25NSIJ2PO1

Exercice 1 (6 points)

Cet exercice porte sur la programmation orientée objet, l'algorithmique et la programmation en Python.

Partie A

1. On écrit le constructeur de la classe `Carte`.

```
class Carte:
    def __init__(self, valeur):
        self.valeur = valeur
        self.TdB = self.calcul_TdB()
```

2. On écrit le code de la méthode `calcul_TdB`.

```
    def calcul_TdB(self):
        TdB = 0
        if self.valeur %11 == 0:
            TdB += 5
        if self.valeur %10 == 0:
            TdB += 3
        if self.valeur %10 == 5:
            TdB += 2
        if TdB == 0:
            TdB = 1
        return TdB
```

3. On écrit le code de la méthode `est_superieure_a`.

```
    def est_superieure_a(self, autre):
        return self.valeur > autre.valeur
```

Partie B

4. On complète le code de la classe `Paquet`.

```
5       def afficher(self):
6           for carte in self.contenu:
7               print(carte.valeur)
8
9       def ajouter_carte(self, carte):
10          self.contenu.append(carte)
```

5. On écrit le code de la méthode `nombre_TdB`.

```
5       def nombre_TdB(self):
6           TdB=0
7           for carte in self.contenu:
8               TdB += carte.TdB
9           return TdB
```

6. On écrit le code de la méthode `distribuer`.

```
5       def distribuer(self, nbr):
6           liste = []
7           for k in range(nbr):
8               liste.append(Paquet())
9           for i in range(10):
10              for k in range(nbr):
11                  liste[k].append(self.contenu[10*i+k])
12          self.contenu = self.contenu[10*nbr:]
13          return liste
```

Partie C

7. L'instruction `J1 = Joueur("Joueur 1", L[0]` convient.
8. On complète les lignes 3, 7 et 9 du script.

```
1  from random import *
2  # créer les 104 cartes du jeu initial grâce à une liste par
3  compréhension
4  jeu = [Carte(i) for i in range (1, 105)]
5  # mélanger cette liste de cartes
6  shuffle(jeu)
7  # instancier le paquet de cartes avec cette liste de cartes
8  jeu_initial = Paquet(jeu)
9  # distribuer 10 cartes aux deux joueurs que l'on instancie en les nommant `J1` et `Ordi`.
10 distri = jeu_initial.distribuer(2)
11 Ordi = Joueur("Ordi", distri[0])
12 J1 = Joueur("J1", distri[1])
```

Exercice 2 (6 points)

Cet exercice porte sur la programmation Python, la programmation orientée objet, les bases de données relationnelles et les requêtes SQL.

Partie A

1. La requête `SELECT nom FROM champignon WHERE lamelle = "oui" AND couleur = "orange";` convient.
2. On peut utiliser la requête `SELECT nom FROM champignon WHERE pied_max = 0 AND chapeau_min <= 15 AND chapeau_max >=15;`.
3. La clé étrangère de la table `champignon` est `id_ordre` qui fait référence à l'attribut `id` de la table `ordre`.
4. La requête `SELECT nom FROM champignon JOIN ordre ON champignon.id_ordre = ordre.id WHERE classe = "agaricomycètes";` convient.
5. La requête `INSERT INTO champignon VALUES (56, "amanite solitaire", 4, "oui", 6, 20, 4, 10);` convien
6. Le schéma relationnel est :
   `champignon(id, nom, #id_ordre, #id_toxicite, lamelle, couleur, chapeau_min, chapeau_max, pied_mi pied_max)`
   `ordre(id, nom, classe)`
   `toxicite(id_tox, type, effets)`
7. La requête `UPDATE champignon SET id_toxicite = 1 WHERE nom = "amanite citrine";` convient.
8. La requête `SELECT nom FROM champignon JOIN ordre ON champignon.id_ordre = ordre.id JOIN toxicite ON champignon.id_toxicite = toxicite.id_tox WHERE ordre.nom = "amanitales" AND type = "très toxique";` convient.

