Tableaux

18 septembre 2015

1 Tableaux et type `list` en `Python`

1.1 Deux structures de données en informatique.

Les données manipulées en informatique sont organisées, que ce soit dans la mémoire où dans la manière d'y accéder, de les manipuler. Une telle organisation porte le nom de *structure de données*, en voici deux grandes.

Tableau Les données sont stockées dans des cases contigües de la mémoire de l'ordinateur, chaque emplacement étant souvent indicé par un entier. A priori, il n'est pas possible d'en rajouter autant que possible : cette place a été préallouée. Pour accéder au contenu de l'emplacement numéro k, la machine a seulement besoin de connaître l'adresse de la première case mémoire et de la largeur de chaque case (faire un dessin). L'accès en lecture et en écriture à une donnée (à partir du numéro de son emplacement) se fait donc en temps constant (on dit $O(1)$). L'ajout d'une nouvelle donnée à un tableau peut alors être problématique !

Liste (chaînée) Les données ne sont pas stockées de manière organisée dans la mémoire, mais de chaque emplacement on pointe l'emplacement suivant. Ainsi, l'accès (en lecture ou en écriture) ne se fait plus en temps constant, mais l'ajout d'une nouvelle donnée se fait simplement en temps constant.

1.2 Réalisation en `Python`

Les objets de type `list` en `Python` sont des tableaux : c'est une dénomination fâcheuse car, partout ailleurs en informatique, le terme *liste* désigne en fait une liste chaînée. Prenez donc l'habitude de dire "tableau", ou "liste `Python`" et non "liste" quand vous parlez de cette structure.
Cependant, il y a une raison à cette dénomination : ces tableaux en `Python` possèdent presque la propriété des listes, dans le sens où l'ajout d'un nouvel élément se fait en temps constant *amorti*. Cela signifie que la plupart de ces ajouts se font en temps constants car en fait `Python` a "gardé des cellules en réserve", mais parfois l'ajout d'un

élément force la création d'un nouveau tableau avec plus de cellules de réserve, et la copie de l'ancien tableau dans le nouveau. Cette opération est coûteuse, mais elle est rare. En moyenne, chaque ajout à un coût constant.
On parle alors de *tableaux dynamiques*.

Pour créer un tableau on peut le donner en extension.

```
>>> t = [23, 41, 101]
>>> t
[23, 41, 101]
```

On peut aussi le donner en compréhension.

```
>>> u = [k**2 for k in range(10)]
>>> u
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

On peut alors accéder à ses éléments via leur indice.

```
>>> t[0]
23
>>> u[1]
1
>>> u[9]
81
```

Attention à utiliser un indice existant !

```
>>> t[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Les indices appartiennent à $[\![0, \texttt{len(t)}[\![$. Mais on peut aussi compter les éléments à partir de la fin, en utilisant des indices négatifs !

```
>>> t[-1] # dernier élément
101
>>> t[-2] # avant-dernier
41
>>> t[-3]
23
```

Attention, là aussi, à ne pas sortir du tableau !

```
>>> t[-4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Enfin, il y a une liste `Python` particulière : la liste vide `[]`

1.3 Opérations usuelles

1.3.1 Concaténation

Concaténer deux tableaux revient à les mettre bout à bout, on l'effectue en Python avec l'opérateur +.

```
>>> t+u
[23, 41, 101, 0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> u+t
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 23, 41, 101]
```

Dans cette opération, les deux tableaux sont intégralement recopiés.

1.3.2 Quelques calculs

La longueur se calcule avec la fonction `len`. Pour des listes Python de nombres, le maximum se calcule avec la fonction `max`, le minimum avec `min` et la somme des éléments de la liste avec `sum`.

```
>>> len(t)
3
>>> max(t)
101
>>> min(t)
23
>>> sum(t)
165
```

On peut aussi tester l'appartenance d'un élément dans une liste Python avec `in`.

```
>>> 42 in t
False
>>> [] in [[]]
True
```

Enfin, on peut recopier plusieurs fois un *même* tableau avec l'opérateur *.

