
Lua

Dans ce premier chapitre, nous allons progressivement introduire le minimum nécessaire pour pouvoir commencer à écrire des petits programmes très simples en Lua.

Premier module

L'espace module

Pour écrire un programme en Lua, nous avons, tout d'abord, besoin d'un endroit pour l'écrire. De même que les modèles s'écrivent dans des pages nommées : **Modèle:«Nom de la page»**, les programmes Lua s'écriront dans des pages nommées : **Module:«Nom de la page»**. On dira que l'on écrit dans l'espace module.

Une fois le programme écrit dans l'espace module, nous devons être capable de pouvoir l'utiliser à partir d'une autre page (de même qu'un modèle peut être inclus dans une page en écrivant `{{Nom du modèle}}`).

Exemple de module

Nous allons prendre un exemple très simple pour bien comprendre. Supposons que l'on veuille écrire un programme qui, lorsqu'on l'appelle, nous renvoie le message : « Coucou, c'est moi ! ». Nous commencerons par créer une page du style : Module:Exemple simple et dans cette page, nous écrirons :

```
local p = {}

function p.Salutation()
    return "Coucou, c'est moi !"
end

return p
```

Nous allons vous expliquer ce que signifie chacune des lignes du module.

La table *p*

Tout d'abord, nous avons une première ligne : `local p = {}`. Le mot `local` est réservé par le langage Lua (de tels mots-clés apparaissent en couleur). Cela signifie que lorsqu'il sera employé, il aura un sens qui a été défini par les concepteurs du langage Lua. Ce mot nous indique que ce qui est écrit juste après n'est valable que dans la page ou à l'intérieur d'un bloc d'instructions (nous reviendrons sur cette notion quand nous préciserons ce que l'on entend par « bloc d'instructions »). Après le mot `local`, nous voyons : `p = {}`. *p* est une table (dans d'autres langages, on dit plutôt tableau). Dans un langage de programmation, une table (ou tableau) peut être vue comme une armoire avec des tiroirs, dans lesquels on peut ranger toutes sortes d'objets (nous verrons lesquels, plus tard). Comment savons-nous que *p* est une table ? C'est à cause de `= {}`. Les accolades ouvrantes et fermantes symbolisent une table en Lua (attention, ce n'est pas vrai dans tous les langages !). La ligne : `local p = {}` signifie donc : « *p* est une table qui ne peut être utilisée que dans ce module ».

Retenez bien, si vous ne le savez pas, que ce que vous apprenez ici est propre au langage Lua. Les créateurs de langages, assez mystérieusement, prennent plaisir à nous embrouiller en définissant les choses différemment. En langage C par exemple, un tableau ne se définit pas de la même manière.

La fonction *p.Salutation*

Nous avons ensuite les lignes :

```
function p.Salutation()
    return "Coucou, c'est moi !"
end
```

`function` est un autre mot-clé qui sert à définir une fonction. Ici, on définit la fonction *p.Salutation* qui sera rangée dans la table *p* (l'un des tiroirs de l'armoire *p*, pour reprendre l'analogie utilisée plus haut), c'est pour cela que l'on écrit `function p.Saluation`. Si l'on avait écrit seulement `function Salutation()`, on aurait bien défini une fonction *Salutation* mais elle ne se trouverait pas dans la table *p*. Il est nécessaire de mettre la fonction *p.Salutation* dans la table *p* pour que cette fonction puisse être retournée en même temps que le contenu de la table *p* grâce à l'instruction `return p` se trouvant en fin de programme (nous reviendrons là-dessus plus tard).

En dessous de la définition de la fonction, nous avons la ligne : `return "Coucou, c'est moi !"` qui constitue le bloc d'instructions décrivant le programme de la fonction. Nous disons bloc d'instructions car il y aurait pu y avoir plusieurs instructions si la fonction avait été plus complexe. Ce bloc d'instructions se terminant par le mot-clé `end` indiquant que la programmation de la fonction est terminée.

Si l'on analyse plus en détail l'instruction : `return "Coucou, c'est moi !"`, nous voyons qu'elle débute par le mot-clé `return`. `return` indique ce qui doit être retourné par la fonction. Le rôle d'une fonction est, la plupart du temps, de retourner quelque chose. En mathématiques la fonction **f(x) = 3x + 2** retourne le résultat du calcul de l'expression **3x + 2** en remplaçant **x** par un nombre particulier. En Lua, on programmerait cette fonction ainsi :

```
function f(x)
    return 3 * x + 2
end
```

Nous remarquons les parenthèses ouvrante `(` et fermante `)`, présentes aussi dans notre programme dans la ligne : `function p.Salutation()`. Mais il n'y a rien à l'intérieur car la fonction *p.Salutation* se contente de retourner une phrase sans qu'on lui transmette aucune information (il n'en sera pas toujours ainsi).

Après le mot-clé : `return`, nous avons : `"Coucou, c'est moi !"`, qui est la phrase que doit retourner la fonction *p.Salutation*. Nous remarquons la présence des guillemets qui indiquent que c'est une chaîne de caractères. Ces guillemets ont deux fonctions. D'abord, ils indiquent où commence la chaîne de caractères et où celle-ci finit. Ensuite, ils indiquent que l'on a bien affaire à une chaîne de caractères et pas à une variable (nous reviendrons là-dessus plus bas lorsque nous étudierons plus en détail ce qu'est une variable).

Utiliser un module

Dans notre programme, après l'écriture de la fonction, nous voyons apparaître la ligne : `return p`. Cette instruction, écrite en dehors de la fonction, indique que l'on retourne le contenu de la table *p* pour le rendre disponible en dehors du module. Nous

précisons que nous retournons le contenu de la table *p* et pas la table *p* elle-même, celle-ci n'étant disponible qu'à l'intérieur du module à cause de l'instruction `local p = {}`.

Comment allons-nous pouvoir récupérer le contenu de la table *p*, à savoir la fonction *p.Salutation* en dehors du module ? Pour cela, il nous suffit d'écrire dans une autre page qui n'est pas dans l'espace module, la commande : `{{#invoke:Exemple simple|Salutation}}`.

En effet, en tapant : `{{#invoke:Exemple simple|Salutation}}`, on obtient bien : « Coucou, c'est moi ! ».

Nous retiendrons la syntaxe de cette commande sous la forme provisoire : `{{#invoke:nom du module|nom de la fonction}}`. Nous allons voir dans les paragraphes suivants que l'on peut y rajouter des paramètres.

Paramétrer une fonction

Exemple avec un seul paramètre

Pour illustrer le passage d'un paramètre à une fonction écrite dans un module et ceci à partir d'une page extérieure au module, nous allons écrire une nouvelle fonction dans un nouveau module que l'on appellera, par exemple, Module:Autre exemple.

Nous nous proposons cette fois d'écrire un programme qui va traduire les jours de la semaine en anglais.

