Instructions itératives

1 Boucles inconditionnelles

1.1 Instruction for

Soit à calculer, pour un k donné, l'expression

$$S(p) = \sum_{i=0}^{i=p} i^k$$

Nous allons répéter une instruction de la forme valeur_somme=valeur_somme + i ** k

Le problème est que nous ne savons pas combien de fois nous aurons besoin de le faire. Une série d'instructions conditionnelles (par exemple if p==2 : puis if p==3 : puis etc..., dont la condition porterait sur la valeur de p) ne suffirait pas pour résoudre ce problème (il en faudrait une infinité).

On a donc la nécessité d'avoir la possibilité de répéter un certain nombre de fois (fixe ou variable) certaines instructions : ceci est possible grâce au mot-clé **for**. Sur notre exemple :

```
def S(p):
    valeur_somme=0
    for i in range(1,p+1):
        valeur_somme=valeur_somme+i**k
    return valeur_somme
```

1.2 Méthodologie de l'écriture de boucles for

Une boucle for est répétée un certain nombre de fois ; dans le cas de le boucle for i in range(N) elle est parcourue N fois ; de 0 à N-1 (comme souvent en informatique on commence à numéroter de zéro).

La variable i (déclarée et affectée à zéro implicitement) va ainsi prendre dans notre exemple toutes les valeurs de 1 à p. Cette variable s'appelle un *compteur* et il est possible d'y faire référence tout au long des instructions de la boucle.

La variable valeur_somme joue le rôle de la somme partielle. Il faut l'initialiser avant l'entrée de la boucle.

Pour écrire une boucle :

1. On détermine combien de fois la boucle devra s'exécuter
2. On choisit une variable pour le compteur et on détermine si celle-ci joue un rôle dans la boucle
3. On écrit le corps de la boucle
4. On vérifie si on ne doit pas initialiser certaines valeurs avant l'entrée de la boucle et si il ne faut post-traiter des résultats

Exercice 1 Détailler ces étapes pour écrire un programme calculant la n-ième puissance itérée de k, autrement dit $k^{k^{\cdot^{\cdot^{\cdot^{k}}}}}$ formé de n exemplaires de k.

1.3 Valeurs itérables

Il existe une autre situation dans laquelle on aimerait pouvoir parcourir une boucle : lorsqu'on souhaite parcourir toutes les valeurs d'un type composé (n-uplet ; chaîne de caractères, liste..).

Lorsqu'on sait identifier le premier élément à traiter et que l'on sait passer d'un élément au suivant, on parle de valeur itérable. Pour construire une boucle qui parcourt un itérable on utilise la syntaxe suivante :

for element **in** iterable :
bloc d'instructions

Exercice 2 Considérons une liste L de booléens. Que permet de calculer la fonction suivante ?

```
def f(L):
    tmp=True
    for i in L:
        tmp=(not tmp and i) or (tmp and not i)
    return tmp
```

On pourra tester ce programme avec L=[True,True,False] et L=[True,True,True] respectivement

1.4 Cas particulier de l'itérable range

L'expression range(n) est en fait également un itérable. Celle-ci renvoie une liste qui contient les entiers compris entre 0 et n-1 rangés dans l'ordre croissant.

Cette expression admet deux formes plus générales : range(m,n) qui renvoie une liste contenant les entiers compris entre m et n-1 et range(m,n,p) qui renvoie une liste dont le plus petit entier est m ; telle que l'élément après le i-ème est incrémenté de p par rapport à celui-ci et tel que le dernier élément de la liste soit strictement plus petit que p.

2 Boucles imbriquées

2.1 Principe et exemple

Quand l'instruction à l'intérieur d'une boucle est elle aussi répétitive, le corps de cette boucle contient lui-même une seconde boucle. On dit que ces deux boucles sont imbriquées. Les bornes de la boucle interne peuvent dépendre du compteur de la première boucle (et ce sera le plus souvent le cas).

Écrivons par exemple le programme qui calcule la quantité :

$$S(n) = \sum_{1 \leqslant j \leqslant i \leqslant n} \frac{1}{i+j} = \sum_{i=1}^{i=n} \sum_{j=1}^{j=i} \frac{1}{i+j}$$

```
def Somme(n):
    valeur_somme=0
    for i in range(1,n+1): # i parcourt les entiers de 1 à n
        for j in range(1,i+1): # j parcourt les entiers de 1 à i; noter l'indentation
            valeur_somme=valeur_somme+1/(i+j) # nouveau niveau d'indentation
    return valeur_somme # niveau d'indentation en sortie de la boucle externe
```

2.2 Approche de la notion de complexité

Exercice 3 : Combien de fois l'instruction de sommation est-elle exécutée dans le programme précédent ?