```
>>> 4*t
[23, 41, 101, 23, 41, 101, 23, 41, 101, 23, 41, 101]
>>> t*4
[23, 41, 101, 23, 41, 101, 23, 41, 101, 23, 41, 101]
```

1.3.3 Un peu de méthodes

Python est un langage *orienté objet* : en Python, tout est un *objet*, et ces objets sont regroupées en *classes*. Une *méthode* est une fonction qui s'applique aux objets d'une classe particulière. Si `method` est une méthode de la classe `c`, et si `a` est un objet de cette classe, alors on applique `method` à `a` avec la syntaxe suivante : `a.method()`, les parenthèses pouvant contenir des paramètres.

On peut ajouter un élément à un tableau avec la méthode `append`. Le point important est que cette opération s'effectue en temps constant amorti.

```
>>> t.append(-42)
>>> t
[23, 41, 101, -42]
```

À votre avis, pour un objet `x`, est-il équivalent d'effectuer `t+[x]` et `t.append(x)` ?

On peut aussi enlever le dernier élément d'une liste Python avec la méthode `pop()`.

```
>>> x = t.pop()
>>> t
[23, 41, 101]
>>> x
-42
```

Les autres méthodes disponibles sont rassemblées dans le tableau 1. Elles ne sont pas exigibles.

Méthode	Description
`append(x)`	Ajoute `x` en fin de liste
`extend(L)`	Concatène la liste `L` en fin de liste
`insert(i,x)`	Insère `x` à la position `i`
`remove(x)`	Supprime la première occurence de `x` (erreur si impossible)
`pop([i])`	Supprime l'élément en position `i` (si vide, le dernier)
`index(x)`	Renvoie l'indice de la première occurence de `x` (erreur si impossible)
`count(x)`	Renvoie le nombre d'occurences de `x`
`sort(cmp,key,rev)`	Trie la liste (nombreuses options)
`reverse()`	Renverse la liste

FIGURE 1 – Méthodes applicables aux listes.

1.4 Tranchage

On peut directement accéder à une sous-liste Python (ou tranche —*slice* en anglais— c'est-à-dire bloc de cases consécutives) d'une liste, c'est ce que l'on appelle le tranchage (*slicing*).

On utilise les syntaxes

— `u[i:j]` pour accéder à la tranche de la liste `u` allant des indices `i` à `j-1` ;

— `u[i:]` pour accéder à la tranche allant de l'indice `i` à la fin de la liste `u` ;

— `u[:j]` pour accéder à la tranche allant du début de la liste `u` à l'indice `j-1` ;

— `u[:]` pour accéder à la tranche complète (toute la liste `u`) ;

— `u[i:j:p]` pour accéder à la tranche de la liste `u` allant des indices `i` à `j-1`, par pas de `p`.

```
>>> u
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> u[2:5]
[4, 9, 16]
>>> u[2:]
[4, 9, 16, 25, 36, 49, 64, 81]
>>> u[:5]
[0, 1, 4, 9, 16]
>>> u[1:8:2]
[1, 9, 25, 49]
```

1.5 Tableaux multidimensionnels

On peut représenter une matrice avec des listes `Python` par exemple en la décrivant ligne par ligne. Par exemple, on peut représenter la matrice

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

par

```
>>> M = [[1,2,3],[4,5,6],[7,8,9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M[0][2]
3
```

Cependant, cela ne permet pas d'effectuer les opérations classiques sur les matrices. On préférera utiliser les possibilités de la bibliothèque de calcul numérique `numpy`.

2 Alias

Parfois, lorsque l'on manipule des tableaux, on observe un phénomène étrange qui laisse pantois : c'est l'*aliasing*. C'est une notion assez subtile, mais qu'il est bon de

connaître pour éviter les problèmes qui en découlent, ou au moins avoir une parade lorsqu'on s'y trouve confronté.

Commençons par un exemple où tout se passe intuitivement :

```
>>> x = 3
>>> y = x
>>> x = 42
```

Que vaut alors `y` ?

