Partie 1

Partie 1 : Les Files (FIFO)

1.1 Exercice 1 : Découverte des files ★

Une file est une structure de données FIFO (First In, First Out). On dispose d'une file implémentée avec une liste Python et des fonctions `enfiler(f, element)` et `defiler(f)`.

Questions :

1. Quelle est la différence entre une file et une pile ?
2. On effectue les opérations suivantes sur une file vide `f` :

```
enfiler(f, 5)
enfiler(f, 8)
enfiler(f, 3)
x = defiler(f)
enfiler(f, 12)
```

Quel est le contenu de la file après ces opérations ? Quelle est la valeur de `x` ?

1.2 Exercice 2 : Utilisation d'une file ★★

On simule une file d'attente à une caisse. Voici l'implémentation fournie :

```
def creer_file():
    return []

def est_vide(f):
    return len(f) == 0

def enfiler(f, element):
    f.append(element)

def defiler(f):
    if est_vide(f):
        raise IndexError("File vide")
    return f.pop(0)
```

Question : Écrire une fonction `taille_file(f)` qui renvoie le nombre d'éléments dans la file.

1.3 Exercice 3 : Application - Gestion d'imprimante ★★

Une imprimante reçoit des documents à imprimer. Chaque document est représenté par son nom.

Question : Écrire une fonction `traiter_impression(documents)` qui :

- Prend en paramètre une liste de noms de documents

- Utilise une file pour gérer l'ordre d'impression
- Affiche "Impression de : [nom]" pour chaque document dans l'ordre FIFO
- Renvoie le nombre de documents traités

Exemple :

```
traiter_impression(["rapport.pdf", "photo.jpg", "facture.pdf"])
# Affiche :
# Impression de : rapport.pdf
# Impression de : photo.jpg
# Impression de : facture.pdf
# Renvoie : 3
```

Partie 2

Partie 2 : Les Listes Chaînées

2.1 Exercice 4 : Compréhension ★

Une liste chaînée est composée de maillons. Chaque maillon contient une valeur et une référence vers le maillon suivant.

Questions :

1. Quel est l'avantage d'une liste chaînée par rapport à un tableau ?
2. Dessiner une représentation schématique d'une liste chaînée contenant les valeurs 7, 3, 9, 1.

2.2 Exercice 5 : Liste chaînée avec dictionnaire simple ★★

On implémente une liste chaînée à l'aide d'un dictionnaire où chaque clé correspond à une valeur et pointe vers la valeur suivante.

Exemple de liste chaînée contenant 10, 20, 30 :

```
liste = {10: 20, 20: 30, 30: None}
```

Ici, 10 est la première valeur (la tête de liste), 20 suit 10, 30 suit 20, et None indique la fin.

Questions :

a) Écrire une fonction `creer_liste(valeur)` qui crée et renvoie un dictionnaire représentant une liste chaînée contenant uniquement `valeur`.

b) Écrire une fonction `ajouter_fin(liste, tete, valeur)` qui ajoute une nouvelle valeur à la fin de la liste chaînée. La fonction prend en paramètres : - `liste` : le dictionnaire représentant la liste chaînée - `tete` : la valeur du premier élément de la liste - `valeur` : la nouvelle valeur à ajouter

La fonction doit parcourir la liste jusqu'au dernier élément (celui qui pointe vers `None`), puis ajouter la nouvelle valeur.

Exemple d'utilisation :

```
liste = creer_liste(10)
ajouter_fin(liste, 10, 20)
ajouter_fin(liste, 10, 30)
# liste contient maintenant : {10: 20, 20: 30, 30: None}
```

c) Écrire une fonction `afficher_liste(liste, tete)` qui affiche toutes les valeurs de la liste dans l'ordre, en partant de `tete`.

d) Question de réflexion : Quelles sont les limites de cette implémentation ? Que se passe-t-il si on veut stocker deux fois la même valeur dans la liste ?

2.3 Exercice 6 : Liste chaînée avec dictionnaires imbriqués ★★

On implémente une liste chaînée à l'aide de dictionnaires. Chaque maillon est un dictionnaire avec deux clés : - `'valeur'` : la donnée stockée - `'suivant'` : le maillon suivant (ou `None` si c'est le dernier)

Exemple de liste chaînée contenant 5, 8, 3 :

```
liste = {
    'valeur': 5,
    'suivant': {
        'valeur': 8,
        'suivant': {
            'valeur': 3,
            'suivant': None
        }
    }
}
```

Questions :

a) Écrire une fonction `creer_maillon(valeur)` qui crée et renvoie un maillon contenant `valeur` et dont le suivant est `None`.

b) Écrire une fonction `ajouter_fin(liste, valeur)` qui ajoute un nouveau maillon contenant `valeur` à la fin de la liste chaînée. La fonction doit parcourir la liste jusqu'au dernier maillon, puis modifier son champ `'suivant'`.

