Asie - juin 2025 - sujet 1

Exercice 1 (Décidabilité, algorithmique et programmation python - 6 points)
En Python, on peut utiliser le triple guillemet pour écrire une chaîne de caractères sur plusieurs lignes. Par exemple, on peut définir une variable `programme1` qui contient la chaîne de caractères correspondant à un petit programme Python de la manière suivante :

```
1 programme1 = """
2 x = 10
3 y = 10
4 while x > 0:
5     x = x - 1
6     y = y + 1
7 """
```

De même, on peut définir la variable `programme2` :

```
1 programme2 = """
2 def boucle_infinie():
3     while True:
4         pass # Ne rien faire
5 boucle_infinie()
6 """
```

1. On suppose que l'on exécute le programme contenu dans la variable `programme1`. Donner les valeurs de `x` et de `y` après exécution de ce programme.
2. Expliquer pourquoi tout programme Python peut être vu comme une chaîne de caractères.

En Python, la fonction `exec` permet d'exécuter le programme correspondant à une chaîne de caractères passée en paramètre.

```
>>> exec("r = 42")
>>> r
42
>>> exec(programme1)
>>> x + y
20
>>> exec(programme2)
[ne termine pas]
```

On considère les quatre variables `programme3`, `programme4`, `programme5` et `programme6` suivantes :

```
programme3 = """
x = 10
while x != 0:
    x = x - 2
"""

programme4 = """
x = 10
while x > 0:
    x = x + 2
"""

programme5 = """
x = 10
while x < 0:
    x = x + 4
"""

programme6 = """
x = 10
while x != 0:
    x = x - 4
"""
```

3. On exécute les variables `programme3`, `programme4`, `programme5` et `programme6` avec la fonction `exec`. Déterminer lesquelles terminent et lesquelles ne terminent pas.

On cherche à écrire une fonction `arret` telle que `arret(programme)` renvoie `True` si `exec(programme)` termine et `False` si `exec(programme)` ne termine pas. Cette fonction `arret` doit donc s'arrêter dans tous les cas.

4. Indiquer ce que réalise le programme suivant et s'il permet de répondre au problème posé ci-dessus.

```
def arret_essai1(programme):
    exec(programme)
    return True
```

On suppose disposer d'une fonction `recherche(mot, texte)` qui renvoie `True` si une chaîne de caractères `mot` est présente dans une chaîne de caractères `texte` et `False` sinon.

5. Expliquer succinctement le principe de l'algorithme de Boyer-Moore qui permet d'implémenter cette fonction `recherche`.

Une idée est d'écrire une fonction qui décrète qu'un programme s'arrête s'il ne contient pas de `while` et ne s'arrête pas s'il en contient un.

6. Ecrire une fonction `arret_essai2(programme)` qui renvoie `True` si la chaîne de caractères `"while"` n'est pas utilisée dans la chaîne de caractères programme et `False` sinon.
7. Montrer qu'il est possible que :
   - `arret_essai2(programme)` renvoie `True` alors que le programme ne s'arrête pas ;
   - `arret_essai2(programme)` renvoie `False` alors que le programme s'arrête.

   Indication : il n'y a pas que les boucles `while` qui peuvent poser des problèmes de non terminaison.

Nos tentatives pour écrire une telle fonction `arret` sont restées vaines. Nous allons montrer qu'il est en réalité *impossible* d'écrire une telle fonction. On va supposer qu'une telle fonction `arret` existe. On va montrer que cette supposition aboutit à un paradoxe ce qui prouvera que la supposition est fausse.

8. Ecrire une fonction `terminaison_inverse` telle que l'appel `terminaison_inverse(programme)` termine si la chaîne de caractères `programme` représente un programme qui ne termine pas et ne termine pas si la chaîne de caractères `programme` représente un programme qui termine. On pourra utiliser la fonction `boucle_infinie` de `programme2` ainsi bien sûr que la fonction `arret` dont on a supposé l'existence.

