Informatique — MPSI/PCSI, 2023–2024

Correction du TP n° 4

Exercice 4.1 (Mutable *or not* ?)

1. Comme l'opération `M=L` recopie sur `M` l'adresse du pointeur de `L`, alors modifier `M[2]` ou `L[2]` revient au même et donc `L[2]` renvoie `666`.
2. L'opération `[e for e in L]` permet de recopier les éléments de `L` sans les lier. Ainsi, si `M` est modifiée, les éléments de `L` ne le sont pas et donc `L[2]` renvoie bien `2`.
3. La recopie des éléments par compréhension ne passe pas au-dessus de la notion de pointeur. Ainsi, lorsque les éléments d'une liste sont des éléments mutables, alors ce sont leurs pointeurs qui sont recopiés et non leurs valeurs. On observe ici que la deuxième liste de `L` et la liste `B` ont maintenant `666` en dernier élément.
4. Lorsque la recopie est faite avec la fonction `deepcopy` de la bibliothèque `copy`, tous les éléments mutables sont eux-mêmes recopiés et aucun pointeur n'est lié. Ainsi, modifier `M` n'a aucun effet sur `L` ou `B` (ni `A`, ni `C`).
5. Comme c'est trois fois le pointeur de `L` qui est saisi pour constituer la liste `M`, modifier `L[2]` induit la modification de chaque sous-liste de `M`. Donc `[e[2] for e in M]` renvoie `[666, 666, 666]`.
   Comme pour toute séquence (liste, tuple, chaine de caractères), l'opération `*n` est une opération de réplication n fois. Or répliquer un élément mutable ne recopie que son pointeur. Ainsi, pour toute liste `L`, on a

   ```
   >>> [L,L,L]==3*[L]
   True
   ```

   Attention, `3*L` concatène 3 fois la liste `L` et ne recopie donc aucun pointeur au rang 1.
6. Comme par réplication, `[[]]*n` fait n fois référence au même pointeur et donc modifier une des n séquences les modifie toutes. Ainsi, `L` renvoie `[[2], [2], [2]]`.

Exercice 4.2 (Mutation)

1. Pour nettoyer la liste `L` de tous ses `a`, il est possible d'utiliser la fonction `pop` qui permet de supprimer un élément. Par contre on veillera à ce l'on puisse continuer de parcourir la liste (initialement de longueur n) même si on a déjà supprimé $k > 0$ valeurs. En partant de la fin, on peut écrire :

   ```
   def Nettoyer(L,a):
       n=len(L)
       for i in range(n):
           if L[n-1-i]==a:
               L.pop(n-1-i)
   ```

2. Pour permuter deux éléments d'une séquence, il suffit de sauvegarder de façon temporaire un des deux objets pour réaliser la permutation. Tenant compte du fait que l'on agit sur la liste elle-même (pointeur), il n'est pas nécessaire de renvoyer quoi que ce soit.

   ```
   def Permuter(L,i,j):
       tmp=L[i]
       L[i]=L[j]
       L[j]=tmp
   ```

   La version suivante utilise le *swap* natif des objets en Python.

   ```
   def Permuter(L,i,j):
       L[i],L[j]=L[j],L[i]
   ```

3. Pour inverser une liste de longueur n, il suffit d'utiliser un éléments de plus puis, en partant de l'avant dernier, copier un à un les termes à la fin puis les supprimer selon le principe décrit par les schémas ci-dessous.

C'est ce que fait le code suivant.

```
def Inverser(L):
    n=len(L)
    for k in range(n-1):
        L.append(L[n-2-k])
        L.pop(n-2-k)
```

où `L.pop(n-2-k)` est (quasiment) équivalent à `del(L[n-2-k])`. Notez, que nous aurions aussi pu utiliser le *swap* de la fonction `Permuter` :

```
def Inverser(L):
    n=len(L)
    for k in range(n//2):
        Permuter(L,k,n-1-k)
```

ou étendre la liste avec son inverse obtenue par *reverse slicing* puis supprimer la portion initiale (en ne gardant en place que le dernier élément) :

```
def Inverser(L):
    n=len(L)
    L.extend(L[n-1::-1])
    del(L[:n-1])
```

où `L.extend(L[n-1::-1])` est (quasiment) équivalent à `L=L+L[n-1::-1]`. Notons toutefois que cette solution coute temporairement le double en RAM pour l'objet `L`, ce qui parfois peut être un problème.

