> 19
```

La variable "x" contient la *fonction Calcul()* qui elle-même contient la *fonction Ajouter()*.

x est désormais une fonction dans laquelle la variable *nombre* existe encore. *nombre* sera toujours égal à 10.

Les paramètres de **x**, **2**, **5**, et **9**, sont passés comme arguments à la *fonction Ajouter*.

6-h - La portée lexicale

Quand une fonction est incluse dans une autre fonction, elle a pleinement accès aux variables locales de la fonction englobante : cette fonctionnalité est appelée **portée lexicale**.

Commençons par un exemple simple :

Supposez que vous ayez une liste de noms d'élèves et un tableau qui associe ces noms à des notes.

Vous voulez trier, dans un ordre décroissant, la liste des noms, selon leur note.

Vous pouvez effectuer cette tâche comme suit :

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
noms = {"Peter", "Paul", "Mary"}
notes = {Mary = 10, Paul = 7, Peter = 8}

-- Affichage des noms avant le tri
for k, v in ipairs(noms) do
  Affiche(k, v)
end

-- Le tri
table.sort(noms, function(n1, n2)
  return notes[n1] > notes[n2] -- compare les notes
end)

-- Affichage des noms après le tri
for k, v in ipairs(noms) do
  Affiche(k, v)
end
```

Maintenant, si vous voulez créer une fonction qui remplisse la même tâche :

```
noms = {"Peter", "Paul", "Mary"}
notes = {Mary = 10, Paul = 7, Peter = 8}

function Tri_Par_Note(noms, notes)
  table.sort(nom, function(n1, n2)
  return notes[n1] > notes[n2] -- compare les notes
  end)
end
```

Ce qui est intéressant dans cet exemple, c'est que la *fonction anonyme* utilisée pour trier, a effectivement accès aux paramètres *notes* qui appartiennent à la fonction Tri_Par_Note(noms, notes).

Dans cette fonction anonyme, *notes* n'est ni une variable globale, ni une variable locale. C'est une **variable locale externe** ou **upvalue.**

```
function newCounter()
  local i = 0
  return function() -- fonction anonyme
    i = i + 1
    return i
  end
end

h2 = newCounter()
print(h2())  --> 1
print(h2())  --> 2
```

La fonction anonyme, utilise la **upvalue i** , qui lui permet de conserver le compteur.

Car le temps d'appeler la fonction anonyme, la variable locale **i** est déjà hors de portée puisque la fonction qui a créé cette variable (newCounter) est rentrée dans sa phase return .

Mais Lua gère correctement cette situation en utilisant la notion de **closure** (ou clôture).

Rappelez-vous, une **closure** est une fonction supplémentaire qui permettra à la fonction anonyme, d'accéder correctement à ses **upvalue** .

En appelant une nouvelle fois *newCounter()* , il y aura création d'une nouvelle variable locale **i** , puis nous obtiendrons une nouvelle **closure** qui agira par-dessus la nouvelle variable locale **i** , reprenant si besoin est, la suite du premier compteur.

```
c2 = newCounter()
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
(c2())   --> 1
(h2())   --> 3
(c2())   --> 2
```

Dans cet exemple, le premier c2 et le deuxième c2 opèrent une **closure** différente au cours de la même fonction, en agissant indépendamment sur la variable locale **i**.

Les fonctions étant stockées dans des variables régulières, il est facile de redéfinir les fonctions prédéfinies.

Supposez maintenant, que vous souhaitiez redéfinir la fonction *sin()* afin de fournir un résultat en degrés à la place de radians.

```
oldSin = math.sin
math.sin = function(x)
  return oldSin(x * math.pi/180)
end
```

Observez ci-dessous, une façon plus « propre » de faire la même chose :

```
do
  local  oldSin = math.sin
  local k = math.pi/180
  math.sin = function(x)
               return oldSin(x * k)
             end
end
```

Cette fonctionnalité est aussi utilisée pour créer un environnement « sécurisé ».

Les environnements sécurisés sont essentiels, lors du lancement d'un programme exécuté sur un serveur au travers d'Internet.

L'exemple suivant, va redéfinir la fonction *io.open* (de la bibliothèque in/out), en utilisant des *closures* .

```
do
  local oldOpen = io.open
  io.open = function(filname, mode)
     if access_OK(filename, mode) then
        return oldOpen(filename, mode)
     else
        return nil -- accès refusé
     end
  end
end
```

Après cette redéfinition, il n'y a plus aucun moyen d'appeler la fonction *io.open* normalement :

io.open est maintenant une fonction *sécurisée* .

6-i - Les variables non-globales

Une conséquence évidente des fonctions Lua, est de pouvoir être stockées non seulement dans les variables globales, mais aussi dans le champ des tables et dans les variables locales.

La plupart des bibliothèques Lua utilisent ce mécanisme de fonctions contenues dans des tables. (*io.read*, *math.sin*, etc.).

Pour créer de telles fonctions dans Lua, vous devez utiliser la syntaxe habituelle pour les fonctions et les tables.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
Librairie = {}
Librairie.f1 = function (x, y) return x + y end
Librairie.f2 = function (x, y) return x - y end

-- ou de la façon suivante

Librairie = {
f1 = function (x, y) return x + y end,
f2 = function (x, y) return x - y end
         }
```

Lorsque vous enregistrez une fonction dans une variable locale, vous obtenez une fonction locale.

Une fonction locale est limitée au domaine du bloc qui la contient.

Ces définitions sont particulièrement utiles pour les *packages*. (paquets).

Lua gère chaque bloc comme une fonction et un bloc peut déclarer des fonctions locales, qui ne seront visibles qu'à l'intérieur du bloc.

La **portée lexicale** veille à ce que d'autres fonctions dans le package puissent utiliser ces fonctions locales.

```
local f = function (arg)
  ... ici votre code ...
end

local g = function (arg)

  f() -- local f est visible ici

end
```

Un point subtil se pose dans la définition des fonctions locales récursives.

```
local fact = function (n)
  if n == 0 then return 1
  else return n * fact(n-1) -- BUG
  end
 end
```

Lorsque Lua compile fact(n-1), dans le corps de la fonction, la variable locale *fact* n'est pas encore définie.

Par conséquent, cette expression appelle une variable globale, et non locale.

Pour résoudre ce problème, vous devrez d'abord définir la variable locale et ensuite définir la fonction.

```
local fact = nil
fact = function (n)
  if n == 0 then return 1
  else return n * fact(n-1)
  end
end
```

Maintenant, *fact* est à l'intérieur de la fonction qui fait bien référence à la variable locale.

Sa valeur, lorsque la fonction est définie, n'a aucune importance, au moment où la fonction s'exécute, *fact* a déjà la bonne valeur.

D'où l'importance de définir d'abord la variable « f » (local f = nil), puis de lui affecter la fonction.(f = function() ... end)

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
local function fact (n)
  if n == 0 then return 1
  else return n * fact(n-1)
  end
end
```

Bien entendu, l'exemple précédent ne fonctionne pas, puisque vous utilisez une fonction récursive indirecte.

Dans de tels cas, vous devez utiliser l'équivalent d'une déclaration anticipée explicite.

```
local f, g    -- déclaration anticipée
function g ()
    f()
end

function f ()
    g()
end
```

6-j - Les appels récursifs

Un appel récursif, est un appel qui peut se répéter un nombre indéfini de fois.

Ceci peut se passer, lorsqu'une fonction appelle une autre fonction, lors de sa dernière action.

L'exemple suivant est un appel récursif.

```
function f(x)
  return g(x) -- appel récursif
end
```

L'appel à g(x) est la dernière action de la fonction f(x), car après, il n'y a rien d'autre à faire : c'est donc bien « un appel récursif ».

Dans cette situation, le programme n'aura pas besoin de revenir à la fonction d'appel, (puisqu'il n'y a rien d'autre à accomplir) et de ce fait ne conservera aucune information dans la pile.

Puisqu'un appel récursif n'utilise pas d'espace de pile, cela signifie qu'il n'y a plus de limite de *boucle* et que la fonction ne sera jamais débordée.

Aussi, la fonction suivante, boucle sans fin et sans « *stack overflow* ».

```
function f1(n)
  if n > 0 then return f1(n - 1) end
end
```

Mais attention, ce qui suit n'est pas un appel récursif.

```
function f1(x
  g(x)
  return
end
```

Car après l'appel à g(x), la fonction f(x) a encore une chose à faire : un return.

Avec Lua seul un appel return g(arg) est un appel récursif.

Puisque que g et ses arguments peuvent être très complexes : return x[i].f1(x[j] + a*b, i + j)

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

*L'appel récursif est similaire au **goto du basic**: un renvoi qui ne nécessite pas de retour automatique.*

6-k - À retenir

1. *une fonction est une expression exécutable, dont la valeur est de type fonction ;*
2. *on lui envoie des paramètres, elle reçoit des arguments et retourne des valeurs ;*
   1. *function NomFonction(arg1, arg2) ... corps de la fonction ... end ;*
3. *nomFonction(arg1, arg2, ...) Les 3 petits points attendent des arguments non encore définis ;*
4. *une fonction anonyme est une fonction qui n'a pas de nom :*
5. *function NomFonction() ... corps ... end est équivalent à : nomFonction = function() ... corps ... end*
6. *local x, y, _, _ = NomFonction(a, b) : NomFonction attend en retour les valeurs x et y, mais n'a pas besoin des 2 dernières(_)*
7. *une **closure** est une fonction qui capture les références de variables libres dans l'environnement lexical.*
8. *quand une fonction est incluse dans une autre fonction, elle a pleinement accès aux variables locales de la fonction englobante, cette fonctionnalité est appelée **portée lexicale** ;*
9. *définir d'abord la variable, puis lui affecter la fonction : loca l f = nil; f = function (n) ... end ;*
10. *un appel récursif est similaire au **goto du basic**, un renvoi qui ne nécessite pas de retour automatique.*

7 - Les itérateurs

7-a - Qu'est-ce qu'un itérateur ?

Un itérateur est une construction qui permet de répéter (itérer) une même instruction sur les différents éléments d'un ensemble d'articles.

Dans Lua, les itérateurs sont généralement représentés par des fonctions et chaque fois que vous appellerez cette fonction itératrice, elle retournera l'élément suivant d'une liste.

Tout itérateur a besoin de garder un état précis, entre les appels successifs, de façon à savoir où il en est et comment procéder à partir de là.

Les *closures* fournissent un excellent mécanisme pour cette tâche.

Rappelez-vous qu'une fermeture ou clôture (en anglais, closure) est une fonction qui accède à une ou plusieurs variables locales de sa fonction englobante.

Ces variables conservent leurs valeurs à travers les appels successifs à la clôture, permettant à celle-ci de se rappeler où elle en est dans le parcours.

Bien sûr, pour créer une nouvelle *closure*, vous devrez aussi créer ses variables locales externes. Par conséquent, une telle construction comporte généralement deux fonctions : la *closure* elle-même et la fonction qui créer la *closure*.

Un exemple valant mieux qu'un long discours :

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
function Iterateur(t)
  local i = 0
  local nbElements = table.getn(t)
  return function()
     i = i + 1
     if i <= nbElements then return t[i] end
  end
end
```

La fonction *Iterateur()* utilise la fonction Lua *table.getn*(t) qui compte le nombre d'éléments de la table (t) et retourne chaque élément de la dite table, t[i].

Contrairement à *ipairs*(), cet itérateur ne retourne pas l'index de chaque élément, mais seulement sa valeur.

Dans cet exemple, la fonction Iterateur(t) est la fabrique qui créé l'itérateur.

Chaque fois que vous l'appelez, elle créé une nouvelle clôture (l'itérateur lui-même).

Cette clôture conserve l'état de ses variables externes (t, i et n) de sorte que, chaque fois que vous l'appellerez, elle retournera la valeur suivante de la liste des t.

Quand il n'y a pas plus de valeur dans la liste, l'itérateur retourne nil.

Vous pouvez aussi utiliser ces itérateurs avec while.

```
t = {10, 20, 30}
iter = Iterateur(t)    -- création de l'itérateur
while true do
  local element = iter()    -- appel de l'itérateur
  if element == nil then break end
  Affiche(element)
end
```

Cependant, il est beaucoup plus facile d'utiliser les for génériques qui ont été conçus pour ce genre d'itération.

7-b - Les « for » génériques

Le for générique s'occupe de tout, à partir d'une boucle d'itération.

for appelle l'*usine itératrice*, conserve la fonction itératrice interne, appelle l'itérateur à chaque nouvelle itération, et arrête la boucle quand l'itérateur renvoie nil.

Plus tard vous verrez que le for générique fait encore bien plus que ça.

Avec l'exemple précédent, cela donne :

```
t = {10, 20, 30}
for element in Iterateur(t) do
  Affiche(element)
end
```

Cela se lit de la façon suivante :
Pour *chaque élément* dans *la fonction Iterateur(t)* faire

Afficher(element)

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

fin

Comme exemple plus complexe, vous allez écrire un itérateur afin de parcourir tous les mots d'un fichier quelconque.

Pour faire cette traversée, vous aurez besoin de garder deux valeurs : la ligne actuelle et l'endroit où vous êtes dans cette ligne.

Avec ces données, vous pourrez toujours générer le mot suivant et pour le conserver, vous utiliserez deux variables locales externes, ***line*** et ***pos***.

```
function TousLesMots()
  local line = io.read()     -- ligne courante
  local pos = 1              -- position courante dans la ligne
  return function()          -- fonction d'itération
    while line do            -- on répète tant que ligne existe
      local s, e = string.find(line, "%w+", pos)
      if s then              -- si on trouve un mot?
        pos = e + 1          -- position suivante après ce mot
        return string.sub(line, s, e)  -- on retourne le mot trouvé
      else                   -- mot pas trouvé, on essaie la ligne suivante
        line = io.read()
        pos = 1              -- retour à la pos N°1
      end
    end
    return nil               -- plus de ligne = fin de la traversée
  end
end
```

La partie principale de la fonction itératrice est l'appel à la fonction Lua *string.find*(line, "%w+", pos).

Cet appel cherche un mot dans la ligne actuelle, à partir de la position actuelle.

Elle décrit un « mot » en utilisant le pattern "%w+", ce qui correspond à un ou plusieurs caractères alphanumériques.

Si elle trouve le « mot », la fonction met à jour la position actuelle sur le premier caractère après le « mot » et retourne ce « mot » grâce à la fonction Lua *string.sub*(line, s, e), qui extrait une sous-chaîne de la ligne (line) entre les positions données (s et e).

Sinon (else), l'itérateur lit une nouvelle ligne et répète la recherche.

S'il y a plus de lignes, la fonction d'itération retourne nil pour signaler la fin de la recherche.

Malgré son apparente complexité d'écriture, l'utilisation de la fonction *TousLesMots()* reste simple.

```
for mot in TousLesMots() do
  print(mot)
end
```

Cette situation, difficile à écrire mais facile à utiliser est commune aux itérateurs.

Mais ce n'est pas un gros problème en soi, car le plus souvent, vous n'aurez pas à définir les itérateurs.

Il existe deux fonctions Lua spécialement écrites pour ça. Il s'agit de : *pairs()* et *ipairs()* que vous verrez à la fin de ce chapitre.

Mais il est important de bien comprendre, comment tout ça fonctionne ... c'est un peu le but de ce tutoriel !

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

7-c - La sémantique des « for » génériques

L'inconvénient avec les itérateurs précédents est que vous devez créer une nouvelle clôture pour chaque nouvelle boucle.

Mais dans la plupart des situations ce ne sera pas un réel problème.

Dans l'exemple précédent, (avec l'itérateur *TousLesMots()*), le coût de la création d'une clôture simple est négligeable comparé au coût de la lecture d'un fichier entier.

Cependant, dans certaines situations ce surcoût peut être indésirable.

Et c'est là que les **génériques** interviennent.

Le générique conserve trois valeurs :

1 la fonction d'itération ;
2 un état constant ;
3 une variable de contrôle.

La syntaxe générale pour le for générique est la suivante :

```
for  liste   in   expression   do
  ... ici votre code ...
end
```

Où *liste* est une liste d'un ou plusieurs noms de variables, séparés par des virgules, et *expression* est une liste d'une ou plusieurs expressions, aussi séparées par des virgules.

Le plus souvent, la liste d'expressions ne possède qu'un seul élément.

Par exemple, dans le code :

```
for k, v in pairs(t) do
  print(k, v)
end
```

La liste des variables est **k** et **v**, la liste d'expressions possède des paires de l'élément (**t**).

La liste des variables peut aussi avoir une seule variable, comme dans :

```
for line in io.lines() do
  io.write(line, '\n')
end
```

Explication :

Ce code appelle la première variable de la liste. Sa valeur ne peut pas être nil pendant la boucle, parce que lorsqu'elle devient nil la boucle se termine.

La première chose que fait le for, est d'évaluer les expressions après le in.

Ces expressions aboutissent dans les trois valeurs conservées par le for :

1 la fonction d'itération ;

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

2 un état constant ;
3 une variable de contrôle.

Comme dans une affectation multiple, seul le dernier élément de la liste peut se traduire par plus d'une valeur, et le nombre de valeurs est ajusté à trois, les valeurs supplémentaires seront jetées ou initialisées à nil.

Après cette étape d'initialisation, for appelle la fonction itératrice avec deux arguments : l'état invariant et la variable de contrôle.

Puis for assigne les valeurs retournées par la fonction itératrice dans les variables déclarées par la liste de variables. (Et dans le cas présent : « line »)

Si la valeur retournée est nil, alors la boucle prend fin.

Sinon, for répète le processus.

7-d - Stateless iterator

La traduction française de « Stateless (6) iterator » est : « itérateur apatride ».

Un itérateur apatride, est donc un itérateur qui ne conserve aucun état par lui-même, ce qui permet d'éviter le coût de la création de nouvelles clôtures.

À chaque itération, la boucle for appelle sa fonction itératrice avec deux arguments : l'état invariant et la variable de contrôle.

Un itérateur apatride générera l'élément suivant pour l'itération en utilisant uniquement ces deux arguments.

Un exemple typique de ce genre d'itérateur est la fonction *ipairs*(), qui itère sur tous les éléments dans un tableau, comme illustré ci-dessous.

```
a = {"one", "two", "three"}
for i, v in ipairs(a) do
  print(i, v)    --> 1     one
                 --> 2     two
                 --> 3     three
end
```

L'état de l'itération est :

La table traversée (l'état invariant qui ne change pas au cours de la boucle) et l'index courant (la variable de contrôle).

7-e - pairs(t) et ipairs(t)

Tout ce qui a précédé, était là simplement pour essayer de vous faire comprendre la façon dont fonctionne un itérateur.

Sauf cas exceptionnel, vous ne devriez pas avoir à vous en servir, mais il est toujours bon de bien comprendre « le pourquoi du comment des choses ».

En résumé, deux fonctions Lua sont à votre disposition lorsque vous aurez besoin de parcourir une table ou une liste de noms.

Il s'agit des fonctions *pairs*() et *ipairs*() qui s'utilisent de la façon suivante :

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
for k, v in pairs(t) do ... ici votre code ... end

for i, v in ipairs(t) do ... ici votre code ... end
```

Cela se lit de la façon suivante :
Pour chaque élément (k ou i) dans la table(t) faire

... *ici votre code* ...

fin

- k = key (clé) (lors d'un index alphabétique)
- v = valeur
- t = table (ou liste de noms)
- i = index (lors d'un index numérique)

Chaque fonction va parcourir TOUTE la table (t), en relevant à chaque index ou chaque clé, la valeur qui va avec.

