Langage de script (Python)
CMTP n° 7 : Programmation objet avec Python

I) Petit rappel sur les objets

La Programmation Orientée Objet (POO) est un outil pour résoudre le problème suivant : "J'ai compris le problème et je sais le résoudre mais comment je le programme ?". En plus de maîtriser un langage de programmation, des structures de données et des algorithmes il faut structurer le code. La POO propose une méthode pour cela. Le but est de poser le problèmes sous forme d'objets et de leurs interactions. Cela permet d'organiser le code et de découper le problème initial (gros et compliqué) en sous problèmes (plus petits et plus simples). Il permet aussi de réutiliser du code.
Vous pouvez trouver une introduction aux objets en Python ici :

https://docs.python.org/3/tutorial/classes.html

En particulier, dans le tutorial vous trouvez une section préliminaire sur les espaces de nommage en Python (voir https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces) qui a un intérêt indépendamment du traitement des objets et dont on vous conseille fortement la lecture.

II) La classe Graph

Un graphe est un ensemble de sommets reliés par des arcs. Implémentez la classe suivante pour représenter les graphes orientés.
Télécharger le fichier `graph.py` depuis Moodle comme modèle. La classe `Graph` dans ce fichier contient :

- Un dictionnaire `_edges` représentant un graphe orienté et pondéré. Les clefs de `_edges` sont les sommets du graphe et pour un sommet `n`, la valeur `_edges[n]` est une liste de couples `[(n1, w1)..., (nk, wk)]` tel que chaque `(ni, wi)` correspond à un arc de source `n`, cible `ni` et poids `wi` (il s'agit donc plus précisément d'une *liste d'adjacence pondérée pour le sommet* `n`).
- Une méthode spéciale `__len__()` qui retourne le nombre de sommets du graphe. Cette méthode permet d'obtenir la taille (en nombre de sommets) du graphe en utilisant la fonction standard Python `len()` que vous connaissez déjà.
- Une méthode spéciale `__iter__` qui retourne un itérateur sur les sommets du graphe. Cela permet de itérer sue les (étiquettes des) sommets du graphe avec une boucle for standard : `for s in graph: ....`
- Une méthode spéciale `__getitem__` qui permet d'obtenir la liste d'adjacence d'un sommet `n` avec la syntaxe standard : `graph[n]`.

Les méthodes spéciales dans `graph.py` montrent comment en Python on peut faire en sort que vos classes se comportent comme d'autres types prédéfinis. D'autres possibilités existent (p.ex., pour supporter les opérateurs arithmétiques entre instances de vos classes, ou encore implémenter des containers ou des *context managers*). Pour plus de détails sur les méthodes spéciales existantes consultez la documentation ici : https://docs.python.org/3/reference/datamodel.html#emulating-generic-types

Exercice 1 :

1. Ajouter une méthode `add_node(u)` qui ajoute le sommet au graphe si `u` n'est pas encore un sommet du graphe. Veiller à ne pas ajouter un sommet dont l'identifiant existe déjà. Le nouveau node n'aura pas d'arc incident.
2. Ajouter une méthode `add_edge(source, target, weight = None)` qui prend en entrée deux identifiants de sommets, et un poids (par défaut à `None`). Si les identifiants n'existent pas, les ajouter avec la méthode `add_node(u)`.
3. Redéfinir la méthode spéciale `__str__` pour une représentation lisible du graph en chaîne de caractères. Par exemple,

```
s -> [('a', 10), ('e', 8)]
a -> [('c', 2)]
e -> [('d', 1)]
d -> [('a', -4), ('c', -1)]
c -> [('b', -2)]
b -> [('a', 1)]
```

4. Testez votre code avec le graphe ci-dessus.

Exercice 2 :

L'algorithme de Bellman-Ford permet de trouver les plus courts chemins dans un graphe, d'une source vers toutes les destinations.

```
fonction Bellman_Ford(G, source)
  Pour chaque sommet u du graphe
           dist[u] = +infini
           pred[u] = None
  dist[source] = 0
  Pour k = 1 jusqu'à Nombre de sommets - 1 :
       Pour chaque arc (u, v) du graphe faire
            Si dist[v] >  dist[u] + poids(u,v):
               dist[v] =  dist[u] + poids(u,v)
               pred[v] = u
  retourner dist, pred
```

Implémenter cet algorithme comme méthode de la classe `Graph`. La fonction Bellman-Ford doit retourner les dictionnaires `dist` et `pred`. Par exemple sur le graphe de l'Exercice 1 et comme source `'s'`, la fonction doit retourner :

- `dist` vaut 's' : 0, 'a' : 5, 'e' : 8, 'd' : 9, 'c' : 7, 'b' : 5
- `pred` vaut 's' : None, 'a' : 'd', 'e' : 's', 'd' : 'e', 'c' : 'a', 'b' : 'c'

Quel est le temps d'exécution de cette algorithme en fonction de la taille du graphe ?

