# CHAPITRE 6 : LES LISTES ET LES CHAINES DE CARACTÈRES

## I. Les listes.

### 1.1 Découpage en tranches d'une liste (slicing).

Le découpage d'une liste permet de récupérer une portion de la liste à partir de l'indice des éléments. Le découpage basique d'une liste consiste à indexer une liste avec deux entiers séparés par deux points. Cela renvoie une **nouvelle liste** contenant toutes les valeurs de l'ancienne liste entre les indices.

```
>>> carres = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> carres[2:6] #Attention le premier entier est inclus et le second exclu
[4, 9, 16, 25]
>>> carres[0:1]
[0]
>>> #si le premier entier est omis, on commence au début de  la liste
>>> carres[:7] #le second indice est exclu !
[0, 1, 4, 9, 16, 25, 36]
>>> carres[7:] #si le deuxième entier est omis, on termine à la fin  de  la liste
 [49, 64, 81]
>>> carres[::2] #le troisième entier correspond au pas
[0, 4, 16, 36, 64]
>>> carres[2:8:3] #le huitième élément est exclu !
[4, 25]
>>> carres[1:-1] #du deuxième à l'avant dernier élément
[1, 4, 9, 16, 25, 36, 49, 64]
>>> carres[::-1] #inversion de la liste !!
[81, 64, 49, 36, 25, 16, 9, 4, 1, 0]
```

### 1.2 Méthodes associées aux listes.

➢ **nom_list.append(x)** : ajoute **un seul élément x** à la fin de la liste.

```
>>> li=[1, 2, 3]
>>> li.append(4)
>>> li
[1, 2, 3, 4]
>>> id(li)
2989451425544
>>>
```

Ce qui est équivalent à

```
>>> li += [5]
>>> li
[1, 2, 3, 4, 5]
>>> id(li)
2989451425544
>>>
```

➢ **nom_list.extend(L)** : étend la liste en y ajoutant **tous les éléments de la liste L** fournie.

```
>>> li.extend([6, 7])
>>> li
```

```
[1, 2, 3, 4, 5, 6, 7]
>>> id(li)
2989451425544
>>>
```

```
>>> li = li + [8]
>>> li
[1, 2, 3, 4, 5, 6, 7, 8]
>>> id(li)
2989450544840
>>>
```

Les méthodes **append** et **extend** modifient la liste sur laquelle elles travaillent, alors que l'addition li = li + [x] crée un nouvel objet.

➢ **nom_list.insert(i, x)** : **insère un seul élément** x dans une liste avec un indice i déterminé.

```
>>> li.insert(-1, 10) #insère 10 avant le dernier élément de la liste
>>> li
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>>
```

***Remarques :***

- **nom_list.insert(0, x)** insère l'élément en tête de la liste.
- **nom_list.insert(len(a), x)** est équivalent à **nom_list.append(x)**.

**nom_list.remove(x) :** pour **supprimer un élément** x d'une liste **à partir de sa valeur**.

```
>>> li.remove(5) #une erreur s'affiche si l'élément n'existe pas dans la liste
>>> li
[1, 2, 3, 4, 6, 7, 8, 9, 10, 11]
>>>
```

**del nom_list[i] :** pour **supprimer** un élément d'une liste à partir de son **indice i**.

```
>>> del li[3] #supprime le 4ème élément de la liste
>>> li
[1, 2, 3, 6, 7, 8, 9, 10, 11]
>>> del li[4:] #supprime les éléments à partir du 5ème élément de la liste ou li[4:] = []
>>> li
[1, 2, 3, 6]
>>> del li[:] #supprime tous les éléments de la liste ou li[:] = []
>>> li
[]
>>>
```

**nom_list.pop(i)** : enlève de la liste l'élément situé à la position indiquée, et le retourne.

```
>>> li
[1, 2, 3, 6]
>>> li.pop(1)
>>> 2
>>> li
[1, 3, 6]
>>>
```

Si aucune position n'est indiquée, list.pop() enlève et retourne le dernier élément de la liste.

