TP 18 modules, dictionnaires, graphes

*L'objectif du TP est de construire un module Python pour gérer les **graphes**. Nous allons coder un graphe en Python à l'aide d'une nouvelle structure de données : un **dictionnaire**. Voici un exemple :*

```
>>> mon_dictionnaire = {'E3' : [17, 14], 'E1' : [1, 2, 3], 'E2' : []}
>>> mon_dictionnaire
>>> mon_dictionnaire[0]
>>> mon_dictionnaire['E3']
>>> mon_dictionnaire['E3'][1]
>>> mon_dictionnaire['nouveau'] = 34
>>> mon_dictionnaire
>>> 17 in mon_dictionnaire
>>> 'E1' in mon_dictionnaire
```

Vous l'aurez compris, un dictionnaire est une sorte de liste, mais au lieu que ses éléments soient indexés par des entiers commençant à 0, chaque élément d'un dictionnaire à un nom. Pour le reste, les manipulations sur les dictionnaires sont similaires à celles sur les listes. L'utilisation des accolades définit un dictionnaire. Un dictionnaire vide est défini par la commande `{}`.
*Autre structure de données que nous utiliserons : les **ensembles**. Ce sont des ensembles au sens mathématique du terme. On utilise le mot-clé `set` (qui signifie ensemble en anglais) pour les définir : cela convertit une liste ou un dictionnaire en ensemble. Voici quelques exemples :*

```
>>> mon_ensemble = set({})
>>> ensemble2 = set([])
>>> mon_ensemble.add(4)
>>> mon_ensemble.add(6)
>>> mon_ensemble.add(-8.3)
>>> ensemble2 = {-1, 1.2, 9.5}
>>> ensemble2 == mon_ensemble
>>> 3 in ensemble2
>>> ensemble2.add(3)
>>> 3 in ensemble2
```

1. Créez un fichier python nommé `graphes.py`. Dans ce fichier, ajoutez les commandes `import numpy as np` et `import copy`. Ajoutez la définition d'une classe nommée `graph`. Dans cette classe, codez une fonction `__init__(self)` sans autre argument qui initialisera un graphe en lui donnant trois attributs : un attribut `noeuds` défini comme un ensemble vide, un attribut `taille` valant initialement 0, puis un attribut `dico` initialisé par un dictionnaire vide.
2. Créez un autre fichier `TP18.py`. Dans ce fichier, ajoutez les commandes suivantes puis exécutez :

```
import graphes
G = graphes.graph()
G.dico = {'a' : {'b','c'}, 'b' : {}, 'c' : {'a'}}
G.taille = 3
G.noeuds = {'a', 'b', 'c'}
print(G.dico['c'])
```

3. *Nous avons créé un graphe, représenté par un dictionnaire Python. Ce graphe a trois sommets : a, b et c, ces noms de sommets étant représentés par un caractère Python. À l'entrée `'a'` du dictionnaire, on a l'ensemble des sommets voisins de a. Le graphe est orienté : a est connecté à b mais pas b à a. On notera $a \to b$ une telle connexion. On voit que la connexion entre a et c est réciproque par contre : on a $a \to c$ et $c \to a$.*
   (a) Représentez le graphe précédent sur une feuille de papier.
   (b) Ajoutez un sommet `'d'` à votre graphe, avec une connexion $b \to d$ et une connexion $d \to a$.

4. Coder une méthode `matrice` dans la classe `graph` qui renvoie la matrice d'adjacence du graphe. Vous aurez besoin des commandes suivantes (ceci n'est pas une ébauche de code) :

```
def matrice(self) :
A = np.zeros([n, n])
A[i, j] = 1
```

5. Si le graphe n'est pas orienté, c'est-à-dire que tous les liens du graphe sont bidirectionnels, quelle propriété doit vérifier la matrice d'adjacence du graphe ? Codez une méthode `est_oriente` qui renvoie `'oui'` si le graphe est orienté et `'non'` sinon. Vous aurez besoin de la commande `A.all()` qui renvoie `True` si tous les éléments d'une matrice A sont non nuls et `False` sinon.
6. *Si A est la matrice d'adjacence d'un graphe, on sait que A^p est une matrice dont les éléments i, j donnent le nombre de chemins de longueur p reliant le sommet numéro i au sommet numéro j. Pour savoir si un graphe est connecté ou non, on peut donc calculer $\mathrm{I}_n + A + A^2 + \cdots + A^n$, et si cette matrice a un élément nul, c'est que le graphe n'est pas connecté (n est le nombre de sommets)*