Exemple d'utilisation :

```
liste = creer_maillon(10)
ajouter_fin(liste, 20)
ajouter_fin(liste, 30)
# liste contient maintenant : 10 -> 20 -> 30
```

c) Écrire une fonction `afficher_liste(liste)` qui affiche toutes les valeurs de la liste dans l'ordre.

d) Comparer cette implémentation avec celle de l'exercice 5. Quels sont les avantages et inconvénients de chaque approche ?

2.4 Exercice 7 : Liste chaînée en POO ★★★

On souhaite implémenter une liste chaînée en programmation orientée objet.

Implémentation fournie :

```
class Maillon:
    def __init__(self, valeur):
        self.valeur = valeur
        self.suivant = None

class ListeChainee:
    def __init__(self):
        self.tete = None

    def est_vide(self):
        return self.tete is None

    def ajouter_debut(self, valeur):
        nouveau = Maillon(valeur)
        nouveau.suivant = self.tete
        self.tete = nouveau
```

Questions :

a) Compléter la classe `ListeChainee` avec une méthode `afficher()` qui affiche tous les éléments de la liste.

b) Ajouter une méthode `longueur()` qui renvoie le nombre d'éléments dans la liste.

c) Ajouter une méthode `rechercher(valeur)` qui renvoie `True` si la valeur est présente dans la liste, `False` sinon.

d) Ajouter une méthode `supprimer_debut()` qui supprime le premier élément de la liste et renvoie sa valeur. Si la liste est vide, lever une exception.

2.5 Exercice 8 : Inversion d'une liste chaînée ★★★

En utilisant la classe `ListeChainee` de l'exercice 7, écrire une méthode `inverser()` qui inverse l'ordre des éléments de la liste chaînée.

Exemple :

```
liste = ListeChainee()
liste.ajouter_debut(3)
liste.ajouter_debut(8)
liste.ajouter_debut(5)
# Liste : 5 -> 8 -> 3

liste.inverser()
# Liste : 3 -> 8 -> 5
```

Contrainte : Utiliser uniquement les opérations sur les maillons, sans créer de nouvelle liste.

2.6 Exercice 9 : Fusion de deux listes triées ★★★★

On dispose de deux listes chaînées triées par ordre croissant. Écrire une fonction `fusionner(liste1, liste2)` qui crée et renvoie une nouvelle liste chaînée contenant tous les éléments des deux listes, toujours triés par ordre croissant.

Exemple :

```
# liste1 : 1 -> 4 -> 7
# liste2 : 2 -> 5 -> 9
# Résultat : 1 -> 2 -> 4 -> 5 -> 7 -> 9
```

Partie 3

Notes et Barème indicatif

- ★ : Exercice facile
- ★★ : Exercice moyen
- ★★★ : Exercice difficile
- ★★★★ : Exercice très difficile

Partie 4

Corrections

4.1 Exercice 6 : liste avec dictionnaires imbriqués

questions a) b) et c)

```
def creer_maillon(v):
    return {'valeur':v,'suivant':None}

def ajoute_fin(liste, valeur):
    while liste['suivant'] is not None:
        liste = liste['suivant']
    liste['suivant'] = creer_maillon(valeur)

def afficher_liste(liste):
    while liste['suivant'] is not None:
        print(liste['valeur'])
        liste = liste['suivant']
    print(liste['valeur'])```
```

Exemple d'utilisation :

```
D = creer_maillon(1)
ajoute_fin(D,2)
ajoute_fin(D,3)
```

La liste est alors :

```
{'valeur': 1,
 'suivant': {'valeur': 2,
                        'suivant': {'valeur': 3,
                                                'suivant': None}}}
```