Dans le corps de la fonction, on fait ce dont on a besoin à partir de ces « clé-valeur » ou « index-valeur », comme dans l'exemple suivant :

```
if i == 25  then print(v) end
  ou
if k == "fleurs" then print(v) end
```

Exemple d'utilisation des deux fonctions d'itération :

pairs() (key-valeur) :
Cette fonction s'utilise dans le cadre de tables dont l'index est alphabétique ou numérique ou les deux ou lorsque l'on est pas certain de la nature de cet index. **L'affichage se fera dans un ordre indéfini**.

```
local table = {3, banane = "jaune", 10, pi = 3.14159, 17,
               fruit = "banane"}
for key, valeur in pairs(table) do
print(key, valeur)  --> 1       3
                    --> 2       10
                    --> 3       17
                    --> pi      3.14159
                    --> fruit   banane
                    --> banane  jaune
end
```

ipairs () (index-valeur) :
Cette fonction s'utilise dans le cadre de tables dont l'index est numérique et uniquement numérique . **L'affichage se fera dans un ordre croissant** .

```
local table = {3, "cinq", 10, "vingt-cinq", 17, 18}
for index, valeur in ipairs(table) do
Affiche(index, valeur)  --> 1    3
                        --> 2    "cinq"
                        --> 3    10
                        --> 4    "vingt-cinq"
                        --> 5    17
                        --> 6    18
end
```

Et n'oubliez pas :
pairs *= key, valeur - Ordre indéfini.*
ipairs *= index, valeur - Ordre indéfini.*

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Le i de ipairs est le même que celui de : index ...

7-f - À retenir

1. *un itérateur est une construction qui permet de répéter (itérer) une même instruction sur les différents éléments d'un ensemble d'articles ;*
2. *deux fonctions Lua sont à votre disposition lorsque vous aurez besoin de parcourir une table ou une liste de noms. Il s'agit des fonctions :*
    1. *pairs = key, valeur - Ordre indéfini,*
    2. *ipairs = index, valeur - Ordre croissant,*

- *k = key (clé) (lors d'un index alphabétique),*
- *v = valeur,*
- *t = table (ou liste de noms),*
- *i = index (lors d'un index numérique).*

À utiliser de la façon suivante :

```
for k, v in pairs(t) do ... ici votre code ... end

for i, v in ipairs(t) do ... ici votre code ... end
```

8 - Les tables

8-a - Qu'est-ce qu'une table ?

Tout d'abord, il faut bien comprendre ce qu'est une table aussi appelée *tableau* avec certains langages.

Lua utilise le mot **table** (table en anglais) et nom tableau (array en anglais).

Une table est un ensemble de cases appelées champs (field en anglais) dans lesquelles vous pouvez ranger :

1. des chaînes de caractères ;
2. des constantes ;
3. des variables ;
4. des fonctions ;
5. et aussi des tables, contenant éventuellement d'autres tables qui elles-mêmes ...

En fait, c'est la boîte qui est dans la boîte qui est ...

Imaginez une espèce de grande armoire avec un nombre infini de tiroirs, dans lesquels l'on pourrait ranger tout et n'importe quoi, y compris d'autres armoires avec autant de tiroirs que son contenant et qui eux-mêmes ...

Un tableur dans le style *Excel* ou *OpenOffice-Classeur* représente un excellent exemple de ce qu'est une *table.*

A	B	C	D
champ 1A	champ 1B	champ 1C	champ 1D
champ 2A	champ 2B	champ 2C	champ 2D
champ 3A	champ 3B	champ 3C	champ 3D
champ 4A	champ 4B	champ 4C	champ 4D

Une table est créée à l'aide d'un constructeur, qui est défini par 2 accolades : **{ }**.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Exemple de construction d'une table vide, ayant pour nom **maTable**, pouvant contenir x champs : maTable = { }
Initialisation des champs de la table : maTable = {"Z", 1515, f(x), maVariable, etc.}

L'image de *maTable* est équivalente à :

maTable = {	"Z"	1515	f(x)	maVariable	etc.	}

Notez au passage, que les virgules (,) représentent les bords des champs.

Avec LUA, l'indexation, ou la numérotation des champs, ne commence pas à 0, mais à 1.

8-b - Les tables « made in Lua »

1 **Une table est un OBJET** et le programme ne manipule que des références (pointeurs) ;
2 Une table est toujours anonyme ;
3 Il n'existe aucune relation entre la variable **maTable** et la table elle-même ;
4 Si la variable **maTable** n'existe plus, LUA efface la table de la mémoire ;
5 La dimension des tables n'est pas statique. Elle s'accroît en fonction des besoins ;
6 Comme pour toutes les variables, si un champ de la table (ou case) n'a pas été initialisé, elle est évaluée à nil.

Voici un exemple :

```
maTable = {} -- création d'une table

k = "x" - affectation de la chaîne "x" à la variable k

maTable[k] = 10 -- nouvelle entrée, avec la clé(k) = "x" de valeur = 10

maTable[20] = "grand" - nouvelle entrée, avec la clé(k) = 20 de valeur = "grand"

print(maTable["x"]) ---> 10 (c'est la clé k qui est égal à 20)

print(maTable[k])     ---> grand
```

Il existe deux façons de représenter l'écriture d'une table :

1 maTable.variable = 10 qui est équivalent à : maTable["variable"] = 10
2 print(maTable.variable) qui est équivalent à : print(maTable["variable"])

Les deux formes sont équivalentes, mais **ATTENTION**, "variable" DOIT-ETRE une chaîne de caractères et pas un nombre.

Faire attention aussi au fait que, **1**, **"1"**, **"01"** représentent des valeurs différentes, donc des emplacements différents à l'intérieur de la table.

8-c - Les constructeurs

Les constructeurs **{}**, sont utilisés pour créer et initialiser les tables et tout se fait automatiquement.

Dans la déclaration suivante : jours = {"lundi", "mardi", "mercredi"}.
jours[1] sera initialisé avec la chaîne "lundi", jours[2] avec la chaîne "mardi", etc.

Toutes les tables sont créées de la même façon, seul les constructeurs affectent leur initialisation.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

Vous pouvez bien évidement décider de l'ordre : maTable = {[1] = "rouge", [2] = "vert", [3] = "bleu"}.

Les différents champs, (ou éléments de la table) sont séparés par une **virgule**.
maTable = {x = 10, y = 45, "one", "two", "three"}.

8-d - Les matrices et les tables

Pour faire simple, **une matrice est un tableau de nbLignes et nbColonnes**.

	Colonne 1	Colonne 2	Colonne 3	Colonne 4	Colonne 5	Colonne 6	Colonne n
Ligne 1							
Ligne 2							
Ligne x							

La déclaration d'une matrice en LUA est différente de la déclaration d'une matrice avec un autre langage, comme le C ou le Pascal.

Il y a deux façons principales de créer une matrice en Lua.

- Utiliser une table de table ;
- Rassembler les 2 indices en un seul.

8-d-1 - Utiliser une table de table

C'est peut-être la plus utilisée, car à mon avis la plus évidente.

Il s'agit d'une table où chaque élément est une autre table.

Par exemple, vous pouvez créer une matrice de **1** à **n** dimensions par **m** avec le code suivant :

```
matrice = {}      -- création de la matrice

for i = 1, n do
 matrice[i] = {} -- création d'un nouveau rang
 for j = 1, m do
   matrice[i][j] = valeur  -- création d'une case à l'intérieure du rang
 end
end
```

Et pour remplir le tableau donné en exemple ci-dessus :

```
local tablePrincipale = {} -- création de la table
local nbLignes = 3
local nbColonnes = 7

for ligne = 1, nbLignes do
  tablePrincipale[ligne] = {} -- création d'une nouvelle ligne
  for colonne = 1, nbColonnes do
    tablePrincipale[ligne][colonne] = tostring(ligne)..tostring(colonne)
  end
end
```

Ce qui donne :

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

	Colonne 1	Colonne 2	Colonne 3	Colonne 4	Colonne 5	Colonne 6	Colonne n
Ligne 1	11	12	13	14	15	16	1n
Ligne 2	21	22	23	24	25	25	2n
Ligne x	x1	x2	x3	x4	x5	x6	xn

tostring(x), est une fonction de base Lua, qui comme sont nom l'indique, transforme (x) en chaîne de caractères.

tonumber(y), est son contraire et transforme (y) en nombre, si cela est possible.

Puisqu'avec LUA, **les tables sont des objets**, vous devrez créer les rangées de façon explicite.

L'exemple précédent, pourrait s'écrire de la façon suivante :

```
table = {}          --Construction de la table principale
table.ligne1 = {}  --Construction de la ligne1 à l'intérieur de table
table.ligne2 = {}  --Construction de la ligne2 à l'intérieur de table
table.ligne3 = {}  --Construction de la ligne3 à l'intérieur de table
etc.
```

8-d-2 - Rassembler les 2 indices en un seul

Si les deux indices sont des entiers, vous pourrez multiplier le premier par une constante, puis ajouter le second.

Reprenez l'exemple précédent :

```
mt = {} -- création de la matrice (c'est toujours une table!)

for i = 1, n do
  for j = 1, m do
  mt[i * m + j] = 0
  end
end
```

Si les indices sont des chaînes, vous pourrez créer un indice unique en concaténant de deux indices et en les séparant par un caractère.

Soit une matrice *mt* et 2 indices **s** et **t**. Ce qui donnerait : mt[s..":"..t] ou mt["s:","t"].

8-e - Les listes chaînées

Bien que ce ne soit pas le but de ce tutoriel, un petit rappel sur les listes chaînées me semble indispensable.

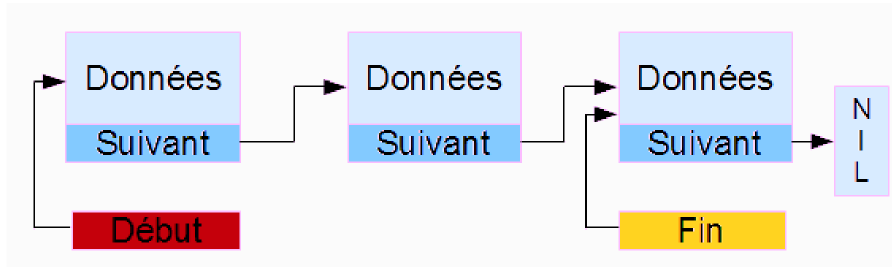
Ce sujet sera abordé succinctement, mais si vous souhaitez l'approfondir n'hésitez pas à vous rendre sur le web où vous pourrez trouver plusieurs sites, traitant plus en profondeur ce sujet.

8-e-1 - Qu'est-ce qu'une liste chaînée ?

Une liste chaînée est une structure informatique qui permet la sauvegarde dynamique de données en mémoire tout comme des variables ou tableaux mais sans se préoccuper de leur nombre et en rendant leur allocation transparente.

On dit liste chaînée, car les données sont chaînées les unes avec les autres. (Voir schéma ci-dessous.)

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.



On accède aux données à l'aide d'un ou deux points d'entrées qui se situent la plupart du temps aux extrémités de la liste.

8-e-2 - Différents types de listes

Il existe trois types de listes chaînées dépendant de la manière dont on se déplace dans la liste :

1. les listes chaînées simples ;
2. les listes doublement chaînées ;
3. les listes circulaires.

La liste chaînée simple est la liste de base, dont chaque élément appelé nœud, contient deux parties :

1. une partie contenant l'information proprement dite ;
2. une partie appelée pointeur qui lie le nœud précédent au nœud suivant.

8-e-3 - Et avec Lua ?

Vous aurez rarement besoin de ces structures, car il existe, avec Lua, un moyen plus simple pour représenter vos données sans l'aide de listes.

Il suffit de représenter une pile à l'aide d'une table illimitée, avec n champs pointant vers le haut.

Or qui dit table Lua, dit table dynamique, donc taille illimitée ...

En effet, les tables Lua étant des entités dynamiques, il est facile d'implémenter des listes chaînées avec ce type de langage.

Chaque nœud serait représenté par une table où les liens seraient tout simplement les champs des tables qui contiendraient des références (pointeurs) à d'autres tables.

Par exemple, pour réaliser une liste basique, où chaque nœud serait composé de deux champs, **next** et **value**.

Création de nouvelles listes :

```
local list = nil     -- initialisation de la variable
for line in io.lines do
  list = {next = list, value = v}
end
```

Ce code lit les lignes de l'entrée standard et les stocke dans une liste liée, dans l'ordre inverse.

Chaque nœud de la liste est une table avec deux champs : le contenu en ligne **value** et **next**, une référence au nœud suivant.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Le code suivant affiche le contenu de la liste :

```
-- Affichage du contenu de la liste:
local l = list
while l do
  print(l.value)
  l = l.next
end
```

Puisque la liste a été implémentée comme une pile, les lignes s'afficheront dans un ordre inverse.

La réalisation d'autres types de listes, comme les listes *doublement chaînées* ou les *listes circulaires*, est également possible.

8-f - Les files d'attente

Bien que vous puissiez utiliser les files d'attente, avec *table.insert*() et *table.remove*() de la bibliothèque table, il n'en reste pas moins vrai que cette méthode reste trop lente pour les grandes structures.

Une autre structure bien plus efficace consiste à utiliser deux indices : **first** pour le premier élément et **last** pour le dernier.

```
function ListNew()
  return {first = 0, last = -1}
end
```

Pour éviter de polluer l'espace global, vous allez définir toutes les opérations de liste dans une table appelée **list**.

L'exemple précédent devient :

```
list = {}
function list.new()
  return {first = 0, last = -1}
end
```

Maintenant, vous pouvez insérer ou retirer un élément aux deux extrémités.

```
function List.pushleft(list, value)
  local first = list.first - 1
  list.first = first
  list[first] = value
end

function List.pushright(list, value)
  local last = list.last + 1
  list.last = last
  list[last] = value
end

function List.popleft(list)
  local first = list.first
  if first > list.last then error("liste vide") end
  local value = list[first]
  list[first] = nil   -- à la corbeille
  list.first = first + 1
  return value
end

function List.popright(list)
  local last = list.last
  if list.first > last then error("liste vide") end
  local value = list[last]
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
  list[last] = nil    -- à la corbeille
  list.last = last - 1
  return value
end
```

Si vous utilisez cette structure de la file d'attente, d'une manière stricte, en appelant seulement **pushright** et **popleft**, à la fois **first** et **last** seront sans cesse augmentés.

Comme LUA représente les rangées à l'aide de tables, vous pourrez les indexer soit à partir de 1 à 20 aussi bien que de 16.777.216 à 16.777.236.

Et comme LUA utilise la double précision, pour représenter les nombres, votre programme pourra s'exécuter pendant deux cents ans, faisant un million d'insertions par seconde, sans rencontrer le moindre problème de débordement (overflow).

8-g - Comment filtrer certains mots définis ?

Supposez que vous souhaitiez lister tous les identificateurs utilisés d'un programme source.

Vous aurez alors besoin de filtrer tous les mots réservés de votre liste.

Certains programmeurs habitués au C pourraient être tentés de représenter l'ensemble des mots réservés sous forme d'un tableau de chaînes de caractères, puis de rechercher dans ce tableau les mots réservés inscrits dans la liste.

Pour accélérer la recherche, ils pourraient même utiliser un arbre binaire ou une table de hachage pour les représenter.

En Lua, un moyen simple et efficace, consiste à définir comme indice (index) les mots réservés que l'on recherche.

Il suffit alors de vérifier, lors d'un test, si le résultat est nil ou non.

Ce qui donne :

```
reserved = {["while"] = true, ["end"] = true,
["function"] = true, ["local"] = true,}

for w in pgrSource() do
  if reserved[w] then -- Si "w" est un mot réservé
    ... ici votre code ...
  end
end
```

while étant un mot réservé en Lua, il ne peut être utilisé en tant qu'identifiant.

Par conséquent, nous ne pouvons pas écrire while = 1, mais nous pouvons écrire : ["while"] = 1.

Et pour avoir une initialisation plus simple, il suffit d'utiliser une fonction auxiliaire pour construire l'ensemble.

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

-- Ce qui donne:
reserved = Set{"while", "end", "function", "local", }
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

8-h - Les tampons pour chaînes de caractères

Si vous voulez construire une chaîne de caractères au *coup par coup*, en lisant un fichier texte *ligne par ligne*, votre code ressemblera très certainement à ce qui suit.

```
ATTENTION: Ce code est désastreux!

local buff = ""
for line in io.lines() do
```

Malgré son air innocent, ce code écrit en Lua peut causer une perte de performance énorme, si vous traitez un gros fichier.

Il faut en effet environ une minute, pour lire un fichier de 350 Ko.

C'est pourquoi Lua fournit la fonction *io.read*() qui utilise un algorithme de collecte spécial, le garbage-collector, qui lira le fichier entier en 0,02 seconde.

Quand Lua s'aperçoit que le programme utilise trop de mémoire, il libère alors les emplacements qui ne sont plus utilisés.

Habituellement cet algorithme a une bonne performance, contrairement à la boucle utilisée ci-dessus qui elle, utilise un très mauvais algorithme.

Explication :

Pour comprendre ce qui se passe, vous allez supposer que vous êtes dans le milieu de la lecture de la boucle.

La variable **buff** est alors une chaîne de caractères de 50 Ko constituée d'une suite de lignes de 20 octets.

Lorsque Lua concatène *buff..line.."\ n"*, il crée une nouvelle chaîne de 50 020 octets, puisqu'il ajoute les 20 octets de la nouvelle ligne aux 50 Ko de *buff ...*

Autrement dit, pour chaque nouvelle ligne, Lua déplace 50 Ko de mémoire (et plus au fur et à mesure).

Après la lecture de 100 lignes nouvelles de seulement 2 Ko (100 fois 20 octets), Lua aura pris plus de 5 Mo de mémoire. (100 fois 50 Ko).

Et pour empirer les choses, l'affectation précédente *buff = buff..line.."\n"* devient après coup, une donnée inutile qui vient gonfler la mémoire.

Or, après 2 cycles, vous aurez donc 100 Ko de données inutiles en mémoire (2 fois 50 Ko)...

C'est à ce moment-là, que Lua décide de lancer son *garbage collector* et de libérer de la mémoire, ces 100 Ko qui ne servent plus à rien.

Le problème est que cela se produit tous les 2 cycles et que Lua gérera alors son *garbage collector* 2000 fois avant de lire le fichier en entier.

Même avec tout ce travail, l'utilisation de la mémoire sera quand même d'environ 3 fois la taille du fichier.

Pour les petites chaînes, la boucle ci-dessus est parfaite et pour lire un fichier entier, il suffira d'utiliser l'option ("*all") avec la fonction io.read(), qui lira le fichier dans son ensemble.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

8-h-1 - Et pour les gros fichiers ?

La seule solution pour les gros fichiers, consiste en la mise en place d'un algorithme plus efficace, comme celui qui va suivre.

La première boucle utilisait une approche linéaire du problème, à savoir : la concaténation de petites "strings" une par une dans un accumulateur.

Le nouvel algorithme utilisera une approche binaire, à savoir : la concaténation de plusieurs petites "strings" entre elles et la concaténation des chaînes résultantes de grande taille en plus grande encore ...

Le cœur de cet algorithme est une pile qui conserve les chaînes de grande taille déjà créées dans le fond de la pile, tandis que les petites chaînes rentreront par le haut.

Cet algorithme est similaire à celui bien connu de certains programmeurs, sous le nom de « la tour de Hanoï ».

Le principe étant qu'une chaîne de caractères ne peut jamais s'asseoir sur une chaîne plus courte.

Chaque fois qu'une nouvelle chaîne est poussée sur une plus courte, alors (et seulement alors) l'algorithme concatène les deux.

Cet enchaînement va créer une chaîne plus grande, qui pourra éventuellement, être plus grande que sa voisine de l'étage précédent. Si cela arrive, elles seront rejointes. Cet enchaînement descendra la pile jusqu'à ce que la boucle atteigne une chaîne encore plus grande, ou le bas de la pile.

```
function newStack()
  return {""}   -- démarrage avec une chaîne vide
end

function addString(stack, s)
  table.insert(stack, s)  -- push 's' dans la pile
  for i = table.getn(stack)-1, 1, -1 do
    if string.len(stack[i]) > string.len(stack[i+1]) then
      break
    end
    stack[i] = stack[i]..table.remove(stack)
  end
end
```

Pour obtenir le contenu final de la mémoire tampon, il suffit de concaténer toutes les chaînes vers le bas. La fonction *table.concat*() fait exactement cela, elle enchaîne toutes les chaînes de cette liste ...

Ce qui donne :

```
local s = newStack()

for line in io.lines() do
  addString(s, line.. "\n")
end
s = toString(s)
```

Ce nouveau programme permet de réduire les temps originaux.

Pour lire un fichier de 350 Ko, le programme passera de 40 secondes à 0,5 seconde.

