Examen Programmation ENSAE première année 2008 (rattrapage)
Examen écrit (1 heure)

Tous documents autorisés.

0.0.1 Suite récurrente (Fibonacci) *

Enoncé

Réécrire la fonction `u` de façon à ce qu'elle ne soit plus récurrente. (4 points)

```
def u (n) :
    if n <= 2 : return 1
    else : return u (n-1) + u (n-2) + u (n-3)
```

Correction

```
def u_non_recursif (n) :
    if n <= 2 : return 1
    u0 = 1
    u1 = 1
    u2 = 1
    i = 3
    while i <= n :
        u  = u0 + u1 + u2
        u0 = u1
        u1 = u2
        u2 = u
        i += 1
    return u
```

fin exo 0.0.1 □

0.0.2 Calculer le résultat d'un programme *

Enoncé

1) Qu'affiche le programme suivant : (1 point)

```
def fonction (n) :
    return n + (n % 2)

print fonction (10)
print fonction (11)
```

2) Que fait la fonction `fonction` ? (1 point)

3) Ecrire une fonction qui retourne le premier multiple de 3 supérieur à `n` ? (2 points)

Correction

1) La fonction `fonction` ajoute 1 si n est impair, 0 sinon. Le programme affiche :

```
10
12
```

2) La fonction retourne le plus petit entier pair supérieur ou égal à n.

3) On cherche à retourner le plus petit multiple de 3 supérieur ou égal à n. Tout d'abord, on remarque que $n + (n\%3)$ n'est pas la réponse cherchée car $4 + 4\%3 = 5$ qui n'est pas divisible par 3. Les fonctions les plus simples sont les suivantes :

```
def fonction3 (n) :
    k = 0
    while k < n : k += 3
    return k
```

Ou encore :

```
def fonction3 (n) :
    if   n % 3 == 0 : return n
    elif n % 3 == 1 : return n + 2
    else : return n + 1
```

Ou encore :

```
def fonction3 (n) :
    if n % 3 == 0 : return n
    else : return n + 3 - (n % 3)
```

fin exo 0.0.2 □

0.0.3 Calculer le résultat d'un programme *

Enoncé

1) Qu'affiche le programme suivant : (1 point)

```
def division (n) :
    return n / 2

print division (1)
print division (0.9)
```

2) Proposez une solution pour que le résultat de la fonction soit 0.5 lors d'une instruction `print division`... ? (1 point)

Correction

1) 1/2 est égal à zéro en *Python* car c'est une division de deux entiers et le résultat est égal à la partie entière. La seconde division est une division entre un réel et un entier, le résultat est réel. Le programme affiche :

```
0
0.45
```

2) Voici deux solutions `print division(1.0)` ou `print division(float(1))`. Il est également possible de convertir `n` à l'intérieur de la fonction `division`. **fin exo 0.0.3** □

0.0.4 Ecrire un programme à partir d'un algorithme **

Enoncé

On considère deux listes d'entiers. La première est inférieure à la seconde si l'une des deux conditions suivantes est vérifiée :

- les n premiers nombres sont égaux mais la première liste ne contient que n éléments tandis que la seconde est plus longue
- les n premiers nombres sont égaux mais que le $n+1^{\text{ème}}$ de la première liste est inférieur au $n+1^{\text{ème}}$ de la seconde liste

Par conséquent, si l est la longueur de la liste la plus courte, comparer ces deux listes d'entiers revient à parcourir tous les indices depuis 0 jusqu'à l exclu et à s'arrêter sur la première différence qui détermine le résultat. S'il n'y pas de différence, alors la liste la plus courte est la première. Il faut écrire une fonction `compare_liste(p,q)` qui implémente cet algorithme.

(3 points)

Correction

Cet algorithme de comparaison est en fait celui utilisé pour comparer deux chaînes de caractères.

```
def compare_liste (p,q) :
    i = 0
    while i < len (p) and i < len (q) :
        if   p [i] < q [i] : return -1  # on peut décider
        elif p [i] > q [i] : return  1  # on peut décider
        i += 1                          # on ne peut pas décider
    # fin de la boucle, il faut décider à partir des longueurs des listes
    if   len (p) < len (q) : return -1
    elif len (p) > len (q) : return  1
    else : return 0
```

On pourrait également écrire cette fonction avec la fonction `cmp` qui permet de comparer deux éléments quels qu'ils soient.

```
def compare_liste (p,q) :
    i = 0
    while i < len (p) and i < len (q) :
        c = cmp (p [i], q [i])
        if c != 0 : return c            # on peut décider
        i += 1                          # on ne peut pas décider
    # fin de la boucle, il faut décider à partir des longueurs des listes
    return cmp (len (p), len (q))
```

fin exo 0.0.4 □

0.0.5 Comprendre une erreur d'exécution *

Enoncé

Le programme suivant est erroné.

```
l = [0,1,2,3,4,5,6,7,8,9]
i = 1
while i < len (l) :
    print l [i], l [i+1]
    i += 2
```

