Centres étrangers - juin 2024 - sujet 2 (corrigé)

Exercice 1 (Programmation Python, POO et récursivité)
Partie A : programmation orientée objet

1. ⋆ Un attribut possible : `self.itineraire`
   ⋆ Une méthode possible : `remplir_grille`
2. On obtient `a = 7` et `b = 4`.
3. La méthode suivante convient :

```
def remplir_grille(self):
    i, j = 0, 0
    self.grille[0][0] = 'S'
    for direction in self.itineraire:
        if direction == 'D':
            i = i + 1
        elif direction == 'B':
            j = j + 1
        self.grille[i][j] = '*'
    self.grille[self.largeur][self.longueur] = 'E'
```

4. La méthode suivante convient :

```
def get_dimensions(self):
    return (self.longueur, self.largeur)
```

5. La méthode suivante convient :

```
def tracer_chemin(self):
    self.remplir_grille()
    for l in self.grille:
        print(l)
```

Partie B : génération aléatoire d'itinéraires

6. Le code suivant convient :

```
def itineraire_aleatoire(m, n):
    itineraire = ''
    i, j = 0, 0
    while i != m and j != n:
        dir = random.choice(['D', 'B'])
        itineraire = itineraire + dir
        if dir == 'D':
            j = j + 1
        else :
            i = i + 1
    if i == m:
        itineraire = itineraire + 'D'*(n-j)
    if j == n:
        itineraire = itineraire + 'B'*(m-i)
    return itineraire
```

Partie C : calcul du nombre de chemins possibles

7. Le tableau de dimension $1 \times n$ est un tableau en une dimension. Le chemin est donc une ligne (on peut uniquement aller vers le bas), on a donc $N(1,n) = 1$.
8. Quand nous sommes sur la case de coordonnées $(m;n)$, nous venons soit d'en haut (donc de la case de coordonnées $(m;n-1)$), soit de la gauche (donc de la case de coordonnées $(m-1;n)$). Le nombre de chemins qui permettent d'atteindre la case de coordonnées $(m;n)$ est donc égal au nombre de chemins qui permettent d'atteindre la case de coordonnées $(m;n-1)$ plus le nombre de chemins qui permettent d'atteindre la case de coordonnées $(m-1;n)$. D'où : $N(m,n) = N(m-1,n) + N(m,n-1)$
9. La fonction suivante convient :

```
def nombre_chemins(m, n):
    if m == 1 or n == 1:
        return 1
    else:
        return nombre_chemins(m-1, n) + nombre_chemins(m, n-1)
```

Exercice 2 (POO, récursivité, arbres binaires et OS)
Partie A

1. On peut, par exemple, utiliser l'une des commandes suivantes :
   - `ls documents`
   - `cd documents`, puis `ls`
2. Le répertoire `multimedia` (et tout ce qu'il contient) se trouve désormais dans le répertoire `documents`.
3. La classe `Arbre` permet de modéliser un arbre binaire (attributs `self.gauche` et `self.droit`). Comme l'arbre de la figure 1 n'est pas un arbre binaire (on peut avoir plus de deux enfants), il n'est pas possible d'utiliser la classe `Arbre` pour modéliser l'arbre de la figure 1.
4. On a un parcours de type préfixe.
5. Avec un parcours en largeur, on obtient :
   `home-documents-multimedia-cours-administratif-personnel-images-videos-films`

Partie B

6. Le code suivant convient :

```
def est_vide(self):
    return len(self.fils) == 0
```

7. Le code suivant convient :

```
var_films = Dossier('films', [])
var_videos = Dossier('videos', [var_films])
var_images = Dossier('images', [])
var_multimedia = Dossier('multimedia', [var_videos, var_images])
```

8. Le code suivant convient :

```
def parcours(self):
    print(self.nom)
    for f in self.fils:
        f.parcours()
```

9. Lors des appels récursifs, on fini toujours par rencontrer un dossier vide (une feuille de l'arbre), ce qui provoque l'arrêt des appels récursifs. Donc la méthode parcours se termine bien.
10. Le code suivant convient :

```
def parcours(self):
    for f in self.fils:
        f.parcours()
    print(self.nom)
```

