Partie III - Les fonctions

par Yves Bailly

Date de publication : 01/07/2005

Dernière mise à jour : 20/11/2005

Après avoir appronfondi la notion de variables et ce qu'il était possible de faire avec, nous avons entr'aperçu le mois dernier une notion importante en programmation structurée : la fonction. Je vous propose aujourd'hui d'étudier ce concept fondamental.

I - Avant propos

Au cours de cet article, nous verrons plusieurs versions d'un même programme. Je vous recommande évidemment de taper les programmes vous-même, afin de vous #faire la main#. Pour les paresseux, vous trouverez ces programmes dans le répertoire qui accompagne cet article.

II - Le besoin des fonctions

Jusqu'à présent, nous avons essentiellement exécuté des instructions à la suite les unes des autres. Mais imaginez une suite d'instructions qui revient sans cesse dans un programme : la retaper à chaque fois peut rapidement devenir fastidieux. Prenons un exemple : vous voulez afficher une suite de nombres, un par ligne, chacun étant séparé par une ligne d'*underscores* (le nom anglais pour le caractère de soulignement "_") et une ligne de tirets. Essayez d'écrire vous-même le programme, par exemple pour afficher les nombres 10, 20, 30 et 40. Le programme pourrait ressembler à ceci (**prog_1.py**) :

```
 1 #!/usr/bin/env python
 2 print 10
 3 print "_"*10
 4 print "-"*10
 5 print 20
 6 print "_"*10
 7 print "-"*10
 8 print 30
 9 print "_"*10
10 print "-"*10
11 print 40
```

Essayez d'imaginer ce que fait ce programme. Puis, saisissez-le dans votre éditeur de texte favori (attention, sans les numéros de ligne !), et exécutez-le, comme nous l'avons appris dans le premier article de cette série. Vous voyez qu'une paire de lignes revient trois fois. Ce ne sont que deux lignes, et que trois fois, mais imaginez si cela représentait cinquante lignes devant être répétées vingt fois !

Répétons-le, les informaticiens sont paresseux (mais pas feignants, attention), donc une solution a été cherchée pour éviter de répéter ainsi une séquence d'instructions. Dans les « temps anciens », on parlait de *sous-programmes*, c'est-à-dire qu'une suite d'instructions étaient regroupées dans une sorte de mini-programme, lequel était exécuté à la demande par ce qui devenait alors le *programme principal*. Ceux d'entre vous qui ont pratiqué des langages comme le Basic se rappellent peut-être des instructions comme **GOTO** ou **GOSUB**, qui étaient les prémisses de la programmation structurée.

Aujourd'hui, on parle plutôt de *fonctions*, parfois de *procédures*. Une fonction est donc une suite d'instructions qui peut être exécutée à la demande par un programme : tout se passe alors comme si on avait inséré les instructions de la fonction à l'endroit où on l'appelle.

III - Définir une fonction

Découvrons sans attendre notre programme modifié, qui définit une fonction (**prog_2.py**) :

```
 1 #!/usr/bin/env python
 2 def Separateur():
 3    print "_"*10
 4    print "-"*10
 5
 6 print 10
 7 Separateur()
 8 print 20
 9 Separateur()
10 print 30
11 Separateur()
12 print 40
```

La fonction est définie ligne 2 et son nom est **Separateur()**. Par habitude, je fais commencer les noms des fonctions par une majuscule, mais il s'agit d'une pure convention d'écriture. Cette ligne vous montre comment définir une fonction :

