Partie IV - Contrôler le déroulement d#un programme

par Yves Bailly

Date de publication : 31/05/2003

Dernière mise à jour : 20/11/2005

Dans l#article précédent, nous avons vu l#intérêt que pouvait avoir les fonctions. Aujourd#hui je vous propose de découvrir comment contrôler le déroulement d#un programme, en réalisant un petit jeu élémentaire.

I - Introduction
II - L#alternative
III - La boucle, ou répétitive
IV - Algorithmique et algèbre de Boole
V - Conclusion
VI - Téléchargement

I - Introduction

Par « *contrôler le déroulement d#un programme* », on entend généralement diriger la succession des intructions, par exemple choisir entre certaines séquences selon divers critères, ou provoquer la répétition de l#exécution d#une certaine séquence. Pour illustrer ces principes, nous allons réaliser le petit jeu suivant : le programme va choisir un nombre au hasard entre 1 et 100, et l#utilisateur devra deviner ce nombre en faisant des propositions. À chaque proposition, le programme affichera « trop grand » ou « trop petit », selon le cas.

Sans plus attendre, je vous donne le début du programme, que je vous invite à saisir dans un fichier **jeu.py** :

```
1 #!/usr/bin/python
2 from random import randint
```

Ici nous anticipons un peu sur l#avenir. Pour faire simple, disons que la ligne précédente nous donne accès à une fonction nommée **randint()**, qui permet d#obtenir un nombre au hasard compris entre deux nombres donnés en paramètre. Rassurez-vous, tout sera éclairci plus tard. Nous utilisons cette fonction immédiatement :

```
3 nombre_hasard = randint(1, 100)
```

Le résultat de la fonction **randint()** est stocké dans la variable **nombre_hasard**. C#est ce nombre que l#utilisateur devra trouver par essais successifs. Mais voyez la ligne suivante :

```
4 chaine = raw_input("Donnez un nombre : ")
```

La fonction **raw_input()** est en quelque sorte l#opposée de la commande **print**, que nous avons déjà beaucoup utilisée. Son rôle est d#attendre que l#utilisateur entre quelquechose au clavier, la chaîne de caractères qui est passée en paramètre étant affichée en début de ligne. Cette chaîne est généralement supposée décrire ce que l#on attend. On la désigne par le terme d#*invite* (car elle invite l#utilisateur à taper quelquechose), ou l#anglicisme *prompt*.

Cette fonction renvoie toujours une chaîne de caractères, or nous voulons un nombre. L#étape suivante consiste donc à transformer cette chaîne en nombre, comme nous avons appris à le faire dans le deuxième article de cette série :

```
5 nombre_joueur = int(chaine)
```

Le nombre que l#utilisateur (le joueur) a donné est donc stocké dans la variable **nombre_joueur**.

II - L#alternative

Une *alternative* est une situation où l#on a le choix entre deux possibilités, deux chemins. Chacune des possibilités est parfois désignée par le terme de *branche* de l#alternative. Plaçons-nous dans la situation où l#utilisateur a saisi un nombre : il nous faut déterminer le message à afficher. Suivez bien, le procédé évoqué maintenant est l#un des plus fondamentaux pour concevoir une partie de programme, voire un programme complet.

Mettez-vous à la place de celui-ci, là où nous en sommes, et agissez mentalement à sa place. Vous disposez d#un nombre de référence, celui tiré au hasard, et d#un nombre donné par l#utilisateur : ce sont les *données d#entrée*, celles que vous devez traiter. Par ce traitement, vous devez produire un résultat qui est un message adapté à la comparaison de ces deux nombres. Décomposez autant que possible votre processus de pensée, en oubliant pour l#instant le cas particuliers où les deux nombres sont égaux. Cela devrait ressembler à ceci :

Cette simple phrase est parfois désignée par le terme de *principe* : une description synthétique et concise que ce que l#on veut faire. Maintenant rapprochons-nous du langage Python, en remplaçant certains termes par les variables qu#ils représentent :

La notion de message, dans notre cas, correspond à un affichage à l#écran, ce que permet la commande **print**. Par ailleurs, la comparaison de deux nombres se fait en utilisant les opérateurs méthématiques usuels que sont les symboles < et >. Poursuivons la « pythonisation » de notre principe :

