

Exercice 1

Graphes et parcours de graphe (extrait AN1 J1 Ex 2)

Le dictionnaire suivant est une implémentation d'un graphe :

```

1 graphe = {'G': ['J', 'N'],
2           'J': ['G', 'Y', 'E', 'L'],
3           'Y': ['J', 'E', 'N'],
4           'E': ['J', 'Y', 'N'],
5           'N': ['G', 'Y', 'E'],
6           'M': ['A'],
7           'A': ['M'],
8           'L': ['J']}

```

1. Donner la définition de *graphe*
2. Quelles sont les clés du dictionnaire ?
3. Comment, à partir de ce dictionnaire parvient-on à atteindre la liste ['G', 'Y', 'E', 'L'] ?
4. Représenter ce graphe.

On donne les 2 algorithmes de parcours en profondeur d'un graphe :

```

1 def DFS(d,s,visited=[]):
2     visited.append(s)
3     for v in d[s]:
4         if v not in visited:
5             DFS(d,v)
6     return visited

```

```

1 def DFS_ite(d,s,visited=[], stack=[]):
2     stack.append(s)
3     while stack:
4         v = stack.pop()
5         if v not in visited:
6             visited.append(v)
7             unvisited = [n for n in d[v] if n not in visited]
8             stack.extend(inverse(unvisited))
9     return visited

```

1. Expliquer ce qu'est un parcours en profondeur.
2. Laquelle de ces fonctions est de type itératif ? Récursif ? Pourquoi ?
3. Ecrire dans chaque cas l'instruction qui appelle la fonction, afin de faire le parcours en profondeur du graphe, à partir du sommet L.
4. Quelles seront les listes retournées par chacune de ces fonctions ?

On donne la fonction de parcours en largeur d'un graphe :

```

1 from collections import deque
2
3 def BFS_ite(d,s,visited=[], queue=deque()):
4     queue.append(s)
5     while queue:
6         v = queue.popleft()
7         if v not in visited:
8             visited.append(v)
9             unvisited = [n for n in d[v] if n not in visited]
10            queue.extend(unvisited)
11            print(queue)
12    return visited

```

Remarque :

La librairie *collection* apporte la structure de donnée dequeue qui se comporte comme une *File*, avec la méthode de classe `popleft()` qui permet de retirer le PREMIER element de la File, en temps constant :

```

1 from collections import deque
2 # Declaring deque
3 queue = deque(['name','age','DOB'])
4 print(queue.popleft())
5 # affiche name
6 print(queue)
7 # affiche deque(['age', 'DOB'])

```

1. Quelles sont les différences avec la fonction de parcours en profondeur?
2. Quelles sera la liste retournée par cette fonction?

Exercice 2

Listes chaînées (extrait Am Nord J2 2024 Ex 3)

Un réseau PaP est utilisé pour effectuer des transactions financières en monnaie numérique *nsicoin* entre les trois utilisateurs. Pour cela, on crée la classe Transaction ci-dessous :

```

1 class Transaction:
2     def __init__(self, expéditeur, destinataire, montant):
3         self.expéditeur = expéditeur
4         self.destinataire = destinataire
5         self.montant = montant

```

2. Dans cycle de transactions, Alice envoie dix *nsicoin* à Charlie puis Bob envoie cinq *nsicoin* à Alice. Écrire la liste Python correspondante à ces transactions. Les transactions réalisées pendant cet intervalle de temps sont regroupées par ordre d'apparition dans une liste Python.

Pour garder une trace de toutes les transactions effectuées, on utilise une liste chaînée de blocs (ou blockchain) dont le code Python est fourni ci-dessous. Toutes les dix minutes un nouveau bloc contenant les nouvelles transactions est créé et ajouté à la blockchain.

```

1 class Bloc:
2     def __init__(self, liste_transactions, bloc_precedent):
3         self.liste_transactions = liste_transactions
4         self.bloc_precedent = bloc_precedent # de type Bloc
5
6 class Blockchain:
7     def __init__(self):
8         self.tete = self.creer_bloc_0()
9
10    def self.creer_bloc_0(self):
11        """
12        Crée le premier bloc qui distribue 100 nsicoin à tous les
13        utilisateurs
14        (un pseudo-utilisateur Genesis est utilisé comme
15        expéditeur)
16        """
17        liste_transactions = [
18            Transaction("Genesis", "Alice", 100),
19            Transaction("Genesis", "Bob", 100),
20            Transaction("Genesis", "Charlie", 100)
21        ]
22        return Bloc(liste_transactions, None)

```

3. La figure 1 représente les trois premiers blocs d'une Blockchain. Expliquer pourquoi la valeur de l'attribut `bloc_precedent` du bloc0 est None.

