Examen Programmation ENSAE première année 2007
Examen écrit (1 heure)

Lors de la correction, je n'ai pas enlevé de points pour les erreurs de syntaxe et accordé les points de la question à partir du moment où l'idée principale de l'algorithme était présente même si le programme ne retournait pas exactement le résultat. Devant un ordinateur, après quelques essais, vous auriez, à partir de votre première réponse, rapidement obtenu un programme correct.

0.0.1 Logarithme en base deux **

Enoncé

On veut écrire une fonction qui retourne n tel que n soit le premier entier qui vérifie $2^n \geqslant k$. Cette fonction prend comme paramètre d'entrée k et retourne également un entier. Elle ne devra utiliser ni logarithme ni exponentielle. On précise qu'en langage *Python*, `2**n` signifie 2 à la puissance n. Une boucle `for` ne semble pas indiquée dans ce cas. (3 points)

Correction

L'indication qui préconisait d'utiliser autre chose qu'une boucle `for` ne voulait pas dire ne pas utiliser de boucle du tout mais une boucle `while`. La majorité des élèves ont réussi à trouver la fonction suivante :

```
def fonction_log2 (k) :
    n = 0
    while 2**n < k :
        n += 1
    return n
```

Même s'il est possible d'utiliser malgré tout une boucle `for` :

```
def fonction_log2 (k) :
    for i in range (0,1000) :
        if 2**i >= k :
            return i
```

Voici un exemple de fonction récursive :

```
def fonction_log2 (k) :
    if k <= 1 : return 0
    else : return fonction_log2 ((k+1)/2)+1
```

fin exo 0.0.1 □

0.0.2 Calculer le résultat d'un programme **

Enoncé

On définit la fonction suivante :

```
def parcours (n) :
    i = 1
    j = 1
    while i+j < n :
        print (i,j)
        i += 1
        j -= 1
        if j < 1 :
            j = i+j
            i = 1
```

Quelles sont les 6 lignes qu'affiche cette fonction si `n = 5` ? (2 points)

Correction

La solution est :

```
(1,1)
(1,2)
(2,1)
(1,3)
(2,2)
(3,1)
```

Et si on continue :

```
(1,4)
(2,3)
(3,2)
(4,1)
(1,5)
(2,4)
(3,3)
...
```

Il fallait voir deux choses importantes dans l'énoncé de l'exercice, tout d'abord, les deux lignes suivantes :

```
        i += 1
        j -= 1
```

Elles signifient que lorsque i augmente de 1, j diminue de 1. L'autre information est que j ne devient jamais inférieur à 1 :

```
        if j < 1 :
            j = i+j
            i = 1
```

Et lorsque ce cas arrive, i devient égal à 1, j devient égale à $i+j$, soit $i+1$ puisque la condition est vérifiée lorsque $j == 1$. Ce programme propose une façon de parcourir l'ensemble des nombres rationnels de la forme $\frac{i}{j}$.

fin exo 0.0.2 □

0.0.3 Calculer le résultat d'un programme **

Enoncé

Que vaut `n` à la fin du programme suivant ? Il n'est en principe pas nécessaire d'aller jusqu'à `i = 10`. (2 points)

```
def suite (n) :
    n = n ** 2 / 2 + 1
    return n

n = 3
for i in range (0,10) :
    n = suite (n) % 17
print n
```

Correction

Il est difficile d'arriver au bout des calculs correctement lors de cet exercice. La première chose à faire est d'identifier la relation entre n et $n+1$. On rappelle que le symbole % désigne le reste d'une division entière et que / désigne une division entière car tous les nombres manipulés dans ce programme sont entiers. La fonction `suite` aboutit à $f(n) = E\left[\frac{n^2}{2}\right] + 1$ où $E[x]$ désigne la partie entière de x. Ajouté aux trois dernières lignes, on obtient :

$$u_{n+1} = f(u_n)\%17 = \left(E\left[\frac{n^2}{2}\right] + 1\right)\%17$$

Même si i n'est pas utilisé dans la boucle `for`, celle-ci s'exécute quand même 10 fois. Pour trouver la valeur finale de n, on calcule donc les premiers termes comme suit :

i	0	1	2	3	4
$u_i = n$	3	5	13	0	1
n^2	9	25	169	0	1
$y = E\left[\frac{n^2}{2}\right] + 1$	4	12	85	**1**	1
$u_{i+1} = y\%17$	5	13	0	1	1

