Informatique de Tronc Commun

Devoir surveillé n°2

La calculatrice n'est pas autorisée

durée : 2 h

Toutes les réponses sont à donner sur la copie.

Exercice 1. Boucles `while`

Dans cet exercice, toutes les boucles doivent être des boucles `while`, l'utilisation de boucles `for` est interdite.

1. Écrire une fonction `cubes1` qui prend pour argument un entier positif et qui affiche les cubes d'entiers positifs qui sont inférieurs ou égaux à cet entier.

Exemples d'appel à la fonction :

```
>>> cubes1(50)
0
1
8
27
```

```
>>> cubes1(64)
0
1
8
27
64
```

2. Modifier la fonction `cubes1` en une fonction `cubes2` qui prend pour argument un entier positif et qui renvoie la liste des cubes d'entiers positifs qui sont inférieurs ou égaux à cet entier.

Exemples d'appel à la fonction :

```
>>> cubes2(50)
[0, 1, 8, 27]
```

```
>>> cubes2(64)
[0, 1, 8, 27, 64]
```

Pour les questions 3. à 6., il est imposé de **ne pas utiliser** de fonction de conversion de type (`int`, *etc*.), ni la division entière (`//`) ou flottante (`/`), ni les fonctions `floor` ou `ceil` du module `math`.

3. Écrire une fonction `e1` qui prend pour argument un flottant `x` positif et qui renvoie la liste des entiers positifs inférieurs ou égaux à `x`.

Exemples d'appel à la fonction :

```
>>> e1(7.22)
[0, 1, 2, 3, 4, 5, 6, 7]
```

```
>>> e1(2.0)
[0, 1, 2]
```

4. Modifier la fonction `e1` en une fonction `e2` qui prend pour argument un flottant `x` positif et qui renvoie le plus grand entier inférieur ou égal à `x`.

Exemples d'appel à la fonction :

```
>>> e2(7.22)
7
```

```
>>> e2(2.0)
2
```

5. Modifier la fonction `e2` en une fonction `e3` qui prend pour argument un flottant `x`, **de signe quelconque**, et qui renvoie le plus grand entier inférieur ou égal à `x` [réalisant ainsi ce que réalise la fonction `floor` de Python].

Exemples d'appel à la fonction :

```
>>> e3(7.22)
7
```

```
>>> e3(-7.22)
-8
```

```
>>> e3(5)
5
```

```
>>> e3(-5)
-5
```

6. Modifier la fonction `e3` en une fonction `e4` qui prend pour argument un flottant `x`, **de signe quelconque**, et qui renvoie le plus petit entier **supérieur ou égal** à `x` [fonction `ceil` de Python].

Exemples d'appel à la fonction :

```
>>> e4(7.22)
8
```

```
>>> e4(-7.22)
-7
```

```
>>> e4(5)
5
```

```
>>> e4(-5)
-5
```

Exercice 2. Recherche dans une liste

1. Écrire une fonction, `rech`, prenant en argument une liste L et une valeur x, et renvoyant un booléen, `True` ou `False`, selon que x appartient à L ou non.

- Exemples d'appel à la fonction :

```
>>> rech([5, 6, 7], 7)
True
```

```
>>> rech(['f', 'ba', '', 'f'], 'a')
False
```

2. L'implémentation donnée en réponse à la question **1.** utilise-t-elle un parcours par les indices de L ou un parcours par les valeurs ?

Donner une seconde version, `rech2` de la fonction `rech`, utilisant l'autre de ces deux parcours de liste.

3. Si la liste L comporte n éléments, combien de comparaisons (avec l'opérateur « == ») sont-elles effectuées, au minimum ? au maximum ?

Donner un exemple de liste de longueur $n = 4$ pour laquelle le nombre de comparaisons est minimal et un exemple de liste de longueur $n = 4$ pour laquelle le nombre de comparaisons est maximal.