Le programme à l'intérieur du Module:Autre exemple sera rédigé ainsi :

```
local p = {}

function p.traduit(frame)
        if frame.args[1] == "Lundi" then return "Monday" end
        if frame.args[1] == "Mardi" then return "Tuesday" end
        if frame.args[1] == "Mercredi" then return "Wednesday" end
        if frame.args[1] == "Jeudi" then return "Thursday" end
        if frame.args[1] == "Vendredi" then return "Friday" end
        if frame.args[1] == "Samedi" then return "Saturday" end
        if frame.args[1] == "Dimanche" then return "Sunday" end
end

return p
```

Nous ne commenterons, bien sûr, que ce qui est nouveau par rapport au paragraphe précédent.

Dans la définition de la nouvelle fonction : `function p.traduit(frame)`, nous remarquons que, cette fois, nous avons entre parenthèses, le mot `frame`.

Par convention, on peut dire que *frame* est une table qui contient les paramètres que l'on a passés au module depuis l'extérieur grâce à la commande vue au paragraphe précédent qui, dans ce paragraphe, aura la syntaxe plus complète suivante : `{{#invoke:''nom du module''|''nom de la fonction''|args[1]|args[2]|args[3]|etc.}}`

. args[1], args[2], args[3], args[4] étant des informations que l'on souhaite transmettre à la fonction que l'on a choisie dans le module. Ces informations, appelées paramètres, se retrouveront dans le module dans la table `frame`. Pour y accéder, il suffira donc d'écrire dans le programme respectivement : frame.args[1], frame.args[2], frame.args[3], etc. Par exemple, supposons que nous voulions utiliser notre programme pour traduire « jeudi » en anglais. Nous écrirons simplement `{{#invoke:Autre exemple|traduit|Jeudi}}` et nous obtenons : « Thursday ».

Dans le corps de la fonction, nous avons cette fois un bloc d'instructions comprenant 7 instructions similaires et nous comprenons aisément que chacune de ces instructions correspond à un jour de la semaine. Prenons la première : `if frame.args[1] == "Lundi" then return "Monday" end`. Nous avons, dans cette instruction deux nouveaux mots-clés : `if` et `then`. `if` se traduit par « si » en français et `then` se traduit par « alors ». Le syntaxe générale de cette instruction est `if` *`condition`* `then` *`instructions`* `end` (ou, en français : « si *condition* alors *instructions* fin »). Autrement dit, de façon plus explicite : « si une certaine condition est réalisée alors le bloc d'instructions sera exécuté ».

Dans notre instruction : `if frame.args[1] == "Lundi" then return "Monday" end`, nous voyons que la condition est `frame.args[1] == "Lundi"`. On se demande si le paramètre transmis à la fonction est la chaîne de caractère « Lundi ». En Lua, `==` signifie « égal » (si l'on met seulement `=`, le sens ne sera pas le même ; nous y reviendrons). Si la condition testée est remplie alors on exécutera le bloc d'instructions qui, dans notre exemple, se limite à `return "Monday"`. Nous voyons que la fonction *p.traduit* va retourner la chaîne de caractères « Monday » si le paramètre transmis est « Lundi » et c'est ce que l'on voulait. Le raisonnement est le même pour les six autres lignes qui suivent.

Exemple avec plusieurs paramètres

Nous allons maintenant essayer de perfectionner le programme vu au paragraphe précédent en lui faisant traduire les jours de la semaine dans une langue que l'on aura choisie. Nous nous limiterons à l'anglais et à l'espagnol. Le lecteur comprendra aisément comment introduire d'autres langues.

Le programme est le suivant :

```
local p = {}

function p.traduit(frame)
        if frame.args[2] == "Anglais" then
                if frame.args[1] == "Lundi" then return "Monday" end
                if frame.args[1] == "Mardi" then return "Tuesday" end
                if frame.args[1] == "Mercredi" then return
"Wednesday" end
                if frame.args[1] == "Jeudi" then return "Thursday"
end
                if frame.args[1] == "Vendredi" then return "Friday"
end
                if frame.args[1] == "Samedi" then return "Saturday"
end
```

```
                if frame.args[1] == "Dimanche" then return "Sunday"
end
        end
        if frame.args[2] == "Espagnol" then
                if frame.args[1] == "Lundi" then return "Lunes" end
                if frame.args[1] == "Mardi" then return "Martes" end
                if frame.args[1] == "Mercredi" then return
"Miércoles" end
                if frame.args[1] == "Jeudi" then return "Jueves" end
                if frame.args[1] == "Vendredi" then return "Viernes"
end
                if frame.args[1] == "Samedi" then return "Sábato" end
                if frame.args[1] == "Dimanche" then return "Domingo"
end
        end
end

return p
```

Nous avons mis ce programme dans le Module:Traduction multilingue. Pour l'utiliser, nous devons préciser deux paramètres. Par exemple, pour obtenir la traduction de dimanche en espagnol, nous devons écrire dans la page appelante : `{{#invoke:Traduction multilingue|traduit|Dimanche|Espagnol}}`, ce qui nous donne : « Domingo ». À l'intérieur du programme, les deux paramètres sont accessibles respectivement par `frame.args[1]` et `frame.args[2]`.

Le programme, en lui-même, ne présente pas de nouvelles instructions. Nous remarquerons toutefois la possibilité d'emboîter les structures `if condition theninstructions end`.

Par exemple, la première structure concernée s'écrit sous la forme :

```
if frame.args[2] == "Anglais" then
        -- Bloc d'instructions --
end
```

Le bloc d'instructions comprenant les 7 traductions en anglais sous forme de 7 structures `if condition then instruction end`.

Concaténer du texte

La concaténation est une opération qui consiste à mettre bout à bout plusieurs chaînes de caractères. L'opérateur de concaténation est `..` (deux points qui se suivent). Pour illustrer cela, nous allons écrire un Module:Faire part qui, à partir d'un ou deux noms entrés en paramètres, forme une phrase annonçant, soit une naissance, soit un mariage, soit un décès.

Le contenu du module est le suivant :

```
local p = {}

function p.naissance(frame)
        return "Nous avons la joie de vous annoncer la naissance de "
.. frame.args[1] .. "."
end

function p.mariage(frame)
        return "Nous sommes heureux de vous annoncer le mariage de "
.. frame.args[1] .. " et " .. frame.args[2] .. "."
end

function p.deces(frame)
        return "Nous sommes au regret de vous annoncer le décès de "
.. frame.args[1] .. "."
end

return p
```

Nous remarquons que ce module contient, cette fois, trois fonctions. Donnons des exemples d'utilisation de ce module.

