janvier 2018

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

Prénom : Nom : Matricule :

Remarques préliminaires

- On vous demande de répondre à **chaque question sur une feuille séparée**.
- N'oubliez pas d'inscrire votre nom, prénom et numéro de matricule sur chaque feuille.
- Vous disposez de trois heures 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 bonne pratique 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 - Fonctionnement de Python (2 points)

Expliquer le mode de fonctionnement d'un compilateur et d'un interpréteur. Python de base est-il interprété ou compilé ? Expliquez quels en sont les avantages et les inconvénients pour ce langage.

Solution. → Voir syllabus et cours. : Il fallait expliquer les 6 phases de la compilation, le cycle de l'interpréteur, dire que de base Python est un langage interprété, ce qui induit une série d'avantages et inconvénients (à détailler). Prénom : Nom : Matricule :

Question 2 - Code Python (5 points)

Pour chacun des codes suivants,

- donnez ce qu'il imprime (avec les appels qui sont donnés) et expliquez de façon générale ce que fait la fonction ou la classe (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,
  - a et n : nombre naturel,
  - b, c, x : liste d'entiers que vous pouvez supposer de taille n,
  - $texte$, s et t : strings que vous pouvez supposer de taille n contenant uniquement des caractères ASCII (128 caractères différents possibles),

  donnez la complexité **moyenne** ($\mathcal{O}()$) de la **fonction** `foo_`i ($i = 1..4$) **dans l'hypothèse où** foo_4 **renvoit** $True$, et des méthodes de la classe Dog (sous-question 5), 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).

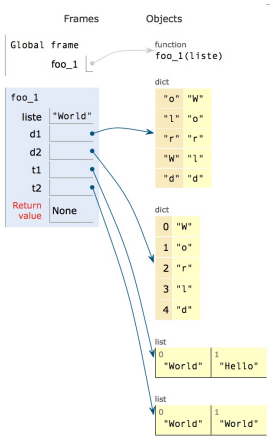
Notez bien : pour expliquer ce que le code fait et donner la complexité, vous pouvez annoter les codes fournis (compléter les docstrings, commenter, ...)

1.
```python
def foo_1(texte):                       # O(n)
    d1 = dict(zip(set(texte),texte))    # O(n)
    d2 = dict(enumerate(texte))         # O(n)
    t1 = [texte, texte]                 # O(1)
    t2 = t1[:]                          # O(1)
    t1[1] = "Hello"                     # O(1)
    print(d1)                           # O(n)
    print(d2)                           # O(n)
    print(t1)                           # O(n)
    print(t2)                           # O(n)
    print(t1 is t2)                     # O(1)
    print(t1[0] is t2[0])               # O(1)
    print(t1[1] is t2[1])               # O(1)
foo_1("World")
```

Solution :
Imprime :

```
{'o': 'W', 'l': 'o', 'r': 'r', 'W': 'l', 'd': 'd'}
{0: 'W', 1: 'o', 2: 'r', 3: 'l', 4: 'd'}
['World', 'Hello']
['World', 'World']
False
True
False
```

Ce code ne semble pas résoudre de problème spécifique.
Diagramme d'état :



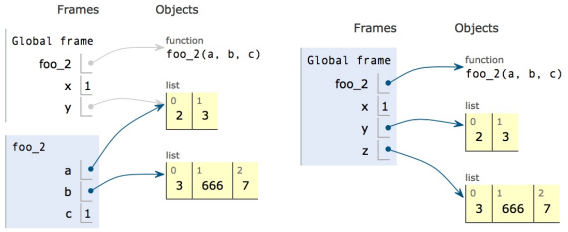
Complexité moyenne de la fonction `foo_1` = $O(n)$ (voir code et détails, voir explications sur la manipulation des structures de données dns le syllabus).

2.
```
def foo_2(a, b, c):     # O(n)
    a = b               # O(1)
    b = c               # O(1)
    c = 1               # O(1)
    b[c] = 666          # O(1)
    print(a)            # O(n)
    return b            # O(1)

x = 1
y = [2,3]
z = foo_2(x,y,[3,5,7])
print(x)
print(y)
print(z)
```

Solution :
Imprime :

```
[2, 3]
1
[2, 3]
[3, 666, 7]
```

Diagramme d'état :



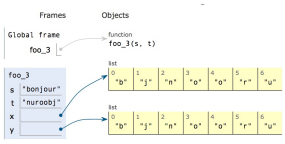
Ce code ne semble pas résoudre de problème spécifique.
Complexité moyenne de la fonction `foo_2` : $O(n)$ à cause du `print(a)` où a est une liste de taille n (voir code). Justification supplémentaire : cf support de notes.