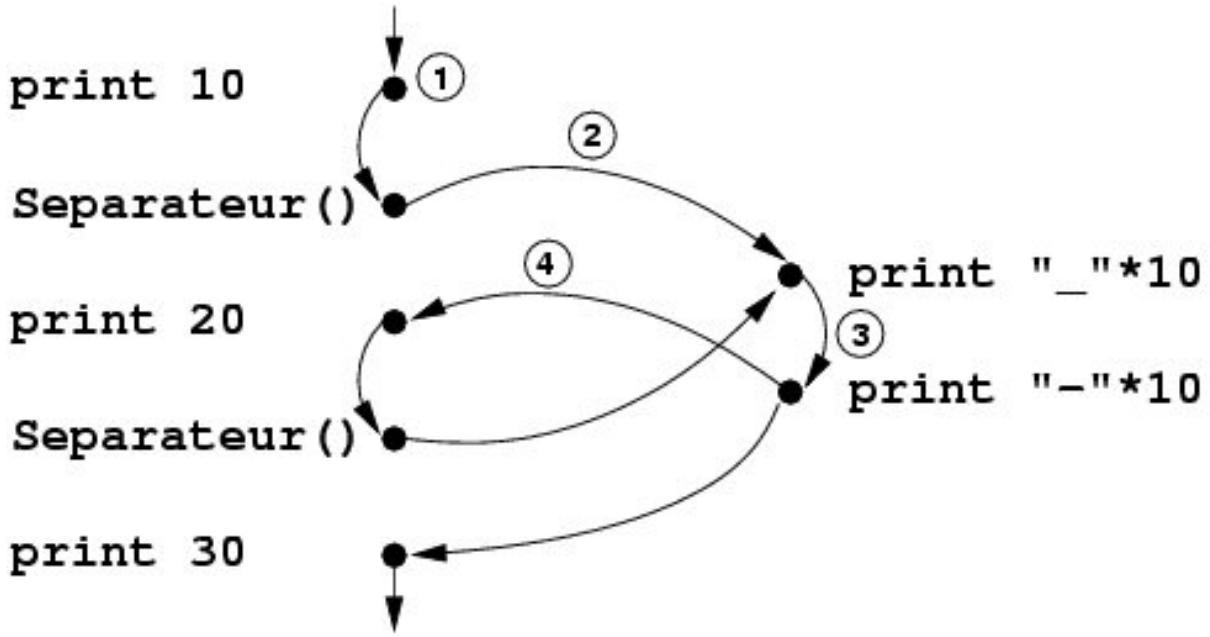
- d'abord, inscrire le mot-clef **def** : c'est par lui que l'on informe Python que ce qui suit est la définition d'une fonction ;
- ensuite, donner le nom de la fonction, ici **Separateur** ;
- puis une paire de parenthèses ; elles sont nécessaires, leur rôle exact apparaîtra dans la suite de cet article ;
- enfin, le caractère deux-points, qui indique à Python que l'on s'apprête à donner les instructions contenues dans la fonction.

Les deux lignes suivantes, 3 et 4, contiennent justement ces instructions. On retrouve les deux lignes qui étaient répétées dans le premier programme, mais naturellement vous pouvez placer autant de lignes que vous voulez. Remarquez que ces lignes sont décalées vers la droite : on dit qu'elles sont *indentées*. En langage Python, cette *indentation* est extrèmement importante. Les instructions qui appartiennent à une fonction constituent ce que l'on appelle un *bloc d'instructions*. Selon les langages, différentes techniques sont utilisées pour repérer le début et la fin d'un bloc : par exemple, en langages Ada ou Pascal, le début est marqué par **begin** et la fin par **end**. En langage C/C++, on utilise les accolades **{** et **}** pour » encadrer » un bloc. Python, lui utilise l'indentation : les lignes décalées du même nombre d'espaces appartiennent au même bloc. C'est de cette façon que Python reconnaît que les lignes 3 et 4 constituent un bloc contenu dans la fonction **Separateur()**, tandis que les lignes suivantes appartiennent à un autre bloc. Le nombre d'espaces en début de ligne n'a pas vraiment d'importance, ce qui compte est que les lignes d'un même bloc soient indentées de la même façon. Personnellement, j'utilise trois caractères : deux ne sont pas suffisants pour un bon confort visuel, tandis que quatre sont un peu trop. Mais il s'agit là d'une convenance personnelle, vous pouvez n'en utiliser qu'un seul.