   Codez une méthode `est_connecte` qui renvoie `'oui'` si le graphe est connecté et `'non'` sinon. Vous utiliserez à nouveau la méthode `.all()`.
7. *On considère maintenant des **graphes pondérés** : chaque lien entre deux sommets est affublé d'un coefficient réel : c'est le poids de ce lien. Il représente un coût de trajet pour aller d'un sommet à l'autre. Beaucoup de problèmes de transport (de données, de marchandises, de personnes) modélisent ainsi les déplacements. Pour coder un graphe pondéré, il suffit que chaque élément du dictionnaire soit lui même un dictionnaire donnant la liste des poids en plus des liens. Voici un exemple :*

```
import graphes
G = graphes.graph()
G.dico = {'a' : {'b' : 4.5, 'c': 3}, 'b' : {}, 'c' : {'a' : 7}}
G.taille = 3
G.noeuds = {'a', 'b', 'c'}
print(G.dico['a']['c'])
```

   Ajoutez à ce graphe un sommet `'d'` avec les connexions $d \to a$ de poids $3,5$, $d \to b$ de poids $1,2$ et $b \to d$ de poids 5.
8. *On cherche maintenant le poids total du plus court chemin partant d'un sommet donné et allant à un autre sommet du graphe. Pour ceci on va coder l'algorithme de Bellman-Ford. On commence avec un dictionnaire contenant les distances du sommet fixé (appelons le s) aux autres sommets. La distance de s à s est bien sûr 0, et on met la distance aux autres sommets à $+\infty$ au départ : on utilise pour cela **`float('inf')`** qui se comporte comme l'infini pour Python : c'est un objet supérieur strictement à tout réel, et si on ajoute un réel à **`float('inf')`** on obtient toujours **`float('inf')`**. On parcourt ensuite plusieurs fois tous les liens du graphe en mettant à jour au fur et à mesure les distances obtenues.*

pour tous les sommets k du graphe :
    $\pi(k) \leftarrow +\infty$
$\pi(s) \leftarrow 0$
$\pi_0 \leftarrow \{\}$
tant que $\pi \neq \pi_0$:
    π_0 est une copie de π
    pour chaque lien u - v du graphe :
        si $\pi(v) > \pi(u) + poids(u, v)$:
            $\pi(v) \leftarrow \pi(u) + poids(u, v)$
retourner π

   Dans votre module `graphe`, codez une fonction `bellman(graph, source)` qui renvoie le dictionnaire `pi` donné par cet algorithme. Testez le sur le graphe de la question précédente.

Formulaire :

Vous aurez besoin des commandes suivantes (dans le désordre)

```
### LES MODULES USUELS
from math import *
import numpy as np
import matplotlib.pyplot as plt
import numpy.random as rd

### LES VARIABLES ALEATOIRE
rd.randint(a,b,N)
rd.random(N)
rd.binomial(n,p,N)
rd.geometric(p,N)
rd.poisson(lambda,N)

## LES LISTES
np.linspace(a,b,n)
for compteur in liste :
for compteur in range(a,b):
[ f(compteur) for compteur in liste]
[ f(compteur) for compteur in range(a,b)]

### COMMANDES CLASSIQUES
while condition:
def nom_fonction( variables ) :
return resultat
print( 'message', variable )
if condition :
else :
log(x)
plt.plot( abscisses , ordonnees )

### CLASSES EN PYTHON
class nom_de_classe :
def __init__(self, autres_parametres):
def __str__(self):

### MATRICES EN PYTHON
np.array([[...],...,[...]])
A.shape
np.zeros([p,n])
np.eye(n)
np.ones([p,n])
A.dot(B)
np.dot(A,B)
A.transpose()
```