ICC, partie Programmation — Série d'exercices 5

GC/MX, 19 octobre 2019

À faire individuellement ou par petits groupes de deux ou trois.

Exercice 1. Matrices: représentation et manipulations

Dans cet exercice, nous allons représenter des matrices par des **`List[List[float]]`** et les manipuler. L'idée de base est de se dire qu'une matrice telle que

$$M_1 = \begin{bmatrix} 1 & 2 & 3 \\ 9 & 8 & 7 \end{bmatrix}$$

à $m = 2$ lignes et $n = 3$ colonnes peut être représentée en Python par une liste de 2 éléments, chacun de ces éléments représentant une ligne de la matrice et étant lui-même une liste de 3 éléments:

```python
from typing import List

# on définit le type Matrix comme étant équivalent à une liste de liste de floats
Matrix = List[List[float]]

# on définit une matrice
m1: Matrix = [[1, 2, 3], [9, 8, 7]]
```

(a) Définissez **`m1`** comme ci-dessus. Définissez ensuite, de manière similaire à la dernière ligne ci-dessous, les matrices suivantes:

$$M_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, M_3 = \begin{bmatrix} 0 & 2 \\ 2 & 0 \end{bmatrix}, M_4 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 5 & 7 \end{bmatrix}, M_5 = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}, M_6 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

(b) Écrivez une fonction **`print_matrix()`** qui prend une matrice en paramètre et qui l'affiche sur le terminal formatée un peu plus joliment que ce que **`print()`** ferait. Par exemple, **`print_matrix(m1)`** doit afficher ceci:

```
[
 1 2 3
 9 8 7
]
```

Indice: la fonction **`print()`** a un paramètre optionnel appelé **`end`**. Par défaut, sa valeur est le caractère de nouvelle ligne: c'est pour ça que, normalement, **`print()`** affiche un retour à la ligne après avoir affiché ses arguments. Cependant, il est possible de changer sa valeur pour ne pas provoquer de retour à la ligne automatique à chaque appel de **`print()`**.

Créez une liste avec toutes les matrices de **`m1`** à **`m6`** et affichez-les avec cette fonction à l'aide d'une boucle **`for-in`**.

(c) Écrivez une fonction **`dim_m()`** qui prend une matrice en paramètre et qui retourne sa dimension m, son nombre de lignes.

(d) Écrivez une fonction **`dim_n()`** qui prend une matrice en paramètre et qui retourne sa dimension n, son nombre de colonnes. Attention aux matrices potentiellement vides. S'il se trouve que vous trouvez plusieurs lignes qui n'ont pas le même nombre d'éléments, retournez **`0`**.

(e) En utilisant **`dim_m()`** et **`dim_n()`**, affichez la dimension de toutes vos matrices.

(f) Écrivez une fonction **`empty()`** qui renvoie la matrice zéro de taille **`m`** par **`n`**, où **`m`** et **`n`** sont des paramètres de la fonction.

(g) Écrivez une fonction **`identity()`** qui renvoie la matrice identité de taille **`n`** par **`n`**, où **`n`** est un paramètre de la fonction. Vérifiez que **`identity(3) == m2`**.

(h) Écrivez une fonction **`mult()`** qui prend deux matrices en paramètres et qui renvoie une nouvelle matrice obtenue par la multiplication des deux matrices d'entrée, ou **`[]`** si la multiplication n'est pas possible.

Rappel: la multiplication de deux matrices A, B de taille $m_1 \times n_1$ et $m_2 \times n_2$, respectivement, est possible si et seulement si $n_1 = m_2$. Le résultat est une matrice C de taille $m_1 \times n_2$, dans laquelle on a pour chaque élément:

$$c_{ij} = \sum_{k=1}^{n_1} a_{ik} b_{kj}.$$

Testez votre fonction **`mult()`** avec quelques exemples.

Exercice 2. MyLittleFacebook

Nous poursuivons ici l'exemple vu au cours:

```python
from typing import Dict, Set

# MyLittleFacebook
friendships: Dict[str, Set[str]] = {}

def add_friends(name1: str, name2: str) -> None:
    if name1 in friendships:
        friendships[name1].add(name2)
    else:
        friendships[name1] = {name2}
    if name2 in friendships:
        friendships[name2].add(name1)
    else:
        friendships[name2] = {name1}

add_friends("Alex", "Victor")
add_friends("Alex", "Emelyne")
add_friends("Alex", "Emelyne")
add_friends("Emelyne", "Rose")
```

(a) Ajoutez sous ce code une fonction **`known_people()`** qui nous donne, dans l'ordre alphabétique, tous les noms pour lesquels le dictionnaire contient des relations d'amitié.