L'appel à *io.read*() ("*all") est encore plus rapide et finit le travail en 0,02 seconde.

io.read() utilise exactement la structure de données que nous venons de présenter, mais écrite en C.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

Plusieurs autres fonctions dans les bibliothèques Lua utilisent également cette application écrite en C.

L'une de ces fonctions est *table.concat*().

table.concat(), collecte toutes les chaînes dans une table, puis les concatène toutes à la fois.

La fonction *table.concat*() accepte un second argument optionnel, qui est un séparateur.

L'utilisation de ce séparateur, dispense d'insérer un saut de ligne après chaque ligne. ("\n" qui doit être séparé par une virgule.)

```
local t = {}

for line in io.lines() do
  table.insert(t, line)
end

s = table.concat(t, "\n").. "\n"
```

L'itérateur *io.lines*() retourne chaque ligne sans saut de ligne.

table.concat() insère le séparateur entre les chaînes, mais pas après la dernière.

Vous devrez donc ajouter le retour à la ligne à la fin de cette dernière.

Cette dernière concaténation duplique la chaîne qui en résulte et qui peut être très grande.

Mais, il n'existe aucune option pour faire en sorte que *table.concat*() insère ce séparateur supplémentaire.

Mais vous pouvez le tromper, par l'insertion d'une chaîne vide supplémentaire en **t** :

```
table.insert(t, "")
s = table.concat(t, "\n")
```

La nouvelle ligne supplémentaire que *table.concat*() va créer avant cette chaîne vide se trouve à la fin de la chaîne qui en résulte.

8-i - Afficher le contenu des champs d'une table

Avec Lua, l'indexation ou la numérotation des champs, ne commence pas à 0, mais à 1.

(oui, je sais, je l'ai déjà dit, mais vous verrez par vous-même, on a facilement tendance à l'oublier ... alors !)

Pour afficher le contenu des champs d'une table, il existe 2 opérateurs, *ipairs*() et *pairs*().

- ***ipairs***() pour les tables numériques, [clé-index] ;
- ***pairs***() pour les tables alphabétiques ou mélangées, [clé-valeur] .

L'affichage se fait dans un ordre croissant avec *ipairs*() et dans un ordre indéfini avec *pairs*().

L'opérateur *ipairs*() est utilisé lorsque les indices sont numériques sous forme de « clé-index ». (i pour index et v pour valeur).

```
for i, v in ipairs (maTable) do
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

```
  print(i.." - "..v)
end
```

L'opérateur *pairs*() est utilisé lorsque les indices sont alphabétiques sous forme de « clé-valeur ». (k pour key (key-valeur) et v pour valeur).

```
for k, v in pairs (maTable) do
  print(k.." - "..v)
end
```

Lorsque la clé contient des espaces, il faut la mettre entre crochets.

maTable = {**[**"ceci est une clé avec des espaces"**]** = 1515}

ou

maTable **[**"ceci est une clé avec des espaces"**]** = 1515

Lorsque la clé est une **clé-valeur** et non une clé-index, la notation peut se faire de la façon suivante :

Écrire *maTable.valeur-de-la-clé* à la place de *maTable["valeur-de-la-clé"]*.

*Et pour un complément d'information sur les tables, n'hésitez pas à vous rendre sur le Wiki de Lua : **à cette adresse.***

8-j - À retenir

1. *avec Lua **une table est un objet** ;*
2. *une table possède **une structure dynamique** qui augmente au fur et à mesure des besoins ;*
3. *la numérotation des champs commence à 1 et non 0 ;*
4. *il y a deux façons principales de créer une matrice en Lua :*
   - *utiliser une table de table ,*
   - *rassembler les deux indices en un seul ;*
5. *utiliser une table avec **n** champs pointant vers le haut pour représenter une liste ;*
6. *pour lire un fichier, utiliser la fonction io.read("*all") ;*
7. *utiliser les itérateurs pairs() et ipairs() pour traverser les tables :*

```
for k, v in pairs(t) do ... corps ... end

for i, v in ipairs(t) do ... corps ... end
```

- *k = key (clé) (lors d'un index alphabétique),*
- *v = valeur,*
- *t = table (ou liste de noms),*
- *i = index (lors d'un index numérique).*

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

9 - Les métatables

9-a - Qu'est-ce qu'une métatable ?

Une métatable n'est ni plus, ni moins qu'une table Lua ordinaire.

Une métatable **mt**, va redéfinir le comportement de la valeur initiale d'une table **t1**, en fonction de certaines opérations spéciales.

Au moment de sa construction, une table ne possède pas de métatable.

```
-- Construction d'une nouvelle table ... Ce n'est pas une métatable!
table0 = {}

-- Recherche de l'existence d'une métatable.
print(getmetatable(table0))

-- On obtiendra: nil, car il n'y a pas de métatable.
```

Il appartient au programmeur de créer la métatable dont il aura besoin et de l'associer à la table de son choix.

```
-- Construction d'une table t0.
t0 = {}

-- Construction d'une table t1.
t1 = {}

-- Assignation de t1 en tant que métatable de t0.
setmetatable(t0, t1)

-- t1 est maintenant la métatable de t0.
```

Toute table peut être la métatable d'une autre table.

Un groupe de tables connexes peut partager une métatable commune (qui décrit leur comportement commun).

Une table peut être sa propre métatable (de sorte qu'elle décrive son propre comportement individuel).

Toute configuration est valide.

9-b - C'est bien tout ça, mais ... ça sert à quoi ?

Les tables en Lua ont habituellement un ensemble d'opérations assez prévisible.

Vous pouvez ajouter des paires « clé-valeur », vous pouvez vérifier la valeur associée à une clé, vous pouvez traverser toutes les paires « clé-valeur », et vous pouvez ... **Oups ! mais, c'est tout.**

Vous ne pouvez pas, par exemple additionner des tables, comparer les tables entre elles, ou appeler une table.

C'est là que les **métatables** et leurs **métaméthodes** interviennent.

Car si la table **t1** est mise en jeu dans une opération qu'elle ne sait pas faire, alors sa métatable **mt** lui indiquera la marche à suivre.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Une métatable contrôle comment un objet se comporte dans les opérations arithmétiques, dans les comparaisons, la concaténation, les opérations « length », et d'indexation.

Chaque opération est identifiée par son nom correspondant.

La clé pour chaque opération est une chaîne avec son nom, précédé de deux caractères de soulignement : __xyz.

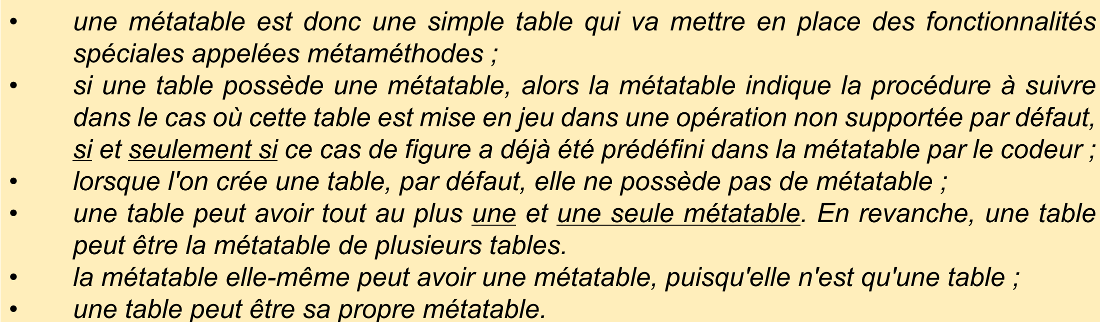
Par exemple, la clé de l'opération ***addition***, est la chaîne __add.

Les métatables vont vous permettre de changer le comportement initial d'une table.

Par exemple, en utilisant les métatables, vous pourrez définir comment Lua calculera l'expression a + b, si a et b sont des tables.

Quand Lua essaye d'ajouter deux tables, il vérifie si l'une d'elle a une métatable et si cette métatable possède un champ __add.

Si Lua trouve ce champ, alors il appelle la valeur correspondante pour calculer la somme.



- *une métatable est donc une simple table qui va mettre en place des fonctionnalités spéciales appelées métaméthodes ;*
- *si une table possède une métatable, alors la métatable indique la procédure à suivre dans le cas où cette table est mise en jeu dans une opération non supportée par défaut, si et seulement si ce cas de figure a déjà été prédéfini dans la métatable par le codeur ;*
- *lorsque l'on crée une table, par défaut, elle ne possède pas de métatable ;*
- *une table peut avoir tout au plus une et une seule métatable. En revanche, une table peut être la métatable de plusieurs tables.*
- *la métatable elle-même peut avoir une métatable, puisqu'elle n'est qu'une table ;*
- *une table peut être sa propre métatable.*

9-c - Et ça fonctionne comment ?

Une forme séquentielle donnerait ceci :

Soit la table indexée suivante : maTable[1].

Lua recherche dans maTable, une entrée avec une clé égale à 1.

La valeur est trouvée : Lua retourne la valeur.

La valeur n'est pas trouvée : Lua retourne nil.

Tous ceci semble logique ... et pourtant, ***c'est faux !*** ou plus exactement ce n'est pas tout à fait exact.

Explication :

Si la valeur n'est pas trouvée, alors Lua recherchera dans une métatable.

Car si Lua ne sait plus comment faire avec un « objet » alors, il fait appel à « sa métatable » pour l'aider.

Regardez maintenant le code suivant :

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
maTable = {
  a = 1,
  b = 2,
  }

maMetaTable = {}
function maMetaTable.__index(la_table, la_key)
  print(la_table, la_key)
  return 666
end

setmetatable(maTable, maMetaTable)
```

Explication :

Supposez que vous vouliez voir le contenu de (maTable.a).

Lua va chercher dans la table, trouver ce que vous lui demandez et retourner le chiffre **1**.

Maintenant vous voulez voir (maTable.c)

Lua va chercher dans la table ce que vous lui demandez, mais ne va rien trouver, puisque **c** n'existe pas, mais avant de vous retourner nil, Lua ira chercher dans sa métatable ... (si cette dernière existe).

Maintenant, Lua appelle la fonction maMetaTable__index.

la_table est l'argument de la table qui a été indexé et **la_key** est la clé qui va essayer de lire la table.

La valeur qui sera retournée est celle que la fonction indexée retournera et dans ce cas : **0**.

En tout état de cause, Lua imprimera donc **la_table**, **la_key**, et la valeur **0**, puisque **c** n'existe pas.

En résumé :

Pour utiliser une métatable, vous devez :

1 la définir : maMetaTable = {} ;
2 définir son comportement. (ce qu'elle doit faire.) ;
3 l'associer à une table. (setmetatable(table, maMetaTable)).

9-d - Et concrètement, on fait comment ?

Supposez que vous ayez deux tables distinctes, t1 et t2 dans lesquelles sont stockées des quantités d'articles.

t1 = {25} et t2 = {18}

Ce sont les mêmes articles et vous voulez les additionner.

Si vous faites, t1 + t2 ... cela générera une erreur.

Donc, pour additionner le contenu de ces deux tables, vous allez :

1 Construire une métatable ;
2 Définir les conditions d'addition pour t1 et t2 ;
3 Associer cette métatable à une des deux tables t1 ou t2.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
-- Les tables t1 et t2
t1 = {25}
t2 = {18}

mt = {}    -- Construction de la table qui servira de métatable.

-- Définition de l'opération addition.
mt.__add = function(a, b)
  return{valeur = a[1] + b[1]}
end

setmetatable(t1, mt)    -- Association de mt à t1.

Et maintenant si l'on fait:
  print(unpack(t1 + t2))

   ---> Cela donnera: 43
```

Pourquoi print(*unpack*(t1 + t2)) ?

Parce que vous ne pouvez pas faire directement print(table).

Si vous faites : print(t1 + t2), vous n'obtiendrez pas son contenu, mais le type et son emplacement mémoire. --> table: 01092738.

Il convient donc d'utiliser la fonction *table.unpack*(), qui retourne les éléments de la table donnée et qui bien sur peut s'écrire directement : *unpack*(table).

__add, est aussi appelé **clé de la métatable** ou encore événement. Dans l'exemple précédent l'événement est **add** et la métaméthode est la fonction qui effectue l'addition.

On ne peut pas utiliser n'importe quoi comme clé et il existe 18 clés déjà définies dans le noyau de Lua.

Un petit rappel :

mt.__add = function(a, b) ... end est équivalent à : function mt.__add(a, b) ... end.

Tableau des 18 clés définies :

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

clés ou arguments	explication
__add(t1, t2)	l'opération addition (+)
__sub(t1, t2)	l'opération soustraction (-)
__mul(t1, t2)	l'opération multiplication (*)
__div(t1, t2)	l'opération division (/)
__mod(t1, t2)	l'opération modulo (%)
__pow(t1, t2)	l'opération exponentielle (^)
__unm(objet)	l'opération unaire (-) [Désigne une valeur négative]
__concat(t1, t2)	l'opérateur de concaténation (..)
__len(object)	l'opération length (#)
__eq(t1, t2)	l'opération égale (==) [les deux valeurs comparées doivent être du même type]
__lt(t1, t2)	l'opération « plus petit que » (<) [*a > b est équivalent à b < a]*
__le(t1, t2)	l'opération « plus petit ou égal » (<=) [*a >= b est équivalent à b <= a]*
__index(objet, clé)	utilisée lorsqu'on tente d'accéder à un élément de la variable inexistant. Sera exploité pour la création d'une classe dans la partie POO. La fonction reçoit la variable et la clé demandée.
__newindex(objet, clé, val)	utilisée lorsqu'on tente de modifier ou de créer un élément de la variable. La fonction reçoit la variable, la clé et la valeur.
__call(objet, ...)	utilisée lorsque la variable est appelée comme une fonction. La fonction reçoit la variable, et tous les arguments de l'appel dans l'ordre où ceux-ci ont été passés.
__metatable(objet)	utilisée pour rendre inaccessible la metatable : donnez-lui la valeur que vous voulez, et c'est cette valeur qui sera retournée par getmetatable(), à la place la métatable.
__tostring(objet)	liée à la fonction d'affichage print(). Permet d'afficher le contenu d'une table en utilisant la fonction print()
__mode	Cette métaméthode permet de définir le niveau de « faiblesse » de la référence vers une table. Aptitude qu'aura une table à être prise en compte ou non par le ramasse-miette.

Un petit résumé :

1 *construire la métatable : mt = {} ;*
2 *définir son action : mt.__xyz = function() ... return ... end ;*
3 *l'associer à une table : setmetatable(table, mt).*

9-e - Les métaméthodes arithmétiques

Pour chaque opérateur arithmétique, il existe une clé correspondante :

__add	addition
__sub	soustraction
__mul	multiplication
__div	division
__pow	exponentiel
__unm	négation

En règle générale, lorsque vous additionnez deux tables, vous ne vous posez aucune question, sur la métatable à utiliser et encore moins sur la métaméthode.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Pour sélectionner la métaméthode adéquate, Lua va effectuer les opérations suivantes :

1. Si la première valeur dispose d'une métatable avec un champ __add, Lua utilisera cette valeur comme étant LA métaméthode à utiliser ;
2. Si la deuxième valeur dispose d'une métatable avec un champ __add, Lua utilisera cette deuxième valeur comme étant LA métaméthode à utiliser ;
3. sinon : Lua déclenchera un message d'erreur.

Il vous appartient, de prévoir un message d'erreur dans la langue que vous souhaitez utiliser, sinon c'est le message d'erreur du système, bien évidement en anglais, qui se déclenchera en cas de problème.

Et pour illustrer tout ça, rien ne vaut un petit exemple d'addition de 2 tables.

Soit deux hangars où sont stockés des fruits.

Pour faire le bilan de l'entreprise, vous allez devoir additionner tous les stocks de marchandises communes.

```
-- Construction des tables nécessaires.
local total   = {}
local hangar1 = {}
local hangar2 = {}

local mt = {}    -- Construction de la table qui servira de métatable.

mt.__add = function(a, b)    -- Création de l'opération d'addition (__add)
  for k1, v1 in pairs(hangar1) do
     for k2, v2 in pairs(hangar2) do
       if k1 == k2 then
         total[k1] = v1 + v2
         break
      end
     end
  end
end

setmetatable(hangar1, mt)    -- Association de la mt à hangar1.

local function addition(a, b)    -- On ajoute un message d'erreur ... au cas ou!
  if (getmetatable(a) == nil) and (getmetatable(b) == nil) then
    error("Vous essayez d'additionner 2 tables sans métatable.")
  else
    local t = a + b
  end
end

-- Les inventaires ont donné le résultat suivant:
  hangar1.banane = 12
  hangar2.banane = 31
  hangar1.orange = 15
  hangar2.orange = 22
  hangar1.pomme  = 11
  hangar2.pomme  = 35

addition(hangar1, hangar2)    -- On lance l'addition.

-- Et on affiche le résultat.
local liste = ""
for k, v in pairs(total) do
  liste = k.." = "..v.."\n"..liste
end
print(liste)

 Ce qui donne:
  banane = 43
  orange = 37
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

```
pomme = 46
```

La construction pour les autres méthodes mathématiques est similaire à celle que vous venez de voir.

Il suffit de remplacer le signe (+) par le signe de l'opération que vous souhaitez effectuer : __sub, __unm, __mul, __div, __mod, __pow et __len.

Faites attention à ne pas confondre l'opérateur de soustraction (-) avec l'opérateur de changement de signe d'un nombre : l'opérateur unaire (-).

9-f - Les métaméthodes relationnelles

Les métatables vous permettront aussi de donner un sens aux opérateurs relationnels au travers des métaméthodes suivantes :

__eq	égalité (==)
__lt	plus petit que (<)
__le	inférieur ou égal (<=)

Il n'existe pas d'autres métaméthodes pour les trois autres opérateurs relationnels qui sont :

a ~= b	que Lua traduite par : not(a == b)
a > b	que Lua traduit par : b < a
a >= b	que Lua traduit pat : b <= a

Là aussi, le système de construction reste le même que précédemment.

9-g - Les autres métaméthodes

9-g-1 - __concat(t1, t2)

Cette métaméthode est liée à l'opérateur de concaténation de chaînes de caractères.

Ce qui permet de concaténer 2 tables : table0 = table1..table2

```
mt.__concat = function(a, b)
local t = a
for k in ipairs(b) do
  table.insert(t, b[k])
end
return t
end
```

9-g-2 - __tostring(objet)

Cette métaméthode modifie les caractéristiques de print().

Ce qui permet d'afficher le contenu d'une table en utilisant la fonction print().

table1 = {1, 2, 3}

Sans la métaméthode : print(table1) --> table: 01092738

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

Avec la métaméthode : print(table1) --> 1 2 3

```
mt.__tostring = function(b)
local s = ""
for k in ipairs(b) do
  s = s.." "..b[k]
end
return s
end
```

9-g-3 - __call(objet, ...)

Cette métaméthode permet d'appeler une table comme s'il s'agissait d'une fonction.

```
mt.__call = function(objet)
  ... ici les nouvelles informations concernant la table ...
end
```

9-g-4 - __metatable(objet)

Cette métaméthode permet de bloquer l'accès à une table, à un utilisateur non autorisé.

```
mt.__metatable = "ici le message ... "
```

9-g-5 - __index(table, clé)

Cette métaméthode permet de gérer l'accès au contenu d'une table.

Lorsque vous tentez d'accéder à un champ non-existant d'une table, Lua cherche la métatable de cette table et la métaméthode __index.

S'il la trouve, il vous indique quoi faire. Sinon, il renvoie nil.

La métaméthode __index peut être une table ou une fonction.

1- Si c'est une table, le champ absent sera recherché dans cette table.

```
table_type = {x = 100, y = 150}     -- Construction d'une table.

table = {}     -- Construction de la table.

mt = {}     -- Construction de la métatable.

mt._index(table_type)     -- Construction de __index.

setmetatable(table, mt)     -- Association table et mt

print(table.x, table.y) --> 100, 150
```

2- Si c'est une fonction, __index peut prendre 0, 1 ou 2 arguments, dans ce cas, ces arguments sont respectivement la table appelante et le champ recherché. __index(), __index(table) ou __index(table, champ).

```
-- Sans argument.
mt.__index = function() return 100 end

-- Avec 1 argument.
mt.__index = function(k) return k end
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
-- Avec 2 arguments.
mt.__index = function(k, v) return v end
```

9-g-6 - __newindex(objet, clé, valeur)

Si __index permet de gérer l'accès au contenu d'une table, __newindex permet de gérer la modification du contenu d'une table.