A partir de $i = 3$, la suite reste constante et égale à 1. Il n'est pas nécessaire d'aller plus loin.

fin exo 0.0.3 □

0.0.4 Comprendre une erreur d'exécution *

Enoncé

Le programme suivant est incorrect.

```
def compte_lettre (s) :
    nombre = {}
    for c in s :
        nombre [c] += 1
    return nombre

print compte_lettre ( "mysteres" )
```

Il retourne l'erreur suivante :

```
KeyError: 'm'
```

1) Quelle est la cause de l'erreur ? (1 point)

2) Quelle est la correction à apporter pour que cette fonction compte les lettres d'un mot ? (1 point)

3) Qu'afficherait le programme une fois corrigé ? (1 point)

4) La fonction `compte_lettre` est-elle réservée aux chaînes de caractères ? Voyez-vous d'autres types de données auxquels elle pourrait s'appliquer ? (1 point)

Correction

1) L'erreur provient de la ligne `nombre [c] += 1` pour `c` $=$ `'m'`, cette ligne équivaut à `nombre[c] = nombre[c] + 1`. Cela signifie que `nombre[c]` doit exister avant l'exécution de la ligne, c'est-à-dire que le dictionnaire `nombre` doit avoir une valeur associée à la clé `c` $=$ `'m'`, ce qui n'est pas le cas ici.

Dans ce programme, `nombre` est un dictionnaire et non une liste (ou un tableau) et les dictionnaires acceptent des indices autres que des entiers, ils acceptent comme indice ou clé toute variable de type immuable : les nombres (entiers, réels), les caractères, les t-uples.

2) 3) La solution compte le nombre d'occurrences de chaque lettre dans le mot `s`.

```
def compteur (s) :
    nombre = {}
    for c in s : nombre [c] = 0        # ligne ajoutée
    for c in s :
        nombre [c] += 1
    return nombre
```

Ou encore :

```
def compteur (s) :
    nombre = {}
    for c in s :
        if c not in nombre : nombre [c] = 0       # ligne ajoutée
        nombre [c] += 1
    return nombre
```

Ces deux programmes retournent :

```
{'e': 2, 'm': 1, 's': 2, 'r': 1, 't': 1, 'y': 1}
```

La dernière solution pour ceux qui n'aiment pas les dictionnaires :

```
def compteur (s) :
    alpha = "abcdefghijklmnopqrstuvwxyz"
    nombre = [ 0 for c in alpha ]
    for c in s :
        i = alpha.index (c)
        nombre [i] += 1
    return nombre
```

Le programme retourne :

```
[0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 2, 1, 0, 0, 0, 0, 1, 0]
```

4) La fonction `compteur` accepte d'autres types de données à condition que les deux lignes `for c in s:` et `nombre[c]+=1` aient un sens et elles ne sont valables que si le paramètre `s` contient des éléments susceptibles de servir de clé pour le dictionnaire `s`, c'est-à-dire un autre dictionnaire, un tuple, une liste d'éléments de type immuable. Autrement dit, les lignes suivantes sont correctes :

```
print compteur ( "mysteres" )
print compteur ( compteur ("mysteres") )
print compteur ( [0,1,1,4,-1, (6,0), 5.5, "ch"] )
print compteur ( { 1:1, 2:2, 1:[] } )
```

Mais pas celle-ci :

```
print compteur ( [0, [0,0] ] )
```

Toutefois, cette dernière ligne est valide si la fonction `compteur` se contente seulement de compter le nombre d'éléments, c'est-à-dire la première solution citée aux questions 2) et 3).

