1 Rappels sur les Dictionnaires

On accède aux éléments d'une liste grâce aux indices qui sont des entiers $k \in [\![0\,;n-1]\!]$ si n est la longueur de la liste. Ainsi, si `L=[5,"a",["c",3.2]]`, alors

- `L[0]` vaut 5
- `L[1]` vaut `"a"`
- `L[2]` vaut `["c",3.2]`

La commande `print(L[3])` provoque une erreur : `out of range`. On aimerait généraliser le concept de listes, avec des indices qui ne seraient plus forcément des entiers entre 0 et $n-1$. Et c'est précisément ce que font les dictionnaires. La notion d'indices une liste va être remplacée par celle de clé dans un dictionnaire. La notion d'élément d'indice k va être remplacée par celle de valeur associée à la clé dans un dictionnaire. Par exemple, imaginons qu'on veuille compter le nombre de pommes de poires et de fraises. On pourrait stocker dans une liste : `L=[5,2,25]` et retenir que `L[0]` compte le nombre de pommes, `L[1]` le nombre de poires et `L[2]` le nombre de fraises. Mais ce n'est pas commode, imaginez que la liste est un million d'éléments, vous n'avez pas très envie d'apprendre par cœur que `L[987123]` compte le nombre d'arbouses. Un dictionnaire (ou tableau associatif) est une collection de paires de la forme `(clé,valeur)`. Ainsi, on veut un dictionnaire avec trois clés : "pommes", "poires" et "fraises", la valeur associée à "pommes" valant 5, de même, la valeur associée à "poires" vaut 2.

1.1 Créer un dictionnaire

Il existe trois façons de créer un dictionnaire.

1. Créer un dictionnaire vide : `D = {}`.
2. Créer un dictionnaire par extension `D = {"pommes":5,"poires":2,"fraises":25}`.
3. Créer un dictionnaire par paramétrage `D = {L[i]:M[i] for i in range(len(L))}`, avec, par exemple, `M = [5,2,25]` et `L = ["pommes","poires","fraises"]`.

Il n'y a pas d'ordre dans un dictionnaire, `{"pommes":5,"poires":2,"fraises":25} == {"poires":2,"pommes":5,"fraises":25}` vaut `True`. Les accolades font penser à la notion d'ensembles en mathématiques où il n'y a pas d'ordre dans non plus. La commande `len(D)` donne le nombre de couples (clé,valeur) dans le dictionnaire, et donc le nombre de clés.

1.2 Vérifier si un élément est une clé dans un dictionnaire

La commande `k in D` est un booléen qui vaut `True` si `k` est une clé de `D` et `False` sinon. Ainsi, pour notre exemple, `"pommes" in D` vaut `True` tandis que `5 in D` et `"kakis" in D` valent `False`.

1.3 Obtenir la valeur associée à une clé

La commande `D["pommes"]` permet d'obtenir la valeur associée à `"pommes"`.

1.4 Modifier les valeurs d'un dictionnaire

La commande `D["pommes"] = 10` modifie la valeur de `D` associée à la clé `"pommes"`

1.5 Rajouter une clé et une valeur à un dictionnaire

Vous venez de recevoir 10 kakis, alors indiquez-le avec la commande `D["kakis"]=10`. Noter que rajouter une clé avec une valeur se fait de la même façon que modifier la valeur associée à une clé.

1.6 Accéder à la collection des clés et boucle

La commanda `D.keys()` renvoie la collection des clés. Ce n'est pas une liste, mais on peut la convertir en liste grâce à `list(D.keys())`.

```
for cle in D.keys():#for cle in D: fait la même chose en plus court
    print(cle)
    print(D[cle])
```

1.7 Effet de bord

Tout comme les listes, les dictionnaires sont soumis aux effets de bord. Une modification d'un dictionnaire (ajout d'une clé avec une valeur, modification de la valeur d'une clé) modifie le dictionnaire de façon globale même sans `return`.

1.8 Quels types de variables pour les clés et les valeurs ?

- Les valeurs peuvent être de n'importe quel type : `int`, `float`, `bool`, `str`, `list`, `dict`, `tuple` etc.
- Pour les clés, il n'est pas possible qu'une liste ou qu'un dictionnaire soit une clé [1].

1.9 Copie d'un dictionnaire

Si `L` est une liste, alors la commande `M = L` ne crée pas une liste distincte de `L` toute modification de `L` ou de `M` affectera l'autre liste. Il faut connaître les valeurs de `L` et `M` après les commandes suivantes :

```
L = [1,2,3]
M = L
M[2] = 5
```

La commande `M=L.copy()` pare ce problème :

```
L = [1,2,3]
M = L.copy()
M[2] = 5
```

À noter que cette parade ne fonctionne pas si les éléments de `L` sont eux-même des listes, testez :

```
L = [[1,2,4],2,3]
M = L.copy()
M[0][2]  = 5
```

Pour éviter ce désagrément, on peut effectuer une copie profonde grâce à une commande (hors programme). Pour les dictionnaires, c'est le même problème, le code suivant ne donne pas le résultat escompté :

```
D = {"a":1,"b":2,"c":3}
Dp = D
D["b"] = 5
```

En revanche, la commande `copy` fonctionne de la même manière que pour les listes :

```
D = {"a":1,"b":2,"c":3}
Dp = D.copy()
Dp["b"]=5
```

Mais comme pour les listes, cela ne marche pas si les valeurs sont eux-mêmes des listes ou des dictionnaires.