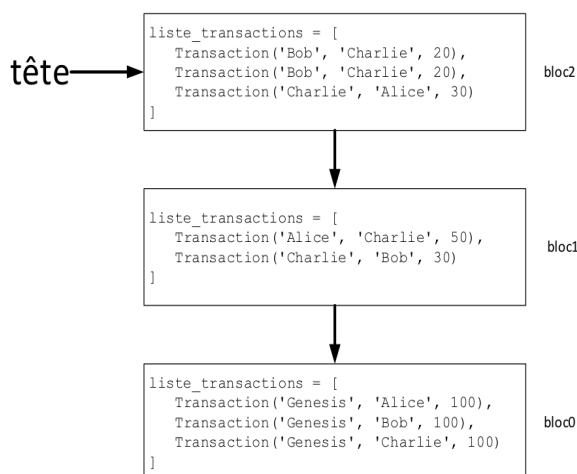


FIGURE 1 – Blockchain

- À l'aide des classes `Bloc` et `Blockchain`, écrire le code Python permettant de créer un objet `ma_blockchain` de type `Blockchain` représenté par la figure 1.
- Donner le solde en nsicoin de Bob à l'issue du bloc2.
- On souhaite doter la classe `Blockchain` d'une méthode `ajouter_bloc` qui prend en paramètre la liste des transactions du dernier cycle et l'ajoute dans un nouveau bloc. Écrire le code Python de cette méthode ci-dessous.

```

1 def ajouter_bloc(self, liste_transactions):
2     # A compléter

```

9. On souhaite doter la classe Bloc d'une nouvelle méthode `calculer_solde` permettant de renvoyer le solde à l'issue de ce bloc. Recopier et compléter sur votre copie le code Python de cette méthode :

```

1 def calculer_solde(self, utilisateur):
2     if self.precedent is None: # cas de base
3         solde = 0
4     else:
5         solde = ... # appel récursif : calcul du solde au bloc précé
6         dent
7         for transaction in bloc.liste_transactions
8             if ... == utilisateur:
9                 solde = solde - ....
10            elif ... :
11                ...
12        return solde

```

10. Écrire l'appel à la fonction `calculer_solde` permettant de calculer le solde actuel de Alice

Exercice 3

Dictionnaires (extraits 2024 ANJ1 Ex3 et sujet 0B Ex 3)

3.1 Import et traitement csv

Voici un exemple d'utilisation de `DictReader` du module `csv` :

```

1 fichier exemple.csv
2
3 champ1, champ2
4 a,7
5 b,8
6 c,9

```

code Python

```

1 import csv
2 with open('exemple.csv','r') as fichier:
3     donnees = list(csv.DictReader(fichier,delimiter=','))
4 print(donnees)

```

Affichage :

```

1 [{'champ1': 'a', 'champ2': '7'},
2  {'champ1': 'b', 'champ2': '8'},
3  {'champ1': 'c', 'champ2': '9'}]

```

Voici un extrait du fichier `flashcards.csv` :

```
1 discipline;chapitre;question;reponse
2 histoire;crise de 1929; jeudi noir - date;24 octobre 1929
3 histoire;crise de 1929; jeudi noir - quoi;krach boursier
4 histoire;2GM;l'Axe;Allemagne, Italie, Japon
5 histoire;2GM;les Alliés;Chine, Etats-Unis, France, Royaume-Uni, URSS
6 philosophie;travail;Marx;alienation de l'ouvrier
```

1. Ecrire une fonction `charger` qui prend `nom_fichier` comme paramètre et charge le contenu d'un fichier pour retourner un tableau (liste de dict) comme dans l'exemple ci-dessus.
2. Ecrire l'instruction qui charge `flashcards.csv` dans la variable `flashcard`, de type tableau.
3. Ecrire une fonction `choix_discipline` qui prend pour paramètres `donnees`, un tableau, et `disc`, la discipline choisie. Cette fonction retourne un tableau à partir de la sélection sur les disciplines égales à `disc`.
4. Ecrire une fonction `question_reponse` qui prend pour paramètres `donnees`, `disc` et qui retourne une liste de listes constituées de couples (question, réponse) relatifs à la discipline choisie. Utiliser la fonction `choix_discipline`.

3.2 Livres et notes maxi, recherche dans une table

Dans cette première partie, on utilise un dictionnaire Python. On considère le programme suivant :

```
1 dico_livres = {'id' : [1, 2, 14, 4, 5, 8, 7, 15, 9, 10],
2               'titre' : ['1984', 'Dune', 'Fondation', 'Ubik', 'Blade
3                           Runner', 'Les Robots', 'Ravage', 'Chroniques martiennes', 'Dragon dé
4                           chu', 'Fahrenheit 451'],
5               'auteur' : ['Orwell', 'Herbert', 'Asimov', 'K.Dick', 'K.
6                           Dick', 'Asimov', 'Barjavel', 'Bradbury', 'Hamilton', 'Bradbury'],
7               'ann_pub' : [1949, 1965, 1951, 1953, 1968, 1950, 1943, 1950,
8                           2003, 1953],
9               'note' : [10, 8, 9, 9, 8, 10, 6, 7, 8, 8]
10              }
11 a = dico_livres['note']
12 b = dico_livres['titre'][2]
```