On considère `"terminaison_inverse(programme_paradoxal)"` qui n'est rien d'autre qu'une chaîne de caractères, et on définit une variable que l'on appelle `programme_paradoxal` à laquelle on affecte cette chaîne de caractères :

```
programme_paradoxal = "terminaison_inverse(programme_paradoxal)"
```

9. Etudier si le programme paradoxal termine ou non, c'est-à-dire si `exec(programme_paradoxal)` termine ou non.
10. Indiquer ce que l'on peut conclure sur la fonction `arret`.
11. Expliquer si l'impossibilité d'écrire une telle fonction `arret` est due aux limitations du langage Python.

Exercice 2 (Arbre et compression de fichier texte - 6 points)
Quand il s'agit de transmettre de l'information sur un canal non bruité, l'objectif prioritaire est de minimiser la taille de la représentation de l'information : c'est le problème de la *compression de données*. Le code de Huffman (1952) est un code de longueur variable optimal, c'est-à-dire tel que la longueur moyenne d'un texte codé est minimale. On observe ainsi des réductions de taille de l'ordre de 20% à 90%. Ce code est largement utilisé, souvent combiné avec d'autres méthodes de compression.

Partie A : coder du texte
On donne, en Figure 1 ci-dessous, la table d'encodage hexadécimal des caractères ISO/CEI 8859-1, dite ASCII Latin 1.

ISO/CEI 8859-1																
	x0	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**	**x8**	**x9**	**xA**	**xB**	**xC**	**xD**	**xE**	**xF**
0x	*positions inutilisées*															
1x																
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8x	*positions inutilisées*															
9x																
Ax	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
Bx	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

FIGURE 1 – Table ISO/CEI 88-59-1

Chaque caractère est codé sur 8 bits, soit deux chiffres hexadécimaux, correspondant respectivement à la ligne et à la colonne à l'intersection desquelles il figure.

Par exemple :

- Pour la lettre `H` figurant à l'intersection de la ligne `4x` et de la colonne `x8`, le code hexadécimal est `48`.
- La chaîne de caractère `Hello_World_!` est codé par : `48 65 6C 6C 6F 5F 57 6F 72 6C 64 5F 21`.
- Le caractère `ESPACE` est symbolisé par `SP`.

Soit la chaîne de caractères `txt = "SIX ANANAS"`.

1. Calculer la taille en octets du texte contenu dans la variable `txt`. En déduire la taille en bits nécessaire pour le stocker.
2. Donner le codage de la chaîne de caractères `txt`.

Partie B : compression de Huffman

Nombre d'occurrences

On appelle *nombre d'occurrences* d'un symbole le nombre de répétitions de ce symbole dans le texte étudié. Ainsi, dans la phrase `"DEECDDEBFACCECCEDBAEE"` on peut associer le tableau d'occurrences ci-dessous :

Symbole	A	B	C	D	E	F
Nombre d'occurrences	2	2	5	4	7	1

3. Ecrire le tableau d'occurrences associé à la chaîne de caractères `txt`.
4. Préciser à quoi correspond la somme des nombres d'occurrences.

Ce tableau d'occurrences peut être stocké dans un dictionnaire Python où les clés sont les symboles rencontrés dans le texte et les valeurs les nombres d'occurrences de chaque symbole. Ainsi, pour l'exemple ci-dessus, le dictionnaire serait `{'D': 4, 'E': 7, 'C': 5, 'B'`

5. Recopier et compléter le code de la fonction `occurrence` ci-dessous qui, pour un texte passé en paramètre, renvoie le dictionnaire d'occurrences associé.

```
1 def occurrence(texte):
2     dico = {}
3     for lettre in ... :
4         if lettre in ... :
5             dico[lettre] = dico[lettre]+1
6         else:
7             ...
8     return ...
```

Arbre de Huffman

L'algorithme de Huffman met en oeuvre plusieurs structures de données. Il opère sur un ensemble dynamique d'arbres binaires pondérés (une *forêt*), structuré en file à priorité.