Exercice 3. Fonctions mystères opérant sur des listes

1. On considère les fonctions `mystere1` et `mystere2` suivantes pour des listes d'entiers de longueur au moins 2 :

```
1. def mystere1(L):
2.     for i in range(len(L) - 1):
3.         if L[i] == L[i + 1]:
4.             return False
5.         return True
```

```
1. def mystere2(L):
2.     for i in range(len(L) - 1):
3.         if L[i] == L[i + 1]:
4.             return False
5.     return True
```

a. Que renvoie les appels `mystere1([4, 5, 6, 3, 7, 6])` ? `mystere1([4, 5, 6, 6, 3, 7])` ?

b. Que renvoie les appels `mystere2([4, 5, 6, 3, 7, 6])` ? `mystere2([4, 5, 6, 6, 3, 7])` ?

c. Indiquer ce que fait chacune de ces deux fonctions. (Il s'agira de préciser ce que renvoie chacune des deux fonctions en fonction des caractéristiques de la liste en entrée et non de paraphraser le code de la fonction).

2. On considère la fonction `mystere3` suivante, prenant en argument des listes d'entiers :

```
def mystere3 (L):
    A = []
    for i in range(1, len(L)):
        if L[i - 1] <= L[i] and L[i] >= L[i + 1]:
            A.append(L[i])
    return A
```

a. L'appel `mystere3([1, 2])` déclenche une erreur. Laquelle et pourquoi ?

b. Caractériser les listes en entrée qui déclenchent la même erreur qu'en **a.** et celles qui ne la déclenchent pas.

c. Proposer une modification minimale du code de la fonction `mystere3`, de sorte qu'aucun appel sur des listes d'entiers ne déclenche l'erreur précédente.

Dans les deux questions suivantes, on suppose que l'erreur précédente a été corrigée :

d. Que renvoie les appels `mystere3([4, 5, 6, 3, 7])`? `mystere3([4, 5, 6, 3, 7, 6])`?

e. Que renvoie les appels `mystere3([])`? `mystere3([8, 4, 5, 6, 1, 3, 3, 7, 6, 3, 4, 2])` ?

f. Que fait la fonction `mystere3` ? (on caractérisera la liste en sortie en fonction de la liste en entrée).

3. On considère la fonction `mystere4` suivante, prenant en argument des listes d'entiers :

```
def mystere4(L):
    A = []
    i = 0
    while i < len(L):
        A.append(L[i])
        i = i + 2
    return A
```

a. Que renvoient les appels

- `mystere4([5, 8, 2, 3])` et
- `mystere4([5, 8, 2, 3, 4])` ?

b. De façon générale, que renvoie la fonction `mystere4` ?

Exercice 4. Suppression des doublons dans une liste

On rappelle que l'opérateur `in`, permet de réaliser sur une liste L, des tests `x in L`, évalués à `True` si $x \in L$, et à `False` sinon. On rappelle également l'opérateur `not` qui renvoie le complémentaire d'un booléen.

- par exemple :

```
>>> 2 in []
False
```

```
>>> 2 in [3, 2 5]
True
```

```
>>> not 2 in [4, 3, 5]
True
```

On rappelle que l'instruction `del L[k]`, pour $0 \leq k < \text{len(L)}$, supprime la valeur `L[k]` dans une liste `L`.
Par exemple, si `L = [4, 8, 2]`, après exécution de l'instruction `del L[0]`, la liste `L` est la liste `[8, 2]`.

On suppose ici que toutes les listes considérées sont des listes non vides d'entiers.

1. Écrire une fonction `estcroissante`, de paramètre `L`, renvoyant un booléen, `True` ou `False`, selon que la liste `L` en entrée est croissante ou non.
2. On rappelle la fonction suivante renvoyant une copie d'une liste L :

```
def copie(L):
    A = []
    for i in range(len(L)):
        A.append(L[i])
    return A
```

   a. Pour cette question, **on suppose que la liste L en entrée de la fonction est triée dans l'ordre croissant**. Écrire une fonction `sans_doublons1(L)` construisant et renvoyant une copie de la liste `L` en en supprimant les doublons. Pour une liste L de longueur n, la fonction ne devra pas effectuer plus de n comparaisons.

