IE Informatique et Société Numérique 2

FC, AS, EUR, AMER

Avril 2023

Durée: 2h
Documents et calculatrice interdits

Attention : Un programme **mal indenté**, ou avec de **mauvais choix de noms de variables** sera **sanctionné** *(jusqu'à -1 point)*.

« Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. » *John F. Woods*

Exercice 1 Lecture de code (3pts)

(Q1.1) Qu'affiche le code suivant ?

```
1  def fonct_1(n):
2      mat = []
3      for i in range(n):
4          ligne = []
5          for j in range(n):
6              ligne.append(255)
7          mat.append(ligne)
8      return mat
9
10 def fonct_3(mat):
11     s= ""
12     for i in range(len(mat)):
13         for j in range(len(mat[i])):
14             s=s+str(mat[i][j])+ "  "
15         s= s+ "\n" #'\n' est le caractère qui permet un retour à la ligne
16     print(s)
17
18 def fonct_2(k,mat,val):
19
20     for i in range(k,len(mat)-k):
21         for j in range(k,len(mat) -k):
22             if (i == k or i== len(mat)-1-k) or (j == k or j== len(mat)-1-k):
23                 mat[i][j] = mat[i][j] - k*val
24
25 mat = fonct_1(4)
26 fonct_3(mat)
27 d = 1
28 f = len(mat)-2
29 while d+f//2 != d: #l'opérateur // effectue une division euclidienne
30     fonct_2(d,mat, 10)
31     d = d+ 1
32     f = f-1
33 fonct_3(mat)
```

(Q1.2) Qu'affiche le code suivant ?

```
1  def plus_un(a):
2      a = a + 1
3
4  def ajoute(l):
5      for i in range(len(l)):
6          plus_un(l[i])
7
8  def fonction2(l,i):
9      l[i] = plus_un(l[i])
10
11 a = 2
12 plus_un(a)
13 print(a)
14 l = [3, 3, 3]
15 ajoute(l)
16 print(l)
17 fonction2(l,2)
18 print(l)
```

Exercice 2 Correction de code (1,5pts)

Le code suivant doit afficher la liste [18, 4, 2] qui correspond à la liste des sommes des valeurs par colonnes de la matrice *mat*.
Cependant, la version actuelle du code ne donne pas le résultat attendu. 5 erreurs sont à déplorer. Localisez-les (en précisant le numéro de ligne) et expliquez chaque erreur.

```
1  def init_sommes(n):
2      sommes = []
3      for i in range(n):
4          sommes[i] = 0
5
6  def calculer_sommes_colonnes(mat):
7      sommes = init_sommes(len(mat))
8      for i in range(len(mat)):
9          for j in range(mat[i]):
10             sommes[j]+=mat[i,j]
11     return sommes
12
13 mat =[[5,2,-1],[3,4,3],[10,-2,0]]
14 print({calculer_sommes_colonnes(mat)})
```

Exercice 3 Candy Crush (3,25pts)

Dans le cadre du projet *Candy Crush*, un groupe de projet a proposé une fonction pour faire descendre des bonbons en présence de "trous" dans la grille (représentés par la valeur -1). Le principe de cette fonction consiste à récupérer, dans une liste, les bonbons restants dans chaque colonne puis à les placer en bas de la grille, au bon endroit, pour simuler leur descente.

Le code proposé est le suivant :

```python
def faire_descendre_bonbons(grille):
    for i in range(len(grille[0])):
        # récuperation dans une liste des bonbons encore présents
        # dans la colonne
        les_bonbons = []
        for j in range(len(grille)):
            if grille[i][j] != -1:
                les_bonbons.append(grille[j][i])

        # placement en bas de la colonne des bonbons  de la liste
        j = len(grille)-1
        for b in les_bonbons:
            grille[i][j]= b
        j = j-1

def afficher_grille(grille):
    for i in range (len(grille)):
        for j in range (len(grille[i])):
            print(f"{grille[i][j]}  ",end='')
        print()

grille=[[1,3,2,4],[2,-1,-1,-1],[1,2,4,1], [3,1,3,2]]
afficher_grille(grille)
print()
print()
faire_descendre_bonbons(grille)
afficher_grille(grille)
```

En testant leur code, les élèves voient bien que la fonction ne se comporte pas comme prévu, à travers leur test. Au lieu d'obtenir la grille, après la descente des bonbons :

3	-1	-1	-1
1	1	3	2
2	2	4	1
1	3	2	4

ils obtiennent la grille suivante :

1	3	2	3
2	-1	-1	3
1	2	4	3
3	1	3	2

(Q3.1) Pourriez vous les aider à trouver leurs erreurs et à les corriger ? Précisez les lignes concernées et expliquez.