Initialement, la *forêt* est constituée d'arbres binaires,

- tous restreints à leur seule racine, dont l'étiquette est un symbole du texte ;
- et respectivement dotés d'un poids correspondant à l'effectif de ce symbole.

Une opération de greffe de deux arbres pondérés est possible : l'arbre résultant est un arbre binaire dont :

- la racine est un nœud sans étiquette ;
- les sous-arbres gauches et droits sont les deux arbres greffés ;
- le poids est la somme des poids de ces deux sous-arbres.

La file à priorité, qui contient tous les arbres considérés, est une structure permettant

- l'extraction : le premier arbre disponible est un arbre de priorité maximale parmi tous les arbres ;
- l'insertion : tout nouvel arbre pondéré est inséré
  - après tous ceux qui ont une priorité strictement plus grande que la sienne ;
  - avant tous ceux qui ont une priorité inférieure ou égale à la sienne.

Pour construire l'arbre de Huffman, tant que la *forêt* compte au moins deux arbres,

- les deux arbres prioritaires sont extraits de la file ;
- ils sont greffés en un nouvel arbre pondéré ;
- et celui-ci est inséré dans la file à priorité.

Une fois l'arbre construit, on pondère les arêtes en partant de la racine : 0 pour les arêtes menant aux enfants gauches, 1 pour les arêtes menant aux enfants droits.

Le schéma de la figure ci-dessous indique comment on construit un arbre de Huffman en fonction du tableau d'occurrences. Pour plus de clarté, les étiquettes de tous les nœuds ont été remplacées par le poids de l'arbre dont ils sont la racine.

