TD – Collections & Sérialisation

Objectifs du TD : Collections, itérateur, sérialisation

1. Liste personnalisée (v1)

Objectifs de l'exercice (concepts abordés) : créer une liste personnalisée
Questions :

1. Listes natives en Python (rappel)
   a. Définir une liste `maListe` contenant les éléments suivants : `10`, `3.14`, `"Paris"`
   b. Ecrire un programme qui :
      - Affiche le contenu de la liste sur la console
      - Ajoute l'élément `True` à la liste
      - Affiche le nombre d'éléments de la liste
      - Affiche le 3ème élément de la liste
      - Supprime l'élément `"Paris"` de la liste.
2. Création d'une classe `ListeNombres` pour manipuler que des nombres entiers.
   a. Définir la classe `ListeNombres` qui a pour attributs : `data` et `nbElem`. Ces deux attributs seront respectivement initialisés à liste vide (i.e. `[]`) et `0` à l'instanciation.
   b. Définir les méthodes suivantes :
      - `afficher` : qui affiche tous les éléments de la liste sous la forme suivante :
        `[ elt1 | elt2 | elt3 | … ]`
      - `ajouter` : qui ajoute un nombre entier nb à la liste. Cette méthode doit afficher un message d'erreur si nb n'est pas un nombre entier. Si nb est bien ajouté à la liste, l'attribut `nbElem` est incrémenté.
        Pour tester si une variable `var` est de type `Type` :

```
if type(var) is Type :
   print("var est bien de type Type")
else :
   print("var n'est pas de type Type")
```

      - `element` : qui retourne un élément spécifié par son indice naturel (c'est-à-dire en commençant par 1 au lieu de 0.
      - `longueur` : qui retourne le nombre d'éléments de la liste sans utiliser la fonction `len` sur la liste.
      - `enlever` : qui supprime de la liste toutes les occurrences d'un élément nb passé en paramètre. L'attribut `nbElem` est décrémenté en conséquence.

2. Liste personnalisée (v2)

Objectifs de l'exercice (concepts abordés) : créer une liste personnalisée en utilisant les classes abstraites des collections offertes par Python.
Questions :

1. Voici un extrait de la définition de la classe abstraite `UserList` définie dans le module `collections`:
   a. `__str__` : une méthode qui doit retourner une chaine de caractères représentant la liste
   b. `__len__` : une méthode qui doit retourner le nombre des éléments de la liste
   c. `__getitem__` : une méthode qui doit retourner un élément de la liste spécifié par son indice qui est passé en paramètre

d. `append` : une méthode qui doit ajouter un élément à la liste
e. `remove` : une méthode qui doit enlever un élément de la liste

2. Définir maintenant la classe `ListeNombresV2` en héritant de la classe `UserList` et en implémentant les bonnes méthodes pour recréer le comportement initial (`ListeNombres` de l'exercice 1)
3. Ecrire un programme principal pour illustrer toutes les méthodes (questions 1.a, 1.b, 1.c, 1.d et 1.e)
   a. Commenter la façon d'invoquer les 3 premières méthodes (1.a, 1.b et 1.c)

3. Itérateurs en Python

Objectifs de l'exercice (concepts abordés) : Utiliser un itérateur d'une collection
Description de l'exercice :

1. Soit la liste de l'exercice 1 :
   `maListe = [10, 3.14, "Paris", True]`
   a. Définir un itérateur `it` sur cette liste.
   b. Afficher tous les éléments de la liste en utilisant une boucle POUR en faisant appel à la fonction `next()`
   c. Afficher tous les éléments de la liste sans utiliser la fonction `next()`
   d. Conclure quant à l'utilisation de la boucle POUR sur des collections itérables.
2. Reprendre la classe `ListeNombresV2` de l'exercice 2 et redéfinir l'itérateur par défaut (hérité de `UserList`) pour qu'il renvoie les nombres négatifs puis les nombres positifs.
   Rappel : pour (re)définir un itérateur dans une classe collection, il faut implémenter la méthode `__iter__()`. Cette méthode renvoie chaque élément de la collection en utilisant le mot clé `yield`.

