 juin 2019

INFO-H-100 – Informatique – Prof. Th. Massart
1ère année du grade de Bachelier en Sciences de l'Ingénieur
Examen de juin – ***Solutions***

Prénom : Nom : Matricule :

Remarques préliminaires

- On vous demande de répondre à **chaque question sur une feuille séparée**.
- N'oubliez pas d'inscrire votre prénom, nom et numéro de matricule sur chaque feuille.
- Vous disposez de deux heures et 45 minutes et vous ne pouvez pas utiliser de notes.
- Vous pouvez utiliser le verso des feuilles pour répondre.
- Si du code vous est demandé,
  - la réponse à la question doit comprendre le code *Python* structuré et conforme aux règles de bonnes pratiques et conventions,
  - sauf mention contraire, vous ne pouvez utiliser aucune fonction de librairies (pas d'`import`)
  - veillez à découper votre réponse en fonctions de manière pertinente
  - veillez à utiliser des structures de données appropriées.

Question 1 - Code Python (5 points)

Pour chacun des codes suivants, et en supposant qu'ils sont exécutés à la console et que si une instruction provoque une exception, l'exécution continue à l'instruction suivante ;

- si une ou des instructions du code provoquent une exception, dites la ou lesquelles ;
- donnez ce qu'imprime le code (avec les appels qui sont donnés) et expliquez de façon générale ce que fait la fonction (si elle résout un problème, lequel ...) ;
- donnez l'état du programme lors de chaque groupe de `print` grâce à des diagrammes d'état, comme fait au cours ;
- sachant que les types des paramètres sont,
  - n : nombre naturel,
  - a, b, c, d, s, t : listes d'entiers que vous pouvez supposer de taille n,

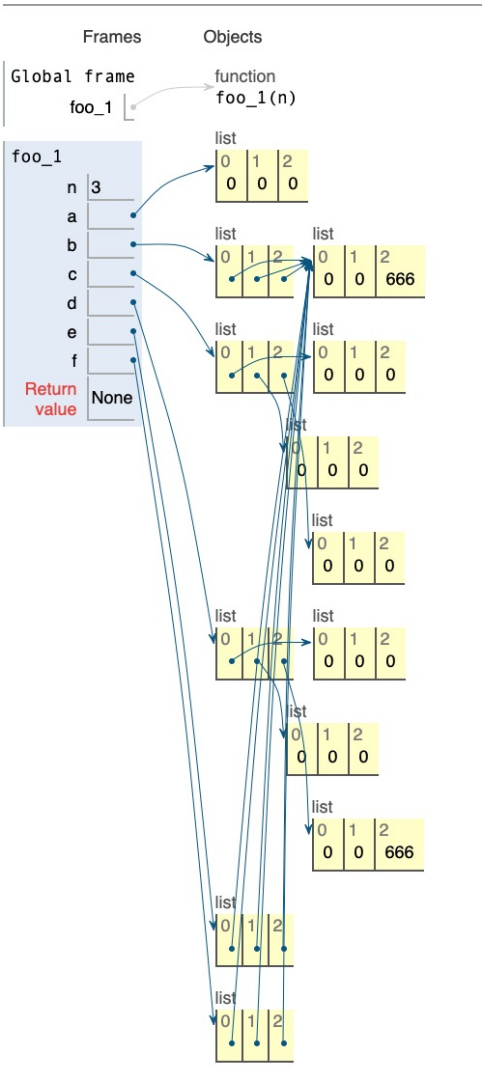
  donnez la complexité **moyenne** ($\mathcal{O}()$) de la **fonction** `foo_`i ($i = 1..3$), en temps d'exécution. Précisez bien les paramètres utilisés pour exprimer la complexité et justifiez bien vos réponses (le résultat seul ne suffit pas pour obtenir des points). Pour calculer la complexité de `foo_3`, faites l'hypothèse que la fonction renvoie le résultat `True`.

Notez bien : pour expliquer ce que le code fait et donner la complexité, vous pouvez annoter les codes fournis (compléter les docstrings, commenter, ...)