1. De manière général, un conteneur contenant une liste ou un dictionnaire ne peut pas être une clé d'un dictionnaire

2 Exercice classique utilisant les dictionnaires

Exercice 1. 1. Écrire une fonction `DictionnaireOccurrences(L)` qui, à une liste `L`, renvoie le dictionnaire des occurrences, c'est-à-dire que les clés du dictionnaire sont les éléments de `L` et la valeur associée à une clé vaut le nombre d'occurrences de cet élément dans la liste `L`. Par exemple, si `L=["a","ba","a","c","a","c"]`, cette fonction renvoie le dictionnaire `{"a":3, "ba":1,"c":2}`. Pour cela :

- Partir d'un dictionnaire vide
- Boucler sur chaque élément de la liste :
  - Soit cet élément est déjà une clé du dictionnaire et dans ce cas, rajouter 1 à la valeur correspondante (en effet, on avait déjà rencontré x fois cet élément avant et comme on le rencontre une fois de plus, on l'a donc rencontrée $x+1$ fois au total)
  - Soit cet élément n'est pas dans le dictionnaire et dans ce cas, rajouter le comme clé du dictionnaire avec une valeur de 1 (car c'est la première fois qu'on le rencontre donc on la rencontré une fois)

Cette fonction va maintenant être utilisée dans les questions suivantes :

2. En déduire une fonction `DeuxàDeuxDistincts(L)` qui renvoie `True` si les éléments de la liste sont deux à deux distincts et `False` sinon. Ainsi, la fonction renvoit `False` pour si `L=[1,3,1]` car `L[0]=L[2]`.
3. Écrire une fonction `LePlusSouvent(L)` qui renvoie la liste des éléments qui apparaissent le plus souvent dans la liste. Ainsi, la fonction renvoie `["a","c"]` si `L=["a","b","c","c","a","de"]` et `["a]` si `L=["a","b","c","c","a","de","c"]`.
4. Écrire une fonction `Permutation(L,M)` qui renvoie `True` si les éléments de `L` et `M` sont les mêmes avec le même nombre d'occurrences, autrement dit on passe de `M` à `L` par une permutation des éléments de la liste et `False` sinon.

5. Écrire une fonction `MêmesÉlements(L,M)` qui renvoie `True` si `L` et `M` ont les mêmes éléments mais pas forcément avec le même nombre d'occurrences.

3 Parcours en profondeur dans un graphe

3.1 Rappels sur les graphes

On rappelle qu'un graphe est formé d'un ensemble de sommets tel que si on prend une paire de sommets distincts, ces sommets sont reliés ou non par un arête (dans le cas où on peut aller dans les deux sens) ou un arc (le cas où on peut aller seulement dans un seul sens). Si on peut aller du point A au point B par une arête ou un arc, on dit que B est un voisin de A. Sur le dessin de graphe, ci-dessous, 1 a quatre voisins : 0, 3, 4 et 5 mais, 6 n'est pas un voisin de 1.

3.2 Concept de pile

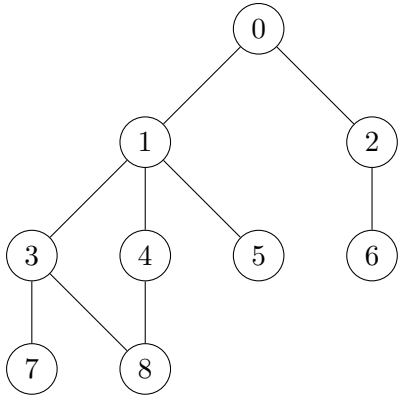
Imaginez une pile d'assiettes à laver. Quand une assiette sale arrive, on la met en haut de la pile, on dit qu'on l'«empile». Lorsqu'on prend une assiette dans la pile, c'est la dernière mise sur la pile qui est choisie, on dit qu'on la «dépile». C'est le principe, complètement dégueulasse, du «dernier arrivé, premier servi». En BCPST2, on modélise une pile par une liste munie des opérations suivantes : la création d'une pile vide, l'ajout d'un élément sur la pile, le retrait de l'élément situé au sommet de la pile.

3.3 Parcours en profondeur

Si on veut parcourir un graphe partant d'un sommet, on effectue ce que l'on appelle un **parcours en profondeur**. On constitue deux listes, une pile que l'on va remplier et dépiler tout au long de l'algorithme et une liste `Visités` contenant les sommets visités que l'on va remplir. On utilise une `pile` afin d'aller au bout de chaque branche l'algorithme s'arrête quand la pile est vide[2]. Plus précisément :

1. On place le sommet de départ dans la pile.
2. Tant que la pile n'est pas vide :
   - On dépile un élément de la pile (c'est-à-dire qu'on prend le dernier élément de la pile et on le retire). Nommons `S` cet élément. On rajoute `S` à la liste `Visités` sauf s'il est déjà dedans.
   - On rajoute à la pile les voisins de `S` sauf s'ils ont déjà été visités.