1. Déterminer les valeurs des variables `a` et `b` après l'exécution de ce programme.

La fonction `titre_livre` prend en paramètre un dictionnaire (de même structure que `dico_livres`) et un identifiant, et renvoie le titre du livre qui correspond à cet identifiant. Dans le cas où l'identifiant passé en paramètre n'est pas présent dans le dictionnaire, la fonction renvoie `None`.

```
1 def titre_livre(dico, id_livre):
2     for i in range(len(dico['id'])):
3         if dico['id'][i] == id_livre :
4             return dico['titre'][i]
5     return None
```

2. Recopier et compléter les lignes 3, 4 et 5 de la fonction `titre_livre`.

3. Écrire une fonction `note_maxi` qui prend en paramètre un dictionnaire `dico` (de même structure que `dico_livres`) et qui renvoie la note maximale.
4. Écrire une fonction `livres_note` qui prend en paramètre un dictionnaire `dico` (de même structure que `dico_livres`) et une note `n`, et qui renvoie la liste des titres des livres ayant obtenu la note `n` (on rappelle que `t.append(a)` permet de rajouter l'élément `a` à la fin de la liste `t`).
5. Écrire une fonction `livre_note_maxi` qui prend en paramètre un dictionnaire `dico` (de même structure que `dico_livres`) et qui renvoie la liste des titres des livres ayant obtenu la meilleure note sous la forme d'une liste Python.

Exercice 4

Arbres, parcours BFS (extrait 2024 centres etrangers J2, Ex 3)

4.1 Partie B

On considère l'arborescence de fichiers de la figure suivante :

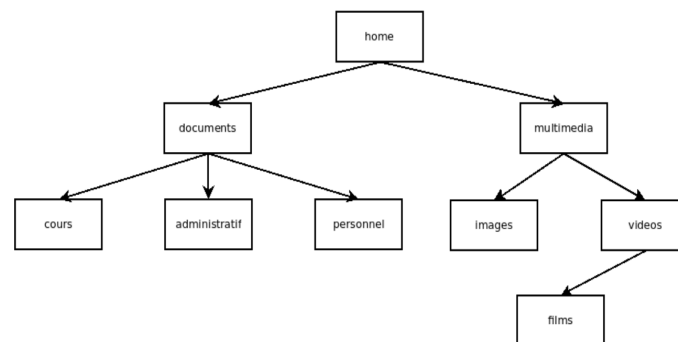


FIGURE 2 – arborescence fichiers

on propose l'implémentation suivante. L'attribut `films` est une variable de type `list` contenant tous les dossiers fils. Cette liste est vide dans le cas où le dossier est vide.

```

1 class Dossier:
2     def __init__(self, nom, liste):
3         self.nom = nom
4         self.films = liste # liste d'objets de la classe Dossier
  
```

1. Écrire le code Python d'une méthode `est_vide` qui renvoie `True` lorsque le dossier est vide et `False` sinon.
2. Écrire le code Python permettant d'instancier une variable `var_multimedia` de la classe `Dossier` représentant le dossier `multimedia` de la figure précédente. Attention : cela nécessite d'instancier tous les nœuds du sous-arbre de racine `multimedia`.
3. Recopier et compléter sur votre copie le code Python de la méthode `parcours` suivante qui affiche les noms de tous les descendants d'un dossier en utilisant l'ordre préfixe.

```

1 def parcours(self):
2     print(...)
  
```

```
3     for f in ...:
4         ...
```

4. Justifier que cette méthode parcours termine toujours sur une arborescence de fichiers.
5. Proposer une modification de la méthode parcours pour que celle-ci effectue plutôt un parcours suffixe (ou postfixe).
6. Expliquer la différence de comportement entre un appel à la méthode parcours de la classe Dossier et une exécution de la commande UNIX `ls`

On considère la variable `var_videos` de type `Dossier` représentant le dossier `videos` de la figure précédente. On souhaite que le code Python `var_videos.mkdir("documentaires")` crée un dossier `documentaires` vide dans le dossier `var_videos`.

7. Écrire le code Python de la méthode `mkdir`.
8. Écrire en Python une méthode `contient(self, nom_dossier)` qui renvoie `True` si l'arborescence de racine `self` contient au moins un dossier de nom `nom_dossier` et `False` sinon.
9. Avec l'implémentation de la classe `Dossier` de cette partie, expliquer comment il serait possible de déterminer le dossier parent d'un dossier donné dans une arborescence donnée. On attend ici l'idée principale de l'algorithme décrite en français. On ne demande pas d'implémenter cet algorithme en Python.
10. Proposer une modification dans la méthode `__init__` de la classe `Dossier` qui permettrait de répondre à la question précédente beaucoup plus efficacement et expliquer votre choix.