Indices: Quel devra être son type de retour? Comment obtenir l'ensemble des clés connues par un dictionnaire? Vous pouvez utiliser la fonction **`sorted()`** de Python.

(b) Ajoutez une fonction **`friends_of()`**, qui prend un paramètre **`name`** de type **`str`** et qui renvoie l'ensemble des amis liés à ce nom. Prenez soin de retourner un set vide lorsque le dictionnaire n'a aucune info sur la personne demandée.

Vous pouvez tester avec ceci:

```python
print(friends_of("Emelyne")) # {'Rose', 'Alex'}
print(friends_of("Rachel"))  # set()
```

(c) Ajoutez une fonction **`are_friends()`**, qui indique si les deux noms passés comme paramètres sont déclarés comme amis. Quel est le type de retour?

Testez avec ceci:

```python
print(are_friends("Alex", "Victor")) # True
print(are_friends("Victor", "Alex")) # True
print(are_friends("Alex", "Rose"))   # False
print(are_friends("Alex", "Rachel")) # False
print(are_friends("Rachel", "Alex")) # False
```

(d) Si l'on passe par la méthode **`add_friends()`**, la modification du dictionnaire est toujours symétrique par rapport aux deux noms passés en paramètre. Mais on pourrait faire exprès de créer une incohérence en allant modifier le dictionnaire de manière asymétrique:

```python
friendships["Alex"].add("Rachel") # Modification asymétrique
```

Pour détecter cette situation problématique, ajoutez une fonction **`is_data_consistent()`**, qui dira si oui ou non l'ensemble du contenu du dictionnaire est cohérent, c'est-à-dire qui renverra **`True`** lorsque A fait partie des amis de B si et seulement si B fait partie des amis de A, et **`False`** sinon.

Indice: vous pouvez itérer sur chaque clé **`k`** liée à sa valeur **`v`** dans un dictionnaire **`d`** comme ceci:

```python
for k, v in d.items():
    ...
```

Testez avec ceci:

```python
print(f"Consistent before modification? {is_data_consistent()}")  # True
friendships["Alex"].add("Rachel") # Modification asymétrique
print(f"Consistent after modification? {is_data_consistent()}")   # False
```

Exercice 3. Recherche dichotomique

Ici, nous profitons des propriétés d'une liste triées pour chercher efficacement si elle contient un élément donné et, si c'est le cas, à quelle position.

La recherche dichotomique (*binary search* en anglais) fonctionne avec une liste triée (ici, dans l'ordre croissant) et un élément à rechercher, disons e. On regarde d'abord l'élément du milieu de la liste, disons m: s'il a la même valeur que e, on a terminé la recherche (avec, admettons-le, pas mal de chance), et on a trouvé la position de e dans la liste. Sinon, il y a deux cas de figure: soit $e < m$, et alors on continue à chercher e, mais dans la moitié inférieure de la liste seulement — donc depuis le début jusqu'à la position qui précède m. Sinon, si $e > m$, on continue aussi à chercher e, mais cette fois dans l'autre moitié de la liste: celle qui commence à l'élément qui suit m et jusqu'à la fin de la liste.

La recherche dans la moitié inférieure ou supérieure de la liste se déroule de la même façon: on prend l'élément qui se trouve au milieu de la moitié de la liste dans laquelle on recherche, et soit on trouve e, soit on procède comme ci-dessus, et on continue de chercher dans un quart de la liste initiale.

On finit forcément soit par trouver e quelque part, soit par se retrouver à chercher e dans une zone de la liste qui ne contient plus d'éléments, et alors on sait que la liste ne contient pas e. Ceci, avec ces critères précis, fonctionne uniquement, rappelons-le, si la liste d'entrée est triée dans l'ordre croissant.

L'exercice consiste à compléter la fonction **`binary_search()`** dans le code ci-dessous pour implémenter une recherche dichotomique selon cette description. Cette fonction doit retourner l'index de **`item`** si **`item`** est un élément de **`values`**, ou **`-1`** si ce n'est pas le cas.

```python
from typing import List

# on définit une liste de nombre et on la trie
numbers = sorted([1, 5, 6, 2, 8, 12, 5])
print(numbers)

def binary_search(values: List[int], item: int) -> int:
    ... # à compléter

# pour chaque nombre dans la liste, la fonction de recherche doit le retrouver
for v in numbers:
    print(binary_search(numbers, v))

# pour des éléments non existants, la fonction de recherche doit retourner -1
print(binary_search(numbers, 3))
print(binary_search(numbers, 7))
print(binary_search(numbers, 42))
```