1.
```python
def foo_1(n):                          # O(n^2)
    a = [0 * n] * n                    # O(n)
    b = [[0] * n] * n                  # O(n)
    c = [e[:] for e in b]              # O(n^2)
    d = [[0] * n for i in range(n)]    # O(n^2)
    e = [elem for elem in b]           # O(n)
    f = b[:]                           # O(n)
    a[-1][-1] = 666                    # produit une exception
    b[-1][-1] = 666                    # O(1)
    d[-1][-1] = 666                    # O(1)
    print("Pour a :")                  # O(1)
    print(a == b)                      # O(1) (arrêt dès le premier élément)
    print(a == c)                      # O(1) (arrêt dès le premier élément)
    print(a == d)                      # O(1) (arrêt dès le premier élément)
    print(a == e)                      # O(1) (arrêt dès le premier élément)
    print(a == f)                      # O(1) (arrêt dès le premier élément)
    print("Pour b :")                  # O(1)
    print(b == c)                      # O(n)
    print(b == d)                      # O(n)
    print(b == e)                      # O(n^2)
    print(b == f)                      # O(n^2)
    print("Pour c :")                  # O(1)
    print(c == d)                      # O(n^2)
    print(c == e)                      # O(n)
    print(c == f)                      # O(n)
    print("Pour d :")                  # O(1)
    print(d == e)                      # O(n)
    print(d == f)                      # O(n)
    print("Pour e :")                  # O(1)
    print(e == f)                      # O(n^2)

foo_1(3)
```

Solution :
Résultats et diagrammes d"état :

```
Pour a :
False
False
False
False
False
Pour b :
False
False
True
True
Pour c :
False
False
False
Pour d :
False
False
Pour e :
True
```

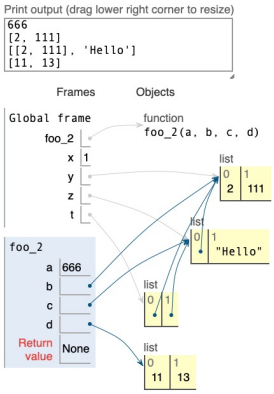


Complexité moyenne de la fonction `foo_1` = $O(n^2)$ (voir code). Notez que :

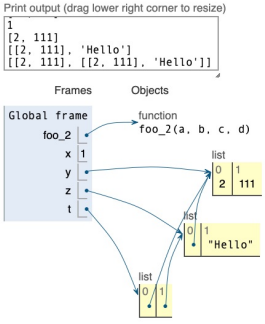
- la partie d'instruction `[[0]*n]*n` est en complexité moyenne (et maximale) en $\mathcal{O}(\backslash)$ étant donné que le code ne crée qu'une seule sous liste ($\mathcal{O}(\backslash)$) et ensuite la liste principale ($\mathcal{O}(\backslash)$) ; la somme des deux est en $\mathcal{O}(\backslash)$.
- la comparaison de tous les objets avec `a` est en $O(1)$ puisque dès le premier élément, il y a une différence.
- la complexité des comparaisons entre `c` ou `d` d'une part et `b`, `e` et `f` d'autre part est en $\mathcal{O}(\backslash)$ puisque dès la sous-liste d'indice 0, il y a une différence.

2.
```
def foo_2(a, b, c, d): # O(n)
    a = 666                 # O(1)
    b[1] = 111              # O(1)
    c[1] = "Hello"          # O(1)
    d = [11, 13]            # O(1)
    print(a)                # O(1)
    print(b)                # O(1)
    print(c)                # O(n) taille de c
    print(d)                # O(1)

x = 1
y = [x + 1, x + 2]
z = [y, y]
t = [y, z]
foo_2(x, y, z, t)
print(x)
print(y)
print(z)
print(t)
```

Solution : Résultats et diagrammes d'état : à la fin de l'exécution de `foo_2`



Résultats et diagrammes d'état : à la fin de l'exécution du script



Ce code ne semble pas résoudre de problème spécifique.
Complexité moyenne de la fonction `foo_2` : $O(n)$. Justification supplémentaire : cf support de notes.