```
>>> y
3
```

Avec des listes `Python` cela ne fonctionne pas exactement de cette façon :

```
>>> x = [0]*5
>>> y = x
>>> x[0] = 42
```

Que vaut alors `y` ?

```
>>> y
[42, 0, 0, 0, 0]
```

Tâchons de donner une explication à ce phénomène.

— Pour les objets de type *non mutable* (on ne peut pas les modifier, comme les types `int`, `float`, `bool`, `str`, `tuple`), une instruction du type `x = y` crée une nouvelle variable : la variable `x` pointe vers une case contenant la valeur de `y`, et on dit que `x` et `y` sont des *alias* de cette valeur. Réattribuer à `x` ou `y` une nouvelle valeur casse cet alias : les deux variables ne pointent plus vers la même valeur.

— Pour les objets de type *mutable*, les choses sont plus compliquées : on peut modifier leur contenu sans les réassigner. C'est le cas des listes `Python`. Après une instruction du type `x = y` , les variables `x` et `y` pointent vers le même objet : ce sont des *alias* de cet objet là aussi. Mais si l'on modifie le contenu de `x`, sans réaffecter `x`, l'alias n'est pas cassé et on modifie à la fois `x` et `y`.

La méthode `id()`, qui affiche pour chaque objet son "numéro d'identité", permet de mettre cela en évidence :

```
>>> x = 3
>>> y = x
>>> id(x),id(y)
(10455104, 10455104)
>>> x = 42
>>> id(x),id(y)
(10456352, 10455104)
```

et

```
>>> x = [0]*5
>>> y = x
>>> id(x),id(y)
(47160620206472, 47160620206472)
>>> x[0] = 42
>>> id(x),id(y)
(47160620206472, 47160620206472)
```

Si l'on veut que `x` et `y` ne soient plus des alias, on pourra plutôt utiliser la méthode `copy()`.

```
>>> x = [0]*5
>>> y = x.copy()
>>> id(x),id(y)
(47160620184904, 47160620205256)
>>> x[0] = 42
>>> y
[0, 0, 0, 0, 0]
```

Il existe d'autres manières pour des tableaux de casser cet alias : `y = list(x)`, `y = x[:]` ou encore `y = x+[]`.

Cependant, en insérant des tableaux dans d'autres tableaux, on a une notion de "profondeur". Les techniques données ci-dessus ne permettent de rompre un alias qu'au niveau de l'enveloppe externe. Il existe la fonction `deepcopy` de la bibliothèque `copy`, qui effectue une copie totale d'un objet, en profondeur comme son nom l'indique. Cela dit il est peu probable que nous en ayions besoin.

3 Algorithmes

Vous devez être capables de reproduire quelques algorithmes sur les listes `Python` : ils figurent au programme de l'informatique tronc commun.

3.1 Calcul de la moyenne et de la variance.

Pour un tableau de nombres $t = [t_0, \dots, t_{n-1}]$ de longueur n, on voudra souvent calculer sa moyenne

$$\bar{t} = \frac{1}{n} \sum_{k=0}^{n-1} t_k$$

ainsi que sa variance

$$\sigma^2(t) = \frac{1}{n} \sum_{k=0}^{n-1} \left(t_k - \bar{t}\right)^2 = \left(\frac{1}{n} \sum_{k=0}^{n-1} t_k^2\right) - \left(\bar{t}\right)^2 .$$

On réalise cela de manière naïve. Exécutons donc le script suivant

```
def moyenne(t):
    """Calcule la moyenne de t
       Précondition : t est un tableau de nombres non vide"""
    s = 0
    for x in t:
        # Invariant : s == somme des éléments de t avant x
        s = s + x
    return s/len(t)

def variance(t):
    """Renvoie la variance de t
       Précondition : t est un tableau de nombres non vide"""
    sc = 0
    for x in t:
        # Invariant : sc == somme des carrés des éléments de t avant x
        sc = sc + x**2
    return sc/len(t) - moyenne(t)**2

t = [i for i in range(101)]
print(moyenne(t))
print(variance(t))
```

Cela renvoie alors :

```
50.0
850.0
```

Une solution plus naturelle dans d'autres langages (mais pas `Python`) pour calculer la moyenne, serait de parcourir `t` suivant ses indices. Par exemple, pour le calcul de la moyenne, cela s'écrit comme suit.