Exemples d'appel à la fonction :

```
>>> sans_doublons1([5, 8, 8, 10, 11])
[5, 8, 10, 11]
```

```
>>> sans_doublons1([5])
[5]
```

   b. Modifier la fonction précédente en une fonction `sans_doublons2` de sorte qu'elle réalise le même cahier des charges, mais sans qu'il soit nécessaire de supposer que la liste en entrée est triée.
   c. Écrire une fonction `sans_doublons3` qui prend en argument une liste L, non nécessairement triée, et **qui modifie la liste L** de sorte à en supprimer les doublons.
   On devra utiliser l'instruction `del`, une boucle `while` et le test `in`.
   d. Modifier la fonction `sans_doublons3` en une fonction `sans_doublons4` en remplaçant le test `in` par une boucle.
   Combien de comparaisons effectue la fonction `sans_doublons4` pour une liste en entrée dont toutes les valeurs sont égales ? pour une liste en entrée dont toutes les valeurs sont distinctes ?

Exercice 5. Chaînes de caractères

On rappelle que la fonction suivante permet de construire et de renvoyer une copie d'une chaîne de caractères, `ch`.

```
def copie(ch):
    ch1 = ""
    for i in range(len(ch)):
        ch1 = ch1 + ch[i]
    return ch1
```

1. Écrire une fonction `remplace(ch, old, new)` renvoyant une copie d'une chaîne `ch` en entrée dans laquelle les occurrences du caractère `old` ont été remplacée par le caractère `new`. Par exemple, l'appel `remplace("toto", "o", "i")` renverra la chaîne `"titi"`.
2. Écrire une fonction `renverse(ch)` renvoyant une copie d'une chaîne `ch` dans laquelle l'ordre des caractères est inversé. Par exemple, l'appel `renverse("toto")` renverra la chaîne `"otot"`.
3. Écrire une fonction `teste_palindrome(ch)` renvoyant un booléen, `True` ou `False`, selon que la chaîne en entrée est un palindrome ou non. La fonction fera appel à la fonction `renverse`.
   Par exemple, l'appel `teste_palindrome("toto")` renverra `False`, et `teste_palindrome("kayak")` renverra `True`.

Exercice 6. Occurrences d'un motif dans une chaîne de caractères

On rappelle la syntaxe du slicing pour les chaînes de caractères :

Pour une chaîne `ch`, et deux entiers a et b tels que $0 \leq a \leq b < \text{len(ch)}$, `ch[a:b]` est une copie de la portion de la chaîne `ch` constituée des caractères de la chaîne `ch` situés aux positions $a, a+1, \ldots, b-1$.

Par exemple, si `ch = "abracadrabra"`, de longueur 11, `ch[4:5]` vaut `"c"`, `ch[8:11]` vaut `"bra"` et `ch[3:3]` est une chaîne vide.

On rappelle que dans ces conditions, la chaîne `ch[a:b]` a pour longueur $b - a$.

On considère la fonction `nombre1`, définie ci-dessous, dont l'implémentation est incomplète :

```
1. def nombre1(motif, texte):
2.     M = len(motif)
3.     T = len(texte)
4.     nb_occ =  ?
5.     for i in range(   ?   ):
6.         if texte[   ?   ] ==  ?  :
7.             ?
8.     return  ?
```

La fonction proposée, `nombre1`, une fois complétée correctement, doit renvoyer le nombre d'occurrences de la chaîne `motif` dans la chaîne de caractères `texte`.

Exemple d'appel à la fonction :

```
>>> nombre1("les", "les MPSI et les PCSI sont en devoir d'informatique")
2
>>> nombre1("ique", "les MPSI et les PCSI sont en devoir d'informatique")
1
```

1. Recopier et compléter les lignes 4 à 8 du programme précédent.

Pour éviter des copies inutiles de chaînes de caractères, on modifie la fonction `nombre1` en une fonction `nombre2` dans une implémentation n'utilisant pas le *slicing*.

Une implémentation incomplète de la fonction `nombre2` est proposée ci-dessous :

```
1. def nombre2(motif,texte):
2.     M = len(motif)
3.     T = len(texte)
4.     nb_occ =  ?
5.     for i in range(   ?   ):
6.         k = 0
7.         while k < M and texte[   ?   ] == motif[   ?   ]:
8.             k = k + 1
9.         if k == ? :
10.            ?
11.    return  ?
```

2. Recopier et compléter les lignes 6 à 10 du programme ci-dessus (les lignes 4, 5 et 11 restant, elles, identiques aux lignes 4, 5 et 8 complétées à la question 1.)