**nom_list.sort() : trie** une liste par ordre croissant par défaut

```
>>> li=[4,5,4,1,3,2,5,7,5]
>>> li2 = sorted(li) #la liste initiale n'est pas modifiée
>>> li
[4, 5, 4, 1, 3, 2, 5, 7, 5]
>>> li2
[1, 2, 3, 4, 4, 5, 5, 5, 7]
>>> li.sort() #la liste initiale est modifiée
>>> li
[1, 2, 3, 4, 4, 5, 5, 5, 7]
>>> li.sort(reverse = True) #trie la liste par ordre décroissant
>>> li
[7, 5, 5, 5, 4, 4, 3, 2, 1]
>>>
```

**nom_list.reverse() :** pour **inverser** une liste.

```
>>> li.reverse()
>>> li
[1, 2, 3, 4, 4, 5, 5, 5, 7]
>>>
```

**nom_list.index(x) :** donne l'**indice** d'un élément x.

```
>>> li.index(4)
3
>>>
```

*Remarques :*

- Si l'élément n'est pas trouvé, alors une erreur s'affiche.
- En cas de doublon, c'est le rang du premier élément qui est renvoyé.

**nom_list.count(x) : compte** le nombre d'éléments x dans une liste.

```
>>> li.count(5)
3
>>>li.count(9)
0
>>>
```

**choice(nom_list)** : **choisit** un élément aléatoire d'une liste non vide.

```
>>> from random import choice
>>> choice(li)
2
>>>
```

**shuffle(nom_list)** : **mélange** les éléments d'une liste. (Importer le module random)

```
>>> from random import choice
>>> shuffle(li)
>>> li
[7, 4, 5, 3, 4, 5, 2, 1, 5]
```

**sample(nom_list,k)** : génère un **échantillon** de taille k d'une liste.

```
>>> from random import sample
```

```
>>> sample(li,3)
 [2, 3, 7]
>>>
```

La méthode append() est particulièrement pratique car elle permet de construire une liste au fur et à mesure des itérations d'une boucle. Pour cela, il est commode de définir préalablement une liste vide de la forme li=[]. Voici un exemple où une chaîne de caractères est convertie en liste :

```
>>> seq = 'CAAAGGTAACGC'
>>> li = []
>>> li
[]
>>> for el in seq:
        li.append(el) #copie de la liste seq dans la liste li ; ou li += [el]

>>> li
['C', 'A', 'A', 'A', 'G', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
>>>
```

Remarquez que vous pouvez directement utiliser la fonction **list()** qui prend n'importe quel objet séquentiel (liste, chaîne de caractères, etc.) et qui renvoie une liste :

```
>>> seq = 'CAAAGGTAACGC'
>>> list(seq) #permet de convertir une chaîne de caractères en liste de caractères
['C', 'A', 'A', 'A', 'G', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
>>>
```

Cette méthode est certes plus simple, mais il arrive parfois que l'on doive utiliser les boucles tout de même, comme lorsqu'on lit dans un fichier.

### 1.3 Test d'appartenance.

L'inscription in permet de tester si un élément fait partie d'une liste.

```
>>> li = [1, 3, 5, 7, 9]
>>> 3 in li
True
>>> 4 in li
False
>>> 3 not in li
False
>>> 4 not in li
True
```

### 1.4 Copie de listes.

L'affectation d'une liste à partir d'une liste préexistante crée en réalité une copie de la liste initiale **par référence** et non une copie **par recopie** :

La fonction id() renvoie un entier, représentant l'identifiant interne de n'importe quel objet, quel que soit son type. Concrètement, il s'agit de l'adresse mémoire dans laquelle il est stocké

```
>>> x = [1, 2, 3]
```

```
>>> x, id(x)
([1, 2, 3], 2930103606792) #Votre id n'est pas forcément le même !
>>> y = x
>>> y,id(y)
([1, 2, 3], 2930103606792)
>>> x[1] = 4
>>> y
[1, 4, 3]
```

Vous voyez que la modification de x modifie y aussi. Rappelez-vous de ceci dans vos futurs programmes car cela pourrait avoir des effets désastreux ! Techniquement, Python utilise des pointeurs (comme dans le langage C) vers les mêmes objets et ne crée pas de copie à moins que vous n'en ayez fait la demande explicitement. Exemple :

```
>>> x = [1, 2, 3]
>>> x, id(x)
([1, 2, 3], 2930103086472)
>>> y = x[:]
>>> y, id(y)
([1, 2, 3], 2930103607240)
>>> x[1] = 4
>>> x, id(x)
([1, 4, 3], 2930103086472)
>>> y, id(y)
([1, 2, 3], 2930103607240)
```

Dans l'exemple précédent, x[:] a créé une copie **par recopie** de la liste x. Vous pouvez utiliser aussi la fonction list() qui renvoie explicitement une liste :

```
>>> x = [1, 2, 3]
>>> y = list(x)
>>> x[1]= -15
>>> y
[1, 2, 3]
```

