À réaliser individuellement ou en groupes de deux ou trois personnes.

Exercice 1. Ensembles : représentation et opérations

Dans cet exercice, nous explorons le type **`set`** de Python et l'utilisons pour effectuer des opérations sur des collections d'éléments.

Rappelons qu'un ensemble est une collection non ordonnée d'éléments uniques. En Python, on peut créer un ensemble à partir d'une liste à l'aide de **`set()`**, ou le définir directement avec des accolades :

```
words = ["apple", "banana", "apple", "cherry", "banana", "apple"]
unique_words = set(words)
print(unique_words)  # p.ex. {'apple', 'banana', 'cherry'}

a = {1, 2, 3, 4, 5}   # ensemble défini directement
```

Les doublons sont automatiquement supprimés. On peut également vérifier efficacement l'appartenance à un ensemble avec **`in`** :

```
print("apple" in unique_words)   # True
print("mango" in unique_words)   # False
```

Python fournit aussi des opérateurs intégrés pour les opérations courantes sur les ensembles. Étant donné deux ensembles **`a`** et **`b`**, l'expression **`a | b`** retourne leur union (tous les éléments apparaissant dans l'un ou l'autre), **`a & b`** retourne leur intersection (uniquement les éléments présents dans les deux), et **`a - b`** retourne leur différence (les éléments de **`a`** qui ne sont pas dans **`b`**). Enfin, **`len(s)`** retourne le nombre d'éléments d'un ensemble **`s`**, et **`s.add(x)`** insère un élément **`x`** dans l'ensemble **`s`**.

(a) Créez les deux ensembles suivants :

```
a = {1, 2, 3, 4, 5}
b = {4, 5, 6, 7, 8}
```

Puis, sans utiliser les opérateurs intégrés **`|`**, **`&`**, **`-`**, implémentez les trois fonctions suivantes manuellement à l'aide de boucles :

- **`my_union(a, b)`** — retourne un nouvel ensemble contenant tous les éléments apparaissant dans **`a`** ou dans **`b`** (ou dans les deux).
- **`my_intersection(a, b)`** — retourne un nouvel ensemble contenant uniquement les éléments présents à la fois dans **`a`** et dans **`b`**.
- **`my_difference(a, b)`** — retourne un nouvel ensemble contenant les éléments de **`a`** qui ne sont pas dans **`b`**.

Vérifiez chacune de vos fonctions en la comparant à l'opérateur intégré correspondant (**`a | b`**, **`a & b`**, **`a - b`**).

(b) Considérez la liste de mots suivante :

```
text = [
    "the", "cat", "sat", "on", "the", "mat",
    "the", "cat", "sat", "on", "a", "mat"
]
```

Écrivez une fonction **`unique_words(words)`** qui prend une liste de mots en paramètre et retourne un **`set`** contenant tous les mots apparaissant au moins une fois. Affichez le résultat pour **`text`**.

(c) Écrivez une fonction **`is_subset(a, b)`** qui retourne **`True`** si chaque élément de **`a`** est également dans **`b`**, et **`False`** sinon — sans utiliser les opérateurs intégrés **`<=`** ou **`>=`**.
Vérifiez votre fonction avec quelques exemples, par exemple :

```
is_subset({1, 2}, {1, 2, 3})   # True
is_subset({1, 6}, {1, 2, 3})   # False
is_subset({1, 2}, {1, 2})      # True
```

(d) Écrivez une fonction **`my_equals(a, b)`** qui retourne **`True`** si deux ensembles contiennent exactement les mêmes éléments, et **`False`** sinon — sans utiliser **`==`**.
Indication : réfléchissez à ce que signifie l'égalité de deux ensembles en termes de la fonction **`is_subset()`** que vous venez d'écrire.

Exercice 2. Matrices : représentation et manipulation

Dans cet exercice, nous allons représenter des matrices sous la forme **`list[list[float]]`** et les manipuler. L'idée de base est qu'une matrice telle que

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

avec $m = 2$ 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
# ceci est une définition de type : on indique que chaque fois
# que l'on écrit Matrix comme type, cela signifie list[list[float]].
# On ne fait cela qu'une seule fois dans le programme. Notez que
# cela ne définit pas une matrice elle-même, mais précise comment
# une matrice est représentée à l'aide des types Python.
Matrix = list[list[float]]

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

Une liste de listes a la propriété particulière que l'on utilise un double indexage pour désigner une valeur dans la matrice. Si l'on écrit **`m1[0]`**, on désigne la sous-liste **`[1, 2, 3]`** — qui peut elle-même être indexée. Pour obtenir la valeur 2, par exemple, on écrirait **`m1[0][1]`**, ce qui peut se lire ainsi : « accéder à la variable **`m1`**, aller à son élément numéro 0 qui est une sous-liste, puis aller à l'élément numéro 1 de cette sous-liste ».