Exercice 7. Fonctions récursives produisant des effets d'affichage

On considère la fonction récursive suivante :

```
12. def f(n, i):
13.     if i != n:
14.         print((n - i) * 'x' + i * 'y')
15.         f(n, i + 1)
```

On rappelle que l'opération `n * ch` ou `n` est un entier et `ch` une chaîne de caractères, produit, pour $n > 0$, une chaîne qui est la concaténation de n copies de la chaîne `ch`, et si n est négatif produit une chaîne vide.

1. Quels affichages obtient-on en exécutant l'appel `f(4, 2)` ?
2. Combien d'appels récursifs sont engendrés par l'appel `f(3, 0)` ? Quels sont les valeurs des paramètres n et i pour ces appels ?
3. Que se passe-t-il si on exécute l'appel `f(4, 5)` ? Donner une modification minimale du code de la fonction permettant de modifier ce comportement. Que se passe-t-il alors si on exécute l'appel `f(4, 5)` ?
4. **Si on échange les lignes 3 et 4 du code de la fonction `f`,** quels affichages obtient-on en exécutant l'appel `f(4, 2)` ? l'appel `f(3, 0)` ?

Exercice 8. Implémentation itérative *vs* implémentation récursive

La « suite de Syracuse » relative à un nombre entier N ($N \geq 0$), est définie par récurrence, de la manière suivante :

$$u_0 = N, \text{ et, pour tout entier } n \geq 0,\ u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}.$$

On admet ici la conjecture dite « de Syracuse », selon laquelle quelle que soit la valeur de l'entier N, il existe un indice n tel que $u_n = 1$.

Partie A programmation impérative

Dans cette partie, on utilisera si nécessaire des boucles, mais aucune fonction récursive.

1. Calculer les cinq premiers termes de la suite de Syracuse de premier terme $N = 10$.
2. Définir une fonction `f` qui prend en argument un entier a et qui retourne la valeur $\frac{a}{2}$ si a est pair, et la valeur $3a + 1$ si a est impair. Les valeurs renvoyées doivent être de type entier.
3. Définir une fonction `syracuse`, utilisant la fonction `f`, qui prend en argument les entiers N et n et renvoie le terme de rang n de la suite de Syracuse.
4. Définir une fonction `syracuse_vol`, utilisant la fonction `f`, qui prend en argument l'entier N et renvoie le temps de vol de la suite de Syracuse associée à N, c'est-à-dire le plus petit indice n tel que $u_n = 1$.

Partie B programmation récursive

Dans cette partie, on utilisera aucune boucle.

5. Écrire une fonction récursive `g` à deux paramètres, N et n, et renvoyant la valeur du terme de rang n de la suite de Syracuse de premier terme N.
   Par exemple, `g(8, 0)` renverra 8, `g(15, 2)` renverra 23.
6. Écrire une fonction récursive `h(N, L)`, telle que l'appel `h(N, [])` (ou `h(N)` si on a donné à `L` la valeur par défaut `[]`) renvoie la liste de tous les termes de la suite de Syracuse de premier terme N, depuis son terme de rang 0, $u_0 = N$, jusqu'à son terme de rang $n_0 \geq 0$, où n_0 est le plus petit entier tel que $u_{n_0} = 1$.
   Par exemple, `h(8, [])` renverra `[8, 4, 2, 1]`, `h(3, [])` renverra `[3, 10, 5, 16, 8, 4, 2, 1]`.

Exercice 9. Dictionnaires

On rappelle la syntaxe pour définir un dictionnaire vide, `d` :

```
d = {} # ou d = dict()
```

On rappelle la syntaxe pour ajouter une paire (cle, valeur) dans un dictionnaire `d` :

```
d[cle] = valeur
```

On rappelle que la syntaxe est la même pour modifier la valeur associée à une clé déjà existante.