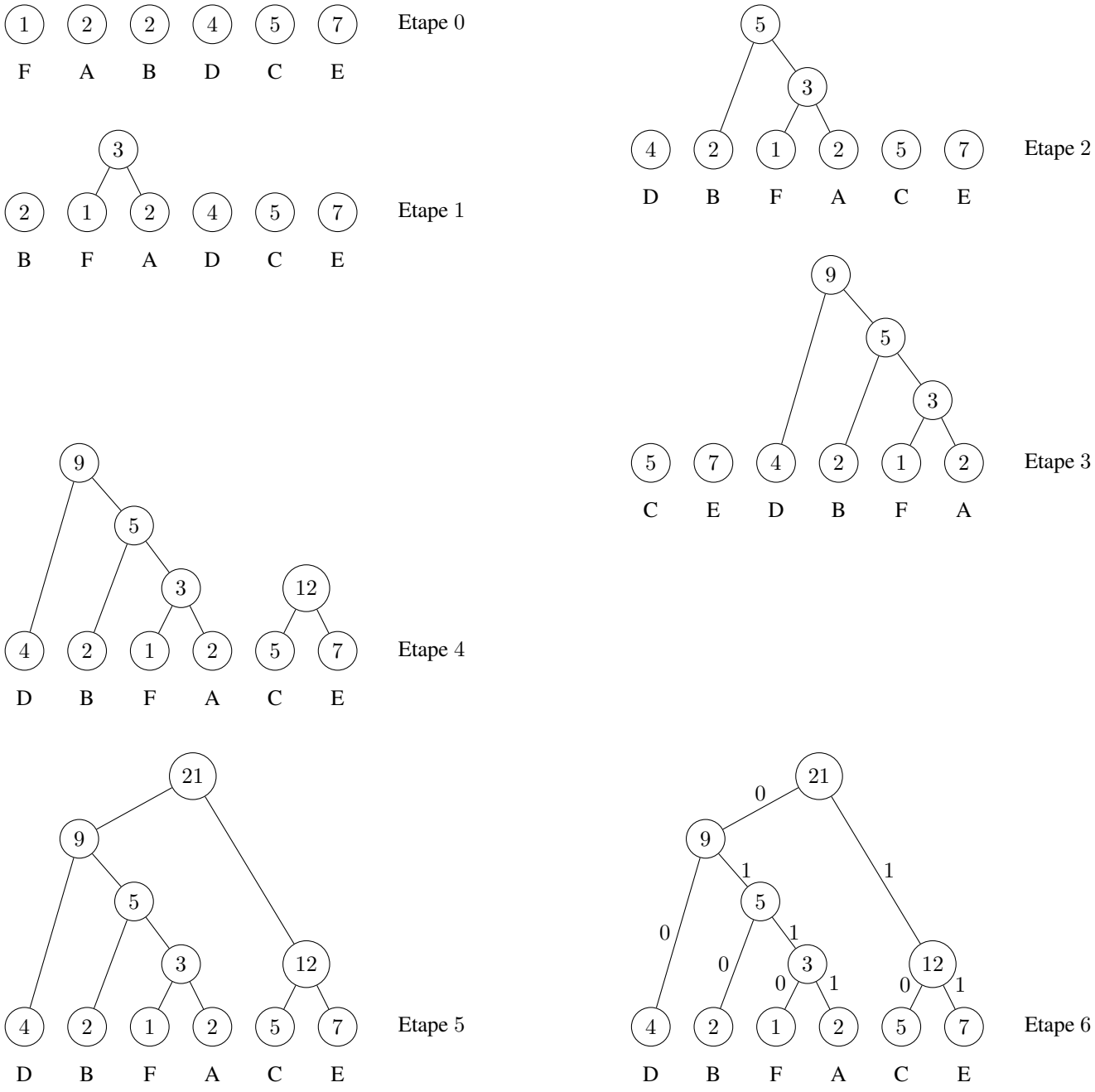

FIGURE 2 – Construction de l'arbre de Huffman

6. Construire l'arbre de Huffman associé à la chaîne de caractères `txt`.
7. Préciser à quoi correspond le poids de la racine de cet arbre.

Codage et compression à l'aide de l'arbre de Huffman

À l'aide de l'arbre de Huffman, on peut créer une table de codage où chaque symbole est codé par les bits lus sur le chemin entre la racine de l'arbre et la feuille correspondant au symbole. Dans l'exemple ci-dessus, la lettre `F` serait codée par 0110 et la lettre `E` par 11.

8. Indiquer le type de parcours à utiliser sur l'arbre de Huffman pour réaliser cette table de codage.
9. Donner la table de codage pour la chaîne de caractères `txt`.
10. Justifier le fait que le code de Huffman est un code de longueur variable.

Le codage du texte se fait ensuite caractère par caractère en utilisant la table de codage.

11. Coder la chaîne de caractères `txt` à l'aide du code de Huffman et de l'arbre construit à la question 6.
12. En reprenant le résultat déterminé dans la partie A, en déduire le taux de compression en $\%$ pour la variable `txt` et vérifier l'assertion du texte d'introduction : « *On observe ainsi des réductions de taille de l'ordre de* 20% *à* 90%. »

On rappelle que le taux de compression est le ratio $\dfrac{\text{encombrement initial} - \text{encombrement final}}{\text{encombrement initial}}$.

Exercice 3 (Dictionnaires, données en table, sécurisation des communications et programmation python - 8 points)
Lorsque l'énoncé demande la manipulation de la structure de données abstraites liste, on utilisera les `list` en Python avec la méthode `append`.

Un club de judo souhaite développer un système d'informations pour faciliter les traitements administratifs en cours d'année (inscriptions, communication, compétitions…).

Cet exercice comporte deux parties indépendantes.

Partie A
On pourra utiliser les mots du langage SQL suivants : `SELECT`, `FROM`, `WHERE`, `JOIN ON`, `INSERT INTO`, `VALUES`, `COUNT`, `DELETE`.

Le club de Judo souhaite proposer à ses adhérents une location de kimonos. Pour cela, il décide de mettre en place une base de données contenant les relations `adherent`, `kimono` et `location`.