fin exo 0.0.4 □

0.0.5 Comprendre une erreur de logique ***

Enoncé

On précise que l'instruction `random.randint(0,1)` retourne un nombre aléatoire choisi dans l'ensemble $\{0,1\}$ avec des probabilités équivalentes ($\mathbb{P}(X=0)=\frac{1}{2}$ et $\mathbb{P}(X=1)=\frac{1}{2}$). La fonction `ligne_nulle` doit compter le nombre de lignes nulles de la matrice `mat` donnée comme paramètre.

```
def ligne_nulle (mat) :
    nb = 0
    for i in range (0, len (mat)) :
        lig = 0
        for j in range (0, len (mat [i])) :
            if mat [i][j] > 0 : lig += 1
            if lig == 0 : nb += 1
    return nb

matri = [ [ random.randint (0,1) for i in range (0,4) ] for j in range (0,20) ]
print ligne_nulle (matri)
```

Après avoir exécuté le programme trois fois de suite, les résultats affichés sont successivement `15`, `19`, `17`. Bien que l'exécution du programme ne provoque aucune erreur, le concepteur de la fonction s'interroge quand même sur ces résultats.

1) Sachant que chaque case de la matrice reçoit aléatoirement 0 ou 1, quelle est la probabilité qu'une ligne soit nulle ? Quelle est la probabilité d'avoir 15 lignes nulles dans la matrice sachant que cette matrice a 20 lignes ? Ces deux réponses peuvent être littérales. Donnez-vous raison à celui qui a écrit le programme (il pense s'être trompé) ? (1 point)

2) Si vous lui donnez raison, ce qui est fort probable, où serait son erreur ? (1 point)

Correction

1) La probabilité d'avoir une ligne nulle est la probabilité d'avoir 4 zéros, c'est donc :

$$\mathbb{P}(\textit{ligne nulle}) = \left(\frac{1}{2}\right)^4 = \frac{1}{16}$$

Comment calculer la probabilité d'avoir 15 lignes nulles parmi 20 ? Cela revient à estimer la probabilité de tirer 15 fois sur 20 une boule blanche lors d'un tirage à remise sachant que dans l'urne, il y a 1 boule blanche et 15 boules noires. Le nombre de boules blanches tirées suit une loi binomiale de paramètre $p=\frac{1}{16}$. On en déduit que :

$$\mathbb{P}(15 \textit{ lignes nulles sur } 20) = C_{20}^{15}\left(\frac{1}{16}\right)^{15}\left(1-\frac{1}{16}\right)^{5} \leqslant \frac{2^{20}}{10^{15}}$$

Cette probabilité est très faible, il est donc presque impossible d'obtenir trois fois de suite un nombre de lignes supérieur à 15. Le programme est sans aucun doute faux.

2) La construction de la matrice est manifestement correcte, c'est donc le comptage des lignes nulles qui est faux. Cette erreur intervient lors de la ligne `if lig == 0: nb += 1`. Celle-ci est incluse dans la seconde boucle `for` ce qui a pour effet d'incrémenter `lig` dès qu'un premier zéro est rencontré sur une ligne. Au final, la fonction retourne le nombre de lignes contenant au moins un zéro. Voici donc la correction à apporter :

```
def ligne_nulle (mat) :
    nb = 0
    for i in range (0, len (mat)) :
        lig = 0
        for j in range (0, len (mat [i])) :
            if mat [i][j] > 0 : lig += 1
        if lig == 0 : nb += 1                    # ligne décalée vers la gauche
    return nb
```

fin exo 0.0.5 □

0.0.6 Récursivité *

Enoncé

Le programme suivant provoque une erreur dont le message paraît sans fin.

```
class erreur :
    def __init__ (self) :
        self.e = erreur ()
e = erreur ()
```