11. La commande `ls` affiche uniquement les descendants directs alors que la méthode `parcours` affiche aussi les descendants des descendants.
12. Le code suivant convient :

```
def mkdir(self, nom):
    nvx = Dossier(nom, [])
    self.fils.append(nvx)
```

13. Le code suivant convient :

```
def contient(self, nom_dossier):
    file = []
    for d in self.fils:
        if d.nom == nom_dossier:
            return True
        file.append(d)
    while len(file) != 0:
        dossier = file.pop(0)
        for d in dossier.fils:
            if d.nom == nom_dossier:
                return True
            file.append(d)
    return False
```

14. Il est possible de légèrement modifier le programme de la question 13 : à la place de renvoyer `True`, on renverrait le nom du dossier parent (`self.nom` pour la première boucle et `dossier.nom` pour la seconde boucle). Le code suivant convient donc :

```
def parent(self, nom_dossier):
    file = []
    for d in self.fils:
        if d.nom == nom_dossier:
            return self.nom
        file.append(d)
    while len(file) != 0:
        dossier = file.pop(0)
        for d in dossier.fils:
            if d.nom == nom_dossier:
                return dossier.nom
            file.append(d)
    return False
```

15. On pourrait ajouter un attribut `self.parent` qui contiendrait le dossier parent.

Exercice 3 (Codage binaire, bases de données et SQL)
Partie A : encodage binaire

1. 11 en base 10 correspond à 1011 en binaire. Le bit de rang 0 (bit de poids faible) nous indique que c'est un équipier, le bit de rang 1 nous indique que c'est un chef d'équipe, le bit de rang 2 nous indique que ce n'est pas un chef d'agrès et le bit de rang 3 nous indique que c'est un conducteur. Par conséquent, il s'agitd bien 'uun chef d'équipe conducteur.
2. Un chef d'agrès conducteur a pour code binaire 1111, ce qui correspond à 15 en décimale
3. 4 en décimale correspond à 100. On aurait donc un chef d'agrès qui ne serait ni chef d'équipe et ni équipier, ce qui est impossible d'après l'énoncé.
4. En passant à un octet (8 bits), on aurait donc 4 bits supplémentaires et on pourrait ainsi coder quatre aptitudes supplémentaires.
5. Si on part du principe qu'il faut 10 caractères pour une aptitude et qu'un caractère est codé sur un octet, il faut donc 10 octets pour une aptitude et donc 40 octets pour 4 aptitudes. L'utilisation d'un octet pour coder quatre aptitudes nous fait donc « économiser » 39 octets, soit environ 98% d'économie mémoire (si on avait économisé 20 octets on aurait eu une économie de 50%, ici nous avons plus). Le seul choix possible est donc 98%).

Partie B

6. La clé primaire permet de distinguer chaque entrée car elle est unique pour chaque entrée. La clé étrangère permet d'effectuer une jointure entre deux tables, c'est-à-dire permet de relier deux tables entre elles.
7. Cette requête génère une erreur car il n'existe pas d'agrès ayant pour identifiant `1` dans la table `agres`.
8. La requête suivante convient :

```
UPDATE intervention
SET heure = '10:44:06'
WHERE jour = '2024-02-15' AND heure = '01:44:06'
```

9. On a le résultat suivant :

Charlot
Red
Kevin

10. La requête suivante convient :

```
SELECT nom
FROM personnel
WHERE qualif >= 16 AND actif = 1
```

11. La requête A renvoie `2` et la requête B renvoie `1`
12. La requête suivante convient :

```
SELECT DISTINCT(nom)
FROM personnel
JOIN agres ON idchefagres = matricule
WHERE jour = '2024-02-15'
```

13. La requête suivante convient :

```
SELECT DISTINCT(nom)
FROM moyen
JOIN agres ON moyen.idagres = agres.id
JOIN intervention ON moyen.idinter = intervention.id
JOIN personnel ON agres.idchefagres = personnel.id
WHERE intervention.jour = '2024-06-11'
```