Correction du TP 5

Exercice 1 — Plus petit élément d'une liste
La fonction suivante prend en argument une liste non vide de flottants et renvoie son minimum.

```
def minimum(L):
    """
    Entrée : une liste L non vide de nombres flottants.
    Sortie : le plus petit élément de la liste L.
    """
    mini = L[0]
    # Le minimum courant est initialisé au premier élément.
    for element in L:
    # La variable element parcourt les éléments de la liste L.
        if element < mini:
            mini = element
            # L'élément rencontré devient le minimum courant :
            # c'est la plus petite valeur qu'on ait rencontrée.
    return mini
```

Exercice 2 — Recherche d'un élément dans une liste
La fonction suivante prend en arguments une liste L et un élément x, et renvoie `True` si x est un élément de la liste L, `False` sinon.

Pour cela, on parcourt les éléments de la liste : dès que l'on rencontre l'élément x, on renvoie `True`, ce qui arrête immédiatement la fonction.
Si on arrive à la fin du parcours sans avoir trouvé x, c'est que l'élément x n'apparait pas dans la liste L et on renvoie donc `False`.

```
def recherche(L, x):
    """
    Entrées : une liste L et un élément x.
    Sortie : True si l'élément x appartient à la liste L,
             False sinon.
    """
    for element in L:
    # La variable element parcourt les éléments de la liste L.
        if element == x:
            return True
            # Si l'élément rencontré est l'élément recherché,
            # on renvoie True, ce qui arrête la fonction.
    return False
    # Si la fonction ne s'est pas arrêtée avant,
    # c'est que l'élément x n'apparait pas dans la liste.
```

Exercice 3 — Recherche des occurrences d'un caractère
La fonction suivante prend en arguments une chaine de caractères et un caractère, et renvoie la liste des indices de toutes les occurrences du caractère dans la chaine.

Pour cela, on commence par créer une liste `Occ`, initialement vide, qui contiendra les indices de toutes les occurrences du caractère que l'on a rencontrées dans la chaine.

On parcourt ensuite les indices de la chaine de caractères (ici, on a besoin de la position de chaque caractère, donc on parcourt la chaine sur ses indices plutôt que sur ses caractères directement). Si l'on rencontre le caractère recherché, on ajoute à la liste `Occ` l'indice auquel on l'a rencontré dans la chaine.

À la fin du parcours, on renvoie la liste `Occ`, qui contient les indices de toutes les occurrences du caractère dans la chaine.

```
def occurrences(chaine, lettre):
    """
    Entrées : une chaine de caractères chaine,
                un caractère lettre.
    Sortie : la liste des indices des occurrences
             du caractère lettre dans la chaine chaine.
    """
    Occ = []
    for k in range(len(chaine)):
    # L'entier k va de 0 à len(chaine) - 1,
    # c'est-à-dire que k parcourt les indices de la chaine.
        if chaine[k] == lettre:
        # Si le caractère d'indice k dans la chaine vaut lettre :
            Occ.append(k) # on ajoute l'indice k à la liste Occ.
    return Occ
```

Exercice 4 — Insertion

Pour insérer un élément x à l'indice i dans une liste (en modifiant la liste, sans en créer une nouvelle), on peut procéder de deux manières différentes.

- **Par échanges successifs**

  Une première méthode consiste à ajouter l'élément x à la fin de la liste, puis à l'échanger successivement avec l'élément qui le précède jusqu'à ce qu'il se retrouve à la position i.

  Pour cela, commençons par écrire une fonction (celle du cours) qui prend en arguments une liste et deux entiers i et j, et qui échange dans la liste les éléments d'indices i et j.

```
def echange(L, i, j):
    """
    Entrées : une liste L,
              deux indices i et j de la liste L.
    Échange les éléments d'indices i et j dans la liste L.
    Sortie : aucune, la liste L est directement modifiée.
    """
    aux = L[i]
    L[i] = L[j]
    L[j] = aux
```

