TD2 : Structures de Données « Piles & Files » avec P.O.O

On suppose qu'il existe une classe **Pile** dans un fichier **Pile.py** permettant **d'utiliser les méthodes** suivantes :

- **p** = Pile( ) : création d'une instance **p** de Pile vide
- **p.**estVide( ) : renvoie un booléen **True** si la pile **p** est vide, **False** sinon.
- **p.**depiler( ) : renvoie l'élément au **sommet** de la pile **p** en le **retirant** de la pile
- **p.**sommet( ) : renvoie l'élément au **sommet** de la pile **p,** sans le retirer.
- **p.**empiler(**x**) : ajoute au sommet de la pile **p** l'élément **x** passé en paramètre
- **`x in p`** : permet d'appeler **`p.__contains__(x)`** **méthode spéciale** existant dans la classe Pile

Exercice 1 : (**Pile** - extrait du Devoir Surveillé 2022 **)**

Attention : Après l'appel de chacune des fonctions suivantes, la pile **p** devra avoir retrouvé son état d'origine

1. Créer une fonction **afficher**(**p**) qui affiche tous les éléments de la pile **p,** elle affiche un élément par ligne en partant du sommet vers la base de **p.**

2. Créer une fonction **get_position(p, x)** qui renvoie la **position** d'un élément **x** dans la pile **p.**
   La position est comptée en partant du sommet de **p.** La position du sommet est égale à **1.**

   Exemple: Pour la pile **p** =

6
7
9
5

   `get_position (p, 6) == 1`
   `get_position (p, 9) == 3`
   `get_position (p, 10) == None`

3. Créer une fonction **get_N_items(p, N)** ayant pour paramètres une pile **p** et un entier **N.** Elle renvoie un tuple contenant les **N** derniers éléments empilés en **haut** de la pile **p**
   Pour l'exemple de pile **p** ci-dessus : **`get_N_items(p, 2) == ( 6,7)`**
   `get_N_items(p, 10) == ( 6 , 7 , 9 , 5 )`

   NB : Si **N** dépasse la taille de la pile, alors on retourne tous les éléments de **p**

Exercice 2 : (File avec POO - extrait de l'examen 2024 **)**

```
class File:
    def __init__(self):          # constructeur qui permet de créer une instance de File vide
        self.data = list( )      # attribut data de type list qui contiendra les éléments de la file

    def enfiler(self, element):  # méthode pour ajouter un élément à la fin de la file
        self.data.append(element)
```

Partie 1 : Dans la classe File (donnée en page 1), ajouter l'implémentation des méthodes suivantes

1.1. `__len__(self)` : renvoie la taille de la file. (méthode spéciale utilisable avec la fonction `len`)

1.2. `estVide(self)` : retourne un booléen indiquant si la file est vide

1.3. `defiler(self)` : retire et renvoie l'élément en tête de la file

1.4. `__contains__(self, element)` : retourne un booléen indiquant si un élément appartient à la file (méthode spéciale utilisable avec l'opérateur `in`)

1.5. `__eq__(self, other)` : retourne un booléen indiquant si les deux files `self` et `other` contiennent les mêmes éléments dans le même ordre. (méthode spéciale utilisable avec l'opérateur `==`)

1.6. `__add__(self, other)` permet de concaténer deux files : Elle créer une nouvelle file qui contient tous les éléments de la première file `(self)` suivis de tous les éléments de la seconde file (`other`) (méthode spéciale utilisable avec l'opérateur `+`)

1.7. `Copie(self)` : retourne une nouvelle file contenant les mêmes éléments que la file **f.**

1.9. À l'extérieur de la classe File, écrire la fonction inverser_file(f) qui inverse les éléments de la file **f** passée en paramètre. On a le droit d'utiliser une instance de Pile (voir page 1)

Partie 2 : Une entreprise agricole souhaite développer une application simple pour gérer les interventions effectuées auprès des agriculteurs. Lorsqu'un agriculteur a besoin d'une intervention, il fait appel à cette entreprise qui attribue la tâche à l'un de ses intervenants (ouvriers, techniciens, ingénieurs) :

- Chaque intervenant peut effectuer **au maximum une intervention par jour**.
- Une intervention est réalisée par un **seul intervenant**.
- Une intervention peut durer **un ou plusieurs jours**.

Les interventions à réaliser seront stockées dans une **file d'attente** (FIFO - Premier Entré, Premier Sorti).

2.1. Définir une classe **Intervention** ainsi que son constructeur qui initialise les attributs suivants :

- **num** : un entier strictement positif identifiant l'intervention de manière unique.
- **cinAg** : le CIN (numéro carte d'identité) de l'agriculteur demandant l'intervention (entier strictement positif de 8 chiffres).
- **cinIn** : le CIN de l'intervenant chargé de l'intervention (entier strictement positif de 8 chiffres).
- **nbj** : un entier strictement positif représentant le nombre de jours nécessaires pour réaliser l'intervention.