3. Nous faisons l'hypothèse que s et t sont bien des anagrammes.

```
def foo_3(s, t):                            # O(n^2) (car O(n) appels recursifs imbriqués)
    print(s, t)
    res = len(s) == 0 and  len(t) == 0      # O(1)
    if not res:                             # O(n) + appel récursif
        long_s = len(s)                     # O(1)
        long_t = len(t)                     # O(1)
        t.append(s[0])                      # O(1)
```

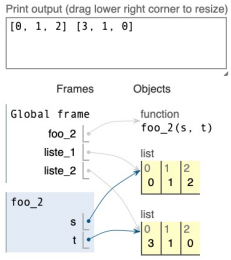
```
        res = long_s == long_t and t.index(s[0]) != long_t # O(n)
        if res:                                 # O(n) + appels récursifs (test en O(1))
            t.pop(long_s)                       # O(1)
            t.pop(t.index(s[0]))                # O(n)
            s.pop(0)                            # O(n)
            res = foo_3(s, t)                   # O(1) + appel récursif
    return res                                  # O(1)

liste_1 = list(range(3))
liste_2 = [3, 1, 0]
print(foo_3(liste_1, liste_2))
```

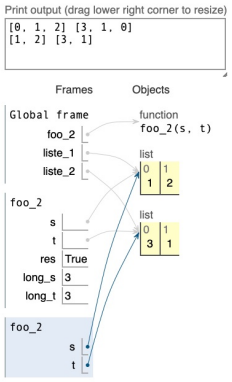
Solution :

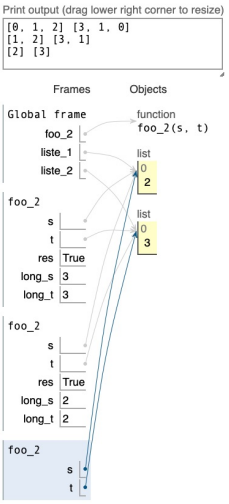
— la fonction `foo_3` renvoie vrai si les mots introduits sont des anagrammes.
Seront imprimés lors de l'exécution du code ci-dessus :

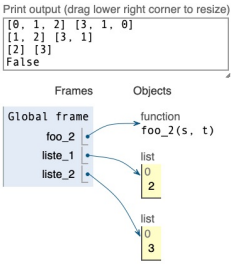
```
[0, 1, 2] [3, 1, 0]
[1, 2] [3, 1]
[2] [3]
False
```

— diagrammes d'état lors des print :









— complexités moyennes et maximales en $O(n^2)$: $n+1$ appels et chaque instance en $O(n)$ (plus précisément $n + n - 1 + n - 2$...).

Question 2 - Théorie (2 points)

En vous aidant des exemples de la question 1, expliquez **en détails** les mécanismes suivants, et montrez où ils sont utilisés dans les codes de la question 1 :

1. La compréhension de liste
2. La récursivité
3. le slicing (tranche)

Solution :
Outre des exemples, votre réponse doit comprendre les éléments suivants :

1. La compréhension de liste
   (a) Syntaxes possibles et effets (0.5)
2. La récursivité (1)
   (a) Principe : fonction que s'appelle elle même directement ou indirectement via des appels à d'autres fonctions
   (b) la gestion de la récursivité par python utilise le stack runtime : à chaque appel, une nouvelle frame est créée, en fin d'appel, on retourne à la frame précédante qui a retenu le contexte local
3. le slicing (0.5)
   (a) `s[debut:fin]` crée une nouvelle liste en copiant les éléments (shadow copy)
   (b) `s[debut:fin] = seq` modifie les élements `s[debut:fin]` `fin` non compris et remplace cette partie par la sous-liste `seq`

Question 3 - Fichiers (5 points)

Dans une librairie, les commandes de livres des clients sont enregistrées dans un fichier texte comme illustré dans l'exemple ci-dessous (fichier "`commandes_clients.txt`") :

```
Jean;Apprendre à programmer avec Python3;Dark Python;Python pour les Nuls
Alain;Programmation Efficace;Learning Python;Learning Python
Jean-Philippe;Python pour les Nuls;Machine Learning avec Python
Ken;Statistique Descriptive;Python pour les Nuls
```

Chaque ligne commence par le nom du client et continue avec une liste des livres commandés, le tout séparé par des points-virgules (";"). Si un client veut plusieurs exemplaires d'un livre, le livre apparaît plusieurs fois comme illustré par la commande d'Alain.