Le schéma relationnel, où les clés primaires sont soulignées et les clés étrangères sont précédées du symbole `#`, est le suivant :

`adherent(numero-licence, taille-adherent, nom, prenom)`
`kimono(id-kimono, taille-kimono)`
`location(#numero-licence, #id-kimono, debut, fin)`

L'attribut `id-kimono` est un nombre entier. Les attributs `taille-adherent` et `taille-kimono` sont des nombres entiers dont l'unité est le centimètre et qui sont tous multiples de 10 (100, 110, 120, 130, etc.).

Les attributs `debut` et `fin` sont des dates au format chaîne de caractères `'AAAA-MM-JJ'`. Tous les attributs doivent être renseignés et valides mais l'attribut `fin` peut éventuellement être égal à la chaîne de caractères vide `''` pour les kimonos en cours de location.

On rappelle qu'en langage SQL la fonction d'agrégation `COUNT` permet de compter un nombre d'enregistrements. Par exemple, pour déterminer le nombre d'adhérents du club, on peut utiliser la requête suivante :

```
SELECT COUNT(numero-licence) FROM adherent
```

1. Ecrire une requête SQL permettant de connaître le numéro des kimonos **en cours de location**.
2. Ecrire une requête SQL permettant de connaître le nombre de kimonos de taille 130 cm possédés par le club.
3. Ecrire la requête SQL permettant de connaître le nom et le prénom de l'adhérent qui possède le kimono 42 uniquement si ce kimono est en cours de location. Cette requête ne doit renvoyer que le nom et le prénom de l'adhérent à qui il est actuellement loué et pas celui de tous ceux qui l'ont éventuellement loué par le passé.

Dans une requête SQL, il est possible de modifier un attribut entier avec une expression du type `a = a + 5`.

4. A la fin de l'année, pour anticiper les locations de l'année à venir, le club de judo modifie arbitrairement la taille de tous ses adhérents mesurant strictement moins de 160 cm en leur rajoutant 10 cm. Ecrire une requête SQL permettant de réaliser cette opération.
5. Le kimono numéro 25 a été déchiré lors d'un combat. Ecrire les requêtes SQL permettant de le supprimer de la base de données.

Partie B
Un adhérent du club de judo est décrit par les descripteurs :

- `nom` : nom de famille de l'adhérent, la donnée est une chaîne de caractères (on supposera dans l'exercice que tous les noms de famille font plus de cinq caractères) ;
- `prenom` : prénom de l'adhérent, la donnée est une chaîne de caractères ;
- `annee` : année de naissance de l'adhérent, la donnée est une chaîne composée de 4 caractères ;
- `mois` : mois de naissance de l'adhérent, la donnée est une chaîne de deux caractères parmi `'0123456789'` : `'01'` (pour janvier) et `'12'` (pour décembre) ;
- `jour` : jour de naissance de l'adhérent, la donnée est une chaîne de deux caractères parmi `'0123456789'` : `'01'` (si l'adhérent est né le premier jour du mois) et `'31'` (si l'adhérent est né le 31 du mois) ;
- `sexe` : sexe de l'adhérent, la donnée est un caractère valant soit `'F'` si l'adhérent est de sexe féminin, soit `'M'` s'il est de sexe masculin.

Pour différencier les pratiquants du judo en France, la fédération française de judo, attribue à chaque pratiquant un « numéro de licence ». Il s'agit d'une chaîne de 16 caractères dont :

- le premier caractère vaut soit `'F'` soit `'M'` selon le sexe de l'adhérent ;
- les huit caractères suivants correspondent à la date de naissance au format `'JJMMAAAA'` ;
- les cinq caractères suivants correspondent aux cinq premières lettres du nom de famille de l'adhérent en majuscules ;
- les deux derniers caractères correspondent à un nombre entre 01 et 99. Ils permettent à la fédération française de judo de différencier des pratiquants, dans le cas rare où ils ont le même sexe, la même date de naissance et les cinq premières lettres de leur nom identiques.

Exemple : Clémence et Stéphanie Dupond, soeurs jumelles nées le 03/07/1997, se voient attribuées les numéros de licences respectifs `'F03071997DUPON01'` et `'F03071997DUPON02'`.