2.2. Écrire une méthode **__repr__** qui retourne une chaîne de caractères au format :
« L'intervention (**num**,**cinAg)** est accomplie par **cinIn** en **nbj** jours»

2.3. Écrire une méthode **__eq__** pour comparer deux interventions. Deux interventions sont considérées égales si leurs numéros (**num**) sont identiques.

Partie 3 : Classe Entreprise

3.1.Définir une classe Entreprise ainsi que son constructeur qui initialise les attributs :

- **LI :** une liste des CINs des intervenants de l'entreprise.
- **FA :** une file (File), **initialement vide**, contenant les interventions à réaliser.

3.2. Écrire une méthode **AjoutIntervention** pour ajouter une intervention **In** (instance de la classe Intervention) passée en paramètre à la file des interventions.

3.3. Écrire une méthode **__getitem__** qui, étant donné le CIN d'un intervenant **cinIn**, retourne une liste **LI** contenant toutes les interventions affectées à cet intervenant.

3.4. Écrire une méthode **DictIntervenant** qui retourne un dictionnaire **DI** où :

- Chaque clé est un CIN d'intervenant.
- La valeur associée est la liste des interventions assignées à cet intervenant.

3.5. Écrire une méthode **__call__** qui vérifie si une intervention **In** passée en paramètre existe dans la file :

- Si oui, calcule et retourne le nombre de jours d'attente avant sa prise en charge.
- Sinon, affiche le message : « Intervention invalide ».

3.6. Écrire une méthode **__str__** qui affiche toutes les interventions de la file d'attente suivie du nombre de jours attendus avant sa prise en charge, chacune sur une ligne indépendante.

4. Gestion des interventions avec un fichier texte

Les interventions sont enregistrées dans un fichier texte nommé **Intervention.txt**. Chaque ligne du fichier a le format suivant : **num#cinAg#cinIn#nbj**

Programme principal :

NB : soit **LI** une liste des CIN des intervenant est supposée donnée

Ecrire un programme principal permettant de

- Créer un objet Entreprise nommé **EP**
- Pour chaque ligne du fichier **« Intervention.txt »** créer une instance de la classe intervention et l'ajouter à la file d'attente de l'entreprise
- Créer et afficher le dictionnaire **DI** décrit en dessus
- Afficher chaque intervention suivie du nombre de jours d'attente avant sa prise en charge

Exercice 3 : Parcours à l'aide de Pile et de File d'un Graphe (défini par Matrice)

Partie 1 : Dans la suite, on souhaite représenter une **Matrice** à l'aide d'un **dictionnaire où**:

les **clefs** du dictionnaire correspondent aux **numéros de lignes**, et les **valeurs** sont des **listes** représentant les **lignes** de la matrice. Voici un exemple de représentation d'une matrice de taille **(m,n)** par un dictionnaire :

$A = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$
 	
Dans le dictionnaire **data**, la **clé 0** est associée à la valeur **data[0] = [0, 1, 2, 3], cette liste** représente la première ligne de la matrice
	Cette matrice A de taille **(m,n) = (3,4)** pourrait être représentée par le dictionnaire **data** suivant :
	
	data = {
	0: [0, 1, 2, 3], # Première ligne
	1: [4, 5, 6, 7], # Deuxième ligne
	2: [8,9,10,11] # Troisième ligne
	}

On donne le constructeur d'une classe **Matrice** contenant un attribut **data** de type **dictionnaire :**

```
class Matrice:
    def  __init__(self, m, n, data={} ):
            self.taille = (m,n)       #avec m = nombre de lignes et n = nombre de colonnes
            self.data = data
            if data == {} :
                  self.data = dict()
                  for i in range(m):
                        self.data[i] = [0] * n          #ligne i = [0,...,0] de taille n
```

Écrire les deux **méthodes spéciales** suivantes dans la classe **Matrice** :

1. `__getitem__(self, index)` : Cette méthode permet **de retourner** l'élément à la position (i, j) de la matrice. Le paramètre **index** est un **tuple** de la forme **(i, j) où i** est l'indice de la ligne et **j** est l'indice de la colonne**. (**Si l'un des indices est hors des limites, on lève une exception `IndexError` avec message d'erreur)

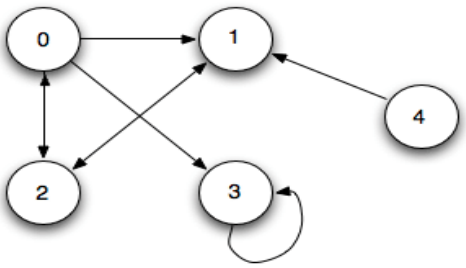
2. `__setitem__(self, index, value)` : Cette méthode permet de **modifier** l'élément à la position (i, j) de la matrice. L'**index** sera de la forme **(i, j)** et le paramètre **value** contiendra la nouvelle valeur à insérer en position **(i , j)** de la matrice.