On rappelle les trois parcours possibles pour un dictionnaire d :

```
## parcours par les clés
for cle in d.keys():
    ...
## parcours par les clés et les valeurs
for cle, val in d.items():
    ...
## parcours par les valeurs
for val in d.values():
    ...
```

1. On considère les notes obtenues à un contrôle, pour lequel on a dénombré, les nombres d'élèves ayant obtenu une note comprise dans chacun des intervalles de notes $[0; 4[$, $[4; 8[$, $[8; 12[$, $[12; 16[$ et $[16; 20]$.
   On a enregistré les résultats dans le dictionnaire suivant :

```
d = {2: 5, 6: 7, 10: 12, 14: 9, 18: 3}
```

   Ainsi, pour ce contrôle, 7 élèves ont obtenu une note comprise entre 4 et 8, strictement inférieure à 4.
   a. Écrire une fonction `effectif_total` prenant en argument un tel dictionnaire, et renvoyant le nombre total d'élèves évalués. Sur l'exemple la fonction renverra 36.
   b. Écrire une fonction `moyenne` prenant en argument un tel dictionnaire, renvoyant la moyenne des notes obtenue en pondérant chaque note, 2, 6, 10, ~~12,~~ 14 et 18 par l'effectif correspondant.

2. Écrire une fonction `histogramme`, prenant en argument une liste de notes entières comprises entre 0 et 20 et renvoyant un dictionnaire construit sur le principe précédent.
On pourra utiliser le squelette de fonction suivant et les opérations de division entière et de modulo.

```
def histogramme(L):
    d = {2: ..., 6: ..., ...}
    for note in L:
        if note ... :
            d[...] = ...
        ...
    return d
```

Exercice 10. Écriture dans un fichier

Donner une séquence d'instructions permettant de créer le fichier d'extension **.py** dont le contenu est le suivant, de sorte qu'il soit exécutable en tant que programme Python.
On utilisera le caractère `'\t'` pour encoder les indentations.

prog.py

```
for k in range(10):
    print('bonjour')
```

Exercice 11. Lecture d'un fichier .csv

Les fichiers **.csv** sont des fichiers de texte, permettant d'encoder des tableaux construits à l'aide d'un tableur.
On considère le tableau suivant dans lequel sont enregistrées des coordonnées (entières) de points du plan, tel qu'il se présente dans un tableur.

	A	B	C	D		
1	point	abscisse	ordonnée			
2	A1	4	5			
3	A2	-1	6			
…	…	…	…			

Enregistré au format **.csv**, ce tableau se présente sous la forme suivante

coords.csv

point;abscisse;ordonnée
A1;4 ;5
A2;-1;6
…

1. Compléter le script suivant afin de recueillir la première ligne dans une variable `entetes`, de type `str`, la première ligne du fichier, et dans une liste `L`, les lignes suivantes. Le premier élément de `L` sera la chaîne `'A1;4;5'`.
On pourra utiliser, selon les besoins, les méthodes `.read()`, `.readline()`, ou `.readlines()`.

```
f = open(...)
entetes = ...
...
```

2. Écrire une fonction `traitement(ch, sep)`, prenant en argument une chaîne `ch` et un caractère `sep` et renvoyant une liste de toutes les sous-chaînes de la chaîne `ch`, situées entre le début ou la fin de la chaîne `ch` et le séparateur `sep` ou entre deux occurrences du séparateur `sep`.
Par exemple, l'appel `traitement('abracadabra', 'a')` renverra la liste `['', 'br', 'c', 'd', 'br', '']`.

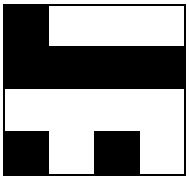
3. Utiliser la fonction `traitement` afin de construire, à partir de la liste `L`, la liste de couples d'entiers, `coords`, contenant les coordonnées de tous les points du fichier.
Sur l'exemple, la liste `coords` sera la liste [(4, 5), (-1, 6), …].

4. Utiliser la fonction `traitement` afin de construire, à partir de la liste `L`, une liste de dictionnaires, `dcoords`, dont chaque élément est un dictionnaire de la forme `{'point' : …, 'x' : …, 'y' : …}`.
Sur l'exemple, le premier élément de la liste `dcoords` serait le dictionnaire :
`{'point' : 'A1', 'x' : 4, 'y' : 5}`.

Exercice 12. Traitement d'images

On considère une liste `L` qui est composée de n listes à n éléments ($n \geq 1$) égaux à 0 ou 1.
La liste de listes `L` est une représentation d'une image en noir et blanc, où 1 représente un pixel noir et 0 un pixel blanc [note : dans la réalité, c'est plutôt le contraire : 0 pour noir, 1 pour blanc].