Il affiche des résultats puis une erreur.

```
1 2
3 4
5 6
7 8
9
Traceback (most recent call last):
  File "examen2008_rattrapage.py", line 43, in <module>
    print l [i], l [i+1]
IndexError: list index out of range
```

Pourquoi ? (1 point)

Correction

Le programme affiche les nombres par groupes de deux nombres consécutifs. Une itération de la boucle commence si la liste contient un élément suivant et non deux. Le programme est donc contraint à l'erreur car lors de la dernière itération, la liste contient un dixième nombre mais non un onzième. Le programme affiche le dixième élément (9) puis provoque une erreur `list index out of range`.

fin exo 0.0.5 □

0.0.6 Précision des calculs **

Enoncé

On cherche à calculer la somme des termes d'une suite géométriques de raison $\frac{1}{2}$. On définit $r = \frac{1}{2}$, on cherche donc à calculer $\sum_{i=0}^{\infty} r^i$ qui une somme convergente mais infinie. Le programme suivant permet d'en calculer une valeur approchée. Il retourne, outre le résultat, le nombre d'itérations qui ont permis d'estimer le résultat.

```
def suite_geometrique_1 (r) :
    x = 1.0
    y = 0.0
    n = 0
    while x > 0 :
        y += x
        x *= r
        n += 1
    return y,n

print suite_geometrique_1 (0.5)  #affiche (2.0, 1075)
```

Un informaticien plus expérimenté a écrit le programme suivant qui retourne le même résultat mais avec un nombre d'itérations beaucoup plus petit.

```
def suite_geometrique_2 (r) :
    x = 1.0
    y = 0.0
    n = 0
    yold = y + 1
    while abs (yold - y) > 0 :
        yold = y
        y += x
        x *= r
        n += 1
    return y,n

print suite_geometrique_2 (0.5)  #affiche (2.0, 55)
```

Expliquez pourquoi le second programme est plus rapide tout en retournant le même résultat. (2 points) Repère numérique : $2^{-55} \sim 2,8.10^{-17}$.

Correction

Tout d'abord le second programme est plus rapide car il effectue moins d'itérations, 55 au lieu de 1075. Maintenant, il s'agit de savoir pourquoi le second programme retourne le même résultat que le premier mais plus rapidement. L'ordinateur ne peut pas calculer numériquement une somme infinie, il s'agit toujours d'une valeur approchée. L'approximation dépend de la précision des calculs, environ 14 chiffres pour *Python*. Dans le premier programme, on s'arrête lorsque r^n devient nul, autrement dit, on s'arrête lorsque x est si petit que *Python* ne peut plus le représenter autrement que par 0, c'est-à-dire qu'il n'est pas possible de représenter un nombre dans l'intervalle $[0, 2^{-1055}]$.

Toutefois, il n'est pas indispensable d'aller aussi loin car l'ordinateur n'est de toute façon pas capable d'ajouter un nombre aussi petit à un nombre plus grand que 1. Par exemple, $1 + 10^{17} = 1,000\,000\,000\,000\,000\,01$. Comme la précision des calculs n'est que de 15 chiffres, pour *Python*, $1 + 10^{17} = 1$. Le second programme s'inspire de cette remarque : le calcul s'arrête lorsque le résultat de la somme n'évolue plus car il additionne des nombres trop petits à un nombre trop grand. L'idée est donc de comparer la somme d'une itération à l'autre et de s'arrêter lorsqu'elle n'évolue plus.

Ce raisonnement n'est pas toujours applicable. Il est valide dans ce cas car la série $s_n = \sum_{i=0}^{n} r^i$ est croissante et positive. Il est valide pour une série convergente de la forme $s_n = \sum_{i=0}^{n} u_i$ et une suite u_n de module décroissant.

fin exo 0.0.6 □

0.0.7 Analyser un programme **

Enoncé

Un chercheur cherche à vérifier qu'une suite de 0 et de 1 est aléatoire. Pour cela, il souhaite compter le nombre de séquences de n nombres successifs. Par exemple, pour la suite 01100111 et $n = 3$, les triplets sont 011, 110, 100, 001, 011, 111. Le triplet 011 apparaît deux fois, les autres une fois. Si la suite est aléatoire, les occurrences de chaque triplet sont en nombres équivalents.

