Des sommets et des graphes

Un graphe dirigé peut être défini comme un couple `G=(V,E)` où `V` est un ensemble de *sommets* et `E` un ensemble de couples `(i,j)` de `VxV` que l'on appelle des *arcs*. Un sommet `j` de `V` est dit *successeur* d'un sommet `i` de `V` si `(i,j)` appartient à `E`. Le sommet `j` est alors un prédécesseur de `i`.

Exemple - `V={0,1,2,3,4,5,6}` - `E={(2,0),(2,1),(2,3),(3,1),(1,4), (6,0),(6,5),(6,3),(0,4),(4,0),(4,5), (4,6)}` - L'ensemble des successeurs du sommet `2` est `{0,1,3}`. L'ensemble des prédecesseurs de `4` est `{0,1}`.

Pour manipuler un graphe dans un programme, une des structures de données les plus utilisées est la liste d'adjacence. Elle consiste en un tableau dont l'entrée i correspond à l'ensemble des successeurs du sommet i.

1- La classe `Sommet`

Objectifs de l'exercice - Manipuler des attributs de classe et des méthodes de classes. - Comprendre la différence entre un attribut/une méthode de classe et un attribut/une méthode d'instance. - Comprendre et manipuler la notion de destructeur. - Comprendre ce qu'est une valeur de hashage et comment rendre une classe hashable. - Apprendre à redéfinir des opérateurs spéciaux du Python. - Apprendre à gérer des exceptions

1.1 - Écrire une classe `Sommet` comprenant un attribut `__id` permettant d'identifier le graphe. Cet identifiant est supposé pour le moment être de n'importe quel type (`int`, `str`, etc.). Définir un constructeur prenant un paramètre permettant d'initialiser cet attribut, ainsi qu'un accesseur en lecture permettant de le connaître. Définir `__str__` de manière à ce que l'affichage d'un sommet provoque l'affichage de son identificateur

1.2 - On souhaite pouvoir connaître le nombre d'objets `Sommet` présents dans le système à n'importe quel moment de l'exécution. Ajouter un attribut de classe `__nbSommets` permettant de compter ce nombre d'objets à chaque instant. Définir un accesseur en lecture pour cet attribut. Modifier le constructeur et ajouter un destructeur de manière à ce que l'attribut de classe `__nbSommets` soit toujours à jour.

Exemple

```
n1=Sommet("INF2")
n2=Sommet(26)
n3=Sommet("INF1")
print(f"nb sommets={Sommet.getNbSommets()}") # 3 sommets
del(n2) # destruction du sommet n2
print(f"nb sommets={Sommet.getNbSommets()}") # 2 sommets
```

1.3 - Afin de pouvoir utiliser des objets `Sommet` en tant que clés d'un dictionnaire, nous avons besoin qu'un tel objet soit *hashable*.

On appelle *hash* un nombre calculé depuis une valeur quelconque et qui est unique et invariable pour cette valeur. Deux valeurs considérées comme égales partageront un même hash, deux valeurs différentes auront dans la mesure du possible des hash différents.

Un objet peut être *haché*. - S'il a un hash qui ne change jamais pendant sa durée de vie. Pour cela, il a besoin d'une méthode `__hash__()`. - S'il peut être comparé à d'autres objets. Pour cela, il a besoin d'une méthode `__eq__()` ou `__cmp__()`. Les objets hachables qui se comparent égaux doivent avoir le même hash.

Tous les objets intégrés immuables de Python sont hachables, alors qu'aucun conteneur mutable (comme les listes ou les dictionnaires) ne le sont. La fonction `hash()` renvoie le hash de n'importe quel objet hashable.

Les objets qui sont des instances de classes définies par l'utilisateur sont hachables par défaut; ils se comparent tous inégaux et leur valeur de hachage est leur id(). Ainsi, bien qu'un objet `Sommet` soit hashable par défaut, deux objets `Sommet` avec le même identificateur ne donnent pas le même hash. Nous allons corriger ce défaut : - À partir de maintenant, nous allons exiger que le type de l'identificateur utilisé pour un Sommet soit hashable : déclencher une exception de type `TypeError` si ce n'est pas le cas. Pour cela, on pourra vérifier si son attribut `__hash__` est égal à `None` - Redéfinir la méthode `__hash__()` de `Sommet` pour qu'elle renvoie le hash de l'identificateur d'un sommet. - Redéfinir la méthode `__eq__()` de `Sommet` : on considère que deux sommets sont égaux si leurs identificateurs sont égaux.