3.
```
def foo_3(s,t):             # O(n log n)
    x = list(s)             # O(n)
    y = [e for e in t]      # O(n)
    x.sort()                # O(n log n)
    y.sort()                # O(n log n )
    print()                 # O(1)
    return x == y           # O(n)

print(foo_3('bonjour','nuroobj'))
```

Solution :
Imprime :

```
True
```

Diagrammes d'état :



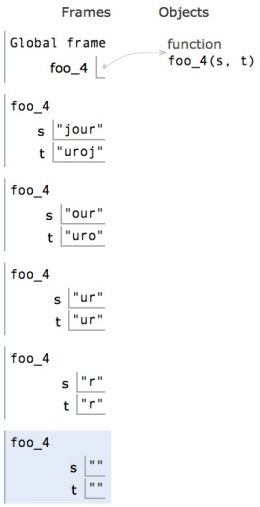
L'autre diagramme d'état est moins intéressant (voir pythontutor pour l'obtenir).
Ce code teste si les deux strings sont des anagrammes c'est-à-dire sont égaux à l'ordre des caractères prêt (bijection entre les deux séquences).
Complexité moyenne : (Hypothèse, s et t de même taille n) : $O(n\ log\ n)$.

4.
```
def foo_4(s,t):                                          # O(n^2)  (O(n) sans l'appel récursif)
    if len(s) > 0 and s[0] in t:                         # O(n)  (sans l'appel récursif) cond en O(n)
        res = foo_4(s[1:], t[:t.find(s[0])]+t[t.find(s[0])+1:])   # O(n) + tps d'exécution de l'appel récursif
    else:                                                # O(n)
        print(s,t)                                         # O(n)
        res = len(t) == 0                                  # O(1)
    return res                                           # O(1)

print(foo_4('jour','uroj'))
```

Solution :
Imprime :

```
True
```

Diagrammes d'état :



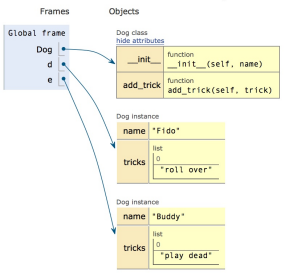
L'autre diagramme d'état est moins intéressant (voir pythontutor pour l'obtenir).
Ce code teste si les deux strings sont des anagrammes c'est-à-dire sont égaux à l'ordre des caractères prêt (bijection entre les deux séquences).
Complexité : Chaque instance prend un temps moyen en $O(n)$ et appel une instance de taille d'un élément en moins. Au total $O(n + (n-1) + (n-2) + ...) = O(n^2)$

5.
```python
class Dog: # complexité moyenne

    def __init__(self, s):     # O(1)
        self.name = s          # O(1)
        self.tricks = []       # O(1)

    def add_trick(self, t):    # O(1)
        self.tricks.append(t)  # O(1)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
print(d.tricks)
print(e.tricks)
```

Solution :
Imprime :

```
['roll over']
['play dead']
```

Diagrammes d'état :



Ce code crée une nouvelle classe d'objet Dog ainsi que 2 objets de cette classe
Complexités moyenne :

- l'initialisation d'un nouvel objet de classe Dog est en O(1) (moyenne).
- la complexité moyenne de la methode `add_trick` est en $O(1)$.

Question 3 (5 points)

Vous êtes engagés par un théatre pour améliorer la logistique d'organisation de leurs scènes.
Ils possèdent des fichiers '.txt' qui indiquent quels acteurs jouent dans quelle scène. Ce type de fichier que nous nommerons 'scenes' se présente de la manière suivant :

```
scene1 Jean Pierre Anne
scene2 Alain Anne Gilles
scene3 Gilles Pierre Paul
scene4 Antoine Jean
```

Chaque ligne représente une scène, où chaque mot est séparé par un espace. Le premier mot est le nom de la scène (en un seul mot) et les mots suivants sont les noms des acteurs qui jouent dans cette scène. Par soucis de facilité, chaque acteur a un nom qui peut s'écrire en un seul mot.
Pour faciliter l'organisation des acteurs, le théatre vous demande d'écrire une fonction qui crée un fichier '.txt' qui résumera pour chaque acteur, dans quelle(s) scène(s) il joue. Ce fichier que nous nommerons 'resume_acteurs' se présenterait comme suit (si l'on utilise le fichier 'scenes' présenté ci-dessus) :

```
Alain scene2
Anne scene1 scene2
Antoine scene4
Gilles scene2 scene3
Jean scene1 scene4
Paul scene3
Pierre scene1 scene3
```

On demande que les noms soient classés par ordre alphabétique. Les méthodes `sort()` ou `sorted()` peuvent être utilisées. Chaque scene correspondant à un acteur sera séparée par un espace.
On vous demande d'écrire une fonction `scenes_acteurs(scenes, resume_acteurs)` qui prend en argument 2 noms de fichier. Le premier représentant celui qui définit quels acteurs jouent dans quelle scène. Le deuxième, que vous devez créer, qui résume pour chaque acteur dans quelle scène il joue.