Attention, les deux techniques précédentes ne fonctionnent que pour les listes à une dimension, autrement dit les listes qui ne contiennent pas elles-mêmes d'autres listes.

```
>>> x = [[1, 2],[3, 4]]
>>> x
[[1, 2], [3, 4]]
>>> y = x[:]
>>> y[1][1] = 55
>>> y
[[1, 2], [3, 55]]
>>> x
[[1, 2], [3, 55]]
>>> y = list(x)
>>> y[1][1] = 77
>>> y
[[1, 2], [3, 77]]
>>> x
[[1, 2], [3, 77]]
```

La méthode de copie qui **marche à tous les coups** consiste à appeler la fonction **deepcopy**() du module copy.

```
>>> import copy
>>> x = [[1,2],[3,4]]
>>> x
[[1, 2], [3, 4]]
>>> y = copy.deepcopy(x)
>>> x[1][1] = 99
>>> x
[[1, 2], [3, 99]]
>>> y
[[1, 2], [3, 4]]
```

### 1.5 Liste de compréhension.

Une des caractéristiques les plus puissantes de Python est la liste de compréhension qui fournit un moyen concis d'appliquer une fonction sur chaque élément d'une liste afin d'en produire une nouvelle.

Exemples :

- Liste des 15 premiers carrés parfaits.

```
>>> carres = [i**2 for i in range(15)]
>>> print(carres)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

La compréhension de listes évite donc d'écrire le code "classique" suivant :

```
carres = []
for i in range(11):
    carres.append(i*i)
```

- Liste des 10 premiers carrés pairs parfaits.

```
>>> li=[i**2 for i in range(20) if i%2 == 0]
>>> print(li)
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

- Simulation de 10 lancers d'un dé à 6 faces.

```
de = [random.randint(1,6) for i in range(10)]
>>> print(de)
[3, 1, 5, 6, 4, 2, 1, 1, 3, 1]
```

Liste des nombres impairs :

```
[n for n in range(1, 40, 2)]
[2 * n + 1 for n in range(20)]
[n for n in range(40) if n%2==1]
```

## II. Les chaînes de caractères.

### 2.1 Exemples.

```
>>> ch = 'Stephane KELLER'
>>> ch
'Stephane KELLER'
>>> len(ch)
```

```
15
>>> ch[3]
'p'
>>>
```

Nous pouvons utiliser les mêmes propriétés des listes concernant le découpage (slicing) :

```
>>> ch[0:4] #Le 0 est optionnel
'Step'
>>> ch[9:]
'KELLER'
>>> ch[:-5]
'Stephane K'
>>> ch[-1]
>>> R
>>>
```

A contrario des listes, les chaînes de caractères présentent toutefois une différence notable, ce sont des listes non modifiables. Une fois définie, vous ne pouvez plus modifier un de ses éléments. Le cas échéant, Python renvoie un message d'erreur.

```
>>> ch[9] = 'M'
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    nom[9]='M'
TypeError: 'str' object does not support item assignment
>>>
```

**Par conséquent, si vous voulez modifier une chaîne, vous êtes obligés d'en construire une nouvelle. Pour cela, n'oubliez pas que les opérateurs de concaténation (+) et de duplication (×) peuvent vous aider. Vous pouvez également générer une liste, qui elle est modifiable, puis revenir à une chaîne (fortement déconseillé ⇒ usine à gaz).**

### 2.2 Caractères spéciaux.

Il existe certains caractères spéciaux comme :

- Le retour à la ligne : \n (newline) ;
- La tabulation : \t (tabulation) ;
- Le caractère d'échappement : \ qui permet d'écrire un guillemet simple ou double (et que celui-ci ne soit pas confondus avec les guillemets de déclaration de la chaîne de caractères), vous pouvez utiliser \' ou \" ou utiliser respectivement des guillemets doubles ou simple pour déclarer votre chaîne de caractères.

```
>>> ch = 'Stephane\nKELLER\tEnseignant d\'informatique'
>>> ch
"Stephane\nKELLER\tEnseignant d'informatique"
>>> print(ch)
Stephane
KELLER        Enseignant d'informatique
>>>
```

### 2.3 Méthodes associées aux chaînes de caractères.

**lower()** et **upper()** : passe un texte en minuscule et en majuscule respectivement. On remarque que l'utilisation de ces fonctions n'altère pas la chaîne de départ mais renvoie la chaîne transformée.

```
>>> ch = 'Stephane KELLER'
>>> ch.lower()
'stephane keller'
>>> ch.upper()
'STEPHANE KELLER'
>>>
```

Pour mettre en minuscule la première lettre seulement, vous pouvez faire :

```
>>> ch[0].lower() + ch[1:]
'stephane KELLER'
>>>
```

**split()** : pour fendre une chaîne de caractères en champs, en utilisant comme séparateur les espaces ou les tabulations. Il est possible de modifier le séparateur de champs :

```
>>> ch = 'Stephane KELLER'
>>> ch.split()
['Stephane', 'KELLER']
>>> ch.split('e') #Attention à la casse
['St', 'phan', ' KELLER']
>>>
```

**find()** ou **index**() : pour avoir le rang d'un élément.

```
>>> ch = 'Stephane KELLER'
>>> ch.find ('K')
9
>>>
>>> ch.index ('L')
11
>>>
```

Si l'élément recherché est trouvé, alors l'indice du début de l'élément dans la chaîne de caractères est renvoyé. Si l'élément n'est pas trouvé, alors la valeur -1 est renvoyée.
En cas de doublon, c'est le rang du premier élément qui est renvoyé.

**replace()** : pour substituer des éléments d'une chaîne de caractère.

```
>>> ch.replace('e', 'o') #Attention à la casse
'Stophano KELLER'
>>>
```

**count() :** pour **compter** le nombre d'éléments (passé en argument) dans une chaîne de caractère.

```
>>> ch = 'Stephane KELLER'
>>> ch.count('e')
2
>>> ch.count('z')
0
>>>
```

**join()** : Jointure de liste.

```
>>> li=['C', 'A', 'A', 'A', 'G', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
>>> "".join(li) #permet de convertir une liste de caractères en chaîne de caractères
'CAAAGGTAACGC'
>>>
```

Programme équivalent :

```
>>> li=['C', 'A', 'A', 'A', 'G', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
>>> ch=''
>>> for el in li:
        ch+=el
>>> ch
'CAAAGGTAACGC'
>>>
```

## III. Liste non exhaustive des avantages et inconvénients des listes et des chaînes de caractères.

| Avantages | Inconvénients |
|---|---|
| **liste** | |
| ➢ Les éléments sont modifiables.<br>➢ On peut ajouter facilement un élément avec la méthode **append**().<br>➢ On peut avoir des éléments de plusieurs types.<br>➢ On peut créer une liste de listes, qui s'apparente à un tableau à 2 dimensions.<br>➢ Tri immédiat des éléments d'une liste avec la méthode **sort**().<br>➢ Inversion des éléments d'une liste avec la méthode **reverse**() ; permet de tester l'existence d'un palindrome. | ➢ Nécessité de l'utilisation d'une boucle For pour l'affichage propre du contenu de la liste.<br>➢ On ne modifie pas la taille d'une liste qui est aussi utilisée dans une boucle for avec un pointeur ou un indice. |
| **Chaîne de caractère** | |
| ➢ L'affichage propre du contenu avec la méthode **print**().<br>➢ Des méthodes comme **lower**(), **upper**() directement utilisables avec une chaîne de caractères.<br>➢ Possibilité d'utiliser certaines des opérations mathématiques avec des chaînes de caractères pour dupliquer une chaîne (×) ou concaténer (+) deux chaînes. | ➢ Un élément n'est pas modifiable.<br>➢ Nécessité d'utiliser la concaténation pour ajouter un élément.<br>➢ Nécessité d'utiliser les techniques de "slicing" pour travailler sur une partie de la chaîne de caractères.<br>➢ Pas de tri des éléments d'une chaîne de caractères. |

## LES LISTES ET LES CHAÎNES DE CARACTÈRES − TESTS

**I.** On utilisera dans les exercices suivants la chaîne de caractères ch = 'azerty0123456789' à titre d'exemple. Vous devrez envisager tous les programmes possibles.
**1°) a)** Créer la fonction **tronq**(ch) avec les paramètres suivants.
En entrée : la chaîne de caractères ch.
En sortie : la chaîne de caractère sans le premier caractère.
**b)** Créer la fonction **ante**(ch) avec les paramètres suivants.
En entrée : la chaîne de caractères ch.
En sortie : la chaîne de caractère jusqu'à l'antépénultième caractère inclus.
**c)** Créer la fonction **portion**(ch,i,j) avec les paramètres suivants.
En entrée : la chaîne de caractères ch et deux nombres entiers positifs tels que j > i.
En sortie : la chaîne de caractère du caractère d'indice i jusqu'au caractère d'indice j.
**2°) a)** Créer la fonction **inverse**(ch) avec les paramètres suivants.
En entrée : la chaîne de caractères ch.
En sortie : la chaîne de caractère à l'envers.
**b)** Afficher la chaîne ch et la chaîne inversée retournée sous la forme suivante :

```
a z e r t y 0 1 2 3 4 5 6 7 8 9
| | | | | | | | | | | | | | | |
9 8 7 6 5 4 3 2 1 0 y t r e z a
```

**3°)** Créer la fonction **subst**(ch, car, i) avec les paramètres suivants.
En entrée : une chaîne de caractères ch, un caractère car et un nombre entier positif i.
En sortie : la chaîne de caractère en substituant l'élément d'indice i par le caractère car.
**4°)** Créer la fonction **indice**(ch, car) avec les paramètres suivants.
En entrée : la chaîne de caractères ch et un caractère car.
En sortie : le caractère et son indice dans la chaine de caractères (-1 si le caractère ne se trouve pas dans la chaine de caractères).
**a)** Utiliser la fonction **find**() dans un premier temps.
**b)** Ne pas utiliser la fonction **find**() dans un second temps.
**5°)** Créer la fonction **occ**(ch) avec les paramètres suivants.
En entrée : une chaîne de caractères ch composée uniquement de chiffres. Par exemple ch = '12223454224650798613460573 1' ou li = [1, 2, 2, 2, 3, 4, 5, 4, 2, 2, 4, 6, 5, 0, 7, 9, 8, 6, 1, 3, 4, 6, 0, 5, 7, 3, 1].
En sortie : une liste li comptabilisant le nombre de caractère correspondant à un chiffre.
Par exemple, la liste li = [0, 1, 1, 0, 3, 0, 1, 2, 1,0] signifie qu'il y un 1, un 2, trois 4, 1 six, 2 sept et un 8.
**a)** Utiliser la fonction **count**() dans un premier temps.
**b)** Ne pas utiliser la fonction **count**() dans un second temps.

**II.** Reprendre tous les exercices précédents en remplaçant "chaîne de caractères" par "liste". On prendra la liste li = ['a', 'z', 'e', 'r', 't', 'y', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'] pour les tests.

**III.** Écrire une fonction similaire à la fonction **shuffle**(), puis à la fonction **choice**().

**IV.** Définir une fonction **trouve(ch,li)** avec les paramètres suivants :
En entrée : une chaîne de caractères ch et une liste de caractères li.
En sortie : un booléen qui confirme ou infirme qu'on puisse écrire le mot de la chaine de caractères avec la liste de caractères. Exemple : 'aimer' et ['a', 'v', 'm', 'i', 'r', 'r', 'e'].
**a)** Utiliser la fonction **count**() dans un premier temps.
**b)** Ne pas utiliser la fonction **count**() dans un second temps.

**V. 7°)** Écrire une fonction qui, à partir d'une liste de 0 et de 1 uniquement, indique si les 0 et les 1 sont alternés.

**VI.** Soit la liste de nombres [8, 3, 12, 5, 45, 25, 5, 52, 1]. Triez les nombres de cette liste par ordre croissant, sans utiliser la fonction **sort**(). Les fonctions **min**(), **append**() et **remove**() vous seront utiles.

**VII.** Générez aléatoirement une séquence nucléotidiques d'ADN de 20 bases en utilisant une liste avec la méthode append() et/ou une chaîne de caractères. La fonction **choice()** est interdite.

**VIII.** Transformez la séquence nucléotidique TCTGTTAACCATCCACTTCG en sa séquence complémentaire inverse. Afficher la séquence initiale, complémentaire et complémentaire inverse. Ne pas utiliser la fonction reverse().

**IX.** Soit la liste de nombres [5, 1, 1, 2, 5, 6, 3, 4, 4, 4, 2]. Enlevez les doublons de cette liste, triez-là par ordre croissant et affichez-là.

**X.** Générez aléatoirement une séquence nucléotidique d'ADN de 50 bases contenant 10% de A, 50% de G, 30% de T et 10% de C.

**XI.** Écrire un programme qui demande à l'utilisateur d'entrer des notes d'élèves comprises par exemple, entre 0 et 20. La boucle se termine lorsque l'utilisateur appuie sur Entrée sans note. Avec les notes ainsi entrées, construire progressivement la liste des notes.
Après chaque entrée d'une nouvelle note, afficher la liste de notes, le nombre de notes entrées, la note la plus élevée, la note la plus basse, la moyenne de toutes les notes et la note médiane. Créer une fonction pour chaque calcul. Vous ne pouvez pas utiliser les fonctions min et max et sort. La fonction **sort()** ne sera utilisée que dans la fonction de calcul de la médiane.