4. Pour vérifier qu'une séquence est égale à son inverse, il suffit de comparer successivement tous les éléments jusqu'au milieu. Dans le cas d'une séquence de nombre impairs d'éléments n, on fera ainsi $\left\lfloor \frac{n}{2} \right\rfloor$ comparaisons, $n/2$ sinon. Le code suivant convient.

```
def palindrome(L):
    n=len(L)
    k=0
    while k < n//2 and L[k]==L[n-1-k]:
        k+=1
    return(k==n//2)
```

Exercice 4.3 (Recherche par dichotomie dans une liste triée)

1. Pour savoir si une séquence $L = \{x_0, x_1, \ldots, x_{n-1}\}$ est triée, il suffit de comparer successivement tous les éléments. En effet, notant $n = \text{Card}(L)$, on doit avoir :

$$\forall k \in [\![0, n-2]\!],\ x_k \leqslant x_{k+1}$$

C'est l'invariant de boucle. Ainsi, partant de $k = 0$, tant que ($k \leqslant n-2$ et $a_k \leqslant a_{k+1}$) est vrai, alors on incrémente k. On a ainsi deux raisons de sortir de la boucle :

— soit $a_k > a_{k+1}$ auquel cas la liste n'est pas triée ;
— soit $k > n-2 \iff k = n-1$ si la liste est triée.

Il vient l'implémentation suivante :

```
def Triee(L):
    n=len(L)
    k=0
```

```
    while k<=n-2 and L[k+1]>=L[k]:
        k=k+1
    return(k==n-1)
```

2. L'algorithme le plus rapide effectue une recherche par dichotomie. Le principe est le suivant : on veut tester l'appartenance d'un entier x à un tableau d'entiers trié dans l'ordre croissant.
   - Si l'élément du milieu vaut x, on peut conclure. S'il est strictement supérieur à x, alors x ne peut pas appartenir à la moitié supérieure : on continue la recherche dans la première moitié (et la seconde dans le cas symétrique).
   - Si l'élément du milieu de la moitié de tableau qui reste vaut x, on peut conclure ; sinon, on continue la recherche dans la seule moitié de moitié possible.
   - Jusqu'au moment où on se retrouve avec une zone avec au plus un élément, et on peut alors conclure !

   Bref, on casse en deux [1] à chaque étape. En regardant l'élément au milieu de la zone, on peut conclure quant à l'appartenance, ou bien diviser par deux la taille de la zone. Pour écrire l'algorithme, on commence par préciser comment représenter l'intervalle de recherche : ce sera $[a, b]$ (a pour début et b pour fin, avec les éléments d'indices a et b inclus). Ensuite, on fait bien attention :
   - à l'initialisation de ces valeurs ;
   - au nombre d'éléments de cet intervalle (c'est $b-a+1$ et non $b-a$) ;
   - au cas de terminaison de la boucle : on choisit de s'arrêter lorsqu'il y a *au plus* un élément.
   - à l'élément « du milieu » : si a et b ont même parité (il y a alors un nombre impair d'éléments dans la zone) c'est $\dfrac{a+b}{2}$; mais sinon, l'un des deux morceaux mis de côté aura un élément de plus que l'autre. En prenant $m = \dfrac{a+b-1}{2}$, les zones $[a, m-1]$ et $[a+1, b]$ auront respectivement k et $k+1$ habitants, avec $k = \dfrac{b-a-1}{2}$·. Dans les deux cas, on prend donc $m = \left\lfloor \dfrac{a+b}{2} \right\rfloor$ et comme les indices a et b sont positifs, c'est aussi la valeur renvoyée en Python par `(a+b)//2`.

   L'algorithme est donc le suivant :

   Entrées : L, x
   $a, b \leftarrow 0, |L|-1$
   tant que $b > a$ **faire**
   | $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$
   | **si** $L[m] = x$ **alors**
   | └ **Résultat :** `True`
   | **si** $L[m] < x$ **alors**
   | | $a \leftarrow m+1$ # continuer à droite
   | **sinon**
   | └ $b \leftarrow m-1$ # continuer à gauche
   # arrivé ici, $b = a$ (il reste un élément) ou $b = a-1$ (il n'y a plus rien)
   si $a = b$ *et* $L[a] = x$ **alors**
   └ **Résultat :** `True`
   Résultat : `False`

   L'adaptation de cet algorithme pour renvoyer l'indice conduit au code suivant :

```
def IndiceObjetListeTriee(L,x):
    n=len(L)
    a=0
    b=n-1
    if L[a]>x or x>L[b]:
        return(False)
    else:
        while b>=a:
            m=(a+b)//2
            if L[m]==x:
                return(m)
            elif L[m]<x:
                a=m+1
```

1. En grec : dikhotomia = « division en deux parties ».

```
            else:
                b=m-1
        return(False)
```