Si dans la page appelante, nous écrivons `{{#invoke:Faire part|mariage|Louis|Christine}}`, nous obtenons :

« Nous sommes heureux de vous annoncer le mariage de Louis et Christine. »

Si dans la page appelante, nous écrivons `{{#invoke:Faire part|naissance|Noémie}}`, nous obtenons :

« Nous avons la joie de vous annoncer la naissance de Noémie. »

Si dans la page appelante, nous écrivons `{{#invoke:Faire part|deces|monsieur Bertelot}}`, nous obtenons :

« Nous sommes au regret de vous annoncer le décès de monsieur Bertelot. »

Par exemple, pour le faire part de mariage, l'instruction `return` retourne une chaîne de caractères obtenue en concaténant avec `..` :

- `"Nous sommes heureux de vous annoncer le mariage de "` (on remarque la présence d'une espace en fin de chaîne pour éviter d'avoir le premier nom collé au mot « de ») ;
- `frame.args[1]` (contenant dans notre exemple la chaîne de caractères formant le prénom *Louis*) ;
- `" et "` (on remarque aussi les espaces en début et en fin de chaîne pour éviter d'avoir *LouisetChristine*) ;
- `frame.args[2]` (contenant dans notre exemple la chaîne de caractères formant le prénom *Christine*) ;

- `"."` (chaîne de caractères se limitant au point final de la phrase).

  Dans ce module, nous remarquons que nous avons écrit la fonction *p.deces* sans accents. En effet, le Lua n'accepte pas les accents dans les noms de fonctions et plus généralement dans les noms de variables.

Variables, types et affectation

Nous allons maintenant aborder une notion importante qui est la notion de variable. D'une façon imagée, on pourrait dire qu'une variable est une boîte dans laquelle on va pouvoir mettre un objet particulier qui sera, soit une chaîne de caractères, soit un nombre, soit une table, soit une fonction, soit un booléen, etc. En Lua, une même variable peut, dans un même programme, contenir, par exemple, une chaîne de caractères dans une partie du programme et dans une autre partie, elle contiendra un nombre. Cette faculté de pouvoir recevoir des objets de nature différentes se traduit en disant que les variables, en Lua, sont dynamiques. Ce n'est pas le cas, dans d'autres langages comme le C ou le Pascal où les variables sont dites statiques (chacune étant spécialisée pour recevoir un seul type d'objet).

La plupart du temps, une variable se déclare grâce à l'instruction :

```
local tirelire
```

Le mot `local` signifie qu'elle n'est opérationnelle que là où on l'a déclarée. Si on la déclare en début de module, elle sera valable dans tout le module (y compris dans les fonctions du module). Si on la déclare au début d'une fonction, elle sera valable uniquement dans la fonction (y compris dans les structures de contrôle comme `if..then`). Si on la déclare au début d'une structure de contrôle, elle sera valable uniquement dans la structure de contrôle.

Il est possible, à la déclaration, de l'initialiser, c'est-à-dire d'y mettre quelque chose dedans. Dans ce cas, la variable adoptera le type de ce que l'on met dedans.

Si on ne l'initialise pas, la variable sera, par défaut, d'un type particulier que l'on appelle *nil* et contiendra `nil`, ce qui signifie « rien ». Elle restera de ce type jusqu'à ce qu'on y mette quelque chose.

Par exemple, pour l'instruction :

```
local tirelire = "Billet de dix euros"
```

la variable *tirelire* sera du type *chaîne de caractères* et sera censée être du type *chaîne de caractères* jusqu'à ce que l'on y mette quelque chose qui ne soit pas une chaîne de caractère.

Le signe `=` présent dans cette instruction ne veut pas dire « égal » mais « affectation ». À la variable *tirelire*, on affecte la chaîne de caractères : `"Billet de dix euros"`.

Si l'on écrit :

```
local tirelire = 10
```

la variable *tirelire* sera du type *nombre* et sera censée être du type *nombre* jusqu'à ce que l'on y mette quelque chose qui ne soit pas un nombre.

Même si une variable peut contenir à des moments différents, des objets de type différent, il faut malgré tout que l'on sache ce qu'elle est susceptible de contenir dans toutes les parties du programme. Dans certains cas, le programme peut ne pas fonctionner si la variable n'a pas le bon type au bon moment.

Prenons un exemple : écrivons un programme qui nous signale si un nombre n'a pas une valeur trop élevée. Dans le Module:Balance, écrivons le programme suivant :

```
local p = {}

function p.alerte1(frame)
        local poids = frame.args[1]
        local reponse = "Votre poids est acceptable"
        if poids > 54 then
                reponse = "Attention, vous commencez à grossir !"
        end
        return reponse
end

return p
```

Si l'on écrit, dans une autre page : `{{#invoke:Balance|alerte1|56}}`, on obtient : « **Erreur Lua dans Module:Balance à la ligne 6 : attempt to compare number with string.** »

Pour comprendre pourquoi le programme ne marche pas, nous allons revenir sur un exemple précédent. Reprenons, par exemple, le programme qui traduisait les jours de la semaine en anglais. Pour traduire jeudi en anglais, nous avions écrit : `{{#invoke:Autre exemple|traduit|Jeudi}}`. Jeudi est une chaîne de caractères et pourtant, nous n'avons pas écrit : `{{#invoke:Autre exemple|traduit|"Jeudi"}}`. La commande `invoke` a interprété jeudi comme étant une chaîne de caractères même si nous n'avons pas mis les guillemets. Pour simplifier l'écriture, la commande `invoke` interprète systématiquement ses arguments comme étant des chaînes de caractères. Par conséquent, lorsqu'on écrit : `{{#invoke:Balance|alerte1|56}}`, le programme reçoit comme argument la chaîne de caractères `"56"` et non pas le nombre `56`. Par conséquent, l'instruction de notre programme :

```
local poids = frame.args[1]
```

affecte à la variable *poids* la chaîne de caractères `"56"`, et l'instruction :

```
        if poids > 54 then
                reponse = "Attention, vous commencez à grossir !"
        end
```

compare donc une chaîne de caractères au nombre 54, ce qui n'a pas de sens.

Pour remédier à cet inconvénient, il faut, avant de faire la comparaison avec 54, transformer le contenu de la variable *poids* en nombre. Comment faire ? Heureusement, dans le Lua, nous disposons d'un certain nombre de petites fonctions préprogrammées que nous étudierons en détail dans les chapitres ultérieurs. Pour les besoins de la circonstance, nous allons utiliser l'une d'elles ici. Cette fonction est la fonction *tonumber* qui convertit une chaîne de caractères en nombre dans la mesure où cela est possible. Par exemple, elle convertira la chaîne de caractères `"12"` en nombre 12.

Nous pouvons mettre en œuvre cette fonction en écrivant dans notre programme :

```
        local poids = tonumber(frame.args[1])
```

Et nous écrirons donc, dans le Module:Balance, une nouvelle fonction *p.alerte2*, qui est la version corrigée de la fonction *p.alerte1*, ainsi :

```
local p = {}

function p.alerte2(frame)
        local poids = tonumber(frame.args[1])
        local reponse = "Votre poids est acceptable"
        if poids > 54 then
                reponse = "Attention, vous commencez à grossir !"
        end
        return reponse
end

return p
```

Maintenant, si dans une autre page, on écrit : `{{#invoke:Balance|alerte2|56}}`, on obtient : « Attention, vous commencez à grossir ! »

On constate que cette fois, ça marche !! (du moins le programme, pas le régime !)