L'indentation fait donc partie de la *syntaxe* du langage Python, tandis qu'elle n'est qu'un ornement pour d'autres langages. Il est toutefois vivement recommandé de l'utiliser pour faire apparaître visuellement les blocs d'instructions : cela facilite grandement la relecture. Aussi n'utiliser qu'un seul espace pour marquer les blocs, à mon avis, nuit beaucoup à la lisibilité. Par ailleurs, je vous recommande également de clairement séparer les fonctions en insérant au moins une ligne vide avant et après : cela ne change rien au programme, mais encore une fois cela facilite sa lecture.

IV - L'appel de la fonction

Voyons maintenant ce qui se passe quand on utilise, quand on appelle notre fonction. Pour appeler une fonction, il suffit d'inscrire son nom, sans oublier la paire de parenthèses. Rappelez-vous, dans notre exemple, tout se passe comme si les lignes 7, 9 et 11 étaient remplacées par le contenu de la fonction. Voyez le schéma suivant : les instructions sont symbolisées par les disques noir, et les flèches représentent le déroulement du programme :



Les événements surviennent ainsi :

1 d'abord, on trouve l'instruction **print 10** : celle-ci est exécutée normalement ;
2 puis, on rencontre **Separateur()** : Python reconnaît qu'il s'agit d'une fonction et va donc aiguiller le programme vers le contenu de cellec-ci ;
3 les deux instructions contenues dans la fonction sont exécutées ;
4 la fonction étant terminée, Python revient au programme principal, à l'instruction suivant l'appel de la fonction : c'est l'instruction **print 20**, qui est exécutée normalement.

Le reste du programme est exécuté selon le même principe. Tapez ce programme, exécutez-le : vous verrez que le résultat est rigoureusement identique au premier programme.

Maintenant, certains esprits chagrins pourraient faire remarquer : « *quel intérêt que tout cela, le deuxième programme est plus compliqué que le premier, il fait la même chose, et il est tout aussi long ?* ». En effet, mais il ne s'agit là que d'un microscopique exemple. Concernant la longueur, imaginez vouloir rajouter l'affichage de trois nombres. Dans le premier programme, vous devrez taper neufs lignes. Dans le deuxième, six seulement. Nous avons donc bien gagné en longueur, au moins pour la vie future du programme. Mais comme nous allons le voir maintenant, nous avons gagné beaucoup plus que cela : nous avons gagné en *souplesse*.

V - Les paramètres

On ne le répétera jamais assez, un programme évolue au cours du temps, au fur et à mesure que les besoins changent. Par exemple, on pourrait estimer qu'une largeur de 10 caractères pour nos lignes séparatrices n'est pas suffisante, et 20 seraient préférables. Dans le premier programme, cela implique de changer six valeurs, alors qu'il suffit d'en changer deux dans le second. Nous avons donc gagné en facilité de maintenance et d'évolution. Toutefois, ce n'est pas encore satisfaisant.

Il est prévisible que plus tard, nous voudrons encore changer la largeur de nos lignes de tirets. Mieux : sans doute voudrons-nous un jour pouvoir afficher des lignes de différentes longueurs ! Par exemple, une ligne de 10 tirets, puis une ligne de 20, puis une ligne de 30... On pourrait imaginer créer une fonction pour chaque largeur de ligne, mais on retomberait alors dans les mêmes problèmes que ceux du premier programme. Ce qu'il nous faut, c'est une fonction plus *générique*, c'est-à-dire qui puisse adapter son action selon la situation. Pour réaliser cela, nous allons utiliser des *paramètres*. Modifiez le programme ainsi (**prog_3.py**) :

```
 1 #!/usr/bin/env python
 2 def Separateur(largeur):
 3     print "_"*largeur
 4     print "-"*largeur
 5 print 10
 6 Separateur(10)
 7 print 20
 8 Separateur(20)
 9 print 30
10 Separateur(30)
11 print 40
```