Elle peut par exemple, empêcher de rajouter un nouveau champ.

```
-- Soit la table suivante.
table = {a = 10, b = 20, c = 30}

-- Mise en place, maintenant classique.
mt = {}
mt.__newindex = function(t, k, v) end
setmetatable(table, mt)

-- Et maintenant.
table.z = 150
print(table.z)

  ==> ce qui donnera: nil
```

9-g-7 - __mode()

Cette métaméthode permet de définir le niveau de « faiblesse » de la référence vers une table.

Le niveau de « faiblesse », est l'aptitude qu'aura une table à être prise en compte ou non par le ramasse-miettes (garbage-collector).

La faiblesse d'une table est contrôlée par le champ de sa métaméthode : __mode.

Si le champ __mode est une chaîne contenant le caractère "k", alors les clés de la table seront considérées comme faibles.

Si le champ __mode contient la lettre "v", alors les valeurs de la table seront considérées comme faibles.

9-h - À retenir

1. *une métatable est une simple table qui va mettre en place des fonctionnalités spéciales appelées métaméthodes.*
2. *pour mettre en place une métatable :*
   1. *construire la métatable : mt = {},*
   2. *définir son action : mt.__xyz = function() ... return ... end,*
   3. *l'associer à une table : setmetatable(table, mt) ;*
3. *il existe 18 clés (ou événements) déjà définies dans le noyau de Lua.*
4. *il existe des métaméthodes arithmétiques, des métaméthodes relationnelles et autres métaméthodes.*
5. *Lua possède huit types primitifs :*
   1. *booléens,*
   2. *nil,*
   3. *string,*
   4. *table,*
   5. *function,*
   6. *number,*

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

    7 *thread,*
    8 *userdata ;*
6 *Tous ces types, les tables, les chaînes et les userdatas sont pilotés par des métatables ;*
7 *Les métatables ne sont pas obligatoires pour développer des applications en Lua, mais n'oubliez pas qu'elles offrent une incommensurable richesse à la syntaxe du langage et que par conséquent il serait dommage de ne pas les utiliser.*

10 - Les strings

10-a - Qu'est-ce qu'une chaîne de caractères ?

En informatique, une chaîne de caractères ou **string** en anglais, est une séquence finie de caractères (Par exemple, des lettres, chiffres, symboles et signes de ponctuation.)

Une caractéristique importante de chaque chaîne, est sa longueur qui représente le nombre de caractères.

La longueur peut être n'importe quel nombre naturel, zéro ou entier positif.

Une chaîne particulièrement utile pour certaines applications de programmation est la chaîne vide, qui est une chaîne ne contenant pas de caractère et ayant une longueur égale à zéro.

10-b - Déclaration, guillemets et crochets

Une chaîne de caractères est enfermée par des guillemets ou des doubles crochets.

```
local nomString = "Une string est enfermée par des guillemets."

print(nomString)

--> Une string est enfermée par des guillemets.
```

Vous pouvez utiliser des apostrophes **'** ... **'**, des guillemets anglais **"** ... **"** ou encore des doubles crochets **[[** ... **]]**.

Cela permet d'inclure un type de guillemets à l'intérieur d'un autre.

```
print(" hello 'les utilisateurs de Lua' ")
--> hello 'les utilisateurs de Lua'

print("Son [[contenu]] n'a pas obtenu de sous-chaîne.")
--> Son [[contenu]] n'a pas obtenu de sous-chaîne.

print([[Ayons plus de "strings".]])
--> Ayons plus de "strings".
```

Des lignes multiples de texte peuvent être jointes par des doubles crochets.

```
print([[Plusieurs lignes de texte, peuvent
être entourées par des doubles
crochets.]])

--> Tout sera affiché sur une seule ligne.
```

10-c - Les séquences d'échappement

Les séquences d'échappement ne sont pas reconnues à l'intérieur des doubles crochets.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Lua peut également manipuler des séquences d'échappement comme en C.

Les chaînes Lua peuvent contenir les caractères d'échappement suivants :

Escapes	Signification
\a	alerte
\b	espace
\f	saut de page
\n	nouvelle ligne
\r	retour à la ligne
\t	tabulation horizontale
\v	tabulation verticale
\\	contre-slash
\"	guillemet anglais
\'	apostrophe
\[	crochet gauche
\]	crochet droit
\?	point d'interrogation
\000	notation octale
\xhh	notation hexadécimale

Exemple d'utilisation :

```
print("bonjour\nNouvelle ligne \tTabulation")

  -- ce qui donne:
  bonjour
  Nouvelle ligne    Tabulation
```

10-d - La concaténation

L'opérateur de concaténation est signifié par deux points (**..**).

Si une donnée est un nombre, Lua le convertit en une chaîne de caractères.

```
print("Hello ".."World")  --> Hello World

print(0..1)  --> 01

a = "Hello"
print(a.." World")  --> Hello World
print(a)  --> Hello
```

Il ne faut pas oublier qu'en Lua, les chaînes de caractères sont immuables et que tout changement à l'intérieur d'une chaîne, entraîne la création d'une nouvelle chaîne.

L'opérateur de concaténation crée une nouvelle chaîne de caractères, mais sans modifier les « opérandes ». Dans l'exemple précédent : (a).

Des nombres peuvent être enchaînés aux chaînes de caractères. Dans ce cas, ils sont **contraints** avant d'être enchaînés.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

10-e - La coercition

Lua exécute **la conversion automatique** des nombres en chaînes de caractères et vice versa si cela est approprié, c'est la **coercition**.

Pendant la coercition, vous n'aurez pas le plein contrôle du formatage de la conversion.

Pour formater le nombre comme vous le souhaitez en chaîne de caractères vous pouvez utiliser la fonction *string.format*() comme dans l'exemple ci-dessous :

```
string.format("%.3f", 5.0) --> 5.000

print("Lua version "..string.format("%.1f", 5.0))

--> Lua version 5.0
```

(%3.f, signifie un nombre à virgule flottante avec trois chiffres après la virgule.)

Il s'agit là d'une conversion explicite en utilisant une fonction pour convertir le nombre, plutôt que d'utiliser la coercition.

10-f - Qu'est-ce qu'un « pattern » ?

L'utilisation de base de certaines fonctions sur les chaînes de caractères, consiste à rechercher un « pattern » dans une chaîne donnée.

Le mot anglais « pattern » est utilisé pour désigner un modèle, une structure, un motif, un type, etc.

La forme la plus simple d'un « pattern » est un mot qui ne correspond qu'à une copie de lui-même.

Par exemple, le *pattern* "bonjour" recherche la chaîne "bonjour" à l'intérieur de la chaîne sujet.

Quand *string.find*() trouve son *pattern*, il retourne deux valeurs : L'indice du début et l'indice de fin.

```
local s = "bonjour le monde"

local i, j = string.find(s, "bonjour")

print(i, j) --> 1 7 (-- les indices de début et de fin)

print(string.find(s, "monde")) --> 12 16
```

Lua possède quelques fonctions sur les strings qui utilisent les patterns.

Parmi elles, vous trouverez :

Fonctions	Explications
string.find(string, pattern)	Cherche la première instance du *pattern* dans *string*.
string.gsub(string, pattern, rempl)	Retourne une nouvelle *string* où toutes les instances de *pattern* ont été remplacées par *rempl*.
string.match(string, pattern,[init]	Retourne une ou plusieurs instances de *pattern* trouvées dans *string*. [init] est optionnel et spécifie l'endroit où démarrer la recherche.

Mais vous pouvez faire des patterns plus performants que "bonjour", avec les **classes de caractères**.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Une classe de caractères est un élément dans un pattern qui peut correspondre à tout caractère dans un ensemble spécifique.

Par exemple, la classe %d correspond à n'importe quel chiffre. Par conséquent, vous pouvez rechercher une date au format dd/mm/yyyy avec le pattern "%d%d/%d%d/%d%d%d%d".

```
local s = "la date limite est le 12/12/2012."

local date = "%d%d/%d%d/%d%d%d%d"

print(string.sub(s, string.find(s, date)))

--> Ce qui donne: 12/12/2012
```

Les combinaisons suivantes sont autorisées à décrire une classe de caractères qui représente :

Classe de caractères	Représentation
%a	N'importe quelle lettre.
%c	N'importe quel caractère de contrôle.
%d	N'importe quel chiffre.
%l	N'importe quelle lettre minuscule.
%p	N'importe quel caractère de ponctuation.
%s	N'importe quel caractère d'espace.
%u	N'importe quelle lettre majuscule.
%w	N'importe quel caractère alphanumérique.
%x	N'importe quel chiffre hexadécimal.
%z	N'importe quel caractère avec une représentation "\0".

Une version MAJUSCULE d'une de ces classes, représente le contraire de la classe. Par exemple : "%A" représente tous les caractères non-lettre.

L'exemple suivant, remplace tout ce qui n'est pas lettre, par l'étoile (*).

```
print(string.gsub("hello, up-down!", "%A", "*"))

--> Ce qui donne: hello**up*down* 4

-- Le chiffre 4, représente ici, le nombre de substitutions.
```

Quelques caractères, appelés **caractères magiques** (ou expressions régulières), ont une signification particulière lorsqu'ils sont utilisés dans un pattern.

Les caractères magiques sont : **() . % + - * ? [^ $**

Listes et signification des caractères magiques :

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Caractères magiques	Signification
()	Pour former un groupe ou une sous-expression qui sera souvent appelé.
. (*le point*)	Représente n'importe quel caractère.
%	Sert d'échappement pour les caractères magiques.
+	Une ou plusieurs répétitions du pattern.
-	Zéro ou plusieurs répétitions du pattern pour les petites séquences.
*	Zéro ou plusieurs répétitions du pattern pour les grandes séquences.
?	Zéro ou plusieurs répétions du pattern. Rends facultatif le pattern suivant.
[	Démarre une séquence : [a-z].
^	Recherche de chaîne de caractères commençant par ...
$	Recherche de chaîne de caractères se terminant par ...

Le point (.) représente un caractère magique qui peut représenter n'importe quel caractère.

"%" fonctionne comme un caractère d'échappement pour les caractères magiques. Ainsi, "%." correspond à un point et "%%" correspond au caractère "%" lui-même.

Si vous avez besoin de mettre une apostrophe dans un pattern, vous devez utiliser les mêmes techniques que vous utiliseriez pour placer un guillemet à l'intérieur d'une autre chaîne : par exemple, vous pouvez échapper le guillemet avec un contre-slash, "\", qui est le caractère d'échappement pour Lua.

Un ensemble de caractères, vous permet de créer vos propres classes, en les combinant à l'intérieur de crochets. Par exemple, "[%w_]" correspond à la fois à des caractères alphanumériques (w) et des caractères de soulignement (_).

L'ensemble "[01]" correspond aux chiffres binaires, et "[%[%]]" correspond à des crochets.

Pour compter le nombre de voyelles CAPITALES et minuscules dans un texte, vous pouvez écrire :

```
local texte = "Pour compter le nombre de voyelles"

local _, nbVoyelles = string.gsub(texte, "[AEIOUYaeiouy]", "")

print(nbVoyelles)

--> Ce qui donne: 12
```

(Le _ est juste un nom fictif de variable.)

Vous pouvez également inclure les plages de caractères dans un ensemble de caractères, en écrivant le premier et le dernier caractères de la gamme séparés par un tiret. Mais vous aurez rarement besoin de cette facilité, car la plupart des gammes utiles sont déjà prédéfinies. Par exemple, "%d" est plus simple que "[0-9]" de même, "%x" est plus simple que "[0-9a-fA-F]". Toutefois, si vous avez besoin de trouver un chiffre octal, alors vous préférerez peut-être "[0-7]", au lieu d'une énumération explicite tel que : "[01234567]".

Vous pouvez obtenir le contraire d'un jeu de caractères en le commençant par "^". Par exemple, "[^0-7]" trouve n'importe quel caractère qui n'est pas un chiffre octal et "[^\n]" correspond à tout caractère différent de saut de ligne.

Mais n'oubliez pas que vous pouvez inverser des classes simples avec sa version capitale : "%S" est plus simple que "[%^s]".

Les classes de caractères suivent la localisation courante définie par Lua. Par conséquent, la classe "[a-z]" peut être différente de "%l". Dans certains cas appropriés, cette dernière forme comprend des lettres telles que "ç" et "ã". Vous

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

devriez toujours utiliser la dernière forme, sauf si vous avez une bonne raison de faire autrement. C'est plus simple, plus portable, et un peu plus efficace.

10-g - Les captures

Le mécanisme de capture offre un pattern pour copier les parties de la chaîne qui correspondent aux parties du pattern.

Vous spécifiez une capture en écrivant la partie de la structure que vous voulez capturer entre parenthèses.

Lorsque vous spécifiez une capture à *string.find*(), la fonction retourne les valeurs capturées en tant que résultats supplémentaires de l'appel.

L'utilisation typique de cette construction est de briser une chaîne en plusieurs parties.

```
pair = "nom = Anna"

_, _, key, value = string.find(pair, "(%a+)%s*=%s*(%a+)")

print(key, value) --> nom Anna
```

Le pattern "%a+" spécifie une séquence non vide de lettres et "%s*" spécifie une séquence d'espaces (éventuellement vide).

Ainsi, dans l'exemple précédent, le motif entier spécifie une séquence de lettres, suivie par une séquence d'espaces, suivi par "=" et de nouveau suivi par des espaces plus une autre séquence de lettres.

Les deux séquences de lettres ont leurs patterns entourés par des parenthèses, de façon à être capturés si une rencontre a lieu.

La fonction *string.find*() retourne toujours le premier des indices, qui dans l'exemple précédent a été stocké dans la variable fictive (_) puis retourne la capture faite au cours des recherches.

Le code suivant va supprimer les « slashs » :

```
date = "14/7/1789"

_, _, d, m, y = string.find(date, "(%d+)/(%d+)/(%d+)")

print(d, m, y)  --> 14 7 1789
```

Vous pouvez également utiliser des captures dans le pattern lui-même. Un pattern, à un élément comme "%d", (où d est un chiffre unique), qui correspond uniquement à une copie de la capture d.

10-h - Exemples d'utilisation

Appréhender les patterns n'est pas forcement chose aisée.

Et j'espère que les quelques exemples mentionnés ci-dessous, vous permettrons de mieux comprendre leur grande utilité.

Plutôt que de répéter toujours les mêmes choses, je vais commencer par créer une fonction intitulée : **FindPattern**() que vous retrouverez tout au long des exemples suivants.

```
function FindPattern(text, pattern, start)
   return string.sub(text, string.find(text, pattern, start))
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
end
```

10-h-1 - Les classes de caractères

```
-- Recherche la position de "an" et "ou"
  print(string.find("banane", "an"))    --> 2 3

  print(string.find("banane", "ou"))    --> nil

local s = "une petite phrase"
-- Recherche un mot commencent par ...
print(FindPattern(s, "p ..."))    --> petite

  print(FindPattern(s, "p ...", 6)) --> phrase

-- Recherche une lettre
  print(FindPattern(s, "%a")) --> u

-- Recherche tout ce qui est lettre
  print(FindPattern(s, "%a+")) --> une

local s = "nous sommes en 2012"
-- Recherche un chiffre
  print(FindPattern(s, "%d")) --> 2

-- Recherche tout ce qui est chiffre
  print(FindPattern(s, "%d+")) --> 2012

-- Recherche une lettre CAPITALE
  print(FindPattern("mAjuscUle", "%u")) --> A

-- Recherche une lettre minuscule
  print(FindPattern("MInUScUle", "%l")) --> n

-- Recherche une minuscule suivie d'une capitale
  print(FindPattern("MInUScUle", "%l%u")) --> nU
```

10-h-2 - Les ensembles

```
-- Recherche la première occurrence d'une des lettres "x, y, j, k, n"
  print(FindPattern("banana", "[xyjkn]")) --> n

-- Recherche la première occurrence d'une des lettres comprise entre j et q
  print(FindPattern("banana", "[j-q]")) --> n

-- Recherche ce qui n'est pas a ou b
  print(FindPattern("banana", "[^ba]")) --> n

-- Recherche ce qui n'est pas compris entre a et n
  print(FindPattern("bananas", "[^a-n]")) --> s

-- Recherche ce qui n'est pas lettre minuscule ou espace
  print(FindPattern("nous sommes en 2012", "[%l%s]")) --> 2
```

10-h-3 - Les captures

```
* pour zéro ou plusieurs caractères du pattern. (* recherche la séquence la plus longue.)
  print(FindPattern("bananas", "az*")) --> a
  print(FindPattern("bananas", "an*")) --> an
  print(FindPattern("bannnnnanas", "an*")) --> annnnn

+ pour une ou plusieurs caractères du pattern
  print(string.find("banana", "az+")) --> nil
  print(FindPattern("bananas", "an+")) --> an
  print(FindPattern("bannnnnanas", "an+")) --> annnnn
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
- pour zéro ou plusieurs caractères du pattern. (- recherche la séquence la plus courte.)
  print(FindPattern("bananas", "az-")) --> a
  print(FindPattern("bananas", "an-")) --> a
  print(FindPattern("bannnnnanas", "an-")) --> a

? pour zéro ou une occurrence du pattern
  print(FindPattern("bananas", "az?")) --> a
  print(FindPattern("bananas", "an?")) --> an
  print(FindPattern("bannnnnanas", "an?")) --> an
```

10-i - À retenir

1 une chaîne de caractères est une séquence finie de caractères ;
2 elle se définit entre apostrophes ou guillemets doubles ou entre doubles crochets ;
3 Lua peut manipuler des séquences d'échappement comme en C ;
*4 l'opérateur de concaténation est signifié par (**..**) ;*
*5 la coercition est la **conversion automatique** des nombres en strings et des strings en nombres ;*
6 Lua utilise les patterns ;
*7 Lua dispose de 14 fonctions, pour manipuler les chaînes de caractères. (**Voir à ce sujet, le paragraphe consacré à la bibliothèque Lua.**)*

11 - Les coroutines

11-a - Qu'est-ce qu'une coroutine ?

Tout d'abord, un petit rappel sur ce qu'est un thread :

Un thread est similaire à un processus, car tous deux représentent l'exécution d'un ensemble d'instructions du langage machine d'un processeur.

Et du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle. Mais, là où chaque processus possède sa propre mémoire virtuelle, les threads d'un même processus se partagent la même mémoire virtuelle.

Comme exemple d'application, on peut citer les traitements de texte où la correction orthographique est exécutée tout en permettant à l'utilisateur de continuer à entrer son texte.

En Lua, une coroutine est similaire à un thread et représente une ligne d'exécution avec sa propre pile, ses propres variables locales et son pointeur d'instruction propre.

La principale différence est que si un programme peut fonctionner avec plusieurs threads simultanés, il ne pourra en revanche ne faire tourner qu'une seule coroutine à la fois.

Le système des coroutines est un système collaboratif, une coroutine s'exécute, puis passe la main à une autre, et ainsi de suite.

***Une coroutine**, va donc permettre de faire exécuter un programme en tâche de fond, sans ralentir celui du premier plan.*

11-b - Les bases de fonctionnement

Vous retrouverez toutes les fonctions utiles aux coroutines dans la bibliothèque Lua : **Fonctions sur les coroutines.**

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

La fonction *coroutine.create*(), crée une nouvelle coroutine et retourne une valeur de type thread, qui représente la nouvelle coroutine.

Le plus souvent, l'argument de la fonction *coroutine.create*() est une fonction anonyme.

```
co = coroutine.create(function()
  print("bonjour")
end)

print(co)     --> thread: 0x8071d98
```

Une coroutine dispose de trois états différents :

1 **suspended** qui signifie : en attente ;
2 **running** qui signifie : en fonctionnement ;
3 **dead** qui signifie : mort !

Lorsque vous créez une coroutine, elle est automatiquement **suspended**, donc en attente de vos ordres.

La fonction *coroutine.status*(nom de la coroutine) vérifie son état.