Nous allons maintenant étudier une particularité du Lua. Dans une affectation, le Lua a la capacité de caractériser le type des variables automatiquement en fonction de ce qui est affecté. Si, par exemple, dans ce qui est affecté, il y a des signes opératoires, la variable sera de type *nombre*. S'il y a des concaténations, la variable sera du type *chaîne de caractères*.

Premier exemple :

```
compte = a + b
```

La variable *compte* sera automatiquement considérée comme étant du type *nombre* à cause de la présence de l'opérateur `+`, sans même s'occuper de ce que contiennent les variables *a* et *b*. Et ceci, même si *a* et *b* contiennent des chaînes de caractères.
Si *a* contient la chaîne de caractères `"1"` et si *b* contient la chaîne de caractères `"2"`, alors la variable compte contiendra, malgré tout, le nombre 3.

À noter toutefois que si *a* ou *b* ne contient pas quelque chose qui puisse être converti en nombre, alors cela génère une erreur et le programme s'arrête. Nous étudierons la gestion des erreurs dans un chapitre ultérieur.

Deuxième exemple :

```
panneau = indication..route
```

La variable *panneau* sera automatiquement considérée comme étant du type *chaîne de caractères* à cause de la présence de l'opérateur de concaténation `..`, sans même s'occuper de ce que contiennent les variables *indication* et *route*. Et ceci, même si *indication* et *route* contiennent des nombres. Si *indication* contient le nombre 1 et si *route* contient le nombre 2, alors la variable panneau contiendra, malgré tout, la chaîne de caractères `"12"`.

Ceci étant dit, nous pouvons revenir à notre programme qui ne marchait pas. Nous avons dit qu'il faudrait que la variable *poids* contienne un nombre et pas une chaîne de caractères. Pour la transformer, compte tenu de ce que nous venons de dire, certains petits malins auraient pu écrire le programme ainsi :

```
local p = {}

function p.alerte1(frame)
        local poids = frame.args[1] + 0
        local reponse = "Votre poids est acceptable"
        if poids > 54 then
                reponse = "Attention, vous commencez à grossir !"
        end
        return reponse
end

return p
```

Ça marche ! Mais :

```
        local poids = frame.args[1] + 0
```

ne fait pas professionnel et ressemble à du bricolage. Nous abandonnerons donc cette idée au profil de :

```
        local poids = tonumber(frame.args[1])
```

La structure `if..then..else`

Nous venons d'étudier la structure `if condition then instructions end`. Il est possible de compléter cette structure en y rajoutant un petit élément qui est `else` et qui signifie « sinon » en français. La
structure **if *condition* then *instruction1* else *instruction2* end** signifie en français : « si

une *condition* est remplie exécuter *instruction1* sinon exécuter *instruction2* ». À titre d'exemple, reprenons notre Module:Balance que nous avons commencé à remplir.

Dans ce module, grâce à l'ajout de `else`, nous pouvons écrire une fonction *p.alerte3*, qui est une autre façon d'écrire la fonction *p.alerte2*, ainsi :

```
local p = {}

function p.alerte3(frame)
        local poids = tonumber(frame.args[1])
        local reponse
        if poids < 55 then
                reponse = "Votre poids est acceptable"
        else
                reponse = "Attention, vous commencez à grossir !"
        end
        return reponse
end

return p
```

Dans une autre page, si nous écrivons `{{#invoke:Balance|alerte3|57}}`, nous obtenons : « Attention, vous commencez à grossir ! »

Traiter plusieurs variables en une seule instruction

Il nous reste à voir une particularité intéressante du Lua qu'on ne trouve pas dans d'autres langages : c'est l'affectation simultanée de plusieurs variables. En effet, plutôt que d'écrire :

```
a = 2
b = 7
```

c'est-à-dire mettre la valeur 2 dans *a*, puis la valeur 7 dans *b*, on peut écrire :

```
a, b = 2, 7
```

et tout se passera comme si l'on avait simultanément mis 2 dans *a* et 7 dans *b*. Vous allez me dire : *bof ! quel intérêt ?* 😐

Il y a plusieurs intérêts à cela. Nous verrons certains de ces intérêts dans les chapitres suivants quand nous étudierons les fonctions qui retournent plusieurs valeurs.

Pour le moment, nous pouvons donner un exemple simple : supposons que nous voulions échanger le contenu de deux variables. En Lua, nous écrirons simplement :

```
a, b = b, a
```

Dans un autre langage qui n'a pas l'affectation simultanée, on aurait été tenté d'écrire :

```
a = b
b = a
```

Mais ça ne marche pas car le contenu de *a* est remplacé par le contenu de *b* dans la première affectation. Le contenu initial de *a* est donc perdu et ne pourra donc pas aller dans *b* à la deuxième affectation.

On peut aussi déclarer simultanément deux variables en les initialisant :

```
local a, b = 2, 7
```

ou déclarer simultanément deux variables sans les initialiser :

```
local a, b
```

Mise au point d'un module

Dans le chapitre précédent, nous avons vu suffisamment de notions pour commencer à faire des petits modules faciles à utiliser. Bien souvent les modules ne seront pas nécessairement petits et faciles à utiliser. Leurs mises au point risque d'être délicates à faire. Par conséquent, avant d'aller plus loin dans l'étude du Lua avec Scribunto, nous allons consacrer ce chapitre à l'étude des moyens dont nous disposons pour faciliter la mise au point des modules.

Augmenter la lisibilité du programme se trouvant dans le module

Une première façon de rendre un programme plus facile à mettre au point est de l'écrire de façon à ce qu'il soit facile à relire par nous-même ou par quelqu'un d'autre. Par nous-même, car même si l'on a l'impression, sur le moment, de bien savoir ce qu'il contient, il se peut que l'on soit amené à y revenir après plusieurs mois et là, on risque d'avoir du mal à retrouver comment il fonctionne. Par les autres, car les modules écrits sur un des projets Wikimédia peuvent être améliorés par d'autres utilisateurs.

L'amélioration de la lisibilité d'un programme se base sur trois techniques:

L'indentation

C'est le fait de décaler vers la droite un bloc d'instructions pour le rendre plus lisible. On décalera vers la droite les instructions se trouvant à l'intérieur d'une fonction, d'une structure `if condition then instruction end`, et les autres structures de contrôle que nous verrons plus tard.