Cette fois, quelque chose apparaît entre les parenthèses dans la définition de la fonction. Il s'agit d'un paramètre : une information donnée à la fonction au moment où on l'appelle. A l'intérieur de la fonction, ce paramètre se comporte comme une variable qui serait nommée **largeur** : là où son nom apparaît, Python le remplace par sa valeur. Voyez la ligne 6. Entre les parenthèses, nous indiquons quelle valeur doit avoir le paramètre au moment de cet appel à la fonction, 10 dans l'exemple. Dans ce cas, la fonction se comporte exactement comme celle définie dans le premier programme. Par contre, ligne 8, nous donnons une autre valeur : cette fois les lignes de tirets seront de 20 caractères ! Essayez ce programme, et constatez que la même fonction produit différents résultats, selon la valeur du paramètre qu'on lui donne.

V-A - Plusieurs paramètres

Il est heureusement possible de passer plus d'un paramètre à une fonction. Par exemple, nous pourrions vouloir utiliser d'autres caractères pour dessiner nos lignes de séparation. Notre fonction aura donc besoin de trois paramètres : le premier donne la longueur, le deuxième le caractère de la première ligne, et le troisième le caratère de la deuxième ligne. Pour passer plusieurs paramètres, il suffit d'en donner la liste, séparés par des virgules. Voici le nouveau programme (**prog_4.py**) :

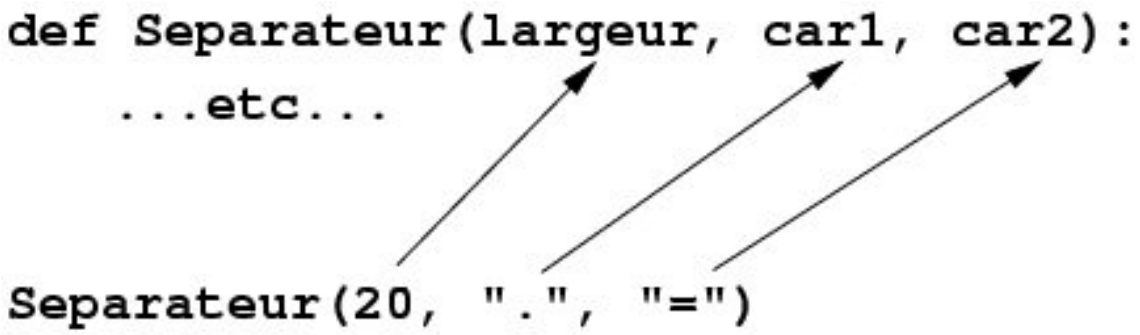
```
 1 #!/usr/bin/env python
 2 def Separateur(largeur, car1, car2):
 3     print car1*largeur
 4     print car2*largeur
 5 print 10
 6 Separateur(10, "_", "-")
 7 print 20
 8 Separateur(car2=".", car1="=", largeur=20)
 9 print 30
10 Separateur(30, "_-", "-_")
11 print 40
```

Les nouveaux paramètres à la fonction sont nommés **car1** et **car2**, comme vous le voyez ligne 2. De même que