III) L'itinéraire de métro

Nous allons utiliser les fonctionnalités de la classe `Graph` pour écrire un script qui calcule un itinéraire sur un plan de métro à Paris. Reprendre la classe `Graph` de la section précédente. Récupérer sur Moodle les fichiers json `stops.json` et `timetable.json`. Ces fichiers contiennent des dictionnaires `stops` et `timetable` que vous pouvez récupérer grâce au module `json` de Python. Le dictionnaire `stops` décrit les arrêts du réseau de la RATP (arrêts de bus et quais de métro et RER) et `timetable` donne les temps de transfert entre les arrêts.

On utilisera deux fonctions du module `json` :

- `json.dump(variable, descripteur_de_fichier)` écrit dans un fichier au format json le contenu de la variable
- `json.load(descripteur_de_fichier)` retourne le contenu de la variable obtenue par `json.dump`.

Par exemple, pour sauvegarder le contenu d'un dictionnaire `dic` :

```
json.dump(dic, open("dic.json", "wb"))
```

Et pour le récupérer à partir du fichier `dic.json` :

```
dic = json.load(open("dic.json", "rb"))
```

Pour plus de détails, n'hésitez pas à consulter la documentation https://docs.python.org/3/library/json.html#module-json.

Les dictionnaires contenus dans les fichiers `stops.json` et `timetable.json` ont la structure suivante :

```
stops[stop_id] = {
    'stop_name': stop_name,
    'stop_desc': stop_desc,
    'route': transp_desc,
}
timetable['stop_id, stop_id'] = tempsensecondes
```

Où `transp_desc` est un dictionnaire décrivant la ligne de bus, métro ou tram marquant cet arrêt. Note : certains arrêts ne sont associés à aucune route. Dans ce cas, `route` est la chaîne de caractères `'NONE'`.[1] Les clés de `timetable` sont des chaînes de caractères représentant une paire des deux `stop_id` séparés par une virgule et un espace.[2] N'hésitez pas à afficher les premières items des deux dictionnaires.

Exercice 3 :

Définir une classe `Transport` qui hérite de `Graph`, et ajouter un attribut `_stops` contenant les descriptions des arrêts. Construire une instance de la classe `Transport` qui représente le réseau de transport de la RATP. Ajoutez tous les sommets (les arrêts), et pour toutes les paires d'arrêts (u, v) dans `timetable`, ajouter les deux arcs (u, v) et (v, u) au graphe, et ayant comme poids le temps en secondes entre les deux arrêts (à ce fin, prévoir un constructeur de `Transport` qui prend en paramètre les noms des fichiers json à décoder).

Exercice 4 :

Écrire une fonction qui prends en paramètre une chaîne `s` et retourne la liste des `stop_id` des stations de métro dont le nom `stops[stop_id]['stop_name']` est donné par `s`.
On reconnaît les stations de métro par le numéro de ligne `stops[stop_id]['route']['num']` qui représente une valeur numérique qui est $\leqslant 14$. (Le numéro de ligne est une chaîne de caractères, qui peut être "A", "B", "C",... pour les lignes de RER.) Pour savoir si une chaîne est numérique, on peut utiliser la fonction `isnumeric()`. Attention, rappelez vous que pour certains `stop_id`, la valeur de `stops[stop_id]['route']` est `'NONE'`, dans ce cas vous ne pouvez pas accéder à `'num'`.
Remarquez aussi que les `stops` représentent un arrêt comme un quai de métro ou un arrêt de bus, pas une station de métro en entier. Les temps de transfert incluent aussi le temps pour aller d'une quai à un autre à l'intérieur de la station de métro. Ainsi, si vous cherchez `'Nation'` vous obtiendrez un ensemble de 8 `stop_id`.

Exercice 5 :

Écrire un script qui permet d'obtenir les plus courts chemins entre deux stations de métro. Le script prend en entrée deux chaînes qui représentent des noms de stations de métro. Le script trouve les arrêts correspondants et lance le calcul du plus court chemin entre les deux stations en utilisant l'algorithme de Bellman-Ford, en affichant le résultat de façon la plus lisible possible.

Exercice 6 :

On peut modifier l'algorithme de Bellman-Ford pour qu'il limite le nombre d'arêtes (dans les plus courts chemins) à k pour un entier k quelconque. Modifiez l'algorithme. Quelle est sa complexité ? Quelle différence constatez-vous au niveau du temps d'exécution ?

1. En principe, il y aurait été bien mieux d'avoir la valeur `None` de Python plutôt qu'une chaîne de caractères, mais ceci n'est pas le format de `stops.json`.
2. Aussi dans ce cas il y aurait été bien mieux d'avoir un couple de valeurs `int`, mais ceci aussi n'est pas dans le format de `stops.json`.