La fonction *coroutine.resume*(nom de la coroutine) démarre son exécution.

La coroutine démarre et fait ce pour quoi elle a été créée, puis s'arrête, c'en est fini pour elle, elle est **dead** (morte).

Jusqu'à présent, les coroutines ne ressemblent à rien de plus qu'à une manière compliquée d'appeler des fonctions. La vraie puissance des coroutines provient de la fonction *coroutine.yield*(), qui autorise une reprise ultérieure de son fonctionnement.

```
co = coroutine.create(function()
  for i = 1, 10 do
    print("co", i)
    coroutine.yield()
  end
end)
```

La coroutine est créée, mais elle ne fait rien, elle attend *coroutine.resume*(co) qui va exécuter une 1ère boucle, puis *coroutine.yield*() va mettre la suite en suspens et attendre de nouveau vos ordres.

À chaque *coroutine.resume*(co), la variable *co* fera un tour, s'arrêtera et attendra de nouveau jusqu'à ce que la boucle soit bouclée.

Elle a fait ses dix tours, maintenant, elle est morte. Vous ne pouvez plus rien en faire, elle ne redémarrera plus.

Vous ne pouvez pas relancer une coroutine qui est « dead ».

Notez que *coroutine.resume*() fonctionne en mode protégé. Par conséquent, s'il y a une erreur dans une coroutine, Lua ne pourra pas afficher de message d'erreur, mais reviendra à la fonction d'appel.

Une fonctionnalité utile en Lua est que cette paire **resume-yield** peut échanger des données.

Le premier **resume** qui n'a pas encore rencontré le premier **yield** peut lui envoyer ses propres arguments en tant que extra-argument de la fonction principale.

L'exemple suivant permettra de mieux clarifier ces propos :

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

```
co = coroutine.create(function(a, b, c)
  coroutine.yield(a + b, a - b)
end)

print(coroutine.resume(co, 20, 10))    --> true 30  10
```

Symétriquement, **yield** retourne les arguments supplémentaires passés par **resume**.

```
co = coroutine.create(function()
  print("co", coroutine.yield())
end)

coroutine.resume(co)
coroutine.resume(co, 4, 5))
--> co  4  5
```

Enfin, lorsqu'une coroutine se termine, toutes les valeurs retournées par sa fonction principale, s'ajoutent à la correspondance de **resume**.

```
co = coroutine.create(function()
  return 6, 7
end)

print(coroutine.resume(co))    --> true 6 7
```

Vous aurez rarement l'occasion d'utiliser tous ces outils dans une même coroutine, mais il est bon de savoir qu'ils existent.

Il est maintenant important de clarifier certains concepts avant de poursuivre.

Lua offre des **coroutines asymétriques**. Cela signifie qu'il existe une fonction pour suspendre l'exécution d'une coroutine et une fonction différente pour reprendre la coroutine suspendue.

Contrairement à certains autres langages qui offrent des *coroutines symétriques*, où il existe seulement une seule fonction de transfert du contrôle d'une coroutine à l'autre.

Certaines personnes appellent les **coroutines asymétriques**, des « demi-coroutines » parce qu'elles ne sont pas symétriques.

Mais, d'autres personnes utilisent le même terme de « semi-coroutine » pour dénoter une application restreinte de coroutines. Par exemple une coroutine qui pourrait suspendre son exécution que si elle est à l'intérieur d'une fonction auxiliaire, c'est-à-dire, quand elle a ses appels en attente dans sa pile de contrôle ...

En d'autres termes, seul le corps principal de ces « semi-coroutines » peut utiliser **yield**.

Un générateur « Python » est un exemple de ce sens de « semi-coroutine ».

Contrairement à la différence entre les *coroutines symétriques* et **asymétriques**, la différence entre les coroutines et générateurs (tel que présenté en Python) est profonde.

Les générateurs ne sont tout simplement pas assez puissants pour mettre en œuvre plusieurs constructions intéressantes que nous pouvons écrire avec les coroutines.

Lua offre de véritables coroutines asymétriques.

Mais ceux qui préfèrent utiliser les coroutines symétriques peuvent les implémenter en les installant sur le dessus des coroutines asymétriques de Lua. Ce qui en soit est une tâche relativement facile puisque chaque transfert d'un **yield** est suivie d'un **resume**.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

11-c - Exemple d'utilisation

Un des exemples d'utilisation des coroutines, est celui de : **producteur --> consommateur.**

Explication :

Supposez une fonction qui produit continuellement les valeurs (par exemple, la lecture d'un fichier) et une autre fonction qui consomme en permanence ces valeurs (par exemple, en écrivant les valeurs récupérées dans un autre fichier).

Typiquement, ces deux fonctions pourraient ressembler à celles-ci :

```
function Producteur()
  while true do
    local x = io.read() -- production d'une nouvelle valeur.
    Envoyer(x)          -- envoie à consommateur.
  end
end

function Consommateur()
  while true do
    local x = Recevoir() -- reçoit de producteur.
    io.write(x, "\n")    -- utilise la nouvelle valeur.
  end
end
```

Dans cette façon de faire, le *producteur* et le *consommateur* tournent continuellement, mais il reste facile de les arrêter lorsqu'il n'y aura plus de données.

Le problème ici est : « comment faire correspondre envoyer et recevoir ? ».

C'est un cas typique du problème de « qui a la main sur la boucle ... »

Tant que le *producteur* et le *consommateur* sont actifs, les deux ont leurs propres boucles principales, et les deux supposent que l'autre est un service appelable.

Dans cet exemple particulier, il serait facile de modifier la structure de l'une des fonctions, en déroulant sa boucle pour en faire un agent passif. Toutefois, ce changement de structure pourrait être plus difficile dans d'autres scénarios réels.

Les coroutines fournissent un outil idéal pour faire correspondre *producteur* et *consommateur*, car une paire de **resume-yield** tourne à l'envers la relation typique entre l'appelant et l'appelé.

Quand une coroutine appelle **yield**, elle n'entre pas dans une nouvelle fonction, mais retourne plutôt un appel en attente (pour **resume**).

De même, un appel à **resume** ne démarre pas une nouvelle fonction, mais retourne un appel à **yield**.

Cette propriété est exactement celle dont vous avez besoin pour faire correspondre *envoyer* à *recevoir*, de telle sorte que chacun agisse comme s'il était le maître et l'autre l'esclave ...

Donc, **recevoir des « resume » de producteur** afin qu'il puisse produire une nouvelle valeur, et **envoyer des « yield » de la nouvelle valeur** pour le consommateur.

```
function Recevoir()
  local status, value = coroutine.resume(producteur)
  return value
end
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
function Envoyer(x)
  coroutine.yield(x)
end
```

Bien sûr, le *producteur* doit désormais être une coroutine :

```
producteur = coroutine.create(function()
  while true do
    local x = io.read()   -- produit une nouvelle valeur
    Envoyer(x)
  end
end)
```

Dans cette conception, le programme appelle en premier la fonction *recevoir()*.

Lorsque celle-ci a besoin d'un élément, elle fait appel à *coroutine.resume*(), qui boucle jusqu'à ce qu'elle ait quelque chose à *envoyer(x)*, puis s'arrête jusqu'au redémarrage de *recevoir()*.

Il s'agit là d'une conception **axée sur la réception**.

Vous pouvez étendre cette conception de « filtres (7) », qui ne sont en fait que des tâches qui se placent entre le *producteur* et le *consommateur* afin d'opérer une sorte de transformation dans les données.

Un filtre va chercher (**resume**) sur un producteur pour obtenir de nouvelles valeurs et retourne (**yield**) des valeurs transformées à un consommateur.

Vous pouvez aussi, ajouter au code précédent, un filtre qui insère un numéro de ligne au début de chaque ligne.

Le code complet, pourrait alors ressembler à ceci :

```
function Receive(prod)
  local status, value = coroutine.resume(prod)
  return value
end

function Send(x)
  coroutine.yield(x)
end

function Producer()
  return coroutine.create(function()
    while true do
      local x = io.read()   -- produit une nouvelle valeur
      send(x)
    end
  end)
end

function Filter(prod)
  return coroutine.create(function()
  local line = 1
    while true do
    local x = receive(prod) -- obtient une nouvelle valeur
    x = string.format("%5d %s", line, x)
    send(x)                  -- envoie au consommateur
    line = line + 1
    end
  end)
end

function Consumer (prod)
  while true do
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

```
    local x = receive(prod)  -- obtient une nouvelle valeur
    io.write(x, "\n")        -- consomme une nouvelle valeur
  end
end
```

Le dernier bit crée simplement les composants dont il a besoin, les relie, et démarre le *consommateur* final : Les coroutines sont une sorte de « multithreading non-préemptif (8) ».

Alors que dans les « pipes (9) », chaque tâche s'exécute dans un processus séparé, les coroutines elles, exécutent chaque tâche dans une sorte de multithreading non-préemptif séparée.

Les pipes fournissent une zone tampon entre l'écrivain (le producteur) et le lecteur (le consommateur) ce qui laisse une certaine liberté dans leurs vitesses relatives.

Ceci est important, car dans ce contexte, le coût de la commutation entre les processus est élevé.

Avec les coroutines, le coût de la commutation entre les différentes tâches est beaucoup plus petit, environ le même que lors d'un appel à une fonction.

11-d - Les coroutines et les itérateurs

Vous pouvez voir les itérateurs comme un exemple très concret du modèle **producteur --> consommateur**.

Un itérateur produit des articles qui seront ensuite consommés par le corps de la boucle. Il semble donc approprié d'utiliser les coroutines pour écrire les itérateurs. En fait, les coroutines fournissent un outil puissant pour cette tâche. Car encore une fois, la principale caractéristique des coroutines est la capacité à retourner la relation entre l'appelant et l'appelé.

Avec cette fonctionnalité, vous pouvez donc écrire un itérateur sans vous soucier de conserver l'état entre les appels successifs à l'itérateur. Pour illustrer ce type d'utilisation, vous allez écrire un itérateur afin de parcourir toutes les permutations d'un tableau donné.

Ce n'est pas une tâche facile et il est plus simple d'écrire une fonction récursive qui génère toutes les permutations. L'idée est simple : mettez chaque élément du tableau en dernière position et générez de façon récursive toutes les permutations possibles des éléments restants.

Ceci étant, le code pourrait ressembler à ce qui suit :

```
function Permgen(a, n)
  if n == 0 then
    printResult(a)
  else
    for i = 1,n do
       -- on met le ième élément à la dernière place.
      a[n], a[i] = a[i], a[n]
      -- on génère les permutations des autres éléments.
      Permgen(a, n - 1)
      -- on restaure le ième élément.
      a[n], a[i] = a[i], a[n]
    end
  end
end
```

Il faut maintenant définir une fonction PrintResult() appropriée et appeler Permgen() avec les arguments adéquats.

```
function PrintResult(a)
  for i, v in ipairs(a) do
    io.write(v, " ")
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
  end
  io.write("\n")
end

Permgen ({1,2,3,4}, 4)
```

Changez maintenant printResult() par une *coroutine.yield*()

```
function Permgen(a, n)
  if n == 0 then
    coroutine.yield(a)
  else
  ...
```

Maintenant définissez une fonction qui fasse que le générateur tourne dans une coroutine, puis crée la fonction d'itération. L'itérateur reprendra simplement la coroutine pour produire la permutation suivante.

```
function Perm(a)
  local n = table.getn(a)
  local co = coroutine.create(function()
  Permgen(a, n) end)
  return function()   -- iterator
    local code, res = coroutine.resume(co)
    return res
  end
end
```

La fonction Perm() utilise un modèle commun en Lua, qui *enveloppe* un appel permettant de reprendre ses coroutines correspondantes à l'intérieur d'une fonction.

Ce modèle est si courant que Lua fournit une fonction spéciale : *coroutine.wrap*().

Comme *coroutine.create*(), *coroutine.wrap*() crée une nouvelle coroutine.

Contrairement à *coroutine.create*(), *coroutine.wrap*() ne retourne pas la coroutine mais, une fonction qui, lorsqu'elle est appelée, reprend la coroutine.

Et contrairement à *coroutine.resume*(), cette fonction ne retourne pas de code d'erreur, mais soulève l'erreur au cas ou.

```
function Perm(a)
  local n = table.getn(a)
  return coroutine.wrap(function() Permgen(a, n) end)
end
```

Habituellement, *coroutine.wrap*() est plus simple à utiliser que *coroutine.create*() car elle donne exactement ce dont vous avez besoin : une fonction de reprise. Mais elle est aussi moins souple, car il n'existe aucun moyen de vérifier son statut et les erreurs éventuelles.

11-e - Le multithreading non-préemptif

Comme vu précédemment, les coroutines sont une sorte de multithreading collaboratif. Chaque coroutine est équivalent à un thread.

Une paire d'interrupteurs **resume-yield** contrôle les deux threads de l'un à l'autre. Cependant, contrairement au « vrai » multithreading, les coroutines sont non préemptives.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Lorsqu'une coroutine est en fonctionnement, elle ne pourra pas être arrêtée de l'extérieur. Elle ne suspendra son exécution que si une demande explicite est faite (via un appel à yield). Pour plusieurs applications, ce n'est pas un problème, bien au contraire. La programmation est beaucoup plus facile en l'absence de préemption. Vous n'avez pas besoin d'être paranoïaque à propos de la synchronisation des bugs, car toute synchronisation entre les threads est explicite dans le programme. Vous n'avez qu'à vous assurer qu'une *coroutine.yield*() intervienne seulement quand elle est en dehors d'une région critique.

Cependant, avec le multithreading non-préemptif, chaque fois qu'un thread appelle une opération de blocage, c'est l'ensemble du programme qui s'arrête jusqu'à la fin de l'opération. Or, pour la plupart des applications, c'est un comportement inacceptable, ce qui entraîne de nombreux programmeurs à faire abstraction des coroutines comme d'une véritable alternative au traditionnel multithreading. Mais comme vous allez le voir ci-dessous, ce problème a une intéressante et évidente solution.

Supposez une situation typique de multithreading : Vous voulez télécharger plusieurs fichiers à distance, via le protocole HTTP.

Bien sûr, pour télécharger plusieurs fichiers à distance, vous devez savoir comment télécharger un fichier distant. Dans cet exemple, vous allez utiliser la bibliothèque « luasocket », développée par Diego Nehab. (**que vous pouvez trouver ici.**)

Pour télécharger un fichier, vous devez ouvrir une connexion à un site, envoyer une demande au fichier, recevoir le fichier (en blocs) et fermer la connexion.

Dans Lua, vous pouvez écrire cette tâche comme suit :

Premièrement, chargez la bibliothèque « luasocket ».

```
require "luasocket"
```

Ensuite, vous définissez l'hôte et le fichier que vous voulez télécharger. Dans cet exemple, vous allez télécharger la spécification HTML 3.2 à partir du site du world wide web consortium.

```
host = "www.w3.org"

file = "/TR/REC-html32.html"
```

Ensuite, vous ouvrez une connexion TCP vers le port 80 (qui est le port standard pour les connexions HTTP) de ce site.

```
c = assert(socket.connect(host, 80))
```

L'opération retourne un objet de connexion que vous utiliserez pour envoyer la demande de fichier :

```
c:send("GET "..file.." HTTP/1.0\r\n\r\n")
```

La méthode de réception retourne toujours une chaîne de caractères, plus une autre chaîne avec l'état de l'opération.

Et finalement, vous fermez la connexion :

```
c:close()
```

Maintenant que vous savez comment télécharger un fichier, revenons au problème de téléchargement de plusieurs fichiers.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

L'approche triviale consisterait à télécharger un fichier à la fois. Mais, cette approche séquentielle, où l'on ne commence à lire un nouveau fichier qu'après avoir terminé la lecture du précédent, est trop lente.

Lors de la lecture d'un fichier distant, un programme passe le plus clair de son temps à attendre que les données arrivent. Plus précisément, il passe le plus clair de son temps bloqué dans l'attente de recevoir.

Aussi, pourrait-il être beaucoup plus performant s'il téléchargeait tous les fichiers simultanément. Lorsqu'une connexion n'aurait plus de données disponibles, le programme pourrait lire une autre connexion.

De toute évidence, les coroutines offrent un moyen pratique de structurer ces téléchargements simultanés. Vous créez un nouveau thread pour chaque tâche de téléchargement. Quand un thread n'a pas de données disponibles, il cède le contrôle à un répartiteur simple, qui invoque un autre thread.

Pour réécrire le programme avec les coroutines, vous devez d'abord réécrire le code précédent, comme une fonction de téléchargement.

```
function Download(host, file)
  local c = assert(socket.connect(host, 80))
  local count = 0    -- compte le nb d'octets lus
  c:send("GET "..file.." HTTP/1.0\r\n\r\n")
  while true do
    local s, status = receive(c)
    count = count + string.len(s)
      if status == "closed" then break end
  end
  c:close()
  print(file, count)
end
```

Puisque vous n'êtes pas intéressé par le contenu du fichier, cette fonction ne comptera que le nombre d'octets du fichier. Dans ce nouveau code, vous utiliserez une fonction auxiliaire (receive(c)) afin de recevoir les données de la connexion. Dans l'approche séquentielle, son code serait comme ceci :

```
function Receive(connexion)
  return connexion:Receive(2^10)
end
```

Pour la mise en œuvre simultanée, cette fonction doit recevoir les données sans blocage. Et s'il n'y a pas assez de données disponibles, alors, *coroutine.yield*() rentrera en fonction.

Le nouveau code ressemble à ceci :

```
function receive(connexion)
  connexion:timeout(0)
  local s, status = connexion:receive(2^10)
  if status == "timeout" then
    coroutine.yield(connexion)
  end
return s, status
end
```

L'appel à *timeout(0)* rend sur la connexion, toute opération « non-bloquante ».

Lorsque l'état de l'opération est « timeout », cela signifie que l'opération retournée n'est pas encore terminée et demande à la *coroutine.yield*(connexion) de faire son office.

L'argument « *non-false* » passé à yield signale au répartiteur que le thread doit toujours remplir sa mission. (Plus tard vous verrez une autre version où le répartiteur a besoin d'une connexion interrompue.)

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

La fonction suivante garantit que chaque téléchargement sera exécuté dans un thread individuel.

```
threads = {} -- liste de tous les threads.
function get(host, file)
  -- création de la coroutine.
  local co = coroutine.create(function()
    download(host, file)
  end)
  -- insère dans la liste.
  table.insert(threads, co)
end
```

Le tableau des threads conserve une liste de toutes les threads en direct, pour le répartiteur.

Le répartiteur est principalement une boucle qui passe par tous les threads, en les appelant un par un.

Il faut aussi enlever de la liste les threads qui terminent leurs tâches.

La boucle s'arrête lorsqu'il n'y a plus de thread.

```
function Dispatcher()
  while true do
    local n = table.getn(threads)
    if n == 0 then break end   -- il n'y a plus de thread
    for i=1,n do
      local status, res = coroutine.resume(threads[i])
      -- le thread a-t-il fini sa tâche?
      if not res then
        table.remove(threads, i)
        break
      end
    end
  end
end
```

Enfin, le programme principal crée les threads dont il a besoin et appelle le répartiteur.

Par exemple, pour télécharger quatre documents sur le site du W3C, le programme principal pourrait ressembler à ceci :

```
host = "www.w3.org"

get(host, "/TR/html401/html40.txt")
get(host,"/TR/2002/REC-xhtml1-20020801/xhtml1.pdf")
get(host,"/TR/REC-html32.html")
get(host,"/TR/2000/REC-DOM-Level-2-Core-20001113
    /DOM2-Core.txt")

-- appel à la boucle principale.
Dispatcher()
```

Si une machine met six secondes à télécharger ces fichiers en utilisant ces quatre coroutines, avec la mise en œuvre séquentielle, il lui faudra deux fois plus de temps (15 secondes).

Mais cette dernière application est loin d'être optimale.

Tout va pour le mieux tant qu'au moins un thread a quelque chose à lire.

Mais, dans le cas contraire, le répartiteur ne reste pas inactif et va de thread en thread à la recherche de quelque chose à se mettre sous la dent ...

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Ce qui signifie que cette application utilise le CPU presque 30 fois plus que la solution séquentielle !

Pour éviter un tel comportement, vous pouvez utiliser la sélection de fonction à partir de *luasocket*.

Ce qui permet au programme d'attendre sagement le changement d'état à l'intérieur du groupe de sockets. Les changements mis en œuvre sont de petite taille. Vous avez seulement changé le répartiteur.