Partie B

9. On complète le programme.

```
for e in liste_champi:
    if e.saison == 'été':
        print(e.nom)
```

10. Quentin compare la chaîne de caractères `'12 minutes à feu moyen'` avec la chaîne de caractères `'feu moyen'`. Ces chaînes sont différentes.
11. On modifie le programme en utilisant la fonction `recherche_textuelle`.

```
1  for c in liste_champi:
2      if c.nom == 'Lactaire délicieux':
3          return recherche_textuelle(c.cuisson, 'feu moyen')
```

Exercice 3 (8 points)

Dans cet exercice il est question de tableaux, de dictionnaires, de recherche de chemins dans un graphe, de piles, de files et de POO.

Partie A

1. Une pile est une structure linéaire où les insertions et les extractions se font sur le principe LIFO last-in first-out soit dernier entré-premier sorti.
2. Une file est une structure linéaire où les insertions et les extractions se font sur le principe FIFO first-in first-out soit premier entré-premier sorti.
3. La relation de voisinage est symétrique donc un graphe non orienté est adapté.
4. On dessine le graphe.

Partie B

5. On complète la fonction `chaine_vers_tab`.

```
def chaine_vers_tab(mot):
    tab_lettres = []
    for lettre in mot :
        tab_lettres.append(lettre)
    return tab_lettres
```

6. La fonction `distance` renvoie le nombre de lettres de `mot1` qui restent quand on a enlevé les lettres présentes dans `mot2`, ce qui correspond bien à la notion de distance utilisée.

```

```

7. On complète la fonction `renvoie_voisins`.

```
def renvoie_voisins(mot):
    tab_voisins = []
    for voisin_possible in TAB_MOTS :
        if distance(mot, voisin_possible) == 1:
            tab_voisins.append(voisin_possible)
    return tab_voisins
```

Partie C

8. On déroule la fonction pas-à-pas, en représentant la file par une liste Python où on enfile en dernière position avec `append()` et où on défile avec `pop(0)`.

Init. `parent={'mars': None}` et `file_voisins=['mars']`

Tour 1 `mot='mars'`, `parent = {'mars' : None, 'gars': 'mars', 'mors': 'mars'}`,
`file_voisins=['gars', 'mors']`

Tour 2 `mot='gars'`, `parent = {'mars' : None, 'gars': 'mars', 'mors': 'mars'}`,
`file_voisins=['mors']`

Tour 3 `mot='mors'`, `parent = {'mars' : None, 'gars': 'mars', 'mors': 'mars', 'ours': 'mors'}`,
`file_voisins=['ours']`

Tour 4 `mot='ours'`,
`parent = {'mars' : None, 'gars': 'mars', 'mors': 'mars', 'ours': 'mors', 'purs': 'ours'}`,
`file_voisins=['purs']`

Tour 5 `mot='purs'`, qui est le mot final donc on sort de la boucle ; la fonction renvoie
`{'mars' : None, 'gars': 'mars', 'mors': 'mars', 'ours': 'mors', 'purs': 'ours'}`.

9. On complète la fonction `renvoie_pile`.

```
def renvoie_pile(parent, mot_final):
    ma_pile = Pile()
    mot = mot_final
    while mot != None :
        ma_pile.enmpiler(mot)
        mot = parent[mot]
    return ma_pile
```

10. On complète la fonction `construit_chemin`.

```
def construit_chemin(ma_pile):
    tab = []
    while not ma_pile.est_vide():
        mot = ma_pile.depiler()
        tab.append(mot)
    return tab
```

11. On code la fonction `chercher_chemin`.

```
def chercher_chemin(mot_depart, mot_final):
    parent = dic_parent(mot_depart, mot_final)
    pile = renvoie_pile(parent, mot_final)
    return construit_chemin(pile)
```