Lycée Les 3 Sources **Bourg-Lès-Valence**
NSI 1re Année **2023-24**

Devoir surveillé n°4 – correction

Exercice 1 : (13,5pt) *Cet exercice porte sur les listes, les dictionnaires et la programmation de base en Python.*

Pour son évaluation de fin d'année, l'institut d'Enseignement Néo-moderne (EN) a décidé d'adopter le principe du QCM. Chaque évaluation prend la forme d'une liste de questions numérotées de 0 à 19. Pour chaque question, 5 réponses sont proposées. Les réponses sont numérotées de 1 à 5. Exactement une réponse est correcte par question et chaque candidat coche exactement une réponse par question. Pour chaque évaluation, on dispose de la correction sous forme d'une liste `corr` contenant pour chaque question, la bonne réponse ; c'est-à-dire telle que `corr[i]` est la bonne réponse à la question `i`.

Par exemple, on présente dans la suite du sujet la liste `corr0` qui correspond à la correction de l'épreuve 0. Cette liste indique que pour l'épreuve 0, la bonne réponse à la question 0 est 4, et que la bonne réponse à la question 19 est 3.

Les copies des élèves sont également sous forme de liste, avec pour chaque question, la réponse donnée. On donne ci-dessous `copTM` qui est la copie de Tom Matt pour l'épreuve 0.

```
corr0 = [4, 2, 1, 4, 3, 5, 3, 3, 2, 1, 1, 3, 3, 5, 4, 4, 5, 1, 3, 3]
copTM = [4, 1, 5, 4, 3, 3, 1, 4, 5, 3, 5, 1, 5, 5, 5, 1, 3, 3, 3, 3]
```

1) a) Calculer la note de Tom, sachant qu'une bonne réponse vaut 1 point et qu'une mauvaise réponse en vaut 0. Il a 6 bonnes réponses, donc 6.

   b) Compléter le code de la fonction `note` qui prend en paramètre `cop` et `corr`, deux listes d'entiers entre 1 et 5 et qui renvoie la note attribuée à la copie `cop` selon la correction `corr`. Ainsi, `note(copTM, corr0)` renvoie la réponse trouvée à la question précédente.

```
def note(cop, corr):
    n = 0
    for i in range(len(cop)):
        if cop[i] == corr[i]:
            n = n + 1
    return n
```

L'institut EN souhaite automatiser totalement la correction de ses copies. Pour cela, il a besoin d'une fonction pour corriger des paquets de plusieurs copies. Un paquet de copies est donné sous la forme d'un dictionnaire dont les clés sont les noms des candidats et les valeurs sont les listes représentant les copies de ces candidats. On peut considérer un paquet `p1` de copies où l'on retrouve la copie de Tom Matt :

```
p1 = {
 'Tom Matt': [4, 1, 5, 4, 3, 3, 1, 4, 5, 3, 5, 1, 5, 5, 5, 1, 3, 3, 3, 3],
 'Lambert Ginne': [2, 4, 2, 2, 1, 2, 4, 2, 2, 5, 1, 2, 5, 5, 3, 1, 1, 1, 4, 4],
 'Carl Roth': [5, 4, 4, 2, 1, 4, 5, 1, 5, 2, 2, 3, 2, 3, 3, 5, 2, 2, 3, 4],
 'Kurt Jett': [2, 5, 5, 3, 4, 1, 5, 3, 2, 3, 1, 3, 4, 1, 3, 1, 3, 2, 4, 4],
 'Ayet Finzerb': [4, 3, 5, 3, 2, 1, 2, 1, 2, 4, 5, 5, 1, 4, 1, 5, 4, 2, 3, 4]
 }
```

2) On souhaite écrire une fonction `notes_paquet` qui prend en paramètre un paquet de copies `p` et une correction `corr` et qui renvoie un dictionnaire dont les clés sont les noms des candidats du paquet `p` et les valeurs dont leurs notes selon la correction `corr`.
   Par exemple :

```
>>> notes_paquet(p1, corr0)
{'Tom Matt': XX, 'Lambert Ginne': 4, 'Carl Roth': 2, 'Kurt Jett': 4,
'Ayet Finzerb': 3}
```