```
function Dispatcher()
  while true do
    local n = table.getn(threads)
    if n == 0 then break end  -- il n'y a plus de thread.
    local connexions = {}
    for i = 1,n do
      local status, res = coroutine.resume(threads[i])
      if not res then  -- le thread a-t-il fini sa tâche?
        table.remove(threads, i)
        break
      else  -- timeout.
       table.insert(connexions, res)
      end
    end
    if table.getn(connexions) == n then
      socket.select(connexions)
    end
  end
end
```

Ce nouveau Dispatcher() recueille les connexions « time-out » dans le tableau connexions.

Rappelez-vous que Receive() passe les connexions à *yield* et que *resume* les retourne.

Lorsque toutes les connexions à « time-out » sont écoulées, le Dispatcher() appelle select().

Au final, cette mise en œuvre tourne aussi vite que celle faite à partir des coroutines.

Mais, compte tenu du fait que lorsque le CPU n'est pas occupé, c'est qu'il attend, on en déduit que son utilisation est juste un peu plus importante que lors de la mise en œuvre séquentielle.

11-f - À retenir

1. *une coroutine en Lua représente une unité d'exécution indépendante à la différence des unités d'exécution des systèmes multi-code. Néanmoins, une coroutine suspend seulement son exécution en appelant explicitement une fonction de production ;*
2. *vous créez une coroutine avec un appel à coroutine.create(). Son seul argument est une fonction qui est la principale fonction de la coroutine. Cette fonction crée seulement une nouvelle coroutine et lui retourne un objet de type « thread ». Elle ne démarre pas l'exécution de la coroutine ;*
3. *quand vous appelez en premier coroutine.resume(), passant en tant que son premier argument, une unité d'exécution renvoyée par coroutine.create(), la coroutine démarre son exécution à la première ligne de sa principale fonction. Des arguments supplémentaires passés à coroutine.resume() sont transmis à la fonction principale de la coroutine. Après que la coroutine ait démarré son exécution, elle fonctionne jusqu'à ce qu'elle se termine ou meure;*
4. *une coroutine peut terminer son exécution de deux façons :*
    1. *normalement, quand sa principale fonction return (explicitement ou implicitement, après la dernière instruction). Dans ce cas, coroutine.resume() renvoie true, plus n'importe quelle valeur renvoyée par la principale fonction de la coroutine,*

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

2. *anormalement, s'il y a une erreur non protégée, dans ce cas, coroutine.resume() renvoie false plus un message d'erreur ;*

5. *une coroutine est produite en appelant coroutine.yield(). Quand une coroutine est produite, la correspondance coroutine.resume() renvoie immédiatement le résultat, même si la production arrive à l'intérieur d'appels imbriqués de fonction (c'est-à-dire, pas dans la fonction principale, mais dans une fonction appelée directement ou indirectement par la fonction principale). En cas de production, coroutine.resume() renvoie également true, plus toutes les valeurs passées à coroutine.yield(). La fois suivante où vous reprenez la même coroutine, elle continue son exécution depuis le point où elle est sortie, avec l'appel à coroutine.yield() renvoyant tout argument supplémentaire passé à coroutine.resume() ;*
6. *tout comme coroutine.create(), la fonction coroutine.wrap() crée également une coroutine, mais au lieu de renvoyer la coroutine elle-même, elle renvoie une fonction qui, une fois appelée, reprend la coroutine. Tous arguments passés à cette fonction sont considérés comme arguments supplémentaires de coroutine.resume(). La fonction coroutine.wrap() renvoie toutes les valeurs retournées par coroutine.resume(), excepté la première (le code d'erreur booléen). À la différence de coroutine.resume(), coroutine.wrap() ne décèle pas les erreurs ; n'importe quelle erreur est propagée à la fonction*

12 - Les tables d'environnement

12-a - Définition

Lua conserve toutes ses variables globales dans une table ordinaire appelée : **table d'environnement.**

Un des avantages de cette structure, est que cette table d'environnement, simplifie l'implémentation interne de Lua, car vous n'avez pas forcément besoin de différentes structures de données pour les variables globales.

Mais le principal avantage reste la possibilité de manipuler cette table comme n'importe quelle autre table.

Car pour faciliter le travail, Lua stocke son propre environnement dans une variable globale, appelée **_G**.

Par exemple, le code suivant imprime les noms de toutes les variables globales définies dans l'environnement courant.

```
for n in pairs (_G) do print(n) end
```

Les paragraphes suivants, vous permettront d'apprendre plusieurs techniques utiles pour manipuler cette **table d'environnement**.

12-b - Affectation

Comment accéder aux variables globales avec des noms dynamiques ?

En règle générale, l'affectation est suffisante pour obtenir et définir les variables globales.

Mais la difficulté survient lorsque vous voulez manipuler une variable globale dont le nom est stocké dans une autre variable, ou lorsque vous voulez la calculer au moment de l'exécution.

Pour obtenir la valeur de cette variable, de nombreux programmeurs sont tentés d'écrire quelque chose comme :

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
loadstring("value = ".. varname)()

ou bien

value = loadstring("retour ".. varname)()
```

Si par exemple, varname a la valeur connue x, la concaténation se traduira par « retour x » (ou « value = x », avec la première forme).

Cependant, ces codes impliquent la création et la compilation d'un nouveau morceau de code et beaucoup de travail supplémentaire.

Vous pouvez accomplir la même chose avec le code suivant, beaucoup plus court et bien plus efficace que le précédent.

```
value = _G[varname]
```

Puisque l'environnement est une table ordinaire, vous pouvez simplement l'indexer avec le nom de la variable.

De la même manière, vous pouvez assigner directement une variable globale en écrivant : _G[varname] = valeur.

Mais attention, certains programmeurs quelque peu excités par ces fonctions peuvent arriver à écrire : _G["a"] = _G["var1"], qui est juste une manière compliquée d'écrire a = var1.

Une généralisation du problème précédent est de permettre d'accéder à tous les champs (mots, caractères, etc.) contenus dans un nom dynamique, comme *io.read* ou *a.b.c.d*.

Vous résoudrez ce problème avec une boucle qui démarre à _G et évolue, champ par champ.

```
function Getfield(f)
  local v = _G   -- début de la table _G
  for w in string.gmatch(f, "[%w_]+") do
    v = v[w]
  end
return v
end
```

La fonction *string.gmatch*(), de la bibliothèque string, itère tous les mots contenus dans **f**. (où « mots » est une séquence d'un ou plusieurs caractères alphanumériques et/ou de soulignement.)

La fonction correspondante utilisée pour définir des champs est un peu plus complexe.

Une affectation comme a.b.c.d.e = v est équivalente à :

```
local temp = a.b.c.d

temp.e = v
```

Autrement dit, pour récupérer le dernier nom, vous devez gérer le dernier champ séparément.

La fonction suivante Setfield() crée également des tables intermédiaires dans un chemin où elles n'existent pas.

```
function Setfield(f, v)
  local t = _G              -- on démarre avec la table Globale
  for w, d in string.gmatch(f, "([%w_]+)(.?)") do
    if d == "." then      -- pas le dernier champ?
      t[w] = t[w] or {} -- on crée une table si absente
      t = t[w]          -- get the table
    else                 -- dernier champs
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
        t[w] = v            -- on assigne
      end
    end
  end
```

Ce nouvel exemple capture le nom du champ dans la variable w et un point optionnel dans la variable d.

Si un nom de champ n'est pas suivi par un point, c'est qu'il est le dernier.

Avec la fonction précédente, l'appel à Setfield("t.x.y", 10) crée une table t globale, et une autre table t.x, et assigne 10 à t.x.y.

```
print(t.x.y)  --> 10

print(Getfield("t.x.y"))  --> 10
```

12-c - Environnement global

Avec Lua, les variables globales n'ont pas besoin d'être déclarées.

Si les petits programmes s'accommodent très bien de cette facilité, il n'en est pas de même pour les gros programmes où une simple faute de frappe peut provoquer d'énormes bogues.

Mais, vous pouvez changer un tel comportement, si vous le souhaitez. Puisque Lua conserve ses variables globales dans une table classique, vous pouvez utiliser une métatable pour modifier son comportement lors de l'accès aux variables globales.

```
setmetatable(_G, {
  __newindex = function(_, n)
  error("tentative d'écrire dans une variable non déclarée, "..n, 2)
  end,
  __index = function(_, n)
  error("tentative de lire une variable non déclarée, "..n, 2)
  end, })
```

Après ce code, toute tentative d'accéder à une variable globale inexistante déclenchera une erreur.

Mais dans ces conditions, comment pouvez-vous déclarer de nouvelles variables ?

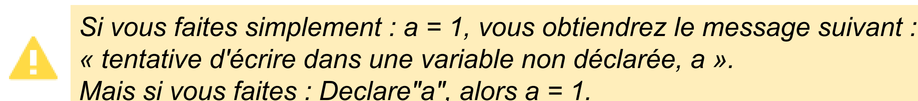
Tout simplement en utilisant la fonction *rawset*(), qui contourne la métaméthode.

```
function Declare(name, initval)
  rawset(_G, name, initval or false)
end
```

La présence de or et de false assure que la nouvelle variable globale obtienne toujours une valeur différente de nil.

N'oubliez pas de définir cette fonction avant d'installer le contrôle d'accès, sinon, vous obtiendrez une erreur.

Cette fonction mise en place, vous permet d'obtenir un contrôle total sur les variables globales.



Si vous faites simplement : a = 1, vous obtiendrez le message suivant :
« tentative d'écrire dans une variable non déclarée, a ».
Mais si vous faites : Declare"a", alors a = 1.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Mais maintenant, pour tester si une variable existe, vous ne pouvez pas simplement la comparer à nil, car si elle est nil, son accès lèvera une erreur.

Alors à la place, vous allez utiliser *rawget*(), afin d'éviter la métaméthode.

```
if rawget(_G, var) == nil then  -- "var" n'est pas déclarée
  ...
end
```

Il n'est pas difficile de changer cela afin de permettre le contrôle des variables globales de valeur égale à nil.

Vous avez simplement besoin d'une table auxiliaire qui maintienne les noms des variables déclarées.

Chaque fois qu'une métaméthode est appelée, elle vérifie dans ce tableau si la variable est ou non déclarée.

Le code pourrait ressembler à ce qui suit :

```
local declaredNames = {}

function Declare(name, initval)
  rawset(_G, name, initval)
  declaredNames[name] = true
end

setmetatable(_G, {
  __newindex = function(t, n, v)
  if not declaredNames[n] then
    error("tentative d'écrire dans une variable non déclarée, "..n, 2)
  else
    rawset(t, n, v)
  end
end,
  __index = function(_, n)
  if not declaredNames[n] then
    error("tentative de lire une variable non déclarée, "..n, 2)
  else
    return nil
  end
end, })
```

Pour ces deux solutions, le surcoût est négligeable.

Avec la première solution, les métaméthodes ne sont jamais appelées en fonctionnement normal.

Dans la seconde, elles peuvent être appelées, mais seulement lorsque le programme accède à une variable de valeur égale à nil.

12-d - Environnement non-global

Un des problèmes avec l'environnement global, est que justement, il est global ... Et toute modification affectera l'ensemble de votre programme.

Par exemple, lorsque vous installez une métatable pour contrôler l'accès global, votre programme dans son ensemble doit suivre ses directives. Et si vous souhaitez utiliser une bibliothèque qui utilise des variables globales sans les déclarer, vous êtes dans la « panade » ...

La version 5.0 de Lua avait introduit les fonctions *setfenv*() et *getfenv*() qui sont maintenant dépréciées.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Pour définir l'environnement d'une fonction Lua, vous devrez utiliser un environnement lexical ou la nouvelle fonction *loadin*(env, ...).

loadin(env, ...) est similaire à *load*(), sauf que *env* définit l'environnement de la fonction créée. Les paramètres après *env* sont similaires à ceux de *load*(). (*env* peut être une fonction Lua ou un nombre qui spécifie le niveau de la pile de la fonction, le niveau 1 étant la fonction d'appel.)

Il est bon de rappeler ici qu'un environnement lexical définit un nouvel environnement courant pour le code situé à l'intérieur de son bloc.

Autrement dit, un environnement lexical change la table utilisée pour résoudre tous les accès au niveau des variables globales, à l'intérieur d'un bloc.

12-e - À retenir

1. *avec Lua, chaque fonction et userdata possède une table qui lui est associée, appelée **table d'environnement** ;*
2. *comme les méta-tables, les tables d'environnement sont des tables normales et plusieurs objets peuvent partager la même table d'environnement ;*
3. *Lua, permet de manipuler l'environnement d'un objet uniquement par l'intermédiaire de la bibliothèque de débogage ;*
4. *les fonctions écrites en C et les userdatas, partagent l'environnement de la fonction C qui a été créée ;*
5. *les fonctions non imbriquées de Lua (créées avec loadfile, loadstring ou load) partagent l'environnement global ;*
6. *les fonctions imbriquées de Lua, partagent l'environnement courant, celui où elles sont définies ;*
7. *les environnements associés à userdata n'ont aucun sens pour Lua. C'est seulement une façon pratique pour les programmeurs, d'associer une table à un userdata ;*
8. *l'environnement associé à une fonction C, peut être directement accessible par le code C. Il est utilisé comme environnement par défaut, pour les autres fonctions C et userdatas créés par la fonction ;*
9. *l'environnement associé à une fonction Lua, est utilisé comme environnement courant pour tout le code de la fonction ;*
10. *un environnement lexical change l'environnement courant à l'intérieur de son champ d'application ;*
11. *dans tous les cas, l'environnement courant est la table que Lua utilise pour résoudre les variables globales et initialiser les fonctions imbriquées.*

13 - La bibliothèque Lua

13-a - Fonctions standards de base

_G : Variable globale.
_G n'est pas une fonction, mais une variable globale qui contient l'environnement global de Lua. Lua n'utilise pas cette variable pour lui-même. Pour définir l'environnement d'une fonction Lua, utilisez la nouvelle fonction loadin().

_VERSION : Variable globale.
_VERSION n'est pas une fonction, mais une variable globale qui affiche la version en cours de Lua.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

assert() : Teste la bonne exécution d'une fonction.

```
assert(test, "ici le message d'erreur")

-- Un exemple concret d'utilisation avec io.open().
file = assert(io.open(filename))
```

Cette fonction est semblable à celle utilisée en C.
Si la condition **test** est false ou nil, alors une erreur est retournée.
Dans le cas contraire le traitement sera effectué.
Un message facultatif peut être retourné en plus du message système.
Voir à ce sujet la fonction error() pour plus de détails.
Le dernier code ouvre **filename** en lecture et l'assigne à la variable **file**, ou retourne un message d'erreur.

collectgarbage() : Nettoie la mémoire.

```
collectgarbage(opt, arg)
```

Cette fonction est une interface générique pour le garbage collector qui remplit des fonctions différentes selon la valeur de **opt** :

stop	Arrête le garbage collector.
restart	Redémarre le garbage collector.
collect	Effectue une pleine collecte.
count	Retourne la mémoire totale utilisée par Lua (en Ko).
step	Effectue une collecte par pas. La taille du pas est contrôlée par arg.
setpause	Définit arg comme la nouvelle valeur pour la pause de garbage collector.
setstepmul	Définit arg comme la nouvelle valeur du multiplicateur d'étape du garbage collector.
isrunning	Retourne un booléen qui indique si le collecteur est opérationnel.

dofile() : Appelle un fichier Lua externe et exécute son contenu.

```
dofile("ici le nom du fichier")
```

Cette fonction ouvre le fichier nommé et exécute son contenu comme un morceau de Lua.
Lorsqu'elle est appelée sans arguments, dofile() exécute le contenu de l'entrée standard (stdin).
Retourne au programme appelant toutes les valeurs retournées par le programme appelé.
En cas d'erreurs, dofile() retourne l'erreur au programme appelant.
Cette fonction ne peut pas être utilisée en mode protégé.

error() : Affiche les erreurs d'une fonction.

```
error("ici le message", niveau)
```

Termine la dernière fonction protégée appelée et renvoie **"ici le message"** comme message d'erreur.
La fonction error() n'a pas de retour.
L'argument **niveau** indique où obtenir la position du message d'erreur.
Avec le niveau 1 (par défaut), la position de la fonction error() est la position où l'erreur sera appelée.
Le niveau 2, pointe sur la fonction qui a appelé error() et ainsi de suite.
Le niveau 0, évite l'ajout de la position des messages d'erreur.

getmetatable() : Recherche l'existence d'une métatable.

```
getmetatable(objet)
```

Cette fonction retourne nil si l'objet n'a pas de métatable ; la valeur associée au champ de la métatable, si cette dernière existe, ou la métatable de l'objet.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

ipairs() : Itération d'une table.

```
ipairs(t)
```

Cette fonction retourne 3 valeurs :
1 une fonction d'itération ;
2 la table t ;
3 nil.

De sorte que la construction :

```
for i, v in ipairs(t) do ... body ... end
```

parcourt l'ensemble de la table(t) de la façon suivante : t[1], t[2], etc.

load() : Charge un morceau (chunk) d'un programme Lua.

```
load(id, source, mode)
```

Si id est une fonction, load() l'appellera à plusieurs reprises afin d'obtenir tous les morceaux. Un retour sur une chaîne vide ou nil, signalera la fin du morceau.
Si id est une chaîne de caractères, alors le morceau sera cette chaîne.
S'il n'y a pas d'erreur, le morceau sera compilé comme une fonction.
S'il y a une erreur load() retournera nil plus le message d'erreur.

source	Est utilisé comme la source du morceau pour les messages d'erreur et les informations de débogage.
mode	Indique s'il s'agit de texte ou du binaire.
t	Signifie qu'il s'agit de texte. Le morceau sera alors pré-compilé.
b	Signifie qu'il s'agit d'un morceau en binaire. (La valeur par défaut est : bt les deux, binaire et texte).

loadin() : Charge un « morceau » en définissant son environnement.

```
loadin(env, ...)
```

Cette fonction est similaire à load(), sauf que **env** définit l'environnement de la fonction créée.
Les paramètres après **env** sont similaires à ceux de load().

loadfile() : Charge un fichier dont le nom est connu.

```
loadfile("nom du fichier")
```

Semblable à la load(), mais charge le fichier **"nom du fichier"** comme chaîne de caractères.
S'il n'y a aucune erreur, le retour peut être une fonction.
S'il y a une erreur, la fonction retourne nil avec un message d'erreur.
L'environnement de la fonction retourné est l'environnement global.

loadstring() : Charge une chaîne de caractères à partir d'un fichier.

```
loadstring(string, message)
```

Semblable à load(), mais charge le morceau de la chaîne donnée.
Le paramètre **message** est facultatif et sert à mentionner un message en cas d'erreur de chargement.
Pour charger et exécuter une chaîne donnée :

```
assert(loadstring(string))
```

next() : Permet de parcourir tous les champs d'une table.

```
next(table, index)
```

Le premier argument est une table et le deuxième un index dans la table.
La fonction retourne le prochain index de la table et la valeur liée à cet index.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

Si le deuxième argument est nil, alors la fonction retournera le premier index de la table et sa valeur associée.
Si la fonction est appelée avec le dernier index, ou avec nil dans une table vide, alors la fonction retournera nil.
Si le deuxième argument est absent, alors il est interprété comme nil.
Lua n'a aucune déclaration de champs. Il n'y a donc aucune différence entre un champ non existant et un champ avec une valeur nil.
Par conséquent, la fonction considère seulement les champs avec des valeurs non nil. L'ordre dans lequel les index sont énumérés n'est pas indiqué, même pour les index numériques.
Pour parcourir une table dans l'ordre numérique, utiliser un index numérique pour la fonction ipairs().

pairs() : Obtient une paire clé-valeur.

```
pairs(t)
```

Cette fonction retourne 3 valeurs :
1 la fonction suivante ;
2 la table t ;
3 nil.