Auriez-vous une explication pour ce qui suit ? (1 point)

```
Traceback (most recent call last):
  File "examen2007.py", line 4, in ?
    e = erreur ()
  File "examen2007.py", line 2, in __init__
    self.e = erreur ()
  File "examen2007.py", line 2, in __init__
    self.e = erreur ()
  File "examen2007.py", line 2, in __init__
    self.e = erreur ()
  ...
```

Correction

L'erreur retournée est une erreur d'exécution et non une erreur de syntaxe. Cela veut dire que le programme de l'exercice est syntaxiquement correct. Il n'est donc pas possible de dire que l'attribut `e` n'est pas défini et de corriger le programme comme suit pour expliquer l'erreur :

```
class erreur :
    def __init__ (self,e) :
        self.e = e
```

En fait, la classe `erreur` définit un attribut qui est également de type `erreur`. Cet attribut va lui aussi définir un attribut de type `erreur`. Ce schéma va se reproduire à l'infini puisqu'à aucun moment, le code du programme ne prévoit la possibilité de s'arrêter. Le programme crée donc des instances de la classe `erreur` à l'infini jusqu'à atteindre une certaine limite dépendant du langage *Python*. C'est à ce moment-là que se produit l'erreur citée dans l'énoncé.

fin exo 0.0.6 □

0.0.7 Compléter un programme **

Enoncé

11 (base décimale) s'écrit 102 en base 3 puisque $11 = 1 * 3^2 + 0 * 3^1 + 2 * 3^0$. L'objectif de cet exercice est d'écrire une fonction qui écrit un nombre entier en base 3. Cette fonction prend comme paramètre d'entrée un entier et retourne une chaîne de caractères. On précise que l'opérateur % calcule le reste d'une division entière et que `11/3` → `3` car c'est une division entière dont le résultat est un entier égal au quotient de la division. La fonction `str` permet de convertir un nombre en une chaîne de caractères. Il ne reste plus qu'à compléter les deux lignes manquantes du programme suivant : (2 points)

```
def base3 (n) :
    s = ""
    while n > 0 :
        r = n % 3
        # ....... à compléter
        # ....... à compléter
    return s
```

Correction

Il y a effectivement deux lignes à corriger. `r` désigne le reste de la division de `n` par 3. Il faut le convertir en chaîne de caractères et l'ajouter à gauche à la chaîne `s`. On passe au chiffre suivant en division `n` par 3 ce qui est rassurant puisque `n` va tendre vers zéro et le programme s'arrêter nécessairement au bout d'un moment. Pour vérifier que ce programme est correct, il suffit de l'appliquer à un nombre, voire appliquer le même algorithme mais en base 10 pour être vraiment sûr.

```
def base3 (n) :
    s = ""
    while n > 0 :
        r = n % 3
        s = str (r) + s
        n = n / 3            # équivalent à n = (n-r) / 3
                             # puisque / est une division entière
    return s
```

fin exo 0.0.7 □

0.0.8 Comprendre une erreur de logique ***

Enoncé

Un professeur désireux de tester une répartition aléatoire des notes à un examen décide de tirer au hasard les notes de ses élèves selon une loi normale de moyenne 15 et d'écart-type 3. Il arrondit ses notes à l'entier le plus proche en n'omettant pas de vérifier que ses notes sont bien dans l'intervalle $[0, 20]$. On précise que l'instruction `float(i)` convertit un nombre `i` en nombre réel, cette conversion est utilisée pour être sûr que le résultat final sera bien réel et non le résultat d'opérations sur des entiers. Cette conversion intervient le plus souvent lors de divisions.

```
import copy
import math
import random

class Eleve :
    def __init__ (self, note) :
        self.note = note

e = Eleve (0)
l = []
for i in range (0,81) :
    e.note = int (random.gauss (15, 3) + 0.5)  # tirage aélatoire et arrondi
    if e.note >= 20 : e.note = 20              # pas de note au-dessus de 20
    if e.note < 0 : e.note = 0                 # pas de note négative
    l.append (e)

moy = 0
var = 0

for e in l :
    moy += e.note
moy = float (moy) / len (l)  # les notes sont entières,
                             # il faut convertir avant de diviser
                             # pour obtenir la moyenne
for e in l :
```

```
    var += (e.note - moy) ** 2
var = math.sqrt ( float (var) ) / len (l)

print "moyenne ", moy
print "écart-type ", var
```