Par exemple, dans le Module:Traduction multilingue, si nous avions écrit :

```
local p = {}

function p.traduit(frame)
if frame.args[2] == "Anglais" then
if frame.args[1] == "Lundi" then return "Monday" end
```

```
if frame.args[1] == "Mardi" then return "Tuesday" end
if frame.args[1] == "Mercredi" then return "Wednesday" end
if frame.args[1] == "Jeudi" then return "Thursday" end
if frame.args[1] == "Vendredi" then return "Friday" end
if frame.args[1] == "Samedi" then return "Saturday" end
if frame.args[1] == "Dimanche" then return "Sunday" end
end
if frame.args[2] == "Espagnol" then
if frame.args[1] == "Lundi" then return "Lunes" end
if frame.args[1] == "Mardi" then return "Martes" end
if frame.args[1] == "Mercredi" then return "Miércoles" end
if frame.args[1] == "Jeudi" then return "Jueves" end
if frame.args[1] == "Vendredi" then return "Viernes" end
if frame.args[1] == "Samedi" then return "Sàbato" end
if frame.args[1] == "Dimanche" then return "Domingo" end
end
end

return p
```

Le programme aurait, tout de même, bien fonctionné mais aurait été moins lisible.

Les noms de variable explicites

Le Lua, ainsi que la plupart des langages de programmation, permettent d'écrire les variables en utilisant plusieurs caractères. On donnera donc aux variables un nom qui exprimera ce qu'elles contiennent. Par exemple, une variable destinée à mémoriser un salaire s'appellera *salaire*.

Les commentaires dans le programme

Nous n'en n'avons pas parlé pour raison pédagogique, au chapitre précédent, car les programmes étaient petits et des commentaires les auraient alourdis inutilement. Mais il est possible et même fortement conseillé, quand les programmes deviennent plus longs, de rajouter des commentaires à l'intérieur même des programmes pour expliquer ce que chaque partie du programme fait.

Un petit commentaire tenant sur une ligne se fera en commençant par mettre un double tirets `--`. On peut même mettre un commentaire après une instruction.

Par exemple, dans le Module:Faire part, on aurait pu rajouter des commentaires sur ce que réalisent les fonctions :

```
local p = {}

function p.naissance(frame)    -- Faire part de naissance
        return "Nous avons la joie de vous annoncer la naissance de "
.. frame.args[1] .. "."
end
```

```
function p.mariage(frame)      -- Faire part de mariage
        return "Nous sommes heureux de vous annoncer le mariage de "
.. frame.args[1] .. " et " .. frame.args[2] .. "."
end

function p.deces(frame)                -- Faire part de décès
        return "Nous sommes au regret de vous annoncer le décès de "
.. frame.args[1] .. "."
end

return p
```

On peut aussi imaginer avoir besoin de plusieurs lignes pour faire un commentaire. Pour cela, on commencera le commentaire par `--[[` et on le terminera par `]]`.

Par exemple pour Module:Exemple simple on aurait pu écrire :

```
local p = {}

function p.Salutation() -- [[mon commentaire ……
 de plusieurs lignes]]
    return "Coucou, c'est moi !"
end

return p
```

L'éditeur Scribunto

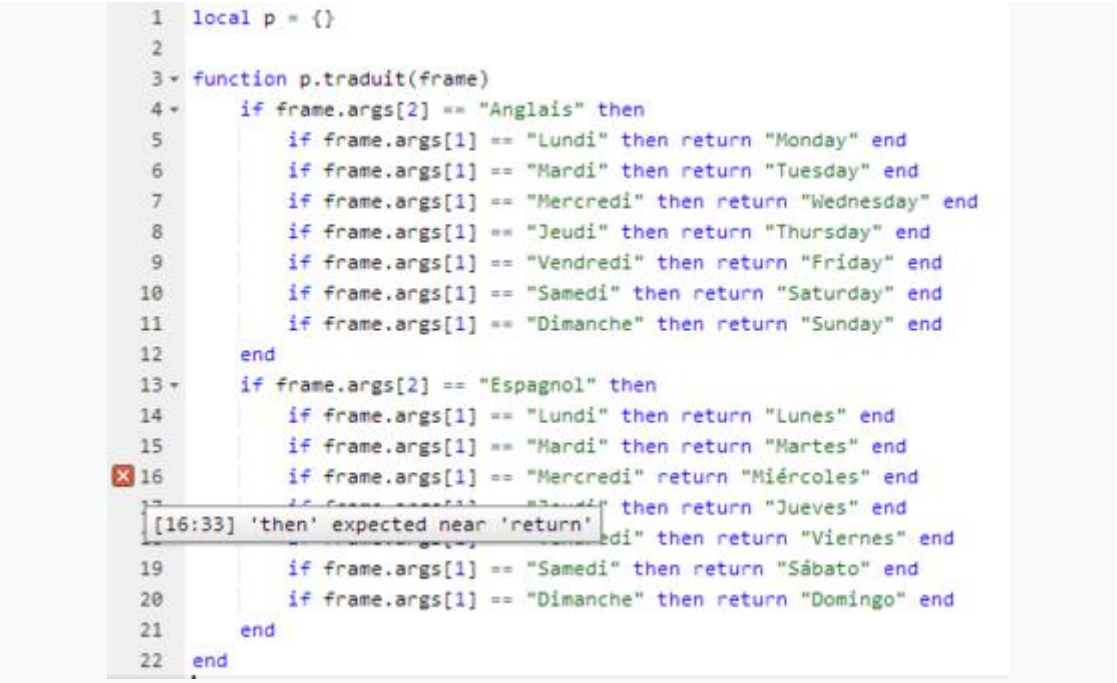


Figure 1

Étudions de plus près l'éditeur dont nous disposons dans l'extension Scribunto. Nous remarquons, tout d'abord, que chaque ligne est numérotée. Lorsque nous écrivons une ligne, celle-ci se détache sur un fond légèrement grisé. Nous le voyons à la figure 2, ligne 16 où se trouve le curseur. L'éditeur dispose d'une première correction pour détecter les erreurs grossières dans la syntaxe des instructions. Si nous n'écrivons pas correctement une instruction, le numéro en début de ligne se retrouve précédé d'une croix dans un carré rouge. Nous le voyons, par exemple, dans la figure 1, ligne 16. Si nous regardons, de plus près la ligne 16, nous verrons que nous avons oublié de mettre le mot-clé `then` qui doit obligatoirement se trouver dans une structure `if` *`condition`* `then` *`instructions`* `end`. Mieux que cela, si un carré rouge avec croix apparaît devant un numéro de ligne, nous pouvons avoir une indication sur le type d'erreur en promenant le curseur dessus (nous voulons dire par là, que nous pointons le carré rouge avec le curseur sans toutefois cliquer dessus). Dans notre exemple, figure 1, nous avons le message « *[16:33] 'then' expected near 'return'* », ce qui signifie que ligne 16, position 33, `then` est attendu avant `return`.

Nous remarquons aussi, en début de certaines lignes, un symbole ▼ juste après le numéro de ligne. Ce caractère permet de masquer un bloc d'instructions. Par exemple, si nous cliquons sur le ▼ de la ligne 3 ou est déclarée la fonction *p.traduit*, nous voyons disparaître toutes les instructions se trouvant entre cette déclaration et le `end` indiquant la fin de l'écriture du contenu de la fonction. Si nous cliquons sur le ▼ de la ligne 4, nous verrons disparaître toutes les instructions du bloc `if`. L'utilité de cette fonctionnalité est double. On peut ainsi masquer certaines parties du programme sur lesquelles on n'est pas en train de travailler. On peut aussi, dans un programme, où il y a beaucoup de structures emboîtées et par conséquent beaucoup de `end`, s'assurer que l'on ne s'est pas « emmêlé les pinceaux » avec les `end`.