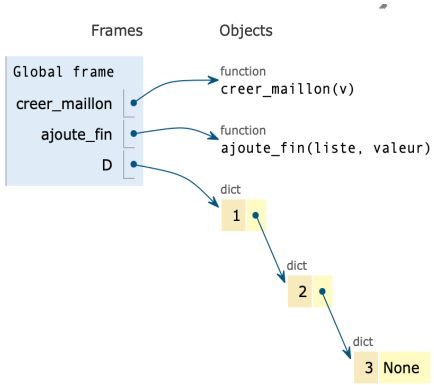

FIGURE 1 – visualisation pythontutor

question c)

```
>>> afficher_liste(D)
1
2
3
```

4.2 Exercice 7 : Liste chainée en POO

Questions a), b), c), d)

```
class Maillon:
    def __init__(self,valeur):
        self.valeur = valeur
        self.suivant = None

class Liste_Chainee:
    def __init__(self):
        self.tete = None

    def est_vide(self):
        return self.tete is None

    def ajouter_debut(self,valeur):
        nouveau = Maillon(valeur)
        nouveau.suivant = self.tete
        self.tete = nouveau

    def afficher(self):
        M = self.tete # M est un objet de la classe Maillon
        while M.suivant is not None:
            print(M.valeur)
            M = M.suivant
        print(M.valeur)

    def longueur(self):
        M = self.tete # M est un objet de la classe Maillon
        i = 0
        while M.suivant is not None:
```

```
            i = i + 1
            M = M.suivant
        i = i+1
        return i

    def rechercher(self,valeur):
        M = self.tete # M est un objet de la classe Maillon
        while M.suivant is not None:
            if M.valeur == valeur:
                return True
            M = M.suivant
        if M.valeur == valeur:
            return True
        return False

    def supprimer_debut(self):
        valeur = self.tete.valeur
        assert self.tete is not None, "impossible de supprimer un
   element dans liste vide"
        self.tete = self.tete.suivant
        return valeur
```

Quelques tests : Construction d'une liste 1->2->3

```
L1 = Liste_Chainee()
L1.ajouter_debut(3)
L1.ajouter_debut(2)
L1.ajouter_debut(1)
L1.afficher()
# affiche
1
2
3
```

Exemple d'utilisation des fonctions programmées :

```
>>> L1.tete.suivant.valeur
2
>>> L1.longueur()
3
>>> L1.est_vide()
False
>>> L1.supprimer_debut()
>>> L1.afficher()
2
3
```

4.3 Exercice 8 : Inversion d'une liste chainée

```
    def inverser(self):
        precedent = None
        courant = self.tete
```

```
        while courant is not None:
            # Sauvegarder le suivant avant de modifier le lien
            suivant = courant.suivant

            # Inverser le lien
            courant.suivant = precedent

            # Avancer d'un cran
            precedent = courant
            courant = suivant

        # Le dernier maillon traité devient la nouvelle tête
        self.tete = precedent
```

Remarque : Cette méthode a une complexité O(n) en temps et O(1) en espace (pas de nouvelle liste créée).

4.4 Exercice 9 : fusion de 2 listes triées

Il faudra ajouter les méthodes de classe suivantes à la classe `Liste_chainee`

```
    def ajouter_fin(self,valeur):
        if self.tete is None:
            self.ajouter_debut(valeur)
        else:
            M = self.tete # M est un objet de la classe Maillon
            while M.suivant is not None:
                M = M.suivant
            M.suivant = Maillon(valeur)


    def atteindre_maillon_fin(self):
        assert self.tete is not None, "impossible de parcourir une liste vide"
        M = self.tete # M est un objet de la classe Maillon
        while M.suivant is not None:
            M = M.suivant
        return M
```

Puis programmer la fonction `fusionner` :

```
def fusionner(liste1,liste2):
    liste = Liste_Chainee()
    while not liste1.est_vide() and not liste2.est_vide():
        if liste1.tete.valeur <= liste2.tete.valeur:
            liste.ajouter_fin(liste1.tete.valeur)
            liste1.supprimer_debut()
        else:
            liste.ajouter_fin(liste2.tete.valeur)
            liste2.supprimer_debut()
    if liste1.est_vide():
        fin = liste.atteindre_maillon_fin()
        fin.suivant = liste2.tete
    else:
```

```
        fin = liste.atteindre_maillon_fin()
        fin.suivant = liste1.tete
    return liste
```

Enfin, exemple d'utilisation :

```
L1 = Liste_Chainee()
L1.ajouter_debut(6)
L1.ajouter_debut(3)
L1.ajouter_debut(1)
L2 = Liste_Chainee()
L2.ajouter_debut(8)
L2.ajouter_debut(4)
L2.ajouter_debut(2)
L = fusionner(L1,L2)
L.afficher()
# affiche
1
2
3
4
6
8
```