Il songe à vérifier néanmoins que la moyenne de ses notes arrondies est bien conforme à ce qu'il a échafaudé.

```
moyenne  16.0
écart-type  0.0
```

La moyenne égale à 16 ne le perturbe guère, il se dit que l'arrondi a été plutôt généreux. Toutefois l'écart-type nul le laisse perplexe.

1) Que signifie un écart-type nul ? Quelle est l'erreur du professeur ? (Elle est située à l'intérieur de la boucle `for i in range(0,81)` :, ce n'est pas une erreur lors du calcul de l'écart-type ni une erreur de définition de la classe `Eleve`.) (1 point)

2) Proposer deux solutions pour corriger ce problème, chacune d'elles revient à remplacer la ligne `l.append(e)`. (2 points)

3) On sait que la variance d'une variable aléatoire X vérifie : $\mathbb{V}(X) = \mathbb{E}\left(X^2\right) - [\mathbb{E}(X)]^2$. Cette astuce mathématique permet-elle de réduire le nombre de boucles du programme, si oui, comment ? (1 point)

Correction

1) Un écart-type nul signifie que toutes les notes sont identiques et égales à la moyenne. Selon le programme, tous les élèves ont donc 16. L'erreur provient du fait que l'instruction `l.append(e)` ajoute à chaque fois la même variable de type `Eleve`. A la fin de la boucle, la liste `l` contient 81 fois le même objet `Eleve` ou plus exactement 81 fois la même instance de la classe `Eleve`. On modifie la note de cet unique objet en écrivant : `e.note = int(random.gauss(15,3) + 0.5)`. 16 est donc la note attribuée au dernier élève, la dernière note tirée aléatoirement.

2) Les deux corrections possibles consistent à créer à chaque itération une nouvelle instance de la classe `Eleve`.

1. `l.append(e)` devient `l.append(Eleve(e.note))`
2. `l.append(e)` devient `l.append(copy.copy(e))`

La variable `e` dans la boucle est un `Eleve` temporaire, il est ensuite recréé ou recopié avant l'ajout dans la liste. Ceci veut dire que l'instance ajoutée dans la liste n'est pas la même que celle utilisée dans la boucle.

Une troisième solution est envisageable même si elle introduit des modifications dans la suite du programme, elle est logiquement correcte : `l.append(e)` devient `l.append(e.note)`. La liste `l` n'est plus une liste de `Eleve` mais une liste d'entiers. Le programme devient :

```
#...
e = Eleve (0)
l = []
for i in range (0,81) :
    e.note = int (random.gauss (15, 3) + 0.5)
    if e.note >= 20 : e.note = 20
    if e.note < 0 : e.note = 0
    l.append (e.note)                           # ligne modifiée

moy = 0
var = 0

for note in l :                                 # ligne modifiée
    moy += note                                 # ligne modifiée
moy = float (moy) / len (l)
for note in l :                                 # ligne modifiée
    var += (note - moy) ** 2                    # ligne modifiée
var = math.sqrt ( float (var) ) / len (l)
```

```
print "moyenne ", moy
print "écart-type ", var
```

3) La formule mathématique permet de réduire le nombre de boucles à deux. Lors de la version initiale du programme, la première sert à créer la liste `l`, la seconde à calculer la moyenne, la troisième à calculer la variance. On regroupe les deux dernières boucles en une seule.

```
moy = 0
var = 0

for note in l :
    moy += note
    var += note * note
moy = float (moy) / len (l)
var = float (var) / len (l)
var = var - moy * moy
var = math.sqrt ( float (var) )

print "moyenne ", moy
print "écart-type ", var
```

fin exo 0.0.8 □

Index

Table des matières