Voilà qui commence à ressembler à un programme ! Il nous reste à traduire les mots « si », « alors » et « sinon ». Voici cette traduction en langage Python :

```
 7   if ( nombre_joueur > nombre_hasard ) :
 8     print "Trop grand !"
 9   else :
10     print "Trop petit !"
```

Le mot *si* se traduit en anglais et dans la plupart des langages de programmation par *if*. Le mot alors se traduit par *then*, mais de nombreux langages, dont Python, omettent simplement ce mot. Enfin, *sinon* se traduit par *else*. Remarquez la fin des lignes 7 et 9, vous retrouvez les deux-points que nous avions rencontrés pour la définition d#une fonction. De même, les lignes 8 et 10 sont indentées : de même que les instructions d#une fonction sont rassemblées dans un bloc d#instructions, les instructions correspondant à chacune des branches de l#alternative sont rassemblées dans un bloc. Ici chacun des deux blocs n#est constitué que d#une seule ligne, mais naturellement vous êtes libre d#en rajouter.

Rappelez-vous qu#en python, l#indentation est très importante : c#est ce décalage vers la droite, obtenu en insérant des espaces en début de ligne, qui détermine ce qui fait partie de tel ou tel bloc.

Ligne 7, ce qui se trouve entre les parenthèses constitue la *condition*, ou le *test*, qui permet de déterminer quelle branche de l#alternative on va exécuter. Si le test réussi, si la condition est vraie, alors on exécute la première branche, celle qui se trouve juste après le **if**. Sinon, on exécute l#autre branche. Une fois retraduit en français, tout cela est en fait très naturel ! Ne négligez pas l#importance de la formulation et de la sémantique en langage naturel, elle est bien souvent un guide précieux pour l#écriture du code informatique.

Les parenthèses ligne 7 sont facultatives, on pourrait ne pas les mettre sans changer quoi que ce soit au test. Elles sont toutefois indispensables dans certains langages, comme le C ou le C++, et personnellement je trouve qu#elles améliorent la lisibilité du programme.

Dernière remarque, dans certaines situations vous n#aurez pas besoin de deuxième branche à l#alternative. Vous voudrez simplement exécuter quelque chose si une condition est remplie, et ne rien faire sinon. Dans ce cas, vous avez essentiellement deux possibilités. La première consiste à explicitement indiquer que la deuxième branche ne fait rien, ce qui s#écrit ainsi en Python :

```
if ( une_condition ) :
   des_instructions
 else :
   pass
suite_du_programme
```

La commande **pass** pourrait être traduite par « *passer sans rien faire* ». L#autre possibilité est de tout simplement omettre la clause **else** :

```
if ( une_condition ) :
   des_instructions
suite_du_programme
```

Le résultat est le même, à vous d#utiliser ce qui vous paraît le plus lisible. Vous voyez encore une fois l#importance de l#indentation avec Python.

Comme exercice, essayez de modifier un peu le principe plus haut pour traiter le cas où les deux nombres sont égaux, et de modifier le code en conséquence. Je vous donne la solution tout de suite, mais ne la regardez pas sans expérimenter ! Pour information, pour tester l#égalité entre deux nombres il faut utiliser l#opérateur == (et non pas le simple signe =, c#est une source très courante d#erreurs) :

```
if ( nombre_joueur == nombre_hasard ) :
   print "Vous avez trouvé!"
 else :
   if ( nombre_jouer > nombre_hasard ) :
     print "Trop grand!"
   else :
     print "Trop petit!"
```

Cette solution utilise une *imbrication de blocs*, que l#on peut représenter de la façon suivante, où un cadre plus foncé désigne un bloc plus « interne » :

```
if ( nombre_joueur > nombre_hasard ) :
    print "Vous avez trouvé !"
else :
    if ( nombre_joueur > nombre_hasard ) :
        print "Trop grand !"
    else :
        print "Trop petit !"
```

III - La boucle, ou répétitive

Nous avons progressé, mais notre programme n#est pas terminé : actuellement l#utilisateur ne peut proposer qu#un seul nombre, puis le programme se termine. Comment faire pour que l#utilisateur puisse proposer plusieurs nombres ? Une solution consisterait à dupliquer les lignes de code. Mais cela ne paraît pas une bonne solution, surtout si on se trouve confronter à des centaines d#instructions plutôt que quelques-unes. Et rappelez-vous qu#en général, un programmeur est paresseux (à ne pas confondre avec feignant).