La valeur `XX` correspond à la réponse à la question 1) a).

a) À l'aide des exemples précédents, donner la valeur correspondant à l'évaluation des expressions suivantes :

```
>>> p1['Carl Roth'][1]
4
>>> notes = notes_paquet(p1, corr0)
>>> notes['Kurt Jett']
4
>>> [c for c in notes if notes[c] < 4]
['Carl Roth', 'Ayet Finzerb']
```

b) Compléter le code de la fonction `notes_paquet`. Il doit y avoir un appel à la fonction `note` définie précédemment.

```
def notes_paquet(paquet, corr):
    notes = {}
    for nom in paquet:
        notes[nom] = note(paquet[nom], corr)
    return notes
```

Un ingénieur de l'institut EN a démissionné en laissant une fonction Python énigmatique sur son poste. Le directeur est convaincu qu'elle sera très utile, mais encore faut-il comprendre à quoi elle sert.
Voici la fonction en question :

```
def enigme(notes):
    a = None
    b = None
    c = None
    reste = {}
    for nom in notes:
        tmp = c
        if a == None or notes[nom] > a[1]:
            c = b
            b = a
            a = (nom, notes[nom])
        elif b == None or notes[nom] > b[1]:
            c = b
            b = (nom, notes[nom])
        elif c == None or notes[nom] > c[1]:
            c = (nom, notes[nom])
        else :
            reste[nom] = notes[nom]
        if tmp != c and tmp != None:
            reste[tmp[0]] = tmp[1]
    return (a, b, c, reste)
```

3) On considère le dictionnaire suivant :

```
notes = {'Tom Matt': 6, 'Lambert Ginne': 4, 'Carl Roth': 2, 'Kurt Jett': 4,
'Ayet Finzerb': 3}
```

a) Déterminer les valeurs de ces variables après cet appel à `enigme` :

```
>>> a, b, c, reste = enigme(notes)
>>> a
('Tom Matt', 6)
>>> b
('Lambert Ginne', 4)
>>> c
('Kurt Jett', 4)
>>> reste
{'Carl Roth': 2, 'Ayet Finzerb': 3}
```

b) Expliquer, en français, ce que calcule la fonction `enigme` lorsqu'on l'applique à un dictionnaire dont les clés sont les noms des candidats et les valeurs sont leurs notes.

Solution : Cette fonction détermine les 3 meilleurs notes dans le dictionnaire et renvoie des couples de la forme `(nom, note)`. La quatrième valeur correspond au dictionnaires des notes qui ne font pas partie des 3 meilleures.

c) Quelles sont les valeurs de `c` et de `reste` renvoyées par `enigme` si le dictionnaire a strictement moins de 3 entrées ?

Solution : Avec moins de 3 entrées, `c` vaut **`None`** et `reste` est un dictionnaire vide.

d) Compléter la fonction `classement` qui prend en paramètre un dictionnaire dont les clés sont les noms des candidats et les valeurs sont leurs notes et qui, en utilisant la fonction enigme, renvoie la liste des couples `(nom, note)` des candidats classés par notes décroissantes.

```
>>> classement({'Tom Matt': 6, 'Lambert Ginne': 4, 'Carl Roth': 2,
                'Kurt Jett': 4, 'Ayet Finzerb': 3})
[('Tom Matt', 6), ('Lambert Ginne', 4), ('Kurt Jett', 4),
 ('Ayet Finzerb', 3), ('Carl Roth', 2)]
```

```
def classement(notes):
    cl = []
    reste = notes
    while len(reste) > 0:
        a, b, c, reste = enigme(reste)
        for v in [a, b, c]:
            if v != None:
                cl.append(v)
    return cl
```