Une autre particularité intéressante de l'éditeur est que lorsque l'on clique juste après une parenthèse, un crochet ou une accolade ouvrante ou fermante, nous voyons un léger encadrement sur la parenthèse, le crochet ou l'accolade fermante ou ouvrante correspondante. Cela peut être utile dans les expressions ayant beaucoup de parenthèses, crochets et accolades pour éviter les erreurs.

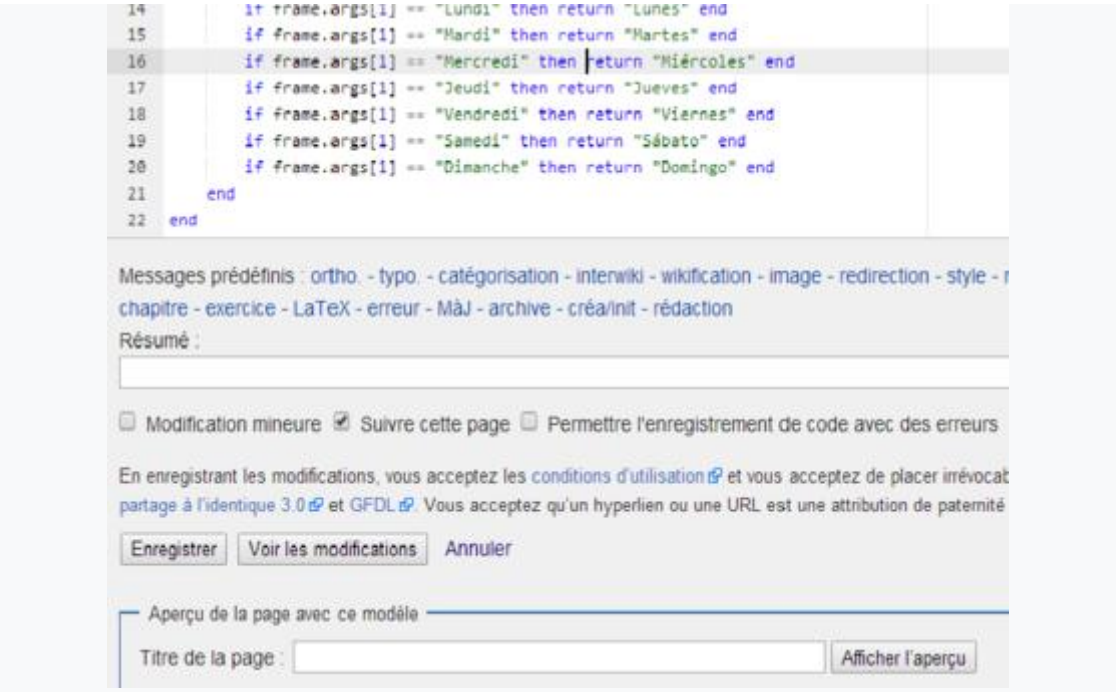


Figure 2

Intéressons-nous maintenant à ce qui apparaît sous le cadre de visualisation. Nous n'allons pas nous intéresser à ce qui est juste en dessous du cadre de visualisation, car il n'y a là rien de bien nouveau. Nous allons nous intéresser à ce qui se trouve plus bas dans le cadre noté « Aperçu de la page avec ce modèle » (voir figure 2). En effet, nous allons pouvoir, avec cet aperçu, voir ce que va donner le module avant même de devoir l'enregistrer. Il est possible, grâce à ce cadre, de faire l'écriture et la mise au point complète du module sans faire une seule édition.

Pour cela, enregistrez tout d'abord dans une page — par exemple Bac à sable — la commande `{{#invoke:''nom du module''|''nom de la fonction''|"arguments"}}` concernant votre module. Dans le cadre « Aperçu de la page avec ce modèle » (voir ci-contre), écrivez le titre de la page où le module est invoqué (ici la page « Bac à sable »). Enfin, cliquez sur « Afficher l'aperçu ». L'aperçu de la page où vous avez invoqué votre module (ici l'aperçu de Bac à sable) s'affiche en haut de la page. Si ce n'est pas correct, vous pouvez corriger le module et cliquer à nouveau sur « Afficher l'aperçu » autant de fois que vous voulez, jusqu'à ce que le module soit au point. Une fois le module au point, vous pouvez cliquer sur le bouton « Enregistrer » et le module sera édité.

Le traitement des erreurs de script

Après avoir corrigé toutes les erreurs indiquées par l'éditeur, nous ne sommes peut-être pas au bout de nos peines. En essayant le programme, nous voyons apparaître le charmant message : « **Erreur de script : vous devez spécifier une fonction à appeler.** »

Cela signifie que nous avons malgré tout fait une erreur que l'éditeur n'a pas décelée mais qui rend l'exécution du programme impossible.

Avec l'expérience, nous pouvons éviter les principales erreurs de script. En attendant d'acquérir cette expérience, nous nous contenterons d'énumérer les principales situations qui provoquent une erreur de script.

Voici une liste à compléter éventuellement :

- Utilisation d'une variable en croyant qu'elle contient un certain type de données, alors qu'elle en contient un autre. Exemple : comparaison d'une variable contenant une chaîne de caractères avec un nombre.
- Utilisation d'une instruction en dehors du contexte où elle devrait être normalement utilisée. Par exemple, emploi de `frame.args[1]` en dehors de la fonction qui devrait normalement recueillir l'argument.
- La fonction appelée n'existe pas. Vous avez, peut-être, fait une faute d'orthographe en écrivant son nom ou simplement oublié `p.` en début de nom.
- Opération avec une variable, qui est bien du bon type, mais que l'on n'a pas initialisée et qui est donc vide au moment où on l'utilise.
- Peut éventuellement être produit par l'oubli de l'instruction `return` dans une fonction (selon comment est utilisée la fonction).

Lorsqu'une erreur de script se produit, vous pouvez avoir une première indication sur la provenance de cette erreur en cliquant sur le message : « **Erreur de script : vous devez spécifier une fonction à appeler.** » (bien qu'il soit rouge et non bleu). L'indication vous permettra, peut-être, de corriger rapidement l'erreur.

Si, malgré tout, l'erreur de script continue à apparaître et que vous ne voyez pas d'où elle provient, vous pouvez utiliser l'astuce suivante :

Vous mettez `--` progressivement au début des lignes, en commençant par celles qui paraissent les plus douteuses, jusqu'à ce que l'erreur de script disparaissent. Ces lignes commençant par `--` seront alors interprétées comme étant des commentaires et ne pourront plus provoquer d'erreur de script. Vous pourrez ainsi repérer la ligne qui provoque l'erreur de script.