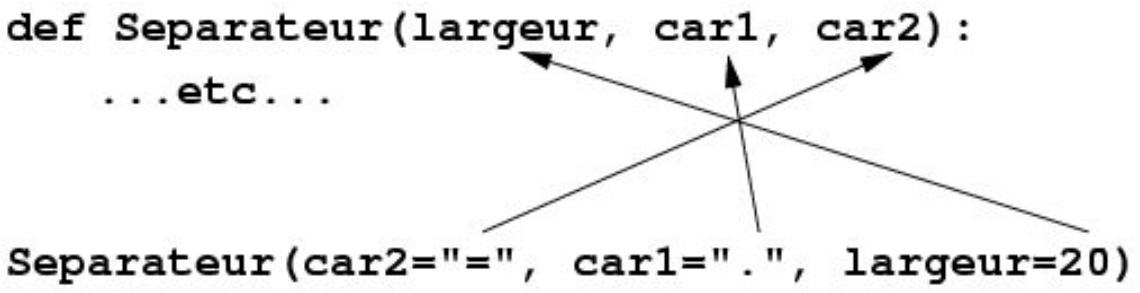
largeur, ils se comportent comme des variables à l'intérieur de la fonction, et n'existent simplement pas en-dehors d'elle. Voyez comme les appels à la fonction ont été modifiés pour utiliser cette nouvelle possibilité. Notre fonction est maintenant (presque) aussi générique que possible, et nous permet de dessiner des lignes de différentes longueurs et de différents caractères.

V-B - Nommer les valeurs passées

L'association entre les valeurs données lors de l'appel de la fonction (lignes 6, 8 et 10) et les paramètres donnés lors de sa définition (ligne 2) se fait normalement par l'ordre dans lequel ils apparaissent : la première valeur se retrouve « dans » le premier paramètre, la deuxième valeur « dans » le second paramètre, et ainsi de suite, ce que l'on peut représenter ainsi :



Devoir respecter cet ordre est parfois contraignant, surtout pour des fonctions ayant beaucoup de paramètres : on oubli rapidement quel paramètre vient avant ou après quel autre. Python nous propose pour cela une facilité, que l'on retrouve également par exemple en langage Ada : la possibilité de nommer les valeurs lors de l'appel de la fonction. Cela se fait simplement en indiquant devant chaque valeur, le nom du paramètre auquel elle est destinée, suivi d'un signe « = » - un peu comme si on déclarait une variable. Reprenons l'exemple précédent, mais en changeant l'ordre des paramètres au moment de l'appel :



Le résultat final sera rigoureusement le même. Par ailleurs, nommer ainsi les paramètres permet encore une fois de rendre le code plus facile à relire, surtout si le relecteur n'est pas l'auteur : cela évite de se référer sans cesse à la définition de la fonction pour connaître le sens de chaque paramètre. Petite recommandation, à ce sujet : donnez toujours un nom « parlant » à vos paramètres. Evitez de les nommer **par1**, **par2**, **par3**... encore une fois, la compréhension du programme en sera facilitée !

V-C - Valeur par défaut

Nous avons donc créé une fonction assez générique, avec plusieurs paramètres. Cela nous donne une grande souplesse et ouvre beaucoup de perspectives, mais dans la pratique, on constate que l'on utilise souvent les mêmes paramètres - par exemple, la plupart du temps nous voulons le même type de lignes. Le gain en *généricité*

(le fait d'être générique, c'est-à-dire être adaptable à différents contextes), en souplesse, semble devoir être payé par une plus grande complexité d'utilisation de la fonction et l'obligation de taper finalement plus de choses. Là encore, Python nous propose un moyen pour éviter de saisir encore et encore les mêmes valeurs pour les mêmes paramètres.

Il s'agit de spécifier, dans la définition de la fonction, une valeur par défaut pour un ou plusieurs paramètres, c'est-à-dire la valeur qu'ils auront s'ils ne sont pas présents à l'appel de la fonction. Je vous propose cette nouvelle définition pour notre fonction (**prog_5.py**) :

```
1 #!/usr/bin/env python
2 def Separateur(largeur, car1="_", car2="-"):
3     print car1*largeur
4     print car2*largeur
5 print 10
6 Separateur(10)
```

La suite du programme est inchangée. Remarquez dans la définition ligne 2, on retrouve la même syntaxe que celle utilisée pour déclarer une variable : **car1** et **car2** ont maintenant une valeur par défaut, ce qui signifie qu'il n'est plus indispensable de leur donner une valeur au moment où l'on appel la fonction. La ligne 6 en est un exemple : seule apparaît la valeur pour le premier paramètre, la seule qui soit laissée obligatoire. Notez bien que, si vous essayez d'appeler la fonction **Separateur()** ainsi définie sans aucun paramètre, vous vous exposez au courroux de l'interpréteur Python ! Dans cet exemple, au moment où la fonction est appelée ligne 6, **car1** contiendra "_" et **car2** contiendra "-".

