Partie V - La liste et la boucle bornée

par Yves Bailly

Date de publication :

Dernière mise à jour : 21/11/2005

Jusqu'à présent, nous avons utilisé des variables simples. Découvrons ce mois-ci un type de données plus sophistiqué, la liste, ainsi qu'une nouvelle forme de boucle adaptée à son utilisation.

I - Introduction

Imaginez cette situation : vous êtes technicien dans une écurie de Formule1. Plusieurs voitures sont créées et testées. Pour déterminer laquelle participera au prochain grand prix, différentes statistiques doivent être réalisées sur les résultats des essais sur circuit. Chaque voiture effectue quelques tours de circuit, la vitesse moyenne de chaque tour étant enregistrée. Il s'agit, dans un premier temps, de trouver la vitesse la plus élevée.

Avec ce que nous connaissons déjà, vous pourriez réaliser ce programme, et je vous encourage même à le tenter. Il suffit d'une variable, dans laquelle serait stockée une valeur donnée par l'utilisateur. Si la valeur donnée est plus grande que celle stockée, alors elle la remplace. Ceci tant que l'utilisateur donne quelque chose. Prenez cela comme un exercice... dont voici immédiatement le corrigé (ce n'est qu'une solution possible parmis une infinité d'autres) :

```
1 vitesse = 0.0
2 vitesse_max = 0.0
3 chaine = raw_input("Vitesse ? ")
4 while ( len(chaine) > 0 ) :
5   vitesse = float(chaine)
6   if ( vitesse > vitesse_max ) :
7     vitesse_max = vitesse
8   chaine = raw_input("Vitesse ? ")
9 print "Vitesse max :", vitesse_max
```

Rien de bien compliqué. Sauf que... sauf qu'un jour, on vous demandera de sauvegarder les vitesses données, de calculer la vitesse moyenne, ce genre de choses. Bref, il est nécessaire de mémoriser les valeurs qui seront données. Avec nos connaissances actuelles, cela implique de passer par plusieurs variables différentes, comme **vitesse_1**, **vitesse_2**, etc. Premier problème, nous ne connaissons pas à l'avance le nombre de tours de circuit qu'une voiture effectuera, même si on peut supposer qu'il ne sera pas très grand, de l'ordre d'une dizaine. Ensuite, cela deviendra rapidement fastidieux, et cette solution serait inapplicable si nous devions mémoriser des centaines de valeurs.

II - La liste

Plutôt que de déclarer cinq variables, nous n'allons en déclarer qu'une seule. Mais ce sera une variable un peu particulière : elle ne contiendra pas une valeure unique, mais une *série* de valeurs - une pour chaque vitesse que nous devrons mémoriser. Une telle variable se déclare en Python de la façon suivante :

```
vitesses = []
```

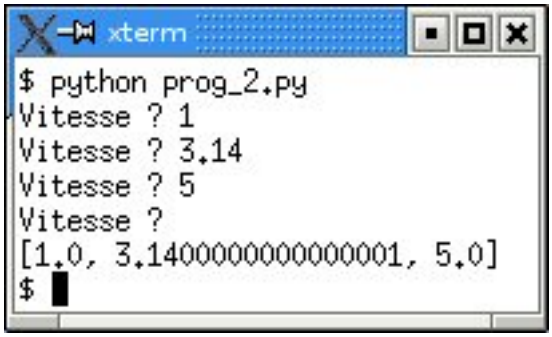
Comme d'habitude, à gauche du signe = se trouve le nom de la variable, à droite la valeur qui lui est affectée. Cette valeur est ici l'écriture reconnue par Python pour signifier « *une liste vide* », c'est-à-dire une liste ne contenant rien. Comme ce n'est pas très utile, il nous faut remplir cette liste avec les valeurs données par l'utilisateur. Voici comment :

```
1 vitesses = []
2 v = 0.0
3 chaine = raw_input("Vitesse ? ")
4 while ( len(chaine) > 0 ) :
5   v = float(chaine)
6   vitesses.append(v)
7   chaine = raw_input("Vitesse ? ")
8 print vitesses
```

L'ajout d'une valeur à la liste est réalisé ligne 6. On utilise pour cela une fonction nommée **append()** (*ajouter à la fin*, en anglais), qui prend en paramètre la valeur à ajouter. Elle est appellée ici d'une façon particulière : nous devons préciser à quelle liste nous ajoutons une valeur. En effet, comme on peut avoir plusieurs variables, on peut avoir plusieurs listes - après tout, une liste n'est jamais qu'une variable d'un type particulier. Cette précision est obtenue en utilisant ce que l'on appel la *notation pointée* : à gauche du point, le nom de la variable sur laquelle on veut appliquer la fonction ; à droite du point, la fonction que l'on veut appliquer à la variable. La ligne 6 peut donc se traduire par :

Nous retrouverons ce type de notation plus tard, notamment lorsque nous aborderons les principes de la programmation orientée objet.