Écrivons alors la fonction d'insertion.

```
def insertion(L, x, i):
    """
    Entrées : une liste L,
              un objet x
              un indice i.
    Insère l'élément x dans la liste L à l'indice i.
    Sortie : aucune, la liste L est directement modifiée.
    """
    n = len(L)
    L.append(x) # Après cet ajout, L est de longueur n + 1.
    # Initialement, l'élément x est à l'indice n.
    for k in range(n, i, -1): # k va de n à i + 1, par pas de -1.
        echange(L, k, k - 1)
        # On échange l'élément d'indice k avec le précédent.
        # Après cet échange, l'élément x est à l'indice k - 1.
    # Le dernier échange se fait donc entre les éléments
    # d'indices i + 1 et i dans L :
    # à la fin de la boucle,
    # l'élément x se retrouve donc à l'indice i.
```

- **Par décalages successifs**

  Une méthode plus efficace consiste à ajouter un élément quelconque à la fin de la liste, afin de créer une place en plus en fin de liste, puis à décaler d'un cran vers la droite tous les éléments de la liste dont l'indice est supérieur ou égal à i (en partant de la fin), pour ensuite placer l'élément x à l'indice i.

  Plus précisément, nommons n la longueur initiale de la liste et a_0, a_1, ..., a_{n-1} ses éléments. L'élément que l'on a ajouté en fin de liste est donc à l'indice n.

  - ⋆ On commence par décaler l'avant-dernier élément vers la droite : l'élément $L[n]$ prend la valeur a_{n-1}. L'élément a_{n-1} prend donc la place de l'élément que l'on avait ajouté et se retrouve en toute dernière position.
  - ⋆ On décale ensuite a_{n-2} vers la droite : l'élément $L[n-1]$ prend la valeur a_{n-2}, donc l'élément a_{n-2} se retrouve à l'avant-dernière position.
  - ⋆ Et ainsi de suite.
  - ⋆ On décale enfin l'élément a_i vers la droite : l'élément $L[i+1]$ prend la valeur a_i, c'est-à-dire que l'élément a_i se retrouve à l'indice $i+1$.

  On peut alors remplacer l'élément $L[i]$ par l'élément x sans perdre la valeur a_i ni les suivantes, puisqu'elles ont toutes été décalées vers la droite (on a «libéré» la place i pour l'élément x).

```
def insertion_efficace(L, x, i):
    """
    Entrées : une liste L, un élément x et un indice i.
    Insère l'élément x dans la liste L à l'indice i.
    Sortie : aucune, la liste L est directement modifiée.
    """
```

```
    n = len(L)
    L.append("place en plus")
    # L'élément "place en plus" est d'indice n.
    for k in range(n, i, -1): # k va de n à i + 1, pas par de -1.
        L[k] = L[k - 1]
        # L'élément L[k] prend la valeur de l'élément qui le précède.
    L[i] = x # On place x à la position i.
```

Exercice 5 — Liste de mots

La fonction suivante prend en entrée une liste.
Elle crée une chaine de caractères `chaine`, initialisée à la chaine vide, qui contiendra la concaténation de tous les éléments de la liste rencontrés.

Puis elle parcourt les éléments de la liste :

- si on rencontre un élément qui n'est pas une chaine de caractères, on renvoie un message d'erreur, ce qui arrête la fonction ;
- si l'élément rencontré est une chaine de caractères, on le concatène à la chaine `chaine`.

Si la fonction ne s'est pas arrêtée avant la fin de la boucle, c'est que la liste était bien une liste de chaines de caractères : on renvoie alors la chaine `chaine`, qui est bien la concaténation de tous les éléments de la liste.

```
def concatenation(L):
    """
    Entrée : une liste L.
    Sortie : un message d'erreur si la liste L
             n'est pas une liste de chaine de caractères,
             la concaténation de tous ses éléments sinon.
    """
    if type(L) != list:
        return "L'argument doit être une liste !"
    chaine = "" # Au début, la chaine-résultat est vide.
    for element in L:
    # La variable element parcourt les éléments de la liste L.
        if type(element) != str:
            return "La liste ne contient pas que des chaines !"
        else:
            chaine = chaine + element
            # On concatène l'élément à la chaine-résultat.
    return chaine
```

Exercice 6 — Répétitions

1. Pour éliminer les doublons d'une liste, on propose une méthode qui modifie la liste *en place* ; on utilisera alors la commande `pop` pour supprimer un élément qui n'est pas nécessairement le dernier élément de la liste. Si l'on veut éviter cette utilisation de `pop` — le programme officiel se limite à `pop` en dernière position — on peut créer une nouvelle liste qui contiendra les éléments de la liste initiale sans les doublons.