Encore une fois, cherchons un principe, formulé en langage clair. Je vous propose le suivant :

La partie « *afficher un message* » a déjà été traitée, nous pouvons donc remplacer ce membre par notre principe précédent (que nous savons déjà traduire) :

Ce qui nous intéresse ici est le premier membre de cette phrase. Les mots « *tant que* » sous-entendent que nous allons effectuer les opérations qui suivent jusqu#à ce qu#une certaine condition soit vérifiée, ou plus précisément ici, tant qu#une condition n#est *pas* vérifiée. Premier problème, comment traduire l#expression « *ne sont pas égaux* » ? Il suffit d#exprimer la négation de la condition « *sont égaux* ». En langage Python, la négation d#une condition s#écrit simplement **not**. En effectuant les remplacements par les variables que nous avons, et en utilisant les opérateurs de comparaison, nous pouvons réécrire la phrase ainsi :

Si je vous dis que les mots « *tant que* » se traduisent en anglais et en Python par **while**, avec un peu d#imagination vous devriez pouvoir écrire le code correspondant à cette phrase. Le voici :

```
 6 while ( not  (nombre_joueur == nombre_hasard) ) :
 7   if ( nombre_joueur > nombre_hasard ) :
 8     print "Trop grand !"
 9   else :
10     print "Trop petit !"
11   chaine = raw_input("Donnez un nombre : ")
12   nombre_joueur = int(chaine)
```

Vous voyez comment a été traduite la dernière partie de la phrase, lignes 11 et 12 : pour obtenir un nouveau nombre, on réutilise simplement le code utilisé au début pour demander le premier nombre. Cette première demande est en fait utilisée pour « amorcer la pompe », c#est-à-dire obtenir une situation dans laquelle le code qui suit va pouvoir s#exécuter.

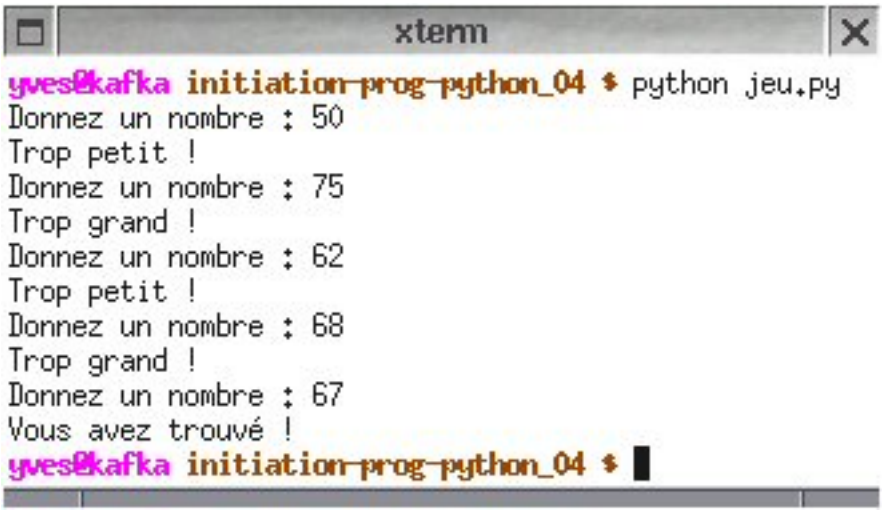
Ce que nous venons de réaliser est parfois désigné par le terme de *répétitive* et plus communément par le terme de *boucle*. Une boucle consiste à répéter une suite d#instructions (rassemblées dans un bloc, comme vous pouvez le constater ici), chaque répétition (ou *tour de boucle*) étant précédée d#une condition. Si cette condition est vérifiée, le bloc de la boucle est exécuté, sinon il est ignoré et le programme se poursuit à l#instruction suivant immédiatement la boucle.

Dans notre exemple, la condition est la *négation* (**not**) de l#égalité de deux nombres. Cela signifie que la boucle s#arrêtera dès que les deux nombres seront égaux. Formulé différemment, la boucle s#arrêtera dès qu#il sera faux de dire que les deux nombres ne sont pas égaux : comme en langage naturel, une double négation équivaut à une affirmation.