Le programme se termine lorsque l'utilisateur ne donne pas de vitesse, c'est-à-dire lorsqu'il presse immédiatement Entrée quand le programme attend une nouvelle valeur. Puis nous affichons une représentation du contenu de la variable **vitesses**. Voici un exemple d'exécution, où l'utilisateur a donné trois valeurs :



Voyez comment est affichée la liste : une série de valeurs délimitée par des crochets, les valeurs étant séparées par une virgule. Chaque valeur étant affichée par une représentation adaptée. Ne soyez pas surpris par la représentation de la deuxième valeur : elle résulte de l'imprécision inhérente aux nombres à virgule flottante (revoir

éventuellement le deuxième article de cette série à ce sujet, LinuxPratique 17). En mémoire, la situation ressemble à ceci :

Remarquez que cette représentation ressemble fort à celle utilisée pour les chaînes de caractères... En fait, telle que fournie par Python, la liste est très proche dans son utilisation de la chaîne de caractères.

Maintenant que nous avons une liste de vitesses, il est temps de chercher la vitesse la plus élevée.

III - Un peu d'algorithmique

Pour ce faire, nous allons créer une fonction qui prendra en paramètre, justement, une liste de vitesses. Pour un programme aussi simple, la création d'une fonction n'est pas vraiment nécessaire. Simplement cela permet de clarifier le programme, en séparant nettement la partie d'interaction avec l'utilisateur (la partie interface) de la partie calculatoire. Les raisons d'une telle séparation ont déjà été évoquée.

Pour rechercher la plus grande valeur dans une liste, problème classique s'il en est, une méthode usuelle consiste à utiliser une variable initialisée à zéro. Puis on parcourt la liste, la valeur de chaque élément étant comparé à la valeur de cette variable. Si l'élément est plus grand, alors la variable prend sa valeur. À la fin du parcours, la variable contient la plus grande valeur dans la liste. Voici une première version de cet algorithme, dans une fonction nommée **valeurMax()** :

```
1  def valeurMax(une_liste) :
2    valeur = 0.0
3    compteur = 0
4    while ( compteur < len(une_liste) ) :
5      if ( une_liste[compteur] > valeur ) :
6        valeur = une_liste[compteur]
7      compteur += 1
8    return valeur
```

Pour parcourir la liste, nous utilisons un compteur. En effet, comme les caractères d'une chaîne, les éléments contenus dans une liste peuvent être obtenu individuellement en utilisant un indice donné entre crochets, le premier élément ayant l'indice 0 (zéro). En fait, les crochets agissent sur une liste de la même façon que sur une chaîne : on peut ainsi découper une liste, assembler plusieurs listes, etc.

Nous retrouvons par ailleurs la fonction **len()** (ligne 4), qui donne le nombre d'éléments dans une liste - sa longueur, comme la longueur d'une chaîne de caractères est le nombre de caractères qu'elle contient. C'est cette valeur qui va limiter notre compteur.

La comparaison entre l'élément considéré de la liste et la valeur précédemment mémorisée est effectuée ligne 5. Si cet élément est plus grand, on le mémorise. Puis le compteur est avancé d'un élément. Enfin la valeur trouvée, la plus grande de la liste, est retournée.

Pour tester cela, ajoutez les lignes précédentes au début du programme remplissant la liste, et ajouter à la fin une simple ligne :

```
print "Vitesse max :", valeurMax(vitesses)
```

Un exemple d'exécution du nouveau programme :

Notre fonction fonctionne. Mais Python nous permet de réaliser la même chose de manière plus concise, et accessoirement plus rapide à l'exécution. Pour cela, nous allons utiliser un autre type de boucle.

IV - La boucle bornée

Il existe en effet une forme de boucle adaptée au parcours d'une liste d'éléments. Cette forme spéciale nous dispense d'utiliser un compteur et de contrôler sa valeur : elle se charge de tout cela automatiquement. Elle va donc se limiter d'elle-même aux bornes de la liste, d'où ce nom de *boucle bornée*. Les instructions contenues en son sein ne seront donc exécutées au maximum qu'autant de fois qu'il y a d'éléments dans la liste. En cela elle diffère de la boucle introduite par **while**, qui ne s'arrête que lorsqu'une certaine condition n'est plus vérifiée. Dans notre fonction, si nous avions oublié d'incrémenter le compteur (ligne 7), nous aurions créé une boucle infinie : si la valeur du compteur ne change pas, il n'atteindra jamais la longueur de la liste, et la boucle ne s'arrêtera jamais. De telles boucles infinies sont parfois intéressantes dans certaines situations, mais dans la plupart des cas elles ne sont qu'une erreur de programmation catastrophique.