Pour chaque élément de la liste, on supprime tous les éléments qui lui sont égaux et qui viennent après lui dans la liste. De la sorte, on aura éliminé tous les doublons de la liste !

Dès qu'un élément est supprimé, la liste change de longueur, ce qui impose l'utilisation d'une boucle `while` plutôt qu'une boucle `for` (dont le nombre de tours serait fixé à l'avance) pour parcourir les indices des éléments de la liste.

```
def sans_doublons(L):
    """
    Entrée : une liste L.
    Supprime les doublons de la liste.
    Sortie : aucune, la liste L est directement modifiée.
    """
    i = 0 # Au début, i est l'indice du premier élément.
    while i < len(L):
        j = i + 1
        # On inspecte les éléments situés
        # strictement après l'indice i
        # et on les compare à L[i].
        while j < len(L):
            if L[j] == L[i]:
                L.pop(j)
                # On supprime l'élément d'indice j de la liste,
                # c'est-à-dire qu'on supprime le doublon.
                # Dans ce cas, l'élément suivant à inspecter
                # est encore d'indice j.
            else:
                j = j + 1
                # On inspecte l'élément suivant.
        i = i + 1
        # On passe à l'indice du prochain élément
        # dont on doit supprimer les doublons.
```

2. La fonction suivante prend en argument une liste L.
   - Elle commence par créer les variables suivantes :
     - ⋆ `element_frequent`, initialisée au premier élément, qui contiendra l'élément le plus fréquent rencontré ;
     - ⋆ `max_occurrences`, initialisée à 1 (on n'a rencontré le premier élément qu'une fois pour l'instant), qui contiendra le nombre d'occurrences de l'élément le plus fréquent.
   - On parcourt ensuite les indices de la liste L à l'aide d'une boucle `while`, en commençant à l'indice 0 (qui est l'indice du premier élément).

     Pour chaque indice i, le tour i de la boucle se déroule ainsi.
     - ⋆ On initialise une variable `occurrences` à 1. Cette variable va compter le nombre d'occurrences de l'élément $L[i]$ dans la liste.
     - ⋆ Comme dans le programme précédent, on parcourt les éléments de la liste situés après l'indice i.

Si l'on rencontre une occurrence de l'élément $L[i]$, alors on incrémente le compteur `occurrences`, puis on supprime cette occurrence, afin de ne pas rencontrer à nouveau la valeur $L[i]$ plus tard dans le parcours de la liste.

- ⋆ Une fois ce parcours de fin de liste terminé, on compare le nombre d'occurrences de l'élément $L[i]$ au nombre maximal d'occurrences. Si l'élément $L[i]$ a strictement plus d'occurrences que le maximum courant, alors il prend la place de l'élément le plus fréquent et son nombre d'occurrences prend la place du maximum.

- À la fin du parcours, on renvoie l'élément le plus fréquent et son nombre d'occurrences.

```
def le_plus_frequent(L):
    """
    Entrée : une liste L.
    Sortie : le couple constitué du premier élément de L
             dont le nombre d'occurrences est le plus grand
             et de son nombre d'occurrences dans la liste L.
    """
    element_frequent = L[0] # L'élément le plus fréquent.
    max_occurrences = 1 # Son nombre d'occurrences.
    # On commence le parcours de la liste.
    i = 0
    while i < len(L):
        occurrences = 1 # Nombre d'occurrences de l'élément L[i].
        j = i + 1
        while j < len(L):
            if L[j] == L[i]:
                occurrences += 1
                L.pop(j) # On supprime cette occurrence de L[i].
            else:
                j = j + 1
        if occurrences > max_occurrences:
            element_frequent = L[i]
            max_occurrences = occurrences
        i = i + 1
    return (element_frequent, max_occurrences)
```

Exercice 7 — Maintenant et pour toujours

Oui, il est possible d'écrire une boucle `for` qui tourne indéfiniment.

Par exemple, la boucle `for` suivante parcourt les éléments d'une liste à laquelle elle rajoute un élément (dont la valeur n'a pas d'importance ici) à chaque tour de boucle. Ainsi, la boucle n'arrivera jamais au bout de la liste et le programme tournera indéfiniment !

```
L = [0]
for element in L:
    L.append(element + 1)
```

Pour mieux voir que la boucle est infinie, on peut afficher l'élément `element` à chaque tour de boucle.