6. Déterminer un numéro de licence possible pour Eddie Nirrer né le 12/10/2021.
7. Un adhérent a le numéro de licence `'M23091974MARTI01'`. Donner sa date de naissance et un nom de famille possible.

Pour manipuler efficacement sa base d'adhérents, le club de judo implémente une table par un tableau nommé `tab_adherents` contenant des dictionnaires en langage Python. Chaque dictionnaire correspond à un adhérent du club. Les clés des dictionnaires, communes à tous les dictionnaires, correspondent aux descripteurs utilisés pour cette table et sont `nom`, `prenom`, `annee`, `mois`, `jour`, `sexe`, `numero-licence`.

On donne en illustration les deux premiers éléments de ce tableau de dictionnaires :

```
tab_adherents= [
{'nom': 'DUPOND', 'prenom' : 'CLEMENCE', 'annee' : '1997',
'mois' : '07', 'jour' : '03', 'sexe' : 'F',
'numero-licence' : 'F03071997DUPON01'},
{'nom': 'DUPOND', 'prenom' : 'STEPHANIE', 'annee' : '1997',
'mois' : '07', 'jour' : '03', 'sexe' : 'F',
'numero-licence' : 'F03071997DUPON02'},
...]
```

8. Donner, sans justifier, la valeur à laquelle on accède avec l'instruction `tab_adherents[1]['prenom']`.
9. Ecrire l'instruction permettant d'obtenir la valeur `'F03071997DUPON01'`.

La direction du club souhaite connaître le nombre de ses adhérents nés une année donnée. Elle demande d'écrire une fonction `nombre_adhere` qui prend en entrée un tableau de dictionnaires `table` correspondant aux adhérents et une chaîne de 4 caractères `annee` correspondant à une année et qui renvoie en sortie l'entier correspondant au nombre d'adhérents nés cette année.

10. Recopier et compléter la fonction suivante pour permettre de répondre à cette problématique.

```
1 def nombre_adherents(table, annee):
2     compteur = ...
3     for adherent in table:
4         if ...:
5             ...
6     ...
```

La direction souhaite également connaître les adhérents les plus âgés du club. Elle nous demande d'écrire une fonction `adherent_plus_age` qui prend en entrée un tableau de dictionnaires `table` correspondant aux adhérents et qui renvoie en sortie une liste de dictionnaires correspondant aux adhérents les plus âgés du club. La comparaison se restreindra uniquement à l'année de naissance. Il peut donc y avoir un ou plusieurs adhérents qui sont nés la même année et qui sont donc les adhérents les plus âgés. On pourra supposer qu'aucun adhérent n'est né après l'année 2024 et qu'il y a toujours au moins un adhérent dans le club. On admet que l'on peut comparer en Python deux chaînes de caractères représentant deux années et que le résultat est le même que si ces années étaient des entiers. Par exemple, on a `tab_adherents[0]['annee'] < '2025'`, car la première adhérente est née en 1997.

11. Ecrire en Python cette fonction `adherent_plus_age`.

Pour tester la cohérence de sa base et des erreurs causées par un mauvais enregistrement de numéro de licence, le club souhaite programmer une fonction `verification_licence`. Cette fonction prend en paramètre un dictionnaire `adherent` correspondant à un adhérent du club et renvoie `True` si le numéro de licence de l'adhérent est conforme à son sexe, sa date de naissance et son nom de famille et `False` sinon. On pourra supposer que l'on dispose d'une fonction `extraire` qui prend en paramètre une chaîne de caractères `s` et deux indices `i` et `j` avec `0 <= i <= j <= len(s)` et qui renvoie la sous-chaîne de `s` entre les indices `i` inclus et `j` exclu. Par exemple :

```
>>> no = 'F03071997DUPON01'
>>> extraire(no, 0, len(no))
'F03071997DUPON01'
>>> extraire(no, 5, 9)
'1997'
```

12. Ecrire la fonction `verification_licence` en Python.