```
def moyenne(t):
    """Calcule la moyenne de t
       Précondition : t est un tableau de nombres non vide"""
    n = len(t) # Longueur de t
    s = 0
    for i in range(n):
        # Invariant : s == sum(t[0:i])
        s = s + t[i]
    return s/n
```

3.2 Recherche du maximum d'un tableau

3.2.1 Obtenir la valeur du maximum

On cherche à obtenir le maximum d'un tableau de nombres. Pour cela, on balaie séquentiellement le tableau en se souvenant du plus grand nombre déjà rencontré.

```
def maxi(t):
    """Retourne le plus grand élément de t.
       Précondition : t est un tableau non vide"""
    m = t[0]
    for x in t:
        # Invariant : m est le plus grand élément trouvé jusqu'ici
        if x > m:
            m = x # On a trouvé plus grand, on met à jour m
    return m

print(maxi([-5,2,9,-6,3]))
```

Cela renvoie alors :

```
9
```

Dans de nombreux autres langages de programmation, on ne parcourt pas directement les éléments d'un tableau mais leurs indices. On écrira quelque chose ressemblant plutôt à

```
def maxi(t):
    """Retourne le plus grand élément de t.
       Précondition : t est un tableau non vide"""
    m = t[0] # Initialisation par le premier élément
    for i in range(1, len(t)):
        # Invariant : m == max(t[0:i])
        if t[i] > m:
            m = t[i] # On a trouvé plus grand, on met à jour m
    return m
```

En Python, c'est moins élégant que l'autre solution.

3.2.2 Obtenir l'indice du maximum

On veut maintenant écrire une fonction `indicemaxi` donnant l'indice d'un plus grand élément d'un tableau, passé en argument.

On reprend la même idée que précédemment en parcourant le tableau par ses indices. On garde alors en mémoire l'indice du plus grand élément déjà trouvé.

Dans la plupart des langages, on écrira le script suivant.

```
def indicemaxi(t):
    """Retourne l'indice du plus grand élément de t.
       Précondition : t est un tableau non vide"""
    im = 0 # Indice du maximum, initialisation par le premier élément
    for i in range(1, len(t)):
        # Invariant : im est indice d'un plus grand élément de t[0:i]
        if t[i] > t[im]:
            im = i # On a trouvé plus grand, on met à jour im
    return im

print(indicemaxi([5,-1,10,3]))
```

Cela renvoie alors :

```
2
```

Cependant, les vrais pythonistes écriraient plutôt :

```
def indicemaxi(t):
    """Retourne l'indice du plus grand élément de t.
       Précondition : t est un tableau non vide"""
    im = 0 # Indice du maximum, initialisation par le premier élément
    for i, x in enumerate(t):
        # Invariant : im est indice d'un plus grand élément de t[0:i]
        if x > t[im]:
            im = i # On a trouvé plus grand, on met à jour im
    return im
```

3.3 Recherche séquentielle d'un élément dans un tableau

On veut écrire une fonction `appartient(t,e)` retournant un booléen disant si un élément `e` appartient au tableau `t`.

Pour cela, on parcourt séquentiellement le tableau en s'arrêtant dès qu'on le trouve. Si on a épuisé le tableau sans le trouver, on renvoie `False`. On écrit alors le script suivant.