Le professeur Paul Tager a élaboré une évaluation particulièrement innovante de son côté. Toutes les questions dépendent des précédentes. Il est donc assuré que dès qu'un candidat s'est trompé à une question, alors toutes les réponses suivantes sont également fausses. Cette fois, il n'enregistre pas les réponses des élèves mais uniquement si la réponse est bonne ou pas. On obtient des listes de booléens avec **`True`** si la réponse est bonne et **`False`** sinon.
M. Tager a malheureusement égaré ses notes, mais il a gardé les listes de booléens associées. Grâce à la forme particulière de son évaluation, on sait que ces listes sont de la forme :

`[True, True, ..., True, False, False, ..., False]`

Pour recalculer ses notes, il a écrit les deux fonctions Python. Voici la première :

```
def renote_express(copcorr) :
    c = 0
    while copcorr[c] == True:
        c = c + 1
    return c
```

La seconde est incomplète :

```
def renote_express2(copcorr) :
    gauche = 0
    droite = len(copcorr)
    while droite - gauche > 1:
        milieu = (gauche + droite)//2
        if copcorr[milieu]:
            gauche = milieu
        else:
            droite = milieu
    if copcorr[gauche]:
        return droite
    else:
        return gauche
```

4) a) Compléter le code de la fonction Python `renote_express2` pour qu'elle calcule la même chose que `renote_express`.

b) Expliquer pourquoi la fonction `renote_express2` va particulièrement plus vite que la fonction `renote_express` pour les longues évaluations.

Solution : Cette fonction utilise la dichotomie. Il faut donc regarder nettement moins de valeurs à regarder pour pouvoir répondre.

c) (Bonus) Expliquer comment adapter `renote_express2` pour obtenir une fonction qui corrige très rapidement une copie pour les futures évaluations de M. Tager s'il garde la même spécificité pour ses énoncés. Cette fonction ne devra pas prendre en paramètre la liste de booléens correspondant à la copie corrigée, mais la copie et le corrigé. Il ne faut pas construire la liste de booléens.

Solution : Il faut remplacer tous les tests cherchant à savoir si `copcorr[i]` est vrai par `cop[i] == corr[i]`.

Exercice 2 : (9,5pt) Une ville souhaite gérer son parc de vélos en location partagée. L'ensemble de la flotte de vélos est stocké dans une table de données représentée en langage Python par un dictionnaire contenant des associations de type `id_velo : dict_velo` où `id_velo` est un nombre entier compris entre 1 et 199 qui correspond à l'identifiant unique du vélo et `dict_velo` est un dictionnaire dont les clés sont : `"type"`, `"etat"`, `"station"`.
Les valeurs associées aux clés `"type"`, `"etat"`, `"station"` de `dict_velo` sont de type *chaînes de caractères* ou *nombre entier* :

- `"type"` : chaîne de caractères qui peut prendre la valeur `"electrique"` ou `"classique"`
- `"état"` : nombre entier qui peut prendre la valeur 1 si le vélo est disponible, 0 si le vélo est en déplacement, -1 si le vélo est en panne
- `"station"` : chaînes de caractères qui identifie la station où est garé le vélo.

Dans le cas où le vélo est en déplacement ou en panne, "station" correspond à celle où il a été dernièrement stationné.

Voici un extrait de la table de données :

```
flotte = {
  12 : {"type" : "electrique", "etat" : 1, "station" : "Prefecture"},
  80 : {"type" : "classique", "etat" : 0, "station" : "Saint-Leu"},
  45 : {"type" : "classique", "etat" : 1, "station" : "Baraban"},
  41 : {"type" : "classique", "etat" : -1, "station" : "Citadelle"},
  26 : {"type" : "classique", "etat" : 1, "station" : "Coliseum"},
  28 : {"type" : "electrique", "etat" : 0, "station" : "Coliseum"},
  74 : {"type" : "electrique", "etat" : 1, "station" : "Jacobins"},
  13 : {"type" : "classique", "etat" : 0, "station" : "Citadelle"},
  83 : {"type" : "classique", "etat" : -1, "station" : "Saint-Leu"},
  22 : {"type" : "electrique", "etat" : -1, "station" : "Joffre"}
}
```

`flotte` étant une variable globale du programme. Elle peut être utilisée dans n'importe quelle fonction sans être donnée en paramètre.
Toutes les questions de cet exercice se réfèrent à l'extrait de la table flotte fourni ci-dessus.
Il y a des rappels sur les dictionnaires en langage Python à la fin de l'exercice.