De sorte que la construction :

```
for k, v in pairs(t) do ... body ... end
```

parcourt l'ensemble de toutes les clés-valeurs de la table(t).

pcall() : Appel d'une fonction en mode protégé.

```
pcall(f, arg1, ...)
```

Cette fonction appelle la fonction f avec les arguments fournis en mode protégé. Cela signifie que toute erreur à l'intérieur de f n'est pas propagée. pcall() attrape l'erreur et renvoie un code d'état.
Son premier résultat est le code d'état (un booléen) :
-true si l'appel réussit sans erreur plus tous les résultats de l'appel.
-false en cas d'erreur, plus le message d'erreur.

print() : Imprime les valeurs contenues entre parenthèses.

```
print(e1, e2, e3, ...)
```

Cette fonction imprime les valeurs séparées par des virgules vers la sortie standard.
Elle n'est pas prévue pour effectuer une sortie formatée, mais seulement comme un moyen rapide d'afficher une valeur.
Pour une sortie formatée, utilisez la fonction string.format().

```
print(string.format("Pi est approximativement %.4f", 22/7))
```

rawequal() : Test d'égalité.

```
rawequal(v1, v2)
```

Vérifie si v1 est égal à v2, sans appeler de métaméthode.
Cette fonction retourne un booléen.

rawget() : Obtient la valeur brute d'un champ de table.

```
rawget(table, index)
```

Obtient la valeur réelle de table[index], sans appeler de métaméthode.
table doit être une table et index peut avoir n'importe quelle valeur différente de zéro.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

rawset() : Définit la valeur brute d'un champ de table.

```
rawset(table, index, valeur)
```

Place **valeur** dans table[index], sans appeler de métaméthode.

- **table** doit être une table ;
- **index** peut avoir n'importe quelle valeur différente de zéro ;
- **valeur** n'importe quelle valeur de Lua.

Cette fonction retourne **table**.

require() : Charge un module.

```
require("nom du module")
```

Charge le module donné.
Cette fonction qui charge le module **"nom du module"**, commence par regarder dans la table _LOADED pour déterminer si le module a déjà été chargé.
S'il l'est, il demande en retour la valeur que le module a renvoyée quand il a été chargé la première fois.
Sinon, il recherche un chemin pour le charger.
Un message d'erreur sera retourné en cas de problème de chargement ou d'exécution du module, ou si require() ne peut pas trouver de chargeur.

select() : Sélectionne les arguments.

```
select(index, ...)
```

Si **index** est un nombre, la fonction retourne tous les arguments après le numéro de l'index.
Si ce nombre est négatif, la fonction retournera tous les arguments à partir de la fin (-1 étant le dernier argument).
Sinon, **index** doit être une chaîne "#", et alors select() retournera le nombre total des arguments supplémentaires qu'il a reçu.

setmetatable() : Définit une métatable associée à une table.

```
setmetatable(table, metatable)
```

Définit une métatable pour la **table** donnée.
Les valeurs ne peuvent pas être créées ou modifiées en Lua, pour ce faire vous devrez utiliser les fonctions « debug » via l'API C.
Si metatable est nil, alors la métatable existante sera supprimée.
Si la métatable originale possède un champ **__metatable**, alors une erreur sera générée.
Cette fonction retourne une table.

tonumber() : Conversion en valeur numérique.

```
tonumber(e, base)
```

tonumber() essaie de convertir l'argument **e** en nombre.
Si l'argument est déjà un nombre ou une chaîne de caractères représentant un nombre, alors la fonction tonumber() renvoie ce nombre, autrement elle renvoie nil.
Un argument facultatif indique la base pour interpréter le numéro.
La base peut être n'importe quel nombre entier entre 2 et 36, inclus.
Dans les bases au-dessus de 10 :

- la lettre A (en majuscule ou minuscule) représente 10 ;
- la lettre B représente 11, et ainsi de suite ;
- ainsi la lettre Z représentant 35.

Dans la base 10 (par défaut), le nombre peut avoir une partie décimale, ainsi qu'un exposant.
Dans d'autres bases, seul les nombres entiers non signés sont acceptés.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
tonumber("10",2)        -->  2
tonumber("F",16)        --> 15
tonumber("01010111",2)  --> 87
```

tostring() : Conversion en chaîne de caractères.

```
tostring(e)
```

tostring() reçoit un argument de n'importe quel type et le convertit en une chaîne de caractères dans un format raisonnable.
Pour un contrôle complet de la conversion de nombre en chaîne de caractères, se référer à la fonction string.format().
Si la metatable de e a un champ **__tostring**, alors tostring() appelle la valeur correspondante avec **e** comme argument, et emploie le résultat de l'appel en tant que résultat.
tostring() convertit son argument en chaîne de caractères. (juste son premier argument, s'il y a en plus d'un.)

type() : Indique le type d'une variable.

```
type(v)
```

Retourne le type de son seul argument, codé comme une chaîne.
Les résultats possibles de cette fonction sont **"nil"** (une chaîne de caractères et non pas la valeur nil), **"number"**, **"string"**, **"booléen"**, **"table"**, **"function"**,**"thread"** et **"userdata"**.

xpcall() : Appel d'une fonction avec définition du message d'erreur.

```
xpcall(f, err, arg1, ...)
```

Cette fonction est semblable à pcall(), sauf que vous pouvez définir une nouvelle fonction de traitement des erreurs en mode protégé : **err**.
Aucune erreur à l'intérieur de f n'est propagée. Au lieu de cela, xpcall() décèle l'erreur, appelle la fonction d'exception avec l'erreur d'origine et renvoie une chaîne de caractères comme message d'erreur (err).
Son premier résultat est le code d'état (un booléen) :

- true si l'appel réussit sans erreur plus tous les résultats de l'appel ;
- false en cas d'erreur, plus le message d'erreur : **err**.

13-b - Fonctions sur les coroutines

coroutine.create() : Crée une coroutine.

```
local maCoroutine = coroutine.create(function()
  ... ici le corps de la coroutine ...
end)
```

coroutine.resume() : Démarre l'exécution d'une coroutine.

```
local maCoroutine = coroutine.create(function()
  return 6, 7
end)

print(coroutine.resume(maCoroutine))

--> 6 7
```

coroutine.running() : Retourne la coroutine en cours d'exécution.
La fonction **coroutine.running()**, retourne la coroutine en cours d'exécution, ou nil.

coroutine.status() : Retourne l'état de la coroutine.
Cette fonction retourne l'état de la coroutine sous forme d'une string :

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

- **running** si elle fonctionne ;
- **suspended** si elle est encore en attente d'appel à yield, ou si elle n'a pas encore commencé son travail ;
- **normal** si elle est active mais ne travaille pas. (C'est-à-dire qu'elle **resume** une autre coroutine.) ;
- **dead** si la coroutine a terminé son travail, ou si elle s'est arrêtée avec une erreur.

coroutine.wrap() : Crée une nouvelle coroutine.

wrap ne retourne pas la coroutine, mais une fonction qui lorsqu'elle est appelée, reprend la coroutine.

```
function Perm(a)
  local n = table.getn(a)
  return coroutine.wrap(function() Permgen(a, n) end)
end
```

coroutine.yield() : Suspend l'exécution de la coroutine.

La fonction **coroutine.yield**, autorise une reprise ultérieure de son fonctionnement. Elle met en attente « maCoroutine ».

```
maCoroutine = coroutine.create(function()
  for i = 1, 10 do
    print("maCoroutine", i)
    coroutine.yield()
  end
end)
```

13-c - Fonctions d'exploitation

os.clock() : Retourne le temps écoulé en secondes.

```
os.clock() --> 11056.989
```

Cette fonction retourne le temps écoulé en secondes, depuis le début du programme.

os.date() : Retourne la date et l'heure courante.

```
os.date(format, time)
```

Retourne dans une table, les informations de date et heure formatées par une chaîne de caractères.
La chaîne de caractères a le même format que la fonction strftime() de C.
Si la chaîne de caractères de formatage utilisé est "*t" (os.date("*t")), alors la table retournée contient l'information de temps.
Si le format est précédé par "!" (os.date("!*t")) alors le temps est converti en coordonnées de temps universel.
os.date() retourne une chaîne de caractères dans le format mm/dd/yy.

```
os.date("%Y/%m/%d %H:%M:%S") --> "2010/03/04 09:05:16"

os.date() --> 11/09/11 10:26:05
```

os.difftime() : Retourne la différence de temps écoulé entre deux dates.

```
os.difftime(t2, t1)
```

Cette fonction retourne le nombre de secondes écoulées entre (t2 et t1).

os.execute() : Lance l'exécution d'un programme.

```
os.execute(nom du programme)
```

Cette fonction exécute une commande shell du système d'exploitation.
C'est comme la fonction system() de C.
Un code d'état, dépendant du système est retourné, fonction de l'exécution ou de **nom du programme**.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

os.exit() : Quitte le programme.

```
os.exit(code, close)
```

os.exit() appelle la fonction de sortie de C, avec un code facultatif, pour terminer le programme du centre serveur.
La valeur par défaut pour le code retour est le code de succès.
La valeur optionnelle du second argument true, ferme l'état dans lequel se trouve Lua avant de fermer proprement le système.

os.getenv() : Retourne le contenu de la variable d'environnement de l'OS.

```
os.getenv(varname)
```

Retourne la valeur **varname** de la variable du processus d'environnement, ou nil si la variable n'est pas définie.

```
os.getenv("BANANE")    --> nil
os.getenv("USERNAME") --> Claude
```

os.remove() : Supprime un fichier.

```
os.remove("nom du fichier")
```

Supprime le fichier et/ou le dossier avec le nom donné (nom du fichier).
Si cette fonction échoue, elle retourne nil, plus une chaîne de caractères décrivant l'erreur.

os.rename() : Renomme un fichier.

```
os.rename("ancien nom", "nouveau nom")
```

Si cette fonction échoue, elle retourne nil, plus une chaîne de caractères décrivant l'erreur.

os.setlocale() : Définit les paramètres régionaux.

```
os.setlocale(locale, category)
```

Permet de définir les paramètres régionaux utilisés par le programme.
category est optionnel et par défaut est **"all"**. ("all", "collate", "ctype", "monetary", "numeric", or "time".)
La fonction retourne le nom de la nouvelle variable locale, ou nil si la demande ne peut pas être honorée.

os.time() : Retourne une table de date et heure.

```
os.time(table)
```

Retourne l'heure actuelle lorsque os.time() est appelé sans arguments, ou un temps représentant la date et l'heure spécifiées par table.
Cette table doit avoir des champs année, mois et jour, et peut avoir des champs heures, minutes, secondes. Pour une description de ces champs, voir le paragraphe : table des codes de formatage.
La valeur retournée est un nombre, dont le sens dépend de votre système.
Dans POSIX, Windows et d'autres systèmes, ce nombre compte le nombre de secondes depuis un temps donné.
Dans d'autres systèmes, le sens n'est pas précisé, et le nombre retourné par le temps ne peut être utilisé que comme un argument de date et de os.difftime().

os.tmpname() : Création d'un fichier temporaire.

```
os.tmpname()
```

Retourne une chaîne avec un nom de fichier qui peut être utilisé comme un fichier temporaire.
Le fichier doit être explicitement ouvert avant son utilisation et explicitement fermé lorsqu'il n'est plus utilisé.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Lorsque c'est possible, IL EST PREFERABLE d'utiliser io.tmpfile(), qui supprime automatiquement le fichier à la fin du programme.

13-d - Fonctions d'entrée/sortie

io.close() : Fermeture de fichier.

```
io.close(file)
```

Équivalent à file:close().
Sans l'argument **file**, cette fonction clôt le fichier de sortie par défaut.

io.flush() : Vide le fichier.
Équivalent à file:flush() sur le fichier de sortie par défaut.

io.input() : Définit le fichier d'entrée par défaut.

```
io.input(file)
```

Lorsque cette fonction est appelée avec **file**, elle ouvre le fichier nommé (en mode texte), et l'utilise comme descripteur d'entrée par défaut.
Lorsque cette fonction est appelée avec un descripteur de fichier, elle établit simplement ce descripteur de fichier comme le fichier.
Lorsqu'elle est appelée sans paramètres, elle renvoie le descripteur de fichier d'entrée actuel par défaut.
En cas d'erreur, cette fonction déclenche l'erreur, au lieu de retourner un code d'erreur.

io.lines() : Lecture ligne par ligne.

```
io.lines("nom du fichier")
```

Ouvre le fichier **"nom du fichier"** en lecture et retourne une fonction d'itération qui, chaque fois qu'elle est appelée, retourne une nouvelle ligne à partir du fichier.

```
for line in io.lines("nom du fichier") do ... end
```

Cette syntaxe parcourt toutes les lignes du fichier.
Lorsque la fonction d'itération détecte la fin du fichier, elle ferme le fichier et renvoie nil pour terminer la boucle.
Un appel à io.lines(), sans **"nom du fichier"** est équivalent à io.input():line() qui parcourt les lignes du fichier d'entrée par défaut.

io.open() : Ouverture de fichier.

```
io.open("nom du fichier", mode)
```

Cette fonction ouvre un fichier, dans le mode spécifié par mode. (mode = chaîne de caractère)
Elle retourne un nouveau descripteur de fichier, ou en cas d'erreur, nil plus un message d'erreur.
La **chaîne mode** est exactement la même que celle qui est utilisée dans la fonction fopen() du langage C standard et peut avoir une des valeurs suivantes :

- **"r"** lecture (par défaut) ;
- **"w"** écriture ;
- **"a"** ajout ;
- **"r+"** mise à jour, toutes les données précédentes sont préservées ;
- **"w+"** mise à jour, toutes les données précédentes sont effacées ;
- **"a+"** mise à jour, les données précédentes sont conservées, et l'écriture n'est autorisée qu'à la fin du fichier.

La **chaîne mode** peut avoir un **b** la fin, ce qui est indispensable pour ouvrir un fichier en mode binaire.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

io.output() : Définit le fichier de sortie par défaut.

```
io.output(file)
```

Comparable à io.input(), mais fonctionne sur le fichier de sortie par défaut.

io.popen() : Lance un programme dans un processus séparé.

```
io.popen(prog, mode)
```

Lance **prog** dans un processus séparé et renvoie un descripteur de fichier que vous pouvez utiliser pour lire les données de ce programme, si le mode est **"r"**, (valeur par défaut) ou d'écrire des données à ce programme si le mode est **"w"**.
Cette fonction est dépendante du système et n'est pas disponible sur toutes les plate-formes.

io.read() : Lecture de fichier.

```
io.read(...)
```

Équivalent à io.input():read(...).

io.stderr() : Erreur standard.
stderr est un flux de redirection qui gère un message d'erreur.

io.stdin() : Entrée standard.
stdin est un flux de redirection qui représente l'entrée standard.
En règle générale, l'entrée standard est connectée au clavier.

io.stdout() : Sortie standard.
stdout est un flux de redirection qui représente la sortie standard.
En règle générale, la sortie standard est connectée à l'écran.

io.tmpfile() : Ouverture d'un fichier temporaire.

```
io.tmpfile()
```

Cette fonction retourne un descripteur de fichier temporaire.
Ce fichier est ouvert en mode mise à jour (append) et est automatiquement supprimé lorsque le programme se termine.

io.type() : Vérification de descripteur de fichier.

```
io.type(objet)
```

Vérifie si objet est un descripteur de fichier valide.
Renvoie la chaîne **"file"** si objet est un descripteur de fichier ouvert, **"closed file"** si objet est un descripteur de fichier fermé, et **nil** si objet n'est pas un descripteur de fichier.

io.write() : Écrit dans un fichier.

```
io.write(...)
```

Équivalent à io.output():write(...).

file:close(): Fermeture de fichier.

```
file:close()
```

Ferme le fichier file.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

file:flush(): Enregistre des données écrites.

```
file:flush()
```

Enregistre les données précédement écrites du tampon vers le fichier file.

file:lines(): Lecture ligne par ligne.

```
file:lines()
```

Retourne une fonction d'itération qui, chaque fois qu'elle est appelée, retourne une nouvelle ligne à partir du fichier file.
Par conséquent, la construction :

```
for ligne in file:lines() do ... corps ... end
```

va parcourir toutes les lignes du fichier file.
Mais contrairement io.lines(), cette fonction ne ferme pas le fichier lorsque la boucle est finie.

file:read(): Lecture de fichier.

```
file:read(format1)
```

Lit le fichier file, selon les formats de données (format1), qui précisent ce qu'il faut lire.
Pour chaque format, la fonction retourne une chaîne (ou un nombre) ainsi que les caractères lus, ou nil si elle ne peut pas lire de données avec le format spécifié.
Lorsqu'elle est appelée sans format, elle utilise un format par défaut qui lit la ligne suivante en entier.
Les formats disponibles sont :

- **"*n"** lit un nombre, c'est le seul format qui retourne un nombre au lieu d'une chaîne ;
- **"*a"** lit le fichier en entier, à partir de la position actuelle. À la fin du fichier, elle retourne une chaîne vide ;
- **"*l"** lit la ligne suivante (en sautant à la fin de ligne), retourne nil à la fin du fichier. Il s'agit du format par défaut ;
- **number** lit une chaîne avec un nombre maximum de caractères, retourne nil à la fin du fichier. Si le nombre est égal à zéro, elle ne lit rien et renvoie une chaîne vide ou nil à la fin du fichier.

```
local bytes = file:read(4) --> lit le fichier file par blocs de 4 octets
```

file:seek(): Se positionne sur l'index dans un fichier.

```
file:seek(start, offset)
```

Définit et retourne la position de l'index pour le fichier file, mesurée à partir du début du fichier, jusqu'à la position donnée par offset en plus d'une base (start) spécifiée par la chaîne, comme suit :

- **"set"** est la position 0 (début du fichier) ;
- **"cur"** est la position courante ;
- **"end"** est la fin du fichier.

En cas de succès, la fonction retourne la position du fichier, exprimée en octets depuis le début du fichier.
Si cette fonction échoue, elle retourne nil, en plus d'une chaîne de caractères décrivant l'erreur.
La valeur par défaut pour start est **"cur"**, et pour offset est **0**.
Par conséquent :

```
file:seek() retourne la position du fichier en cours, sans le modifier.
file:seek("set") fixe la position au début du fichier et retourne 0.
file:seek("end") fixe la position à la fin du fichier, et retourne sa taille.
```

file:setvbuf(): Définit le mode de mise en mémoire tampon.

```
file:setvbuf(mode, size)
```

Définit le mode de mise en mémoire tampon pour un fichier de sortie.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Il y a trois modes disponibles :

- **"no"** pas de tampon. Le résultat de toute opération de sortie apparaît immédiatement ;
- **"full"** mise en mémoire tampon complète. L'opération de sortie est réalisée uniquement lorsque la mémoire tampon est pleine ou lorsque vous videz le fichier explicitement (voir io.flush()) ;
- **"line"** mise en mémoire tampon de ligne. La sortie est mise dans le tampon jusqu'à ce qu'une nouvelle ligne apparaisse ou qu'il y ait apport de fichiers spéciaux (comme ceux d'un terminal).

Pour les deux derniers cas, size spécifie la taille du tampon en octets.
La valeur par défaut est une taille appropriée en fonction du programme en cours.

file:write(): Écrire dans un fichier.

```
file:write(valeur1, ...)
```

Écrit la valeur de chacun de ses arguments dans le fichier file.
Les arguments doivent être des chaînes ou des numéros.
Pour écrire d'autres valeurs, utiliser tostring() ou string.format() avant d'écrire dans le fichier.
En cas de succès, cette fonction retourne file.
Sinon, elle retourne nil, plus une chaîne décrivant l'erreur.

13-e - Fonctions mathématiques

math.abs() : Retourne la valeur absolue de x.

```
math.abs(-100) ---> 100
```

math.acos() : Retourne l'arc cosinus de x. (x étant exprimé en radians.)

```
math.acos(0) ---> 1.5707963267949
```

math.asin(x) : Retourne l'arc sinus de x. (x étant exprimé en radians.)

```
math.asin(1) ---> 1.5707963267949
```

math.atan(x) : Renvoie l'inverse de la tangente.

```
c, s = math.cos(0.8), math.sin(0.8)

math.atan(s/c) ---> 0.8

Ici nous fournissons le résultat de s divisé par c (s/c).
```

math.atan2() : Renvoie l'inverse de la tangente.