Prenons le graphe suivant :



Ainsi, partant du sommet 0, l'algorithme se déroule en plusieurs étapes :

1. `P = [0]`, `V = []`
2. `P = [1,2]`, `V = [0]`
3. `P = [1,6]`, `V = [0,2]`
4. `P = [1]`, `V = [0,2,6]`
5. `P = [3,4,5]`, `V = [0,2,6,1]`
6. `P = [3,4]`, `V = [0,2,6,1,5]`
7. `P = [3,8]`, `V = [0,2,6,1,5,4]`
8. `P = [3,3]`, `V = [0,2,6,1,5,4,8]`
9. `P = [3,7]`, `V = [0,2,6,1,5,4,8,3]`
10. `P = [3]`, `V = [0,2,6,1,5,8,3,7]`
11. `P = []`, `V = [0,2,6,1,5,8,3,7]`

La pile est vide donc on s'arrête.

3.4 Parcours et labyrinthe

Exercice 2. On modélise un labyrinthe comme une matrice (une liste de listes) ayant n lignes et p colonnes les cases blanches sont codées par

2. En BCPST1, vous aviez vu le parcours en largeur qui utilise une file, où c'était l'élément arrivé en premier qui était traité en premier.

des 1 et les cases noires par des 0. On va coder un parcours pour sortir du labyrinthe partant de l'entrée. On modélise ce labyrinthe comme un graphe dont les sommets sont les cases blanches. On considère que deux sommets sont relis par une arête s'ils sont adjacents : Chaque case blanche est représentée par un couple (i, j) où i est l'indice de la ligne et j son indice de colonne. On considère que les voisins d'un sommet (i, j) sont les cases blanches adjacentes à (i, j) soit les couples valides $(i, j-1), (i+1, j), (i, j+1), (i-1, j)$ dont le coefficient de la matrice vaut 1. Par valide, on entend qu'il faut vérifier que $(i, j-1)$ soit bien un coefficient de la matrice (avant de vérifier que c'est une case blanche), c'est-à-dire qu'il faut vérifier que $0 \leqslant i \leqslant p-1$ et $0 \leqslant j-1 \leqslant p-1$.

1. Téléchargez les fichiers `TP13M1.txt` et `TP13M2.txt` qui sont sur CDP et mettez les dans le même dossier que votre script Python actuel.
2. Les commandes suivantes ouvrent le labyrinthe :

```
import matplotlib.pyplot as plt#bibliothèque pour les images
import pickle#bibliothèque pour importer des fichier

with open("TP13M1.txt", "rb") as fichier:
    M = pickle.load(fichier)#M: grille chargée à partir du fichier

print(M[0][0]) # Valeur du pixel première ligne première colo
print(M[0][1]) # Valeur du pixel première ligne, 2ième colonn
print(M[1][1]) # Valeur du pixel 2ième ligne 2-ième colonne
plt.figure()
plt.imshow(M,cmap="gray")#Affiche la matrice en niveau de gri
plt.show()
n,p = len(M),len(M[0])#n: le nb de lignes, p: le nb de colonn
source = (0,1)#entrée du labyrinthe
but = (n-1,p-2)#sortie du labyrinthe
```

   Ainsi, la source $(0, 1)$ a un sommet : le sommet $(1, 1)$, lui-même a trois voisins : celui au-dessus de lui, celui en dessous et celui à sa droite. La case de gauche étant noire ce n'est pas un sommet du graphe et donc pas un voisin de $(1, 1)$.
3. Écrire une fonction `ListeAdj(M,sommet)` qui à un sommet de la forme (i, j) renvoie la liste $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ et $(i, j+1)$.
4. En déduire une fonction `ListeAdjValide(M,sommet)` qui renvoie les éléments de la liste créée par la fonction précédente formée d'indices de lignes et de colonnes valides. Par valide, on entend un couple (x, y) avec $0 \leqslant x \leqslant n-1$ et $0 \leqslant y \leqslant p-1$.
5. En déduire une fonction `ListeVoisins(M,sommet)` qui renvoie la liste des voisins du sommet c'est-à-dire la liste des **cases blanches** qui sont voisins de `sommet`).
6. Coder le parcours partant du sommet `source` en codant une fonction `ParcoursProfondeur(M,source)`
7. Modifier la fonction précédente, pour indiquer à chaque fois que l'on visite un sommet par quel sommet on est passé pour le visité. Cela consiste donc à créer un dictionnaire `P` (dictionnaire des prédécesseurs) tel que `P[v]=sommet` si le sommet `v` a été ajouté à la pile en tant que voisin de `sommet`.
8. Tracer le chemin de l'entrée (sommet `source`) à la sortie (sommet `but`), pour cela partir du sommet `but` et remonter via le dictionnaire des prédécesseurs, chacun de ces sommets sera modifié dans la matrice, le coefficient ne vaudra plus 1 (case blanche) mais 0.5 (case grise).