VI - Retourner une valeur

Il nous reste un dernier pas à faire sur le chemin de la généricité (enfin, pour l'instant). Notre fonction effectue certains calculs (si on considère que la construction d'une chaîne de caractères est un calcul), et affiche le résultat de ces calculs. C'est justement là le dernier problème qui nous reste à résoudre : notre fonction ne devrait pas afficher quoi que ce soit. Je m'explique.

La possibilité de définir des fonctions, ou d'autres types d'entités que nous rencontrerons plus tard, participent du grand principe de la *programmation structurée*. C'est-à-dire que l'on s'efforce d'organiser les programmes selon une méthode cartésienne, consistant à diviser les gros problèmes en plus petits. On constitue ainsi des éléments de programmes, qui communiquent et travaillent ensembles, tout en évitant dans la mesure du possible que ces éléments dépendent trop les uns des autres. Tous les langages de programmation modernes offrent des techniques pour atteindre cet objectif, les fonctions en étant une parmis d'autres. L'objectif étant d'éviter le fouilli que pouvaient être d'anciens programmes écrits par exemple en langage Basic.

De l'expérience accumulée depuis les origines de la programmation, deux grands ensembles se sont dégagés, chacun étant présent dans chaque programme :

l'interface,

qui est la partie du programme responsable de l'interaction avec l'utilisateur, c'est-à-dire de ce que doit afficher le programme et les moyens par lesquels l'utilisateur peut donner des informations au programme ; on utilise parfois le terme savant d'IHM, pour *Interface Homme-Machine*, ou l'anglicisme *look-and-feel* (apparence et manière d'être) ;

le noyau,

ou le c#ur du programme, qui contient ce que l'on pourrait appeler (avec les précautions d'usage) son intelligence, qui est la partie du programme qui effectue des calculs pour produire des résultats - certains devant être affichés, d'autres pas.

L'expérience a montré qu'il était vivement recommandé de séparer nettement ces deux grandes parties. La structure du programme n'en est que plus claire et aisée à appréhender, et la modification d'une partie risque moins d'impliquer des modifications importantes dans l'autre partie. Ne pas respecter ce principe élémentaire implique inévitablement, tôt ou tard, un effort considérable si le programme doit évoluer - car si tout est trop mélangé, un changement d'un coté nécessitera un changement de l'autre, qui à son tour risque de provoquer un changement dans la première partie... cela devient rapidement infernal. Ce « syndrôme » est souvent désigné par le terme d'*effets de bord*.

Notre exemple, aussi trivial soit-il, n'échappe pas à cette règle. Nous avons une partie de programme qui effectue un calcul, à savoir construire une chaîne de caractères. A strictement parler, la fonction qui effectue ce calcul n'a pas à « savoir » ce que l'on va faire de cette chaîne de caractères. Pour l'instant, la fonction provoque un affichage. Mais comment faire si, demain, au lieu d'afficher cette chaîne nous voulons l'écrire dans un fichier, ou la combiner de manière complexe avec d'autres chaînes ? En l'état actuel, une telle évolution du programme nécessiterait des modifications profondes s'étendant sur tout le code.