```
def appartient(e, t):
    """Retourne un booléen disant si e appartient à t
       Précondition : t est un tableau"""
    for x in t:
        # Invariant : e n'est pas positionné dans t avant x
        if e == x:
            return True # On a trouvé e, on s'arrête
    return False

t = [i**3 for i in range(10)]
```

```
print(appartient(3,t))
print(appartient(8,t))
```

Cela renvoie alors :

```
False
True
```

On peut aussi vouloir obtenir l'indice de la première occurence de cet élément. On peut alors modifier la fonction précédente pour parcourir le tableau par ses indices, s'arrêter dès que l'on trouve l'élément et renvoyer `None` sinon.

```
def ind_appartient(e,t):
    """Retourne l'indice de la première occurence de e dans t,
       None si e n'est pas dans t
       Précondition : t est un tableau"""
    for i in len(t):
        # Invariant : e n'est pas dans t[0:i]
        if t[i] == e:
            return i # On a trouvé e à l'indice i
    return None
```

3.4 Recherche dichotomique dans un tableau trié

Dans le problème précédent de recherche d'un élément séquentiellement dans un tableau, on risque de parcourir intégralement le tableau si ce l'élément se trouve à la fin du tableau. Cependant, si le tableau est trié, on peut éviter ceci.

On suppose donc que l'on cherche un élément dans un tableau préalablement *trié par ordre croissant*. On regarde alors l'élément « au milieu du tableau », ce que l'on pourrait appeler un *pivot*. Si ce pivot est plus petit que ce que l'on cherche, on sait qu'il faut chercher à droite du pivot. Si ce pivot est plus grand que ce que l'on cherche, on sait qu'il faut chercher à gauche du pivot.

Pour faire cela, on va garder en mémoire une tranche de notre tableau, que l'on va affiner par dichotomie.

```
def appartient_dicho(e,t):
    """Renvoie un booléen indiquant si e est dans t
       Préconditions : t est un tableau de nombres trié par ordre croissant
                       e est un nombre"""
    g = 0 # Limite gauche de la tranche où l'on recherche e
    d = len(t)-1 # Limite droite de la tranche où l'on recherche e
    while g <= d: # La tranche où l'on cherche e n'est pas vide
        # Invariant : si e est dans t, t[g] <= e <= t[d]
        # Variant : d-g est strictement décroissant
        m = (g+d)//2 # Milieu de la tranche où l'on recherche e
        pivot = t[m]
```

```
        if e == pivot: # On a trouvé e
            return True
        elif e < pivot:
            d = m-1 # On recherche e dans la partie gauche de la tranche
        else:
            g = m+1 # On recherche e dans la partie droite de la tranche
    return False

t = [i**2 for i in range(101)]
print(appartient_dicho(11,t))
print(appartient_dicho(81,t))
```

Cela renvoie alors :

```
False
True
```

On remarquera que, si la tableau `t` (trié) est de taille n, alors la recherche dichotomique effectuera de l'ordre de $\ln(n)$ opérations (comparaisons de flottants, additions et divisions d'entiers, affectations etc.). On dira que l'on effectue $O(\ln n)$ opérations. C'est bien mieux que le pire des cas de la recherche séquentielle, qui nécessite $O(n)$ opérations « en moyenne » (si l'élément recherché est répartie aléatoirement et uniformément dans le tableau) ainsi que dans le pire des cas.

4 Exercices

Exercice 4.0.1. Écrire une fonction `indice(x, t)` retournant un indice `i` tel que `t[i]==x` si `x` apparaît dans le tableau `t` et -1 sinon.

Exercice 4.0.2. Écrire une fonction `tous_les_indices(e,t)` retournant la liste de tous les indices des occurences de `e` dans le tableau `t`.

Exercice 4.0.3. Écrire une fonction `ind_appartient_dicho(e,t)` renvoyant l'indice d'une occurence de `e` dans le tableau `t` (`None` si `e` n'est pas dans `t`), en supposant que `t` est trié par ordre croissant.

Exercice 4.0.4. Écrire une fonction `dec_appartient_dicho(e,t)` renvoyant un booléen indiquant si `e` est dans le tableau `t`, en supposant que `t` est trié par ordre décroissant.