Exercice 4 Jeu de tests (2,25pts)

Une fonction `liste_contient_liste(liste1,liste2)` recherche si `liste2` est une sous-liste de `liste1`, c'est-à-dire si tous les éléments de la liste `liste2` sont des éléments consécutifs (qui se suivent) de la liste `liste1`.

(Q4.1) Proposez un jeu de tests pour cette fonction (en précisant les valeurs de `liste1` et `liste2` et le résultat attendu).

(Q4.2) Ecrivez la fonction `test_liste_contient_liste` qui permet de tester le bon fonctionnement de la fonction `liste_contient_liste` : appelez cette fonction sur un des tests identifiés à la question (Q4.1)

.

Pour les questions des exercices 5 et 6 :
- Vous avez le droit d'écrire des fonctions supplémentaires, si vous pensez que cela facilite l'écriture du code.
- Les docstrings ne sont pas à fournir sauf si elles sont explicitement demandées ou si vous écrivez une fonction supplémentaire.

Exercice 5 Commandes de bonbons (5.5 pts)

Une association gère un stock de n types de bonbons. Chaque type a un numéro, allant de 0 à $n-1$. Le stock est représenté par une liste. Par exemple :
`stock = [5, 340, 80, 234, 10, 90, 14, 56, 124, 450]`
indique qu'il y a en stock 5 bonbons du type 0, 340 bonbons du type 1, 80 bonbons du type 2, etc.

Tout membre de l'association peut commander des bonbons en indiquant, pour chaque type, le nombre de bonbons souhaité. Cette commande est représentée par une liste de listes. Par exemple :
`ma_commande = [[0, 10], [1, 400], [4, 20], [2, 7]]`
représente une commande de 10 bonbons du type 0, 400 bonbons de type 1, 20 bonbons du type 4 et 7 bonbons du type 2.

L'association traite une commande de la manière suivante :

1. Elle établit une liste des bonbons qui manquent en indiquant leur type et le nombre manquant. Par exemple, pour préparer la commande `ma_commande` étant donné le stock `stock`, la liste `manques = [[0, 5], [1, 60], [4, 10]]` est établie : il manque 5 bonbons du type 0, 60 bonbons de type 1 et 10 bonbons du type 4. Cette liste est vide si tous les bonbons de la commande sont en stock en quantité souhaitée.
2. Si aucun bonbon ne manque, la commande est préparée et le stock est mis à jour.
3. Lorsque la commande ne peut être préparée complètement, l'association met à jour une liste `bilan_manques` qui a la forme suivante.
   `bilan_manques = [[0, 5],[0, 10], [2, 20] ]`
   Cette liste est triée par ordre croissant du type de bonbon. Elle indique que des commandes précédentes n'ont pas pu être préparées : il manquait 5 bonbons de type 0 pour l'une, 10 bonbons de type 0 pour une autre et 20 bonbons de type 2 (pour l'une des deux commandes ou sur une autre commande). Après mise à jour avec `manques = [[0, 5], [1, 60], [4, 10]]`, on a :
   `bilan_manques = [[0, 5], [0, 5],[0, 10], [1, 60], [2, 20], [4, 10] ]`

Nous souhaitons écrire des fonctions Python pour aider au traitement d'une commande.

(Q5.1) Écrivez la fonction Python `calcule_manques` qui prend en paramètres un stock et une commande et qui retourne la liste des manques pour cette commande.

(Q5.2) Écrivez la fonction Python `maj_stock` qui prend en paramètre un stock et une commande et qui modifie le stock en conséquence. On suppose que le stock est suffisant pour tous les types de bonbons.

(Q5.3) Écrivez la fonction Python `maj_bilan` qui prend en paramètres une liste `bilan_manques` et les `manques` d'une commande et qui modifie `bilan_manques` en conséquence. Pour mettre en oeuvre cette fonction, vous pouvez utiliser la fonction Python `insert` : `l.insert(i, e)` insère l'élément e en position i dans l.