Grâce à ces deux méthodes spéciales, l'accès à un élément d'une **instance** `M` de la classe **Matrice** sera fait avec la **notation** : `M[i,j]` où `i` désigne l'indice de **ligne** et `j` l'indice de **colonne** :

```
>>> M = Matrice(4,4)
>>> M[(3,2)] = 1    #cette notation permet d'appeler : M.__setitem__((3,2),1)
>>> M[3,2]          # cette notation permet d'appeller : M.__getitem__((3,2))
    affiche  1
```

Partie2 : Nous allons définir une nouvelle structure de données appelée graphe. Un graphe est un ensemble de sommets interconnectés par des **liaisons**, appelées **arcs** (représentés par des **flèches**) :

- Si deux sommets sont liés par un arc, alors ce dernier est associé à la valeur **1**
- Sinon, l'absence d'arc est associée à la valeur **0**



	0	1	2	3	4
0	0	1	1	1	0
1	0	0	1	0	0
2	1	1	0	0	0
3	0	0	0	1	0
4	0	1	0	0	0

Un graphe est défini par une Matrice (appelée matrice d'adjacence) telle que les indices des lignes et des colonnes représentent les sommets. Pour un graphe de **n** sommets numérotés de **0** à **(n-1),** la matrice d'adjacence est de taille ***(n,n)*** avec **chaque coefficients $a_{i,j}$ est égale à la valeur 1** si les sommets **i et j** sont **liés** par un arc**, 0 sinon.**

Travail demandé :

On se propose de construire une classe **Graphe** contenant deux attributs :

- **n** : un entier représentant le nombre de sommets d'un objet Graphe
- **M** : une **instance de la classe Matrice** représentant la matrice d'adjacence d'un objet Graphe

3. Définir la classe **Graphe** avec son constructeur qui initialise les deux attributs avec des paramètres ***n*** et ***M***

4. Définir la méthode **setArc** qui permet de rajouter un arc entre deux sommets ***s*** et ***t*** donnés d'un Graphe.

5. Définir la méthode **_ _getitem_ _** qui , pour un numéro de sommet ***num*** donnée, permet de retourner une liste de numéros des sommets successeurs de ***num.***

6. Définir la méthode **degre** qui retourne le nombre de sommets successeurs d'un sommet ***num*** donnée.

7. Définir la méthode **nbr_arcs** permettant de retourner le nombre d'arcs d'un objet Graphe en appliquant la formule suivante : $nombre\ d'arcs = \sum_{i=0}^{n-1} degre(sommet_i)$

Partie 3 : Parcourir un graphe consiste à lister tous ses sommets une seule fois

Parcours en profondeur : L'idée du parcours en profondeur repose sur l'utilisation d'une **pile (exercice 1)**
Voici la traduction en pseudo-code du parcours en profondeur d'un graphe.

P est une pile vide
On empile le sommet de départ
On initialise une liste vide
Tant que P n'est pas vide
    S = retirer le sommet de la pile P et le sauvegarder dans S
    On ajoute **S** à la liste
    On empile les **successeurs** de **S** qui ne sont pas déjà dans la pile et qui n'ont pas été déjà dépilés
Fin Tant Que

8. Écrire la fonction python **Parcours_prof** qui permet de retourner une liste nommée **listes_sommets** contenant les différents numéros de sommets rencontrés dans un parcours en largeur d'un graphe **G** à partir d'un sommet **numS** donnée. La fonction doit retourner None en cas de non-existence du sommet de départ.

Parcours en largeur ***:*** L'idée du parcours en largeur repose sur l'utilisation d'une **file (exercice 2)**
Voici la traduction en pseudo-code du parcours en largeur d'un graphe décrit ci-dessus.

F est une file vide
On enfile le sommet de départ
On initialise une liste vide
Tant que F n'est pas vide
    S = **retirer le tête de la file F et le sauvegarder dans S**
    On ajoute **S** à la liste
    On enfile les **successeurs de S** qui ne sont pas déjà dans la file et qui n'ont pas été déjà défilés
Fin Tant Que

9. Écrire la fonction python **Parcours_larg** qui permet de retourner une liste nommée **listes_sommets** contenant les différents numéros de sommets rencontrés dans un parcours en largeur d'un graphe **G** à partir d'un sommet **numS** donnée. La fonction doit retourner None en cas de non-existence du sommet de départ.