1) a) Que renvoie l'instruction `flotte[26]` ?
   `{"type" : "classique", "etat" : 1, "station" : "Coliseum"}`
   b) Que renvoie l'instruction `flotte[80]["etat"]` ? `0`

2) Voici le script d'une fonction :

```
def proposition(choix):
  for v in flotte:
    if flotte[v]["type"] == choix and flotte[v]["etat"] == 1:
        return flotte[v]["station"]
```

   a) Quelles sont les valeurs possibles de la variable `choix` ?
   `"electrique"` ou `"classique"`.
   b) Expliquer ce que renvoie la fonction lorsque l'on choisit comme paramètre l'une des valeurs possibles de la variable `choix`.
   Elle renvoie le nom d'une station où un vélo du type demandé est disponible, s'il y en a un et **None** sinon.

3) a) Écrire un script en langage Python qui affiche les identifiants (`id_velo`) de tous les vélos disponibles à la station `"Citadelle"`.

```
for v in flotte:
    if flotte[v]["station"] == "Citadelle":
        print(v)
```

   b) Écrire un script en langage Python qui permet d'afficher l'identifiant (`id_velo`) et la station de tous les vélos électriques qui ne sont pas en panne.

```
for v in flotte:
    if flotte[v]["type"] == "electrique' and flotte[v]["etat"] != -1:
        print(v, flotte[v]["station"])
```

4) On dispose d'une table de données des positions GPS de toutes les stations, dont un extrait est donné ci-dessous. Cette table est stockée sous forme d'un dictionnaire.
   Chaque élément du dictionnaire est du type :
   `'nom de la station' : (latitude, longitude)`

```
stations = {
  'Prefecture' : (49.8905, 2.2967),
  'Saint-Leu' : (49.8982, 2.3017),
  'Coliseum' : (49.8942, 2.2874),
  'Jacobins' : (49.8912, 2.3016)
}
```

On admet que l'on dispose d'une fonction `distance(p1, p2)` permettant de renvoyer la distance en mètres entre deux positions données par leurs coordonnées GPS (latitude et longitude).

Cette fonction prend en paramètre deux tuples représentant les coordonnées des deux positions GPS et renvoie un nombre entier représentant cette distance en mètres.

Par exemple, `distance((49.8905, 2.2967), (49.8912, 2.3016))` renvoie `9591`.

Compléter le code de la fonction `proches` qui prend en paramètre les coordonnées GPS de l'utilisateur sous forme d'un tuple et qui renvoie un dictionnaire dont les clés sont les noms de chaque station à moins de 800 mètres de l'utilisateur. Chaque station est associée à un couple formé de la distance à l'utilisateur et la liste des `id_velo` des vélos disponibles qu'elle contient. Par exemple, `"Baraban": (615, [45])`.

Une station où aucun vélo n'est disponible ne doit pas être dans le dictionnaire.

```
def proches(coord_util):
    reponses = {}
    for v in flotte:
        if flotte[v]["etat"] == 1:
            s = flotte[v]["station"]
            if s in reponses:
                reponses[s][1].append(v)
            else:
                d = distance(coord_util, stations[s])
                if d < 800:
                    reponses[s] = (d, [v])
    return reponses
```

Rappels sur les dictionnaires en Python :

Action	Instruction et syntaxe
Créer un dictionnaire vide	`dico={}`
Obtenir un élément d'un dictionnaire existant à partir de sa clé et renvoie une erreur si cle n'existe pas dans le dictionnaire	`dico[cle]`
Modifier la valeur d'un élément d'un dictionnaire à partir de sa clé	`dico[cle]=nouvelle_valeur`
Ajouter un élément dans un dictionnaire existant	`dico[nouvelle_cle]=valeur`
Tester l'appartenance d'un élément à un dictionnaire (renvoie un booléen)	`cle in dico`
Afficher les associations `cle:valeur` du dictionnaire `dico`	`for cle in dico:` `  print(cle, dico[cle])`