Nous souhaitons maintenant écrire un programme Python qui réalise le travail suivant :
- obtention d'une commande.
- traitement d'une commande : si la commande peut être préparée, le programme met à jour le stock `stock` et affiche : `La commande peut être préparée, mise à jour du stock`. Si la commande ne peut pas être préparée, le programme met à jour `bilan_manques` et affiche : `La commande ne peut pas être préparée, mise à jour de bilan_manques`.
- et ainsi de suite pour les différentes commandes, jusqu'à ce qu'une condition d'arrêt soit vraie.
- Après l'arrêt de la boucle de traitement, le programme affiche `Il faut refaire le stock !`. Il affiche aussi les quantités de bonbons manquantes pour chaque type. Par exemple, avec
  `bilan_manques = [[0, 5], [0, 5],[0, 10], [1, 60], [2, 20], [2, 35], [4, 10] ]`,
  le programme afficherait la liste :
  `[[0, 20], [1, 60], [2, 55], [4,10]]`

On suppose que des fonctions `obtenir_commande` et `arret` ont été écrites. Leur docstring est fournie et vous pouvez les utiliser :

```python
def obtenir_commande():
    '''
    permet d'obtenir une commande
    param : aucun
    return : une liste représentant une commande
    '''

def arret(bilan)
    '''
    verifie si le programme doit s'arrêter
    param : une liste bilan des manques
    return : un booléen, True si le programme doit s'arrêter,
    False sinon
    '''
```

(Q5.4) Écrivez le programme Python qui permet d'obtenir le comportement souhaité. Utilisez des listes nommées `bilan_manques` et `stock` pour désigner respectivement le bilan des manques et le stock. Il n'est pas utile d'écrire le contenu de `stock`.

Exercice 6 Historique des notes des bonbons (4.5 pts)

Cet exercice parle aussi de bonbons mais n'est pas lié au précédent.

Chaque type de bonbon reçoit une (seule) note sur 20 toutes les semaines.

On mémorise la note obtenue par chaque type de bonbon, chaque semaine, dans une liste de listes de réels, `notes`. Le $i^{ème}$ élément de `notes` mémorise les notes données chaque semaine au bonbon de type i. Pour simplifier, on suppose que la $1^{ère}$ semaine de l'année est numérotée 0. Par exemple, avec 4 types de bonbons et pour les 6 premières semaines de l'année :
notes = [[15.3, 16.8, 18.1, 15, 13.2, 17], [16, 16, 15.5, 15, 13, 12], [15.3, 16.8, 18.1, 15, 13.2, 17], [15.3, 16.8, 18.1, 15, 13.2, 17]]
signifie que le bonbon de type 0 a eu la note 15.3 la semaine 0, puis la note 16.8 la semaine 1, 18.1 la semaine 2, etc ; le bonbon de type 1 a eu la note 16 la semaine 0, la note 16 la semaine 1, 15.5 la semaine 2, etc.

L'association souhaite connaître la moyenne des notes de chaque type de bonbon entre deux semaines données.

(Q6.1) Écrivez une fonction Python `moyennes_entre` qui prend en paramètre une liste de listes `notes` et deux numéros de semaine `s1` et `s2` et retourne la liste des moyennes de chaque type de bonbon, entre `s1` et `s2` (toutes deux comprises). Par exemple, l'appel à `moyennes_entre(notes, 2, 4)` fournit la liste : [15.43, 14.5, 15.43, 15.43] où 15.43 est la moyenne des notes du bonbon de type 0 sur les semaines 2, 3 et 4, 14.5 celle du bonbon de type 1 sur ces mêmes semaines, etc.

On identifie un type de bonbon qui pose problème par une note qui décroît strictement sur les n dernières semaines. En fixant $n = 3$, dans `notes`, le type de bonbon 1 serait le seul à poser problème puisque ses notes sur les semaines 3, 4 et 5 sont respectivement 15, 13 et 12.

(Q6.2) Écrivez une fonction Python qui affiche les types de bonbons qui posent problème. Veillez à indiquer l'en-tête de la fonction, avec ses paramètres.

Enfin, pour établir des profils de (types de) bonbons, on souhaite connaître ceux qui sont notés de la même façon. Par exemple, dans `notes`, considérant le type (de bonbons) 2, la liste des types de bonbons notés de la même façon (que le type 2) est : [0, 3]

(Q6.3) Écrire une fonction `meme_notation` qui crée et retourne une liste des types de bonbons qui sont notés de la même manière que le type numéro b depuis le début de l'année. Le numéro b ne doit pas faire partie de la liste résultat. Veillez à indiquer l'en-tête de la fonction, avec ses paramètres.