La boucle que nous allons voir maintenant nous prémunit de ce désagrément, lorsque l'on sait à l'avance combien de fois on veut exécuter les instructions. Voici la nouvelle version de notre fonction :

```
1 def valeurMax(une_liste) :
2   max = 0
3   for val in une_liste :
4     if ( val > max ) :
5       max = val
6   return max
```

La boucle est introduite ligne 3 par le mot-clef **for**, suivi d'un nom de variable, suivi du mot-clef **in**, puis enfin la liste à parcourir. La variable donnée entre **for** et **in**, **val** dans notre exemple, va prendre successivement les différentes valeurs contenues dans la liste, dans l'ordre, de la première à la dernière. À chaque « tour de boucle », sa valeur va donc changer. Ce code est équivalent dans son résultat au précédent, mais il est plus court, plus rapide et moins sujet aux erreurs « bêtes » de programmation, comme l'oubli de l'incrémentation du compteur. Il est donc nettement préférable. La ligne 3 pourrait se traduire par :

...les instructions contenues dans la boucle, ici les lignes 4 et 5. À l'intérieur de la boucle, **val** s'utilise comme une variable classique. Simplement sa valeur est automatiquement changée par la boucle elle-même.

V - Compléments sur les listes

Comme cela a déjà été dit, une liste se comporte *grosso modo* comme une chaîne de caractères quand on utilise les crochets. Mais une liste est plus souple qu'une chaîne, en cela qu'elle peut être modifiée (dans le jargon Python, on dit que la liste est un type *mutable*). Une chaîne de caractères ne peut être modifiée. Par exemple, l'utilisation de la fonction **append()** ne crée pas une nouvelle liste, elle modifie celle sur laquelle elle est appliquée. D'autres fonctions sont disponibles pour manipuler une liste, par exemple :

- **insert()** permet d'insérer un élément au milieu de la liste ; par exemple, si **l** est une liste contenant **[1, 2, 3]**, l'instruction **l.insert(2, 4)** va modifier l, qui contiendra alors **[1, 2, 4, 3]** : le premier paramètre est l'indice après lequel on insère l'élément donné en deuxième paramètre ; cas particulier notable, si le premier paramètre vaut 0, l'insertion se fait en début de liste ;
- **reverse()**, qui ne prend aucun paramètre, retourne la liste ; par exemple, **l.reverse()** va modifier **l**, qui contiendra alors **[3, 2, 1]** ;
- **count()** prend une valeur en paramètre, et retourne le nombre d'éléments dans la liste ayant cette valeur ; par exemple, **l.count(2)** retournera **1** (puisqu'un seul élément a la valeur 2), tandis que **l.count(5)** retournera **0** ; incidemment, cela signifie que plusieurs éléments peuvent avoir la même valeur.

D'autres fonctions existent, nous les rencontrerons à mesure que nous en aurons besoin.

VI - Conclusion

La liste est un moyen puissant et souple de stocker une série de données. Vous pouvez ajouter à une liste autant d'éléments que vous voulez, la limite étant déterminée par votre matériel, la quantité de mémoire dont dispose votre ordinateur. Nous avons manipulé ici une liste *homogène* : tous les éléments sont d'un même type de données, en l'occurrence un nombre à virgule flottante. Il est parfaitement possible de créer des listes de n'importe quel type de données, des listes hétérogènes (contenant des données de types différents), et même des listes de listes ! Conceptuellement, une variable liste peut être considérée comme n'importe quelle autre variable : il n'y a pas de différence avec, par exemple, une variable contenant un nombre entier. Seules les fonctions pouvant s'appliquer à la variable, ainsi que sa représentation à l'écran, sont spécifiques au type.

Ce type de données est très fréquemment utilisé en programmation. Des langages comme Perl ou Ruby proposent un type équivalent à celui présenté ici. Les langages comme C, C++, Ada... ne contiennent pas un type de liste en tant que tel, mais tous les outils sont disponibles pour le créer - la norme du langage C++ impose même de fournir un ensemble d'outils en plus du langage, contenant (entre autres) une liste.

Le mois prochain, nous commencerons à faire nos premières armes en programmation orientée objet.

VII - Téléchargement

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

Langue	Mise à jour	Pages	Taille	Mode FTP	Mode HTTP de secours
	2005.11.21	11	62 Ko	YB05.pdf	YB05.pdf

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