4. Sérialisation & Désérialisation

Objectifs de l'exercice (concepts abordés) : (Dé)Sérialiser des objets en Python
Description de l'exercice :

1. Ecrire un programme python qui :
   a. Définit une variable dictionnaire `dicoEtu` comme suit :
      - `"Nom"` -> Nom de l'étudiant (en chaine de caractères)
      - `"Groupe"` -> Groupe de l'étudiant (en chaine de caractères)
      - `"Notes"` -> Liste des notes de l'étudiant (en réels).
   b. Initialise les valeurs de `dicoEtu` à : `"John Doe"`, `"S5 POO"`, `[19, 20, 18, 17, 20]`
   c. A l'aide la bibliothèque native Pickle (`import pickle`) :
      i. Sérialise la variable `dicoEtu` dans une variable `pickleData` et l'affiche à l'écran.
      ii. Désérialise la variable `pickleData` vers une variable `newDico1` et l'affiche à l'écran.
      iii. Exécuter & Commenter.
   d. A l'aide la bibliothèque native JSON (`import json`) :
      i. Sérialise la variable `dicoEtu` dans une variable `jsonData` et l'affiche à l'écran.
      ii. Désérialiser la variable `jsonData` vers une variable `newDico2` et l'affiche à l'écran.
      iii. Exécuter & Commenter.

2. Définir une classe `Etudiant` ayant les attributs suivants :
   `nom`, Nom de l'étudiant (chaine de caractères)
   `groupe` : groupe de l'étudiant (chaine de caractères)
   `notes` : une liste des notes obtenues par l'étudiant (le nom des modules ou l'ordre des notes dans la liste n'est pas important. Les notes sont des nombres réels.
   A l'instanciation, le nom et le groupe sont passés en paramètres.

La classe doit fournir des méthodes pour ajouter les notes individuellement (`ajouter_note`) ou plusieurs notes à la fois (`ajouter_notes`)

NB : Pour permettre un affichage d'un objet de la classe Etudiant à l'aide de la fonction `print`, il faut ajouter à la classe une méthode `__str__` qui retourne une chaine de caractères représentant l'objet.

3. Ecrire un programme qui :
   a. Instancie un objet `etu` de la classe ayant les valeurs précédentes (question 1).
   b. Sérialise l'objet `etu` en JSON dans la variable `jsonEtu` et l'affiche à l'écran.
   c. Désérialise la variable `jsonEtu` vers un objet `newEtu`.
   d. Exécuter & Commenter.

4. Définir une nouvelle classe `EtudiantEncoder` qui hérite de la classe `JSONEncoder`. Cette classe implémente (redéfinie) la méthode `default` qui prend un objet Etudiant `objEtu` en entrée et retourne un dictionnaire avec les données contenues dans `objEtu`.
   La classe `JSONEncoder` fait partie de la bibliothèque `json` (→ `from json import JSONEncoder`)
   a. Reprendre le programme précédent en passant comme 2ème paramètre à la méthode `json.dumps` le paramètre nommé `cls=EtudiantEncoder`
   b. Exécuter et commenter

5. Définir une nouvelle classe `EtudiantDecoder` qui hérite de la classe `JSONDecoder`. Cette classe implémente (redéfinie) la méthode `decode` qui prend une chaine de caractères (`jsonStr`) en entrée et retourné une instance de classe `Etudiant` avec les informations contenues dans `jsonStr`.
   La classe `JSONDecoder` fait partie de la bibliothèque `json` (→ `from json import JSONDecoder`)
   a. Reprendre le programme précédent en passant comme 2ème paramètre à la méthode `json.loads` le paramètre nommé `cls=EtudiantDecoder`
   b. Exécuter et commenter