Solution.

```python
def load_scenes_in_dico(scenes):
    actors_scenes = {}
    for line in open(scenes):
        infos = line.split()
        for actor in infos[1 : ]:
            actors_scenes.setdefault(actor, []).append(infos[0])
    return actors_scenes

def make_actors_file_from_dico(resume_acteurs, dico):
    fout = open(resume_acteurs, "w")
    for actor in sorted(dico):
        fout.write(actor + " " + " ".join(dico[actor]) + "\n")
    fout.close()

def scenes_acteurs(scenes, resume_acteurs):
    dico = load_scenes_in_dico(scenes)
    make_actors_file_from_dico(resume_acteurs, dico)

scenes_acteurs("scenes.txt", "acteurs.txt")
```

Question 4 - (4 points)

Des chercheurs de fortunes ramènent chacun leur butin chez leur chef. Celui-ci rassemble donc toutes les récoltes dans une liste `butins`. Les butins de chaque chercheur de fortune sont représentés par des sous-listes. Elles contiennent des tuples représentant la nature de l'élément et le volume rapporté (litre). Ces récoltes peuvent être soit du fer (f), soit du bois (b) soit de l'acier (a).

Voici un exemple :

```
butins = [
    [('b', 133), ('a', 25),('f', 100)],
    [('b', 18), ('a', 115),('f', 103)],
    [('b', 13), ('a', 251)]
    ]
```

La valeur et le poids de chaque type d'éléments sont stockés dans un dictionnaire dico où chaque clé correspond au type de l'élément et chaque valeur correspond à un tuple représentant respectivement la valeur par litre et le poids par litre.

Voici un exemple :

```
dico = {'a': (10, 10) , 'b': (6, 5) , 'f': (8, 9)} # dans cet exemple, le bois (b)
                                                    # a une valeur par litre de 6
                                                    # et un poids par litre de 5
```

Nous vous demandons d'écrire une fonction $tri\_recolte(butins, dico)$ qui trie les différents butins selon la règle suivante : Les butins dont la valeur totale de la marchandise est la plus élevée seront placés en première position. Si deux butins ont des valeurs égales, le butin le plus léger sera mis en premier dans la liste. Nous vous demandons d'effectuer ce tri soit par sélection soit par insertion.

Solution.

```
def tri_butins(butins, dico): # tri par sélection
    n = len(butins)
    for i in range(n - 1):
        i_min = i
        for j in range(i + 1, n):
            if vient_avant(butins[j], butins[i_min], dico):
                i_min = j
        butins[i], butins[i_min] = butins[i_min], butins[i]

# Est-ce que le butin b1 vient avant le butin b2 ?
def vient_avant(b1, b2, dico):
    v1, v2 = valeur(b1, dico), valeur(b2, dico)
    p1, p2 = poids(b1, dico), poids(b2, dico)
    return v1 > v2 or (v1 == v2 and p1 < p2)

def valeur(b, dico):
    return eval(b, dico, 0)    # critere 0 = valeur dans dico

def poids(b, dico):
    return eval(b, dico, 1)    # critere 1 = poids dans dico

def eval(b, dico, critere):
    e = 0
    for (m, q) in b:           # Pour chaque matière-quantité du butin
        e += q * dico[m][critere]
    return e
```

Question 5 - Récursivité (4 points)

Votre ancêtre possédait des boîtes magiques. Celles-ci contiennent des pierres de pouvoir bleues, des pierres de pouvoir rouges, ainsi que d'autres boîtes magiques contenant à leur tours d'autres pierres, boîtes, et ainsi de suite.

Afin de comptabiliser les différentes pierres de pouvoir, vous avez créé une liste python dont les éléments sont les suivants :

- 0 pour une pierre rouge,
- 1 pour une pierre bleue,
- une nouvelle liste pour une autre boite magique.

Par exemple, la liste suivante : [0, [0, [[]], 1], 1] représente une boîte contenant :

- une pierre rouge,
- une sous-boîte contenant :
  - une pierre rouge,
  - une sous-boîte contenant :
    - une sous-boîte vide
  - une pierre bleue,
- une pierre bleue

Veuillez écrire une fonction récursive calculant la différence entre la quantité de pierres bleues et la quantité de pierres rouges contenues dans une boîte magique (ainsi que toutes ses sous-boîtes).

Solution.

```
def count_delta(recursive_stones_list):
    diff = 0
    for element in recursive_stones_list:
        if type(element) == list:
            diff += count_delta(element)

        else:
            if element == 1:
                diff += 1
            else:
                diff -= 1
    return diff
```