La recherche d'une erreur dans le programme

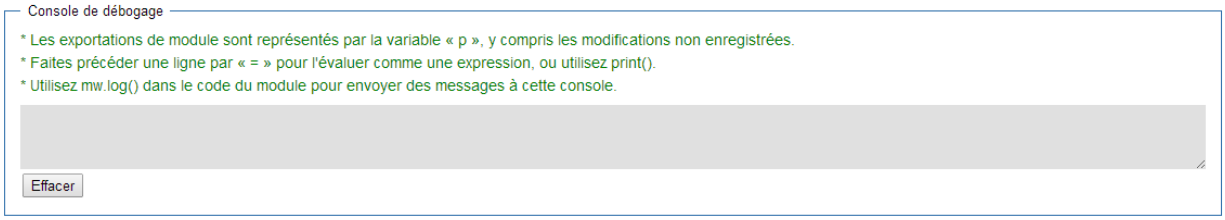
Vous avez écrit un module. L'éditeur n'a pas détecté d'erreur et lorsque vous lancez l'exécution, vous n'avez pas le message : Erreur de script. Le problème, c'est que ce que vous fournit le programme n'est pas conforme à votre attente. Vous avez commis une erreur en écrivant le programme ! Vous essayez donc, dans un premier temps, de relire ce que vous avez écrit pour essayer de comprendre pourquoi cela ne marche pas. Au bout d'un certain temps de réflexion, vous vous rendez à l'évidence, vous n'arrivez pas à comprendre pourquoi cela ne marche pas. Nous allons donc étudier, dans ce paragraphe, des moyens dont nous disposons pour faciliter la recherche de l'erreur.

Introduction d'une variable *espion*

Nous avons d'abord une technique simple qui consiste à introduire dans le programme une variable supplémentaire, que l'on appellera *rapport* par exemple, dans laquelle vous allez, en certains points du programme, concaténer le contenu d'autres variables. À la fin de la fonction, au lieu de retourner la variable prévue, on retournera la variable *rapport* qui nous fournira ainsi une information sur le contenu des variables en certains points du programme et nous permettra de localiser plus précisément dans quelle partie se trouve l'erreur. Une fois que nous avons localisé de façon plus précise la partie du programme défaillante, nous pouvons recommencer en concaténant, dans notre variable *rapport*, plus d'informations sur la partie fautive. Et ainsi de suite jusqu'à repérer l'instruction qui est la cause de nos soucis.

Console de débogage

Lorsque nous sommes en mode modification dans un module, nous avons vu que nous avions un certain nombre de possibilités. Si nous continuons à descendre dans la page, tout en bas, nous découvrons un encadré noté *Console de débogage* représenté ci-dessous :



Nous allons étudier comment cela fonctionne.

Nous commencerons avec le premier exemple dans le premier chapitre, c'est-à-dire la fonction *p.Salutation* dans le Module:Exemple simple.

Après s'être mis en modification et être descendu jusqu'à la console de débogage, taper à l'intérieur de celle-ci :

```
=p.Salutation()
```

puis appuyer sur la touche « Entrée ». Nous voyons alors que ce que l'on a écrit remonte au-dessus de la zone grisée. Puis après un léger temps d'attente apparaît, toujours au-dessus de la zone grisée : « Coucou, c'est moi ! »

Nous avons donc pu tester notre programme. À ce niveau, si quelque chose s'était mal passé, nous aurions eu un message d'erreur nous indiquant la nature de l'erreur et la ligne où l'erreur s'est produite.

Si nous avions tapé :

```
=p.Salutation
```

sans les accolades, nous aurions eu comme réponse : `function`. Nous pouvons avoir ainsi la nature des variables se trouvant dans le programme.

On aurait pu aussi taper :

```
print(p.Salutation())
```

ce qui est équivalent à `=p.Salutation()`.

On peut même se servir de la console de débogage comme calculatrice. En effet, si l'on rentre :

```
=2+3
```

Elle nous répond `5`.

Faisons maintenant une petite expérience et rajoutons la ligne : `mw.log("Il fait beau !")` dans notre programme ainsi :

```
local p = {}

function p.Salutation()
        mw.log("Il fait beau !")
        return "Coucou, c'est moi !"
end

return p
```

Dans la console de débogage tapons à nouveau :

```
=p.Salutation()
```

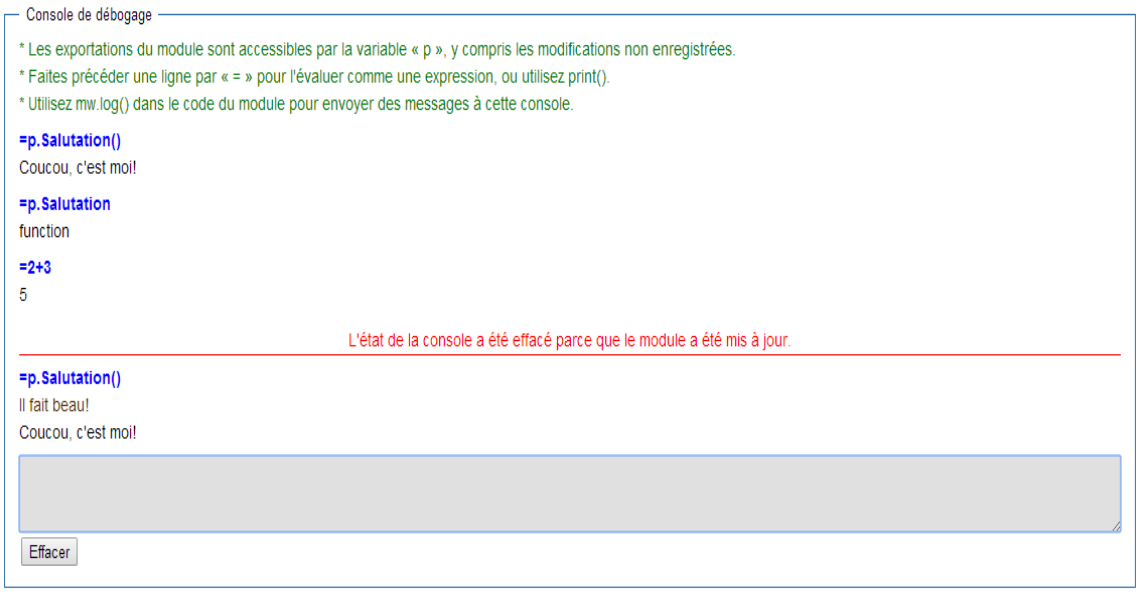
Après nous avoir prévenu que nous avons modifié le programme nous obtenons :

Il fait beau !
Coucou, c'est moi !

`mw.log` est une commande qui nous permet de transmettre des messages à la console de débogage.