De même, pour remplacer le 7 par un 14, on écrirait **`m1[1][2] = 14`**, ce qui se lit : « dans la liste **`m1`**, prendre la sous-liste à l'indice 1, et dans cette sous-liste, aller à l'indice 2 (qui contient actuellement 7) et y écrire la nouvelle valeur 14 ».

(a) Définissez **`m1`** comme ci-dessus. Puis définissez, de manière similaire à la dernière ligne ci-dessus, 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 l'affiche dans le terminal de façon un peu plus lisible que ce que ferait **`print()`**. Par exemple, **`print_matrix(m1)`** devrait afficher :

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

Pour ce faire :

- Affichez le crochet ouvrant sur une ligne
- Utilisez une boucle pour parcourir les lignes de la matrice. À l'intérieur :
  - Utilisez une boucle pour parcourir les colonnes de la ligne courante et afficher chaque composante.
    Indication : la fonction **`print()`** possède un paramètre optionnel appelé **`end`**. Par défaut, sa valeur est le caractère de retour à la ligne : c'est pourquoi, normalement, **`print()`** affiche un saut de ligne après ses arguments. Il est cependant possible de modifier cette valeur pour supprimer le saut de ligne automatique : **`print(my_variable, end=' ')`**
- À la fin des deux boucles, affichez le crochet fermant

Créez une liste contenant 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 retourne sa dimension m, c'est-à-dire son nombre de lignes.

(d) Écrivez une fonction **`dim_n()`** qui prend une matrice en paramètre et retourne sa dimension n, c'est-à-dire son nombre de colonnes. Faites attention aux matrices potentiellement vides. Si plusieurs lignes n'ont pas le même nombre d'éléments, retournez **`0`**.

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

(f) Écrivez une fonction **`empty()`** qui retourne la matrice nulle de taille **`m`** par **`n`**, où **`m`** et **`n`** sont des paramètres de la fonction. Veillez à créer une nouvelle liste de zéros pour chaque ligne de la matrice et à ne pas réutiliser la même. Si nécessaire, utilisez **`my_list.copy()`** pour créer une copie indépendante de la liste **`my_list`**.

(g) Écrivez une fonction **`identity()`** qui retourne 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 retourne une nouvelle matrice obtenue en multipliant les deux matrices d'entrée, ou **`[]`** si la multiplication n'est pas possible.

Rappel : la multiplication de deux matrices A, B de tailles $m_1 \times n_1$ 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 $m_1 \times n_2$, dont chaque élément vaut :

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

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

Exercice 3. Recherche dichotomique

Nous exploitons ici les propriétés d'une liste triée pour rechercher efficacement si elle contient un élément donné et, le cas échéant, à quelle position.

La recherche dichotomique fonctionne avec une liste triée (ici par ordre croissant) et un élément à rechercher, noté e. On examine d'abord l'élément du milieu de la liste, noté m : s'il a la même valeur que e, la recherche est terminée (avec, il faut l'admettre, beaucoup de chance) et on a trouvé la position de e dans la liste. Sinon, deux cas se présentent : soit $e < m$, et on continue la recherche de e, mais uniquement dans la moitié inférieure de la liste — c'est-à-dire du début jusqu'à la position précédant m. Soit $e > m$, et on continue également la recherche de e, mais cette fois dans l'autre moitié de la liste : celle qui commence à l'élément suivant m et va jusqu'à la fin.

La recherche dans la moitié inférieure ou supérieure se déroule de la même façon : on prend l'élément du milieu de la portion de liste dans laquelle on cherche, et soit on trouve e, soit on procède comme ci-dessus en continuant la recherche dans un quart de la liste originale.

On finit inévitablement soit par trouver e, soit par chercher e dans une région de la liste ne contenant plus aucun élément, auquel cas on sait que la liste ne contient pas e. Ceci, avec ces critères précis, ne fonctionne, rappelons-le, que si la liste d'entrée est triée par ordre croissant.

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

Le code de départ se trouve à la page suivante.

```python
# on définit une liste de nombres 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 de la liste, la fonction doit le retrouver
for v in numbers:
    print(binary_search(numbers, v))

# pour les éléments absents, la fonction doit retourner -1
print(binary_search(numbers, 3))
print(binary_search(numbers, 7))
print(binary_search(numbers, 42))
```