Dictionnaires
Rappels et exercices

MP2I Option SI: Informatique tronc commun

1 Dictionnaires

Un dictionnaire est une structure de données permettant de stocker des **associations** entre un ensemble de clés et un ensemble de valeurs. Chaque clé du dictionnaire est associée à une unique valeur, que l'on peut trouver et modifier efficacement. Voici les opérations de base des dictionnaires :

```
# Créer un dictionnaire vide
d_vide = dict()

# Créer un dictionnaire à partir de quelques valeurs
# une ligne K: V signifie "à la clé K, on associe la valeur V"
d = {
  "Euler": "Leonhard",
  "Riemann": "Bernhard"
}

# Associer à la clé k la valeur v dans le dictionnaire d: d[k] = v
d["Fermat"] = "Pierre"
# Si la clé avait déjà une valeur associée, elle est écrasée
d["Fermat"] = "Pierrot"

# Lire la valeur associée à k dans d: d[k]
x = d["Fermat"]
assert(x == "Pierrot")

# Savoir si la clé k est présente dans d
# k in d
if "Euler" in d:
  print(d["Euler"])
else:
  print("Euler n'est pas dans le dictionnaire")
```

Le dictionnaire créé dans le code ci-dessus associe à plusieurs noms de mathématiciens célèbres le prénom correspondant. Ainsi, le dictionnaire à deux associations créé initialement se lit : "Le nom Euler a pour prénom Leonhard, le nom Riemann a pour prénom Bernhard". Dans un dictionnaire, une clé ne peut apparaître qu'une fois. Par exemple, un mathématicien d'un nom donné ne peut pas avoir plusieurs prénoms. En revanche, rien n'empêche une valeur d'être présente plusieurs fois, associée à différentes clés. Par exemple, si l'on rajoute Charles Babbage et Charles Hermite au dictionnaire :

```
d["Babbage"] = "Charles"
d["Hermite"] = "Charles"
```

On peut **itérer** sur les clés d'un dictionnaire avec la syntaxe `for k in d`.

Exemple 1. La fonction `occurrences` suivante prend en entrée une liste L, et construit un dictionnaire dont les clés sont les valeurs de L, et où chaque valeur est associée à son nombre d'occurrences dans L. Le code utilise cette fonction pour calculer puis afficher les occurrences des éléments d'une liste :

```
def occurrences(L):
  d = dict() # dictionnaire des occurrences
  for x in L:
    if x not in d:
      d[x] = 1 # 1ère fois que l'on croise x
    else:
      d[x] = d[x] + 1
  return d

L = ["A", "B", "B", "C", "A", "D", "A"]
d = occurrences(L)
for x in d:
  print(f"{x} apparait {d[x]} fois dans la liste")
```

Exécuter le code ci-dessus affiche :

```
A apparaît 3 fois dans la liste
B apparaît 2 fois dans la liste
C apparaît 1 fois dans la liste
D apparaît 1 fois dans la liste
```

Les dictionnaires Python sont implémentés à l'aide d'une structure efficace appelée **table de hachage** : on peut considérer qu'en pratique les opérations des dictionnaires Python sont en $\mathcal{O}(1)$.

2 Exercices

Exercice 1.

Écrire une fonction `indices` prenant en entrée une liste L de taille n, et calculant un dictionnaire O associant à chaque valeur de L la **liste** des indices où elle apparaît dans L.
Par exemple, le code suivant

```
L = ["A", "B", "B", "C", "A", "D", "A"]
I = indices(L)
print(I)
```

devra afficher :

```
{"A": [0, 4, 6], "B": [1, 2], "C": [3], "D": [5]}
```

Exercice 2.

Écrire une fonction `entrees(D)` prenant en entrée un dictionnaire D et renvoyant une liste de tous les couples clé-valeur de D. Par exemple :

```
D = {"A": 5, "B": 2, "C": 6}
L = entrees(D)
print(L)
```

devra afficher `[("A", 5), ("B", 2), ("C", 6)]`

Exercice 3.

On considère des graphes orientés, représentés par des dictionnaires de listes d'adjacence : à chaque sommet u, on associe la liste de tous ses successeurs. Ainsi, le dictionnaire suivant :

```
G = {
  'A': ['B', 'C'],
  'B': ['A', 'D'],
  'C': ['B', 'E'],
  'D': ['A'],
  'E': ['D'],
}
```

représente un graphe à 5 sommets $A, B, \dots E$, avec comme arcs $(A, B), (A, C), (B, A), \dots, (E, D)$

Q1. Écrire une fonction prenant en entrée un graphe G et une liste L de sommets, et déterminant si elle forme un chemin valide dans G.

Q2. Écrire une fonction prenant en entrée un graphe G et un sommet s de G, et générant un chemin de la manière suivante :

- On fixe $s_0 = s$;
- On choisit s_1 uniformément parmi les successeurs de s_0 ;
- ...
- On choisit s_k uniformément parmi les successeurs de s_{k-1} sans autoriser les sommets $s_0, \dots s_{k-2}$.
- Et ainsi de suite jusqu'à tomber sur un sommet n'ayant aucun successeur pas encore visité.

On pourra utiliser un ensemble ou un dictionnaire pour stocker les sommets visités

Q3. Implémenter l'algorithme du parcours en largeur sur les graphes. On renverra un tableau des prédécesseurs représentant l'arborescence du parcours.

Exercice 4.