1.4 Créer un script qui crée plusieurs sommets, dont éventuellement un sommet avec un identificateur non hashable (par exemple, la liste `[1,2,3]` étant mutable, elle n'est pas hashable). Définir une instruction `try - except - else - finally` qui englobe ces instructions. S'il n'y a pas d'erreur, afficher le message "Sommets créés". S'il y a un problème, rattraper l'exception en affichant l'erreur. Dans tous les cas, afficher ensuite le message "Continuez"

2 - La classe `Graphe` (Exercice optionnel de révision)

Objectifs de l'exercice - Réviser l'ensemble des notions vues jusque maintenant. - Réviser l'utilisation des classes `set()` et `dict` - Réviser la redéfinition des opérateurs spéciaux du Python. - Apprendre à utiliser la fonction `isinstance()` pour manipuler des éléments qui peuvent être de plusieurs types.

Pour représenter un graphe, on utilise un attribut `__succ` de type `dict` où chaque clé est un `Sommet` et où chaque valeur associée à une clé est un `set` d'objets `Sommet`. À chaque sommet sera alors associé l'ensemble de ses sommets successeurs, représentant ainsi tous les arcs du graphe.

2.1 - Définir la classe `Graphe` qui comportera un attribut `__nom` de type `str` qui représentera son nom ainsi que l'attribut `__succ` comme défini précédemment. Le contructeur prendra le nom du graphe en paramètre et initialisera les attributs. Initialement, le graphe est supposé ne contenir ni sommet, ni arc.

2.2 - Définir les méthodes suivantes : - `getNom()` qui renvoie le nom du graphe - `getSommets()` qui renvoie le nombre de sommets du graphe - `getNbArcs()` qui renvoie le nombre d'arcs du sommet du graphe - `__str__()` qui permet d'afficher un graphe. On affichera sur une ligne son nom, son nombre de sommets, son nombre d'arêtes. Puis, on affichera sur chaque ligne un sommet du grahe suivi de la liste de ses successeurs (voir exemple suivant).

NB : Le nombre de sommets correspond au nombres de clés dans le dictionnaire `dict`. Le nombre d'arcs correspond à la somme des cardinaux des ensembles de sommets associés à chaque clé.

2.3 - Ajouter un opérateur `+=` (surcharge de la méthode `__iadd__()`) à la classe `Graphe` qui permettra d'ajouter soit un `Sommet`, soit un arc sous la forme d'un tuple de deux sommets. Le comportement attendu est le suivant : - L'ajout d'un sommet ou d'un arc déjà existant n'a aucun effet. - L'ajout d'un arc entre un ou des sommets qui n'existent dans le graphe pas encore provoque leur création. - La tentative d'ajout d'un élément qui n'est ni un `Sommet` ni un couple de `Sommet` déclanche une exception.

Exemple :

```
 g = Graphe("G1")
print(g)

g += Sommet("INF1")
g += Sommet("INF2")
g += (Sommet("INF1"), Sommet("INF2"))
g += (Sommet("INF1"), Sommet("NF93"))
g += (Sommet("INF1"), Sommet("NF92"))
g += (Sommet("INF1"), Sommet("NF16"))
g += (Sommet("NF92"), Sommet("NF18"))
g += (Sommet("INF2"), Sommet("NF18"))
g += (Sommet("NF92"), Sommet("MI01"))
print(g)
```

provoque l'affichage:

```
 Graphe G1: 0 sommets, 0 arcs


Graphe G1: 7 sommets, 7 arcs
INF1 : NF92 INF2 NF93 NF16
INF2 : NF18
NF93 :
NF92 : MI01 NF18
NF16 :
NF18 :
MI01 :
```

Pour s'entrainer, on pourrait aussi définir l'opérateur '-=' avec le comportement suivant : - La suppression d'un sommet provoque la suppression de tous les arcs liés à ce sommet. - La suppression d'un sommet ou d'un arc inexistant déclenche une exception.