Comme on le constate, la boucle imbriquée est parcourue $\frac{1}{2}n^2$ fois en ordre de grandeur. On dit d'un tel algorithme qu'il a une complexité en $O(n^2)$. L'idée de base est de comparer les temps d'exécution d'un algorithme pour des grandes valeurs de n. Si un algorithme qui nécessite 100n calculs est plus lent qu'un algorithme qui nécessite n^2 opérations tant que n<100 lorsque n devient très grand (penser par exemple à $n = 10^6$ ou $n = 10^9$ qui sont des ordres de grandeur assez typiques de calculs industriels ou scientifiques) c'est le comportement "associé à n" qui devient prépondérant. De ce fait la constante $\frac{1}{2}$ a une importance relativement faible pour les grandes valeurs de n (en termes de comparaisons d'algorithmes) et est tout simplement négligée. Cette idée de complexité fera l'objet d'un cours spécifique en informatique, et la notation O fera l'objet de cours en mathématiques. Retenons simplement pour le moment que les boucles imbriquées sont très coûteuses en temps de calcul et qu'elle devraient être évitées dès lors que c'est possible.

Ainsi, on peut cherche les points de coordonnées entières d'une sphère de rayon R (qui vérifie l'équation $x^2 + y^2 + z^2 = R$). Ceci est possible en faisant parcourir aux variables x,y et z les valeurs entières dans [-R,R] ; ou en utilisant la symétrie du problème les valeurs entières dans [0,R]. Ceci nécessite néanmoins l'écriture d'une boucle triple, dans la complexité est en $\mathcal{O}(n^3)$. Une telle complexité est vite impraticable dès que les valeurs de R deviennent élevées. Il est pourtant possible de ne parcourir qu'une boucle double, dans la complexité en $\mathcal{O}(n^2)$ sera bien plus accessible en pratique.

Exercice 4 : Comment ?

2.3 Méthodologie de l'écriture de boucles imbriquées

Imbriquer des boucles est judicieux lorsqu'on a repéré des procédures doublement (ou triplement ...) répétitives. Un certain nombre de précautions sont à prendre. Dans l'ordre :

1. On repère quelle variable doit être dans la boucle interne et laquelle dans la boucle externe. En particulier, si l'exécution d'une instruction dépend d'une variable qui dépend elle-même d'une autre variable c'est la première qui doit être placée en boucle interne.
2. On s'interroge si le compteur de la boucle interne doit dépendre ou non de celui de la boucle externe
3. On écrit le corps de la boucle interne

4. On écrit le corps de la boucle externe, si besoin est. En général cette étape est courte.
5. On vérifie les niveaux d'indentation

3 Boucles conditionnelles

3.1 Problématisation et syntaxe

Il arrive parfois que l'on souhaite parcourir une boucle mais que l'on ne sache pas à l'avance combien de fois elle devra être parcourue. Supposons par exemple que l'on souhaite trouver le plus petit diviseur d'un nombre entier naturel donné. Dès lors que celui-ci est trouvé, on souhaite arrêter la boucle (et on a la certitude que ceci arrivera). Ainsi l'arrêt de la boucle se fera au moyen d'une condition.

Ce type de boucles est appelé boucle conditionnelle. Elle est introduite en Python par la syntaxe suivante :

while condition vérifiée :
bloc d'instructions

3.2 Choix entre boucle conditionnelle et boucle inconditionnelle

On préférera dans la mesure du possible utiliser des boucles inconditionnelles lorsqu'elles sont adaptées au problème que l'on résout. En effet, on sait à l'avance qu'une boucle for s'arrêtera (et on connaît souvent le nombre de fois où elle sera exécutée ; dès lors que l'itérable que l'on utilise est "range") ce qui n'est pas forcément le cas des boucles while et pose des problèmes dont nous allons discuter par la suite.

Ainsi, si l'on connaît à l'avance le nombre de répétitions que l'on doit effectuer, ou plus généralement si l'on doit parcourir une valeur itérable on choisira une boucle for. À l'inverse si la décision d'arrêter le parcours de la boucle ne peut s'exprimer qu'au moyen d'une condition on utilisera la boucle while

Exercice 5 : Pour chacun des problèmes suivants, choisir le type de boucles qui est adapté à l'écriture du programme :

1. Déterminer le maximum de deux nombres :
2. Déterminer l'ensemble des diviseurs d'un entier naturel :
3. Déterminer le rang du dernier terme strictement positif de la suite récurrente $u_{n+1} = 0.5u_n - n$ où u_0 est un réel positif donné :
4. Calculer la somme des inverses des n premiers entiers strictement positifs :
5. Déterminer la date du prochain vendredi 13 connaissant la date d'aujourd'hui :

3.3 Méthodologie d'écriture de boucles while

Pour écrire une boucle conditionnelle :

1. On cherche la condition pour laquelle la boucle doit être exécutée. Il est souvent plus simple d'exprimer la condition de sortie de boucle, et d'écrire sa négation ou utiliser l'opérateur booléen not
2. On écrit le corps de la boucle en s'assurant que celle-ci modifiera effectivement la valeur de la condition à certaines itérations.
3. On prévoit une initialisation des variables en amont de la boucle
4. On prévoit éventuellement un post-traitement des variables en sortie de boucle

Exercice 6 : En reprenant les étapes précédentes écrire un programme qui affiche le rang du dernier terme strictement positif de la suite récurrente $u_{n+1} = 0.5u_n - n$ où u_0 est un réel positif donné qui sera stocké dans une variable u :