```
c, s = math.cos(0.8), math.sin(0.8)

math.atan2(c, s) ---> 0.8

Cette fonction doit être utilisée de préférence et en particulier quand on convertit
 des coordonnées rectangulaires en coordonnées polaires.
math.atan2 emploie le signe des deux arguments pour placer le résultat dans le
 quart de cercle correct, et produit également des valeurs correctes quand un de ses
 arguments est 0 ou très de près de 0.
```

math.cos(x) : Retourne le cosinus de x. (x étant exprimé en radians.)

```
math.cos(math.pi/4) ---> 0.70710678118655
```

math.cosh(x) : Retourne le cosinus hyperbolique de x.

math.deg(x) : Retourne l'angle x en degrés.

```
math.deg(x)

-- x est donné en radians.
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

math.exp(x) : Retourne la valeur e puissance x.

```
math.exp(x)

e = base des logarithmes normaux
```

math.floor(x) : Retourne le plus grand entier inférieur ou égal à x.

```
math.floor(0.5) ---> 0
```

math.fmod(x, y) : Retourne le reste d'une division x/y.

```
math.fmod(42,5) ---> 2
```

Vous pouvez aussi utiliser l'opérateur %.
1%6 est équivalent à math.fmod(1, 6).

math.frexp(x) : Retourne m et e tel que x = m*2e.

```
math.frexp(x)
```

e est un entier et la valeur absolue de m est dans l'intervalle (0.5, 1) (ou zéro lorsque x est égal à zéro).

math.log() : Retourne le logarithme de ...

```
math.log(x, base)
```

Retourne le logarithme de x en fonction de **base**.
Par défaut, **base** est égale à e. De sorte que la fonction retourne le logarithme naturel de x.

math.max(liste) : Retourne le plus grand nombre contenu dans liste.

```
math.max(1.2, -7, 3) ---> 3
```

math.min(liste) : Retourne le plus petit nombre contenu dans liste.

```
math.min(1.2, -7, 3) ---> -7
```

math.modf(x) : Retourne deux nombres.

```
math.modf(x)
```

Retourne deux nombres, la partie entière de x et la partie fractionnaire de x.

math.pi : Retourne la valeur de pi.

```
math.pi ---> 3.1415926535898
```

math.pow(x, y) : Retourne x à la puissance y.

```
math.pow(x, y)
```

Vous pouvez aussi utiliser l'expression : x^y.

math.rad(x) : Retourne l'angle x. (x étant exprimé en radians.)

```
math.rad(x)
```

x est donné en degrés.

math.random() : Générateur de nombres pseudo-aléatoires.

```
math.random(m, n)
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Cette fonction est une interface pour un générateur de nombre pseudo-aléatoire fournie par ANSI C. (Aucune garantie ne peut être donnée pour ses propriétés statistiques.)
Lorsqu'elle est appelée sans argument, math.random() retourne un nombre pseudo-aléatoire réel dans l'intervalle [0,1].
Lorsqu'elle est appelée avec un nombre entier **m**, math.random() retourne un nombre pseudo-aléatoire entier dans l'intervalle [1, m].
Lorsqu'elle est appelée avec deux nombres entiers **m** et **n**, math.random() retourne un nombre pseudo-aléatoire entier dans l'intervalle [m, n].

math.randomseed() : Générateur de nombres pseudo-aléatoire à partir d'une base.

```
math.randomseed(x)
```

Définit x comme la « base » pour le générateur de nombre pseudo-aléatoire :
Une bonne base est os.time(), mais il faut attendre au moins une seconde avant d'appeler la fonction pour obtenir un autre nombre aléatoire.
Pour obtenir une suite de nombres aléatoires :

```
math.randomseed(os.time())
```

L'initialisation devrait pouvoir être meilleure si Lua pouvait obtenir des millisecondes de os.time().
Mais prenez garde, le premier nombre aléatoire que vous obtenez n'est pas vraiment aléatoire.
Pour obtenir un meilleur nombre pseudo-aléatoire il faudrait pouvoir générer plusieurs nombres aléatoires, avant d'en obtenir un vrai.

math.sin(x) : Retourne le sinus de x. (x étant exprimé en radians.)

```
math.sin(0.123) ---> 0.12269009002432
```

math.sinh(x) : Retourne le sinus hyperbolique de x.

```
math.sinh(1) ---> 1.1752011936438
```

math.sqrt(x) : Retourne la racine carrée de x.

```
math.sqrt(x)
```

Vous pouvez également utiliser l'expression x^0,5 pour calculer cette valeur.

math.tan(x) : Retourne la tangente de x. (x étant exprimé en radians.)

```
math.tan(5/4) ---> 3.0095696738628
```

math.tanh(x) : Retourne la tangente hyperbolique.

13-f - Fonctions sur les chaînes de caractères

string.byte() : Retourne le code numérique d'un ou plusieurs caractères. (code ASCII)

```
print(string.byte("ABCDE"))
 --> Retourne la valeur ASCII du premier caractère: 65

print(string.byte("ABCDE", 2))
 --> Retourne la valeur ASCII du second caractère: 66

print(string.byte("ABCDE", 1, 5))
 --> Retourne la valeur ASCII de tous les caractères: 65 66 67 68 69
```

string.char() : Retourne un caractère ou une chaîne à partir de son code ASCII.

```
print(string.char(65))
  --> Retourne le caractère dont le code ASCII est 65: A

print(string.char(67, 108, 97, 117, 100, 101))
  --> Retourne une chaîne: Claude
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

string.dump() : Retourne une chaîne en une représentation binaire de la fonction donnée.

```
s = string.dump(f)
```

Convertit une fonction **f** en représentation binaire, qui peut être ensuite traitée par loadstring pour récupérer la fonction. La fonction doit être une fonction Lua sans upvalue (10) .

```
function f() print "hello, World" end
s = string.dump(f)
assert (loadstring (s)) () --> hello, World
```

string.find() : Retourne l'emplacement du caractère ou de la sous-chaîne recherché.

```
string.find(string, pattern, init, plain)
-- Recherche la première occurrence du pattern dans une chaîne de caractères
-- et retourne la position du début et de la fin du mot qu'elle a trouvé.

print(string.find("Bonjour tout le monde.", "our"))
--> Résultat =  5 7

print(string.find("Bonjour tout le monde.", "pomme"))
--> Résultat = nil

print(string.find("Bonjour tout le monde.", "our", 6))
-- Recherche "our" à partir du 6e caractère.
--> Résultat = nil

print(string.find("Bonjour tout le monde.", "%s"))
-- Recherche l'emplacement de la première espace.
--> Résultat = 8
```

string.format() : Retourne une chaîne de caractères formatée (**voir liste**).

```
string.format(pattern, e1, e2, ...)
```

Crée une chaîne de caractères formatée et composée des arguments e1, e2 tout en respectant le format défini par **pattern**.

```
s = "Pour manier %s, vous devez être niveau %i"

print(string.format(s, "l'épée", 10))
```

Pour manier l'épée, vous devez être niveau 10. Dans cet exemple %s a été remplacé par l'épée et %i par 10.

```
print(string.format ("%q", 'un chat appelé "Pussy"'))

  --> "un chat appelé \"Pussy"\"
```

%q - formate une chaîne de telle sorte Lua puisse la lire.
%s - affiche une chaîne de caractères (ou quelque chose qui puisse être convertie en une chaîne, comme un numéro).
%% - Un caractère de sortie simple%.

string.match() : Retourne une sous-chaîne de caractères.

```
for w in string.gmatch("Pierre marche dans la rue", "%a+") do
  print(w)
end

Pierre
marche
dans
la
rue
```

Cette fonction renvoie un modèle trouvant un itérateur.
L'itération recherche dans la chaîne, la sous chaîne passée à pattern.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

string.gsub() : Remplace les occurrences d'une sous-chaîne.

```
s, n = string.gsub(string, pattern, remplacement, n)
```

Retourne une copie de string dont les mots représentés par **pattern** ont été remplacés par **remplacement**, pour un maximum de **n** fois.

```
string = "Pierre mange du poisson"

print(string.gsub(string, "du poisson", "des chips"))

--> Pierre mange des chips 1 (1 étant le nombre de substitution).
```

string.len() : Retourne la longueur d'une chaîne de caractères.

```
s = "Retourne la longueur de la chaîne en octets, y compris les espaces."

n = string.len(s)

print(n)

-- Résultat = 67
```

Vous pouvez également utiliser l'opérateur unitaire dièse(**#**) : print(**#**s)

string.lower() : Convertit les CAPITALES en minuscules.

```
s = string.lower(string) -- convertit string en minuscules

print(string.lower("ABCdef"))

-- Résultat = abcdef
```

string.match() : Retourne une sous-chaîne de caractères.

```
s = string.match(string, pattern, index)
```

Trouve la première occurrence de pattern, commençant à la position "index". C'est similaire à string.find(), sauf que les indices de début et de fin ne sont pas retournés.

```
s = "Vous voyez des chiens et des chats"
print(string.match(s, "v...."))

-- Résultat = voyez
```

string.rep() : Retourne une chaîne par concaténation multiple.

```
s = string.rep(string, n)

print(string.rep ("oui ", 4))

-- Résultat = oui oui oui oui
```

string.reverse() : Inverse une chaîne de caractères.

```
s = string.reverse(string)

print(string.reverse("La cigale ayant chanté tout l'été"))
-- Résultat = été'l tuot étnahc tnaya elagic aL

Et bien sûr:

print(string.reverse("été'l tuot étnahc tnaya elagic aL"))
-- Résultat = La cigale ayant chanté tout l'été
```

string.sub() : Retourne une sous-chaîne.

```
s = string.sub(string, start, end)
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

Retourne une sous-chaîne de **string**, à partir de l'index **start** et se termine à l'index **end**. Les deux peuvent être négatifs indiquant ainsi un début à droite.
end est facultatif et est par défaut égal à -1, indiquant ainsi le reste de la chaîne.

```
print(string.sub ("ABCDEF", 2, 3)) -- Résultat = BC

print(string.sub ("ABCDEF", 3)) -- Résultat = CDEF

print(string.sub ("ABCDEF", -1)) -- Résultat = F
```

string.upper() : Convertit les minuscules en CAPITALES.

```
s = string.upper(string)

print(string.upper("ABCdef")) -- Résultat = ABCDEF
```

Liste récapitulative sur les PATTERNS.

```
%c - convertit un nombre en caractère.(identique à string.char()).
print(string.format("%c", 65)) -- Résultat = A

%d et %i - convertit un nombre décimal en entier.
print(string.format("%i", 123.456)) -- Résultat = 123

%o - convertit en octal.
print(string.format("%o", 16)) -- Résultat = 20

%u - convertit un nombre non signé en un entier.
print(string.format("%u", 1234.566)) -- Résultat = 1234
print(string.format("%u", -1234)) -- Résultat = 429496606

%x - convertit en hexadécimal (en minuscules).
print(string.format("%x", 86543)) -- Résultat = 1520f

%x - convertit en hexadécimal (en capitales).
print(string.format("%X", 86543)) -- Résultat = 1520F

%e - notation scientifique, "e" en minuscule.
print(string.format("%e", 15)) -- Résultat = 1.500000e+001

%E - notation scientifique, "E" en capitale.
print(string.format("%E", 15)) -- Résultat = 1.500000E+001

%g - valeur signée imprimée au format %f ou %e, selon le plus
compacte pour une valeur donnée.
print(string.format("%g", 15.656)) -- Résultat = 15.656
print(string.format("%g", 1)) -- Résultat = 1
print(string.format "%g", 1.2)) -- Résultat = 1.2
```

13-g - Fonctions sur les tables

Les fonctions *table.foreach()* , *table.foreachi()* et *table.setn()* sont à ce jour obsolètes et ne figurent donc pas dans cette liste.

table.concat() : Concatène les éléments d'une table.

```
s = table.concat(maTable, séparateur, start, end)
```

Renvoie les entrées numériques de **maTable**, concaténées avec **séparateur,** à partir de l'index **start** et se terminant à l'index **end**.

```
maTable = {"Le", "renard", "roux", "est", "rapide"}

print(table.concat(maTable, " - "))

-- Résultat: Le - renard - roux - est - rapide
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

table.getn() : Retourne la taille d'une table à indexation numérique.

```
n = table.getn(maTable)
-- Ne concerne que les tables à indexation numérique.

print(table.getn{"Le", "renard", "roux", "est", "rapide", nom = "Nick"})

-- Résultat: 5
-- (L'entrée nom = "Nick" n'est pas une entrée dont l'index est numérique.)
```

Comme pour les strings, vous pouvez utiliser l'opérateur unaire dièse (#).

```
maTable = {"Le", "renard", "roux", "est", "rapide", nom = "Nick"}

print(#maTable) --> 5
```

table.insert() : Insère une valeur donnée dans une table à indexation numérique.

```
table.insert(maTable, pos, value)
```

Insère la nouvelle valeur **value** à la position **pos** avec une renumérotation de l'ensemble des champs.

```
maTable = {"Le", "renard", "roux", "est", "rapide"}
table.insert(maTable, 5, "très")
for i, v in ipairs(maTable) do
  print(i.." - "..v)
end

-- Résultat:
1 - Le
2 - renard
3 - roux
4 - est
5 - très
6 - rapide
```

table.pack() : Retourne une nouvelle table.

Cette fonction retourne une nouvelle table avec tous les paramètres stockés dans les clés 1, 2, etc. et un champ **n** représentant le nombre total de paramètres.

table.remove() : Supprime un élément d'une table à indexation numérique.

```
val = table.remove(maTable, pos)
```

Supprime l'élément à la position **pos** de **maTable** et retourne la valeur de l'élément supprimé. Si **pos** est omis, c'est par défaut le dernier élément de maTable, qui sera supprimé.

```
maTable = {"Le", "renard", "roux", "est", "très", "rapide"}
table.remove(maTable, 5)
for i, v in ipairs(maTable) do
  print(i.." - "..v)
end

-- Résultat:
1 - Le
2 - renard
3 - roux
4 - est
5 - rapide
```

table.sort() : Trie les éléments d'une table.

```
table.sort(maTable, f)
```

Trie **maTable** en utilisant la fonction **f** fournie en tant que fonction de comparaison pour chaque élément.

```
maTable = {3, 2, 5, 1, 4}
table.sort(maTable)
print(table.concat(maTable, ", ")) --> 1, 2, 3, 4, 5
```


Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.

```
-- Le tri se fait automatiquement dans un ordre croissant (opérateur <).
```

Et pour le mettre dans un ordre décroissant (utilisation de f) :

```
table.sort(maTable, function(a,b) return a > b end)

print(table.concat(maTable, ", ")) --> 5, 4, 3, 2, 1
```

table.unpack() : Retourne les éléments de la table donnée.

```
print(unpack{10,20,30}) --> 10 20 30
a, b = unpack{10,20,30}
print(a, b)

-- Résultat: 10 20 (30 sera oublié, car pas affecté)
```

Cette fonction spéciale possède plusieurs valeurs de retour. Elle reçoit un tableau et renvoie les résultats de tous les éléments du tableau, en commençant à l'index 1.

13-h - Table des codes de formatage

%a	Nom abrégé du jour de la semaine (local).
%A	Nom complet de jour de la semaine (local).
%b	Nom abrégé du mois (local).
%B	Nom complet du mois (local).
%c	Représentation préférée pour les dates et heures en local.
%C	Numéro de siècle. (L'année divisée par 100 et arrondie entre 00 et 99.)
%d	Jour du mois en numérique de 01 à 31.
%D	Identique à %m/%d/%y.
%e	Numéro du jour du mois. (Les chiffres sont précédés d'une espace.)
%g	Identique à %G sur deux chiffres.
%G	L'année sur quatre chiffres correspond au numéro de la semaine (voir %V).

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

	Même format et valeur que %Y sauf que si le numéro de la semaine appartient à l'année précédente ou suivante, l'année courante sera utilisée à la place.
%h	Identique à %b.
%H	Heure de la journée en numérique et sur 24 heures, de 00 à 23.
%I	Heure de la journée en numérique et sur 12 heures, de 01 à 12.
%j	Jour de l'année en numérique, de 001 à 366.
%m	Mois en numérique, de 1 à 12.
%M	Minute en numérique.
%n	Caractère de nouvelle ligne.
%P	AM ou PM en majuscule, suivant la valeur donnée ou le chaîne correspondante à la localité courante.
%p	am ou pm en minuscule, suivant la valeur donnée ou le chaîne correspondante à la localité courante.
%r	L'heure au format a.m et p.m.
%R	L'heure au format 24h.
%S	Secondes en numérique.
%t	Tabulation.
%T	L'heure actuelle (égale à %H:%M:%S).
%u	Le numéro du jour dans la semaine de 1 à 7. (1 représentant le lundi.)
%U	Numéro de la semaine dans l'année, en considérant le premier dimanche de l'année comme le premier jour de la première semaine.
%V	Le numéro de la semaine comme défini dans l'ISO 8601:1988, sous forme décimale de 01 à 53. La semaine 1 est la première semaine qui a plus de 4 jours ans l'année courante et dont lundi est le premier jour.
%W	Numéro de la semaine dans l'année, en considérant le premier lundi de l'année comme le premier jour de la semaine.
%w	Jour de la semaine en numérique avec dimanche = 0.
%x	Format préféré de représentation de la date sans l'heure.
%X	Format préféré de représentation de l'heure sans la date.
%y	L'année en numérique sur deux chiffres de 00 à 99.
%Y	L'année en numérique sur quatre chiffres.
%Z ou %z	Fuseau horaire, nom ou abréviation en fonction du système d'exploitation.
%%	Un caractère "%" littéral.

14 - Remerciements

Il ne serait pas convenable de terminer ce tutoriel sans adresser un grand remerciement à **LittleWhite** et **ALT** pour toute l'aide technique et orthographique qu'ils m'ont apportée lors de la rédaction de ce document sans oublier bien évidemment le responsable des corrections **Max** et l'ensemble de l'équipe de developpez.com toujours présente et prête à répondre à la moindre question.

À vous tous, MERCI.

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.
http://wxlua.developpez.com/tutoriels/lua/general/cours-complet/

1 : *Un WRAPPER est un programme qui enveloppe un programme différent, permettant ainsi son exécution dans un environnement spécifique. Il s'agit ici de : wxLuaFreeze.exe.*
2 : *Cet anglicisme qui signifie littéralement: « se tenir seul », est un qualificatif qui indique qu'un produit peut être utilisé seul, c'est-à-dire sans modules ou connaissances complémentaires. Un standalone désigne une application à part entière, qui se différencie donc d'une extension (ou add-on) ou d'un plugin (ou greffon).*
3 : *Littéralement, ramassage des déchets, souvent appelé « ramasse-miettes ».*
Lua effectue une gestion automatique de la mémoire en exécutant de temps en temps un garbage collector pour supprimer définitivement de la mémoire tous les objets morts qui ne sont plus accessibles depuis Lua. (Tables, userdatas, fonctions, threads, strings, variables, etc.).
4 : *Se dit d'une fonction qui est passée en argument à une autre fonction.*
5 : *Une variable libre est une variable référencée dans une fonction et qui n'est pas une variable locale, ni un paramètre de cette fonction.*
6 : *En informatique, stateless se rapporte à un système ou à un protocole qui ne garde pas un état persistant entre les transactions. iterator.*
7 : *Nom du programme souvent utilisé dans une configuration de « pipeline ».*
8 : *En informatique, la préemption est la capacité d'un système d'exploitation multitâche à exécuter ou stopper une tâche planifiée en cours. Et donc à contrario, un système d'exploitation* non-préemptif*, ou* collaboratif*, est un système dans lequel c'est le processus en cours d'exécution qui prend la main et est seul juge du moment où il la rend.*
9 : *Pipe, tube ou encore pipeline est un mécanisme qui permet de chaîner des processus de sorte que la sortie d'un processus (stdout) alimente directement l'entrée (stdin) du suivant. Chaque connexion est implantée par un tube anonyme. Les programmes « filtres » sont souvent utilisés dans cette configuration.*
10 : *Une variable locale externe, utilisée à l'intérieur d'une fonction est appelée « upvalue »*

Copyright ® 2013 Claude URBAN. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts.