Remarques sur la programmation en PYTHON

1) Remarques générales

a) **Affectations simultanées** : `x,y = y,x` (affectation de tuples).

Ne pas confondre avec : `x = y ; y = x` qui a pour effet d'attribuer à x et y l'ancienne valeur de y.

Remarque : En fait, PYTHON utilise une variable de stockage : `s = x ; x = y ; y = s`

Attention : L'affectation simultanée ne fonctionne pas avec des variables mutables :

```
M = numpy.array([[1,1],[2,2]]) ; M[0],M[1] = M[1],M[0]
print(M) # renvoie [[2,2],[2,2]]
```

Effectivement, `S = M[0] ; M[0] = M[1] ; M[1] = S` n'échange pas les deux lignes, car après l'instruction `S = M[0]`, la valeur de `S` est la variable `M[0]`(et non sa valeur), donc toute modification de `M[0]` modifie aussi `S`.

Pour permuter les deux lignes de la matrice, il convient par exemple d'écrire :

```
S = M[0].copy() ; M[0] = M[1] ; M[1] = S
```

b) **Evaluation paresseuse** (dans les expressions booléennes). considérons un tableau (liste) L de longueur n.

Ne pas confondre : `while i<n and L[i]==0` avec `while L[i]== 0 and i<n`

Dans le test de la première boucle, si $i \geq n$, l'expression `L[i]==0` n'est pas évaluée.

Dans le second cas, un dépassement de tableau crée une erreur lorsque l'expression `L[n]==0` est évaluée.

c) **Objets mutables**

Les listes (`list`) et les tableaux (`array`) sont des objets mutables (c'est-à-dire qu'ils peuvent être modifiés).

Les méthodes des objets-listes et les affectations de la forme `a[i] =` ... permettent d'effectuer ces transformations.

En revanche, les chaînes de caractères (`str`) ne sont pas mutables.

d) **Fonctions et procédures**

D'un point de vue théorique, on peut distinguer :

- les **fonctions** qui renvoient des objets calculés à partir des valeurs des arguments de la fonction.

- les **procédures** qui modifient les objets (mutables : listes ou tableaux) entrés en arguments. Les procédures sont de type `None` (aucune valeur n'est renvoyée), sauf des procédures-fonctions.

Exemple : `len` est une fonction, `append` est une procédure ; `pop` est une procédure fonction

Exemple : En PYTHON, on peut trier une liste à l'aide de la procédure `sort` : si `L` est une liste d'entiers ou de flottants, la méthode `L.sort()` modifie la liste `L` en la liste triée. En revanche, `sorted` est une fonction qui prend en argument une liste (ou un tableau) : `sorted(L)` renvoie la liste (ou la tableau) des éléments triés par ordre croissant.

Ainsi :

```
L = [5,1,3,2] ; L.sort()  # modifie L en [1,2,3,5]
```

`sorted([5,1,3,2])  #` renvoie la liste $[1, 2, 3, 5]$.

Remarque : Une fonction est dite à effets de bords (*side effect*) lorsqu'elle modifie l'état de l'environnement : c'est le cas des procédures, des fonctions utilisant des variables globales ou faisant intervenir des opérations d'entrées-sorties (écriture dans un fichier par exemple).

2) Conseils de programmation

a) *Remarque* : Une boucle `for` est un cas particulier de boucle `while` :

```
for i in range(a,b) :  instruction(i)
```

équivaut à

```
i = a
while i<b :  instruction(i) ; i = i+1
```

Conseil : Dans une boucle `for i` ..., éviter de modifier la variable i au sein de la boucle.

b) Utilisation de `return` comme un "*break*" au sein d'une boucle `for` dans une fonction :

Exemple : Recherche d'un flottant `x` dans une liste `a` :

```
def cherche(a :  list, x :  float) -> bool :
    for y in a :
        if x == y :  return True
    return False
```

L'autre méthode consiste à utiliser une boucle `while` :

```
def cherche(a,x) :
    n = len(a) ; i = 0
    while i<n and x != a[i] :  i = i+1
    return (i < n)
```

Bien que considérée comme de la programmation barbare, l'utilisation d'un `return` comme un "*break*" est souvent utile pour alléger les programmes.

c) Remarque sur la position des instructions dans une boucle `while`

Premier exemple :

```
x = a ; L = [] ; while test(x) :  L.append(x) ; x = F(x)
```

Second exemple :

```
x = a ; L = [] ; while test(x) :  x = F(x) ; L.append(x)
```

On suppose qu'il existe $p \in \mathbb{N}$ tel que $F^{(p)}(a)$ ne vérifie pas le test (sinon boucle infinie).

Alors, à la sortie de la boucle `while`, la liste L vaut $[a, F(a), F^{(2)}(a), ..., F^{(p-1)}(a)]$ dans le premier exemple et $[F(a), F^{(2)}(a), ..., F^{(p)}(a)]$ dans le second.

d) Remarque sur l'imbrication des boucles `for` ou `while`

On peut naturellement imbriquer deux boucles `for` portant sur des variables différentes :

```
for i in range(n) :
     for j in range(m(i)) :  instruction2(i,j)
```

Mais éviter en général d'utiliser deux boucles `while` imbriquées :

```
i = 0
while i < n :
     j = 0
     while j < p :
         if A[i,j] == 0 :  break
         j = j + 1
     i = i + 1
```

équivaut à

```
i = 0 ;  j = 0
while i < n and j < p :
     flag = (j<p)
     if flag and (A[i,j] == 0) :  break
     if flag :  j = j + 1
     else :  i = i + 1 ; j = 0
```

Remarque : Le drapeau `flag` permet de savoir si on se trouve dans la boucle en i ou en j.

Remarque : On calcule le plus petit $(i, j) \in [\![n]\!] \times [\![p]\!]$ tel que $A[i, j]$ est vrai, pour l'ordre lexicographique.

e) Remarque sur l'utilisation de l'itératif (`while`) et du récursif.

Important : Proscrire en général l'utilisation de boucles `while` dans le corps d'une fonction récursive.