Bien entendu nous supposons qu'un point-virgule ne peut apparaître ni dans le nom d'un client ni dans le titre d'un livre.

Afin d'envoyer les commandes au distributeur, nous voulons collationner ces commandes livre par livre avec le nombre d'exemplaires pour chaque livre. Les livres sont enregistrés dans un autre fichier texte, dans l'ordre décroissant du nombre d'exemplaire commandé.

Ecrivez une fonction `prepare_command(name_in, name_out)` qui crée le fichier de commande de nom `name_out` pour tous les livres au départ des commandes clients dans le fichier de nom `name_in`.

Par exemple, l'appel

```
>>> prepare_command("commandes_clients.txt", "commandes_livres.txt")
```

produirait le fichier de sortie `commandes_livres.txt` avec le contenu suivant :

```
Python pour les Nuls commandé 3 fois
Learning Python commandé 2 fois
Statistique Descriptive commandé 1 fois
Programmation Efficace commandé 1 fois
Machine Learning avec Python commandé 1 fois
Dark Python commandé 1 fois
Apprendre à programmer avec Python3 commandé 1 fois
```

Solution.

```python
def file2dict(file_name):
    d = {}
    for line in open(file_name):
        line = line.strip().split(';')
        for book in line[1:]:
            d[book] = d.get(book, 0) + 1
    return d

def dict2list(d):
    ls = []
    for (book, nb) in d.items():
        ls.append((nb, book))
    return sorted(ls, reverse = True)

def list2file(ls, file_name):
    fout = open(file_name, 'w', encoding = "utf-8")
    for (nb, book) in ls:
        fout.write(book + " commandé " + str(nb) + " fois\n")
    fout.close()

def prepare_command(name_in, name_out):
    list2file(dict2list(file2dict(name_in)), name_out)
```

Question 4 - Tri (4 points)

Vous souhaitez investir votre argent dans un projet durable et qui vous rapportera des bénéfices. Pour ce faire, une agence d'analystes en durabilité classe chaque projet en catégorie de durabilité allant de 1 (très durable) jusque 4 (pas durable du tout), avec comme possibilités $(1, 2, 3, 4)$. Une autre agence d'analystes en rentabilité

vous fournit le taux d'interêt annuel probable pour chacun des projets (quel pourcentage de votre investissement vous toucherez à la fin de l'année).

Vous avez donc accès à une liste `projets` contenant des tuples qui représentent chaque projet dans lequel vous pourriez investir. Chaque tuple contient 3 éléments (nom du projet, categorie de durabilité et taux d'intérêt) et est représenté comme suit : `(nom, durabilité, taux)`.

On vous demande d'écrire une fonction `my_sort(projets)` qui va trier la liste des projets dans l'ordre de votre préférence. Sachant que vous suivez la règle de préférence suivante : Vous préférez les projets les plus durables possibles (de 1 à 4), à une exception près : si un projet est évalué à une catégorie de durabilité d'une unité moins bonne qu'un autre (2 par rapport à 1 par exemple) mais que son taux d'interet le dépasse de plus de 30%, le projet moins durable est préféré. Par ailleurs, s'il y a des égalités parmi la durabilité, vous préférerez les projets qui ont un taux d'intérêt le plus élevé.

Vous êtes libres d'utiliser le tri vu au cours ou au TP de votre choix.

Par exemple :

```
projets = [
    ('BlueBees', 1, 0.04),
    ('Aerospace', 3, 0.02),
    ('CacaoForLife', 2, 0.1),
    ('ChicPen', 2, 0.05),
    ('PetrolForLife', 4, 0.35)
   ]

>>> my_sort(projets)
>>> print(projets)
    [('BlueBees', 1, 0.04), ('CacaoForLife', 2, 0.1),
    ('ChicPen', 2, 0.05), ('PetrolForLife', 4, 0.35),
    ('Aerospace', 3, 0.02)]
```

En effet, CacaoForLife est classé avant ChicPen car son taux d'intéret est plus élevé. Par ailleurs, PetrolForLife est placé avant Aerospace malgré sa moins bonne durabilité, car son taux d'interêt dépasse de 33% celui de Aerospace et n'est qu'une seule catégorie en dessous. Par contre, PetrolForLife ne pourrait pas être placé avant ChicPen par exemple, car il est deux catégories en dessous.