Le chercheur souhaite également faire varier n et calculer les fréquences des triplets, quadruplets, quintuplets, ... Pour compter ses occurrences, il hésite entre deux structures, la première à base de listes (à déconseiller) :

```
def hyper_cube_liste (n, m = [0,0]) :
    if n > 1 :
        m [0] = [0,0]
        m [1] = [0,0]
        m [0] = hyper_cube_liste (n-1, m [0])
        m [1] = hyper_cube_liste (n-1, m [1])
    return m
h = hyper_cube_liste (3)
print h                     # affiche [[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
```

La seconde à base de dictionnaire (plus facile à manipuler) :

```
def hyper_cube_dico (n) :
    r = { }
    ind = [ 0 for i in range (0,n) ]
    while ind [0] <= 1 :
        cle                 = tuple (  ind ) # conversion d'une liste en tuple
        r [cle]             = 0
        ind [ len (ind)-1] += 1
        k                   = len (ind)-1
        while ind [ k ] == 2 and k > 0 :
            ind [k]    = 0
            ind [k-1] += 1
```

```
            k         -= 1
    return r
h = hyper_cube_dico (3)
print h        # affiche {(0, 1, 1): 0, (1, 1, 0): 0, (1, 0, 0): 0, (0, 0, 1): 0,
               #          (1, 0, 1): 0, (0, 0, 0): 0, (0, 1, 0): 0, (1, 1, 1): 0}
```

Le chercheur a commencé à écrire son programme :

```
def occurrence (l,n) :
    d = ....... # choix d'un hyper_cube (n)
    .....
    return d
suite = [ 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1 ]
h = occurrence (suite, 3)
print h
```

Sur quelle structure se porte votre choix (a priori celle avec dictionnaire), compléter la fonction `occurrence`. (4 points)

Correction

Tout d'abord, la structure matricielle est à déconseiller fortement même si un exemple d'utilisation en sera donné. La solution avec dictionnaire est assez simple :

```
def occurrence (l,n) :
    d = hyper_cube_dico (n)
    for i in range (0, len (l)-n) :
        cle = tuple (l [i:i+n])
        d [cle] += 1
    return d
```

Il est même possible de se passer de la fonction `hyper_cube_dico` :

```
def occurrence (l,n) :
    d = { }
    for i in range (0, len (l)-n) :
        cle = tuple (l [i:i+n])
        if cle not in d : d [cle] = 0
        d [cle] += 1
    return d
```

La seule différence apparaît lorsqu'un n-uplet n'apparaît pas dans la liste. Avec la fonction `hyper_cube_dico`, ce n-uplet recevra la fréquence 0, sans cette fonction, ce n-uplet ne sera pas présent dans le dictionnaire `d`. Le même programme avec la structure matricielle est plus une curiosité qu'un cas utile.

```
def occurrence (l,n) :
    d = hyper_cube_liste (n, [0,0])         # * remarque voir plus bas
    for i in range (0, len (l)-n) :
        cle = l [i:i+n]
        t   = d                         #
        for k in range (0,n-1) :        # point clé de la fonction :
            t = t [ cle [k] ]           #                 accès à un élément
        t [cle [ n-1] ] += 1
    return d
```

Si on remplace la ligne marquée d'une étoile par `d = hyper_cube_list(n)`, le programme retourne une erreur :

```
Traceback (most recent call last):
  File "examen2008_rattrapage.py", line 166, in <module>
    h = occurrence (suite, n)
  File "examen2008_rattrapage.py", line 160, in occurrence
    t [cle [ n-1] ] += 1
TypeError: 'int' object is not iterable
```

Cette erreur est assez incompréhensible puisque la modification a consisté à appeler une fonction avec un paramètre de moins qui était de toutes façons égal à la valeur par défaut associée au paramètre. Pour comprendre cette erreur, il faut exécuter le programme suivant :

```
def fonction (l = [0,0]) :
    l [0] += 1
    return l

print fonction ()         # affiche [1,0] : résultat attendu
print fonction ()         # affiche [2,0] : résultat surprenant
print fonction ( [0,0])   # affiche [1,0] : résultat attendu
```

L'explication provient du fait que la valeur par défaut est une liste qui n'est pas recréée à chaque appel mais c'est la même liste à chaque fois que la fonction est appelée sans paramètre. Pour remédier à cela, il faudrait écrire :

```
import copy
def fonction (l = [0,0]) :
    l = copy.copy (l)
    l [0] += 1
    return l
```

Dans le cas de l'hypercube, il faudrait écrire :

```
def hyper_cube_liste (n, m = [0,0]) :
    m = copy.copy (m)
    if n > 1 :
        m [0] = [0,0]
        m [1] = [0,0]
        m [0] = hyper_cube_liste (n-1, m [0])
        m [1] = hyper_cube_liste (n-1, m [1])
    return m
```

fin exo 0.0.7 □

Index

Table des matières