TP Graphes 1

Exercice 1 : dénombrement

Soit $n \in \mathbb{N}$.
Combien y a-t-il de graphes orientés à n sommets ?

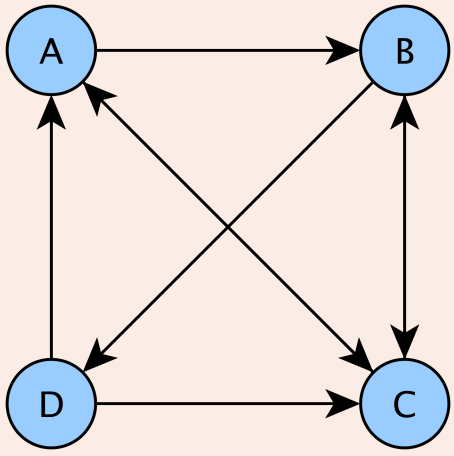
Exercice 2 : matrice ou dictionnaire ?

Soit $n \in \mathbb{N}$. On considère un graphe à n sommets. On comptera comme OPEL toute opération consistant à accéder aux éléments d'une liste.

1. Si on choisit d'implémenter ce graphe à l'aide d'une matrice d'adjacence, combien d'OPEL sont nécessaires pour obtenir la liste de ses successeurs ?
2. Même question avec une implémentation par dictionnaire d'adjacence.
3. Dans quels cas un dictionnaire d'adjacence est-il préférable ?

Exercice 3 : implémentation n°1

On décide d'implémenter la structure de graphe orienté à l'aide d'une matrice d'adjacence encapsulée dans une classe.
On va tester notre implémentation sur le graphe ci-dessous :



Avec cette première approche, on conviendra que A est le sommet 0, B est le sommet 1 et ainsi de suite.

1. Ouvrir dans PyCharm les fichiers `graph_matrix.py` et `test_graph_matrix.py`.

2. Lire les contenus des fichiers.

3. Compléter `graph_matrix.py` pour que `test_graph_matrix.py` fonctionne correctement.
   Aucune programmation défensive n'est demandée.

Exercice 4 : implémentation n°1 améliorée

On aimerait conserver cette implémentation mais faire en sorte que l'utilisateur puisse travailler directement avec les noms des sommets, et pas leurs indices, comme ceci :

```
>>> G = graph_matrix2(['A','B','C'])
>>> G.add_arrow('A','B')
>>> G.is_successor('B','A')
True
>>> G.predecessors('B')
['A']
```

On construit donc un graphe à l'aide d'une liste de sommet qui sera un attribut de l'objet.

Coup de pouce : pour trouver quel est l'indice d'un élément `a` d'une `L`, on peut appeler `L.index(a)` comme dans l'exemple suivant :

```
>>> L=['A','B','C']
>>> L.index('C')
2
```

On va travailler sur le même graphe qu'à l'exercice précédent.

1. Ouvrir dans PyCharm les fichiers `graph_matrix2.py` et `test_graph_matrix2.py`.

2. Lire les contenus des fichiers.

3. Compléter `graph_matrix2.py` pour que `test_graph_matrix2.py` fonctionne correctement.
   Aucune programmation défensive n'est demandée.

Exercice 5 : implémentation n°2

On désire maintenant implémenter la structure de graphe en utilisant un dictionnaire de type `sommet : [liste des successeurs]`.

1. Ouvrir dans PyCharm les fichiers `graph_dict.py` et `test_graph_dict.py`.
2. Lire les contenus des fichiers.
3. Compléter `graph_dict.py` pour que `test_graph_dict.py` fonctionne correctement.
   Aucune programmation défensive n'est demandée.

Coup de pouce : Pour obtenir la liste des clés d'un dictionnaire `d`, on peut utiliser `list(d)` :

```
>>> d = {'Kurt' : 'Cobain', 'Beck' : 'Hansen', 'Jimi' : 'Hendrix'}
>>> list(d)
['Kurt', 'Beck', 'Jimi']
```

Exercice 6 : lister les arcs

Reprendre l'implémentation précédente et ajouter une méthode `arrows` qui renvoie la liste des arcs d'un graphe orienté.
Avec le graphe de l'exercice 1 on doit obtenir ceci :

```
>>> G.arrows()
[('A', 'B'), ('A', 'C'), ('B', 'C'), ('B', 'D'), ('C', 'A'),
 ('C', 'B'), ('D', 'A'), ('D', 'C')]
```

Exercice 7 : pour aller plus loin

1. Créer un environnement virtuel (ne pas cocher `inherit global-site packages` et cocher `make available to all projects`).
2. Ajouter les packages `matplotlib` et `networkx`.
3. Créer un nouveau fichier `graph_dict_nx` contenant une copie de la class `GraphDict`.
4. Ajouter au début

   ```
   import networkx as nx
   import matplotlib.pyplot as plt
   ```

   Ainsi que la méthode `draw` :

```
    def draw(self):
        draw_graph = nx.DiGraph()
        draw_graph.add_edges_from(self.arrows())
        positions = nx.circular_layout(draw_graph)
        nx.draw(draw_graph, positions, with_labels=True)
        plt.show()
```

5. Dans un fichier `test_graph_dict_nx`, créer des graphes aléatoires et les afficher (à vous de voir comment s'y prendre).