**XII.** Construire les listes suivantes en compréhension (une seule ligne d'instructions) :
- [1, 6, 11, 16, 21, 26, 31, 36, 41, 46, 51, 56, 61, 66, 71, 76, 81, 86, 91, 96].
- [58, 55, 52, 49, 46, 43, 40, 37, 34, 31, 28, 25, 22,19] (utiliser range avec un pas négatif).
- La liste des entiers pairs compris entre 0 et 100 (utiliser la commande modulo %).
- La liste des diviseurs de 20876 (utiliser la commande modulo %).
- La liste ['*2*', '*22*', '*222*', '*2222*', '*22222*', '*222222*'] (utiliser range et concaténation de chaînes de caractères : 'a + b' donne 'ab' et 'a' * 5 donne 'aaaaa').
- La liste des couples d'entiers entre 0 et 10 dont la somme est multiple de 5 (double boucle for et test if)

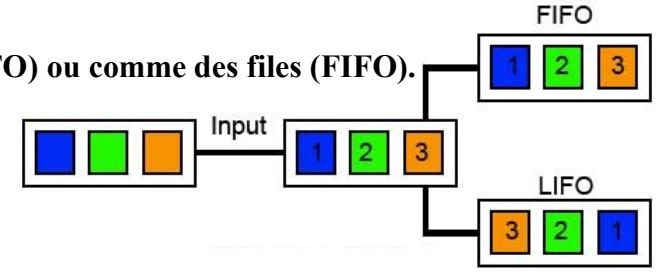
**XIII.** Calcul de sommes et de produits.
1°) Calculer les sommes suivantes.

| | | |
|---|---|---|
| a) $1+2+3+...+100$ | b) $3+6+9+...+201$ | c) $-1-4-7-...-31$ |

2°) Calculer les produits suivants.

| | |
|---|---|
| a) $2\times4\times6\times8\times....\times40$ | b) $24\times12\times6\times3\times....\times0{,}1875$ |

## XIV. Utiliser les listes comme des piles (LIFO) ou comme des files (FIFO).




Les méthodes des listes rendent très facile leur utilisation comme des piles, où le dernier élément ajouté est le premier récupéré ("dernier entré, premier sorti", ou LIFO pour "last-in, first-out").
Il est également possible d'utiliser une liste comme une file, où le premier élément ajouté est le premier récupéré ("premier entré, premier sorti", ou FIFO pour "first-in, first-out").
Simuler, à l'aide de l'exemple précédent, l'utilisation d'une LIFO puis d'une FIFO.

**XV.** Le numéro INSEE indique successivement et exclusivement :

- Le sexe : 1 chiffre, codification : 1 = masculin, 2 = féminin.
- L'année de naissance : 2 chiffres, codification : 00 à 99.
- Le mois de naissance : 2 chiffres, codification : de 01 (janvier) à 12 (décembre).
- Le lieu de naissance : 5 chiffres ou caractères, Le lieu de naissance est chiffré en référence au code officiel géographique (COG) tenu par l'INSEE.).
- Le numéro d'ordre : 3 chiffres, codification de 001 à 999. Il permet de distinguer les personnes nées au même lieu à la même période.
- La clé de contrôle : 2 chiffres. Codification de 01 à 97. La clé de contrôle est le complément à 97 du reste de la division par 97 du nombre de 13 chiffres.

On prendra le numéro INSEE suivant pour les tests : insee1=151024610204325.

**1°)** Concevoir la fonction **nomme(num)** avec les contraintes suivantes :

- En entrée : le numéro INSEE d'une personne.
- En sortie : retourne "*Homme*/*Femme* né(e) le *mois année* dans le *département*".

2°) Concevoir la fonction **cle(num)** avec les contraintes suivantes :

- En entrée : le numéro INSEE d'une personne.
- En sortie : retourne True ou False si la clé de sécurité est valide ou non.

3°) Concevoir la fonction **repart(nums)** avec les contraintes suivantes :

- En entrée : une liste de numéro INSEE.
- En sortie : retourne le nombre de femmes et l'année de naissance de la personne la plus âgée. On prendra la liste linum=[189112610826891, 255081416802538, 151024610204325, 269058606611836, 282127511412312, 189112610826891, 169058606611886]