- Exemple :
  La liste `L0=[[1,0,0,0],[1,1,1,1],[0,0,0,0],[1,0,1,0]]` code l'image suivante :



1	0	0	0
1	1	1	1
0	0	0	0
1	0	1	0

On considère la fonction `traitement` ci-dessous :

```
1. def traitement(L):
2.     n, p = len(L), len(L[0])
3.     for i in range(n // 2):
4.         for j in range(p):
5.             L[i][j], L[n - 1 - i][j] = L[n – 1 - i][j], L[i][j]
```

On appelle la fonction `traitement` avec pour paramètre la liste `L0` définie en exemple.

1. Lors de l'exécution de l'appel `traitement(L0)`, quelles sont les valeurs prises par les variables `n` et `p` ? Quelles valeurs prennent ces variables pour une image rectangulaire quelconque ?

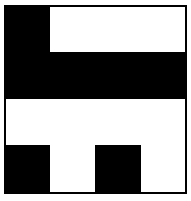
2. A l'issue de l'exécution de l'appel `traitement(L0)`, quel est le contenu de la liste `L0` ? Quelle image est alors codée par `L0` ? [on dessinera cette image sur la copie].
Décrire en une phrase l'effet de l'application `traitement` sur l'image codée.

3. Proposer une implémentation alternative, `traitement1`, de la fonction `traitement` n'utilisant qu'une boucle `for` [et aucune de boucle `while`].

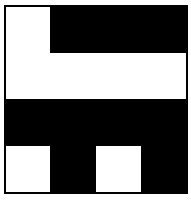
On demande maintenant, d'écrire plusieurs versions modifiées de la fonction `traitement`, qui modifient la liste de liste en entrée, en produisant des effets spécifiques sur l'image codée.

4. Modifier la fonction `traitement` en une fonction `négatif`, transformant la liste de liste en entrée en la liste de liste codant le négatif de l'image codée par la liste en entrée, cette fonction agit sur la liste de liste en entrée en y remplaçant les 1 par des 0 et les 0 par des 1.

Ainsi, par l'appel `negatif(L0)`, la liste `L0=[[1,0,0,0],[1,1,1,1],[0,0,0,0],[1,0,1,0]]`, est modifiée en `[[0,1,1,1],[0,0,0,0],[1,1,1,1],[0,1,0,1]]` et l'image



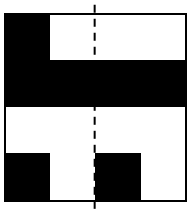
devient



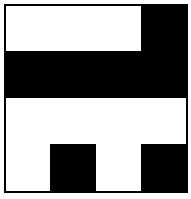
0	1	1	1
0	0	0	0
1	1	1	1
0	1	0	1

5. Modifier la fonction `traitement` en une fonction `symétrie_verticale`, qui agit sur la liste de liste en entrée de sorte que l'image codée soit transformée par une symétrie axiale par rapport à la médiatrice des côtés haut et bas de l'image [en pointillés ci-dessous].

Ainsi, par l'appel `symétrie_verticale(L0)`, où `L0=[[1,0,0,0],[1,1,1,1],[0,0,0,0],[1,0,1,0]]`, l'image codée par `L0`



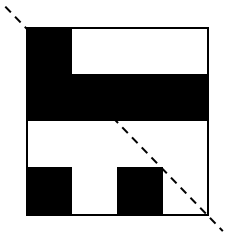
devient



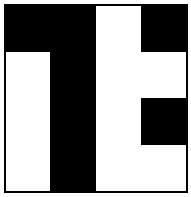
0	0	0	1
1	1	1	1
0	0	0	0
0	1	0	1

6. Modifier la fonction `traitement` en une fonction `symétrie_diagonale`, qui agit sur la liste de liste en entrée de sorte que l'image codée soit transformée par une symétrie axiale par rapport à la diagonale « descendante » [en pointillés ci-dessous] de l'image (supposée carrée).

Ainsi, par l'appel `symétrie_diagonale(L0)`, où `L0=[[1,0,0,0],[1,1,1,1],[0,0,0,0],[1,0,1,0]]`, l'image codée par `L0`



devient



1	1	0	1
0	1	0	0
0	1	0	1
0	1	0	0