On considère une grille carrée $n \times n$ sur laquelle on peut se déplacer de case en case. Certaines cases sont inaccessibles, et à partir d'une case accessible, on peut, en une unité de temps, se déplacer vers n'importe laquelle des 4 cases adjacentes, du moment qu'elle est aussi libre. On représente cette grille par un tableau 2D de booléens : `True` indique une case libre, et `False` une case inaccessible.
De plus, des raccourcis permettent de se déplacer entre certaines cases. On dispose donc d'un dictionnaire `raccourcis` dont les clés sont des coordonnées (i, j), et où la valeur associée à une clé (i, j) est une autre coordonnée (i', j') : une telle association signifie qu'il est possible de passer de la case (i, j) à la case (i', j') en une unité de temps. Attention, les raccourcis ne sont pas a priori à double sens !

Q1. Écrire une fonction `accessibles(T, R, i, j)` prenant en entrée une grille de booléens et un dictionnaire de raccourcis, ainsi qu'une coordonnée de case (i, j), et renvoyant l'**ensemble** des cases accessibles dans T depuis (i, j), en prenant en compte les raccourcis de R. Si la case (i, j) n'est pas accessible, on renverra une liste vide.

Q2. Écrire une fonction `chemin_valide(T, R, L)` prenant en entrée T et R comme dans la question précédente, ainsi qu'une liste L de cases, et vérifiant que la liste forme un chemin valide.

On rappelle qu'un parcours en largeur à l'aide d'une file permet de calculer des plus courts chemins dans un graphe. De plus, au cours d'un tel parcours, les sommets sont énumérés dans l'ordre de leur distance au sommet source. !

Q3. Écrire une fonction `rayon(T, R, i, j, d)` prenant en entrée une grille de booléens T, un dictionnaire de raccourcis R, une case de départ (i, j), et une distance maximale d, et déterminant la liste des cases de T se situant à une distance inférieure à d de (i, j), en prenant en compte les raccourcis de R.

Q4. Écrire une fonction `ajout_optimal(T, R, d)` prenant en entrée une grille et ses raccourcis, ainsi qu'une distance limite, et déterminant un **nouveau** raccourci à ajouter, tel que le nombre de cases se trouvant à une distance inférieure à d de $(0, 0)$ en autorisant ce nouveau raccourci mais pas sans est maximal.

Le problème d'optimisation de la dernière question permettrait par exemple de modéliser (de manière simpliste) une ville dans laquelle on cherche à ajouter une ligne de transports, de façon à ce qu'un maximum d'endroits deviennent accessibles en moins de 10 minutes.

Exercice 5.

Une usine fabrique différents composants mécaniques. Chaque composant est fabriqué à partir d'autres composants plus basiques. Les matières premières sont modélisées comme des composants pouvant être fabriqués immédiatement, sans sous-composants. On représente cette situation par un dictionnaire de recettes : à chaque composant on associe sa recette, qui est une liste de couples (x, k) où x est un des sous-composants nécessaires, et k la quantité requise. Par exemple, on considère une usine fabriquant des moteurs (M), utilisant comme composants intermédiaires des rotors (R), stators (S), écrous (E), tuyaux (P), tiges (T) et fils (F), à partir de trois matières premières : charbon (Ca), fer (Fe) et cuivre (Cu). Son dictionnaire serait :

```
recettes_moteur = {
  "M": [("R", 1), ("S", 1)],

  "R": [("Cu", 12), ("E", 8), ("T", 4)],
  "S": [("P", 3), ("F", 20)],

  "E": [("Fe", 8)],
  "P": [("Ca", 5), ("Fe", 5)],
  "T": [("Fe", 5)],
  "F": [("Cu", 1)],

  "Ca": [],
  "Fe": [],
  "Cu": [],
}
```

Les quantités sont données dans des unités arbitraires. Ainsi, pour produire 1 moteur, il faut 1 rotor et 1 stator, et pour produire un stator il faut 3 unités de tuyaux et 20 unités de fils.

On s'intéresse au calcul des matières premières nécessaires pour produire un composant. Pour commencer, on ignore les quantités, et l'on veut récupérer l'**ensemble** des matières premières nécessaires.

Q1. Écrire une fonction prenant en entrée un dictionnaire de recettes comme décrit plus haut, et renvoyant la liste des matières premières de l'usine. Sur l'exemple, la fonction renverrait la liste `["Ca", "Fe", "Cu"]`.

Q2. Écrire une fonction `mat_necessaires(d, c)` prenant en entrée un dictionnaire de recettes ainsi qu'un composant, et calculant la liste des matières premières réellement nécessaires pour produire ce composant. Par exemple, l'appel `mat_necessaires(recettes_moteur, "R")` renverrait `["Ca", "Fe"]`, car il n'y a pas besoin de cuivre pour produire un rotor.
On pourra remarquer que le dictionnaire décrit en réalité un graphe, où les successeurs d'un composant sont les sous-composants nécessaires à sa fabrication.

Q3. Écrire une fonction `materiaux(d, c)` calculant pour un composant donné, la quantité de chaque matière première devant être utilisée. La fonction renverra un dictionnaire associant à chaque matière première le nombre **total** d'unités de cette matière première devant être utilisées.
Par exemple, l'appel `materiaux(recettes_moteur, "M")` devra renvoyer :

```
{'Cu': 32, 'Fe': 99, 'Ca': 15}
```