L'idée consiste donc à définir des fonctions qui font ce pour quoi elles ont été créées, mais pas plus. Notre fonction calcul une chaîne de caractères, d'accord. Mais ce n'est pas son rôle de l'afficher. Dans notre cas, ce rôle est dévolu au programme principal. Il nous faut donc un moyen pour « récupérer » ce que la fonction aura produit. On dit alors que la fonction va *retourner* un résultat, ce résultat étant la *valeur de retour* de la fonction. Ce que nous ferons de ce résultat est la responsabilité de celui qui appelle la fonction, dans notre cas le programme principal. Voyez la nouvelle - et dernière - version de notre programme, dans le fichier **prog_6.py** :

```
 1 #!/usr/bin/env python
 2 def Separateur(largeur, car1="_", car2="-"):
 3     chaine = car1*largeur + "\n" + car2*largeur
 4     return chaine
 5 print 10
 6 print Separateur(10)
 7 print 20
 8 print Separateur(car2="=", car1=".", largeur=20)
 9 print 30
10 print Separateur(30, "_-", "-_")
11 print 40
```

La ligne 3 déclare une variable nommée **chaine** destinée à recevoir le résultat du calcul. Ce calcul est la concaténation des deux chaînes que nous affichions auparavant, en insérant au milieu un caractère spécial : **\n**. Ce caractère est en fait un code permettant d'insérer un saut à la ligne dans une chaîne : lorsqu'elle sera affichée, notre chaîne occupera donc deux lignes, comme précédemment. Notez bien que cette variable **chaine** est dite *locale à la fonction* : elle n'existe qu'à l'intérieur de la fonction et pour le temps que s'exécute celle-ci. Dès que la fonction a terminé son travail, la variable **chaine** disparaît purement et simplement. Une nouvelle variable portant ce nom ne sera en fait créée qu'au prochain appel de la fonction.

Voyez ligne 4 l'utilisation du mot-clef **return**. Il indique à Python que nous voulons que la fonction retourne quelque chose, une valeur, laquelle est donnée juste après ce mot **return**. Ici, nous demandons de retourner la valeur de la variable **chaine**. Lorsqu'on appelle la fonction, par exemple ligne 6, tout ce passe alors comme si cet appel était remplacé par la valeur retournée par la fonction. Ainsi définie, notre fonction se rapproche un peu de la notion mathématique éponyme.

Par cette dernière manipulation, nous avons finalement atteint nos objectifs :

- un programme structuré, séparant nettement les calculs de l'affichage ;
- une fonction générique, souple et simple d'utilisation, qui pourrait être réutilisée telle quelle dans des contextes totalement différents.

VI - Quand créer une fonction ?

C'est sans doute l'une des premières questions que vous vous poserez lors de vos expérimentations personnelles. Il n'y a pas vraiment de réponse absolue. A mesure que vous gagnerez en expérience, l'intérêt des fonctions vous apparaîtra plus clairement. Pour vous donner une idée, quelqu'un a dit un jour : « *si ça fait plus de trois lignes et que je m'en sers plus de trois fois, alors je fais une fonction* ». C'est un peu simpliste, mais vous voyez l'idée. D'une manière générale, on fait une fonction lorsqu'une séquence d'instructions a toutes les chances d'être utilisée de nombreuses fois, quitte à remplacer quelques valeurs par des paramètres.

VIII - Conclusion

En gros, vous pouvez imaginer une fonction comme étant une machine, qui prend des matières premières d'un coté (les paramètres) et produit un résultat de l'autre coté, à la sortie. C'est également un moyen puissant pour écrire du code que l'on pourra réutiliser.

Jusqu'ici, nos programmes étaient essentiellement linéaires, les instructions étant exécutées l'une après l'autre, la fonction n'étant guère qu'un détour temporaire. Le mois prochain, nous commencerons à aborder les structures de contrôles, qui permettent de modifier le déroulement d'un programme selon des conditions que nous aurons définies.

IX - Téléchargement

Vous pouvez télécharger ce cours sous format **pdf**

Langue	Mise à jour	Pages	Taille	Mode FTP	Mode HTTP de secours
	2005.11.20	14	99 Ko	YB03.pdf	YB03.pdf

Vous pouvez également télécharger les sources de ce cours: prog.zip