Solution.

```
def tri_selection(s):
        n = len(s)
        for i in range(n-1):
                first = i
                for j in range(i+1,n):
                        if better_placement(s,j,first):
                                first = j
                s[i], s[first] = s[first], s[i]


def better_placement(s,j,first):
        if (s[first][1] > s[j][1] + 1) :
                res = True
        elif s[first][1] == s[j][1] + 1 and s[first][2] < s[j][2] + 0.3:
                res = True
        elif(s[first][1] == s[j][1] - 1  and s[first][2] < s[j][2] - 0.3):
                res = True
        elif (s[first][1] == s[j][1] and s[first][2] < s[j][2]):
                res = True
        else:
                res = False
        return res

def better_placement2(s,j,first):
        return (s[first][1] > s[j][1] + 1) or (s[first][1] == s[j][1] + 1 and s[first][2] < s[j][2] + 0.3) or (s[first][1] ==
and s[first][2] < s[j][2] - 0.3) or (s[first][1] == s[j][1] and s[first][2] < s[j][2])
```

Question 5 - Récursivité (4 points)

Les chiffres romains étaient un système de numération utilisé par les Romains de l'Antiquité pour, à partir de seulement sept lettres, écrire des nombres entiers (mais pas le zéro, qu'ils ne connaissaient pas ; ou plus exactement qu'ils ne considéraient pas comme un nombre).

Chiffre romain	i	v	x	l	c	d	m
Nombre décimal	1	5	10	50	100	500	1000

La numérotation dans l'usage actuel repose sur deux principes :

— Toute lettre L_2 placée à la droite d'une autre lettre L_1 s'ajoute à celle-ci si $L_2 \leq L_1$.

— Toute lettre L_1 placée immédiatement à la gauche d'une autre lettre $L_2 > L_1$ se retranche de celle-ci.

Il est a noter qu'une seule lettre peut être soustraite seulement, ainsi `iix` n'est pas un nombre valide pour 8 qui s'écrit `viii`, de même pour `iiixiii` qui n'est pas valide pour 10. Vous pouvez faire l'hypothèse que les nombres donnés sont syntaxiquement corrects.

Quelques exemples de nombres romains :

Entier décimal	604	2658	145	659	1177	3451	1900	2011	154	33
Nombre romain	dciv	mmdclviii	cxlv	dclix	mclxxvii	mmmcdli	mcm	mmxi	cliv	xxxiii

Ecrivez, **obligatoirement de manière récursive**, une fonction permettant de convertir une chaine de caractères (en minuscules) contenant un nombre romain en son equivalent décimal. On considerera que l'argument de la fonction est toujours valide et non-vide. Vous pouvez vous aider du dictionnaire suivant (considéré comme une variable constante globale) :

```
ROMAN_VALUES = { 'm':1000, 'd':500, 'c':100, 'l':50, 'x':10, 'v':5, 'i':1 }
```

Par exemple :

```
>>> print(roman_to_digits("mmdclviii"))
>>> 2658

>>> print(roman_to_digits("xxxiii"))
>>> 33
```

Solution.

```
ROMAN_VALUES = { 'm':1000,'d':500 ,'c':100 ,'l':50, 'x':10 ,'v':5 ,'i':1 }

def roman_to_digits(roman):
    if len(roman) == 1:
        return ROMAN_VALUES[roman]
    elif ( ROMAN_VALUES[roman[0]] < ROMAN_VALUES[roman[1]] ):
        return roman_to_digits(roman[1:]) - ROMAN_VALUES[roman[0]]
    else:
        return roman_to_digits(roman[1:]) + ROMAN_VALUES[roman[0]]
```