L'intérêt de la fonction `mw.log` sur l'instruction `return` est que la fonction `mw.log` ne nous fait pas sortir du programme comme `return` lorsqu'elle est utilisée. On va donc pouvoir utiliser la fonction `mw.log` en plusieurs points du programme pour ramener plusieurs informations visibles sur la console de débogage. On peut ainsi construire tout un rapport d'exécution du programme qui apparaîtra sur la console de débogage et nous permettra ainsi de mettre au point le programme.

Ci-dessous, nous représentons la console de débogage après avoir tapé toutes les opérations décrites ci-dessus :



Le programme précédent était simple à étudier car c'était un programme sans paramètre.

Compliquons un peu les choses en étudiant maintenant la fonction `p.traduit` se trouvant dans le Module:Autre exemple.

Cette fonction, pour fonctionner, doit recevoir en argument un jour de la semaine. Pour parvenir à transmettre cet argument, nous devons taper dans la console de débogage :

```
frame = mw.getCurrentFrame()
```

puis « Entrée ». Le message tapé remonte au dessus de la zone grisée. Nous tapons ensuite :

```
newFrame = frame:newChild{ args = { 'Jeudi' }}
```

puis « Entrée ». Le second message tapé remonte au dessus de la zone grisée. Nous tapons alors :

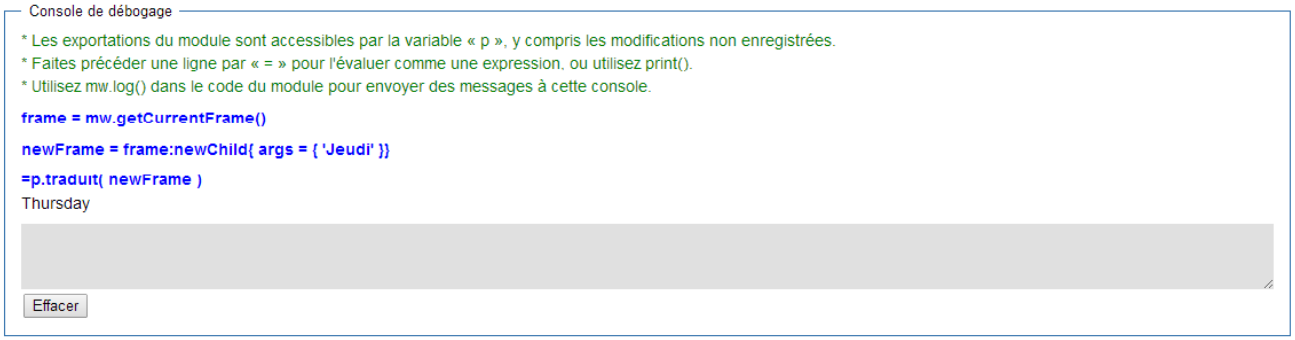
```
=p.traduit( newFrame )
```

Puis « Entrée ». Le troisième message tapé remonte au dessus de la zone grisée mais, cette fois, apparaît en plus :

Thursday

Qui est bien la traduction de jeudi en anglais.

Ci-dessous, nous représentons la console de débogage après avoir tapé toutes les opérations décrites ci-dessus :



Attention, la série des deux commandes :

```
frame = mw.getCurrentFrame()
newFrame = frame:newChild{ args = { 'Jeudi' }}
```

doit être retapée si vous modifiez le programme pour faire des essais (introduction d'une fonction `mw.log` par exemple).

Compliquons encore les choses en essayant de faire une simulation de débogage de la fonction `p.alerte2` se trouvant dans le Module:Balance.

Supposons que le programme ne marche pas (c'est pas vrai ! mais on fait semblant). Pour essayer de comprendre pourquoi le programme ne marche pas, nous allons visualiser sur la console de débogage le contenu de toutes les variables se trouvant dans le programme (en fait, ici, il n'y en a que deux).

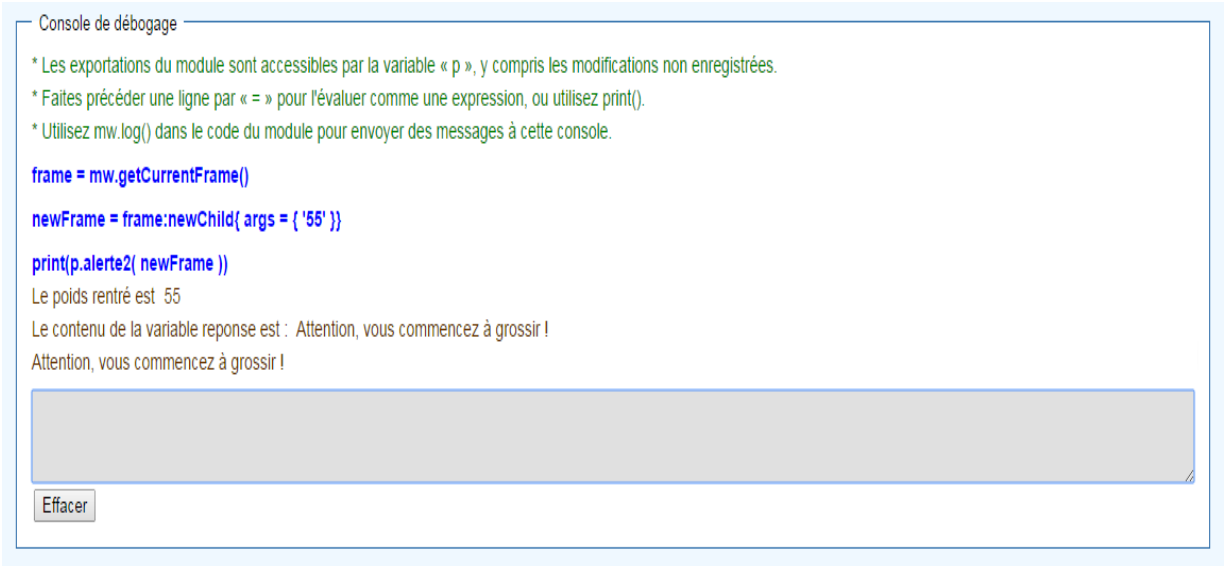
Nous utiliserons donc la fonction `mw.log` à deux endroits différents pour visualiser les contenus des variables *poids* et *reponse*. Le programme sera ainsi complété :

```
local p = {}

function p.alerte2(frame)
        local poids = tonumber(frame.args[1])
        mw.log("Le poids rentré est ", poids)
        local reponse = "Votre poids est acceptable"
        if poids > 54 then
                reponse = "Attention, vous commencez à grossir !"
        end
        mw.log("Le contenu de la variable reponse est : ", reponse)
        return reponse
end

return p
```

Dans la console de débogage, nous taperons successivement les trois commandes :

```
frame = mw.getCurrentFrame()
newFrame = frame:newChild{ args = { '55' }}
print(p.alerte2( newFrame ))
```

Après ces opérations, la console de débogage se présente ainsi :



où nous voyons clairement apparaître le contenu des variables *poids* et *reponse*, ce qui nous permettra éventuellement de mieux comprendre d'où provient l'erreur (s'il y en avait une).