Nous pouvons donc maintenant terminer notre programme. Comme la boucle s#arrête dès que les deux nombres sont égaux, l#instruction que nous allons ajouter juste après la boucle sera l#affichage d#un message indiquant à l#utilisateur qu#il a trouvé le nombre :

```
13 print "Vous avez trouvé!"
```

Voici un exemple d#exécution de ce petit programme :



IV - Algorithmique et algèbre de Boole

Nous venons de voir trois notions très importantes en programmation, dont une de façon implicite.

D#abord la notion d#*alternative*, qui permet d#orienter le flux d#un programme, à la manière d#un aiguillage sur une voie ferrée. Grâce à cela, certaines parties d#un programme ne seront exécutées que selon certaines conditions données.

Puis la notion de *boucle*, qui permet d#exécuter plusieurs fois une même suite d#instructions, comme si la voie ferrée formait un anneau, revenant sur elle-même. La sortie de l#anneau, c#est-à-dire la fin de la boucle, dépend d#une condition donnée (d#un aiguillage).

La troisième notion vient juste d#être mentionnée deux fois, car elle est implicite dans les deux précédentes : la notion de *condition*. La *valeur* d#une condition est donnée par le résultat d#un calcul particuliers, qui est le *test* correspondant à la condition. À strictement parler, le type de cette valeur n#est ni un entier, ni une chaîne de caractères. Il s#agit d#un type dit *booléen* (prononcez « *bou-lé-un* »), par référence au modèle développé par le mathématicien et logicien britannique George Boole (1815-1864, et dire que l#on parle de « nouvelles technologies »...).

Une variable entière (de type entier) peut prendre les valeurs 1, 2, 3, 197635, et bien d#autres. Une variable *booléenne* (de type booléen) ne peut prendre que deux valeurs : soit *vraie* (*true*), soit *faux* (*false*). Par commodité, on note souvent chacune de ces valeurs respectivement 1 et 0, mais ce n#est qu#une représentation. Les règles qui régissent les calculs que l#on peut effectuer sur de telles variables, et bien d#autres choses encore, sont rassemblées dans ce que l#on appelle l#*algèbre de Boole*, conçu justement par monsieur Boole. Nous avons rencontré l#une de ces opérations : la *négation*. Cette opération donne une valeur opposée à la valeur de la variable sur laquelle elle est appliquée : si une variable a la valeur *vrai*, sa négation est la valeur *faux*. Si une variable a la valeur *faux*, sa négation a la valeur *vrai*. Ceci a été utilisé dans l#écriture de la condition de la boucle, ligne 6 du programme.

Ces trois notions font parties des fondements de ce que l#on appelle l#*algorithmique*, la science qui étudie les algorithmes. Un *algorithme* est la description d#une suite d#actions ou d#opérations à effectuer dans un certain ordre. Au cours de cette suite peuvent intervenir des boucles ou des alternatives. La notion d#algorithme n#est pas nouvelle non plus. On doit à Euclide un procédé systématique pour trouver le plus grand diviseur commun (PGCD) de deux nombres, vers le IIIème siècle avant l#an 1. À peine plus jeune, Érathostène a imaginé une méthode, appellée *crible d#Érathostène*, qui permet d#obtenir les nombres premiers. Cette notion a été formalisée par Lady Ada Lovelace Byron, épouse du poète du même nom au XIXème siècle, qui donna son nom au rigoureux langage de programmation Ada.

Nous venons de pratiquer un peu l#algorithmique, en imaginant les successions d#actions nécessaires pour obtenir le résultat désiré. L#action de traduire cet algorithme en langage Python est désignée par le terme d#*implémentation de l#algorithme*. Tout travail de programmation est en fait composé au minimum de ces deux étapes : conception de l#algorithme, puis implémentation de celui-ci. Avec l#expérience, dans bien des situations la première phase devient presque inconsciente et l#on a tendance à se jeter sur le clavier. Il faut se méfier de cette tendance, qui conduit parfois à omettre des cas particuliers importants.

V - Conclusion

J#espère que ce que nous venons de voir vous donnera des idées de programmes, pour expérimenter ces nouvelles notions. Dans le prochain numéro, nous verrons un nouveau type de données, la liste, avec lequel nous pourrons découvrir une nouvelle forme de boucle. Bonne programmation !

VI - Téléchargement

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

Langue	Mise à jour	Pages	Taille	Mode FTP	Mode HTTP de secours
	2005.11.20	12	96 Ko	YB04.pdf	YB04.pdf

Vous pouvez également télécharger les sources de ce cours: jeu.py