

TP 02 - Tableaux

October 18, 2021

1 Tableaux

1.1 Rappels sur les Tableaux

1.1.1 Type et définition

Les tableaux qui en Python sont représentés par le type `list` permettent de grouper des valeurs linéairement et d'y accéder par leur indice, qui est l'ordre de la valeur en partant de l'indice 0. On parle souvent de la case à l'indice i par référence aux cases mémoires et à une forme d'implémentation de ces tableaux.

Le tableau `t = [2,1,3]` contient la valeur 2 à l'indice 0. On y accède en écrivant `t[0]` en lecture comme en écriture. En fait, tout se passe comme si le tableau `t` était une collection de variables nommées `t[0]`, `t[1]` et `t[2]`.

Pour obtenir le nombre de valeurs on utilise la fonction `len`, ainsi `len(t)` est la valeur entière 3. L'indice du dernier élément d'un tableau `t` est donc toujours `len(t)-1`. Il est possible en Python d'utiliser le raccourci `t[-1]` à la place de `t[len(t)-1]`. De même, on peut utiliser `t[-2]`, `t[-3]`,
....

```
[1]: t = [ 'a', 'b', 'c' ]
      print(t[1]) # Affiche 'b' qui est l'élément à l'indice 1
      t[1] = 'd' # on place 'd' à la place de l'élément 'b' dans l'indice 1
      t # maintenant t vaut [ 'a', 'd', 'c' ]
```

b

```
[1]: ['a', 'd', 'c']
```

```
[2]: t = [0, 0, 0, 0, 0]
      # Modification des éléments au sein de ce tableau.
      n = len(t)
      for i in range(0,n):
          t[i] = 2*i
          print(t)
      t # la valeur finale
```

```
[0, 0, 0, 0, 0]
[0, 2, 0, 0, 0]
[0, 2, 4, 0, 0]
```

```
[0, 2, 4, 6, 0]
[0, 2, 4, 6, 8]
```

```
[2]: [0, 2, 4, 6, 8]
```

Les opérations d'accès à une case en lecture ou en écriture sont supposées prendre un temps (nombre d'instructions processeurs, temps effectif sur une architecture de référence, ...) qui est majoré par une constante, c'est-à-dire indépendant du nombre d'éléments présents dans le tableau. On dit que les opérations s'effectuent en **temps constant**. On notera parfois que la complexité temporelle en pire cas est en $O(1)$, notation que l'on justifiera dans la suite mais qui servira uniquement pour le vocabulaire dans un premier temps.

1.1.2 Parcours

Il est possible de parcourir un tableau avec une boucle `for` :

```
for x in tableau:
    # corps de boucle
```

Les instructions dans le corps de boucle vont être exécutées pour chaque valeur présente dans le tableau, de la première à la dernière. On peut accéder à cette valeur en utilisant le nom de variable `x`.

```
[3]: t = [1,2,3,4]
    for x in t:
        print(x) # ici tout se passe comme si on exécutait l'instruction
                  # avec x qui prend tour à tour chaque valeur de t DANS L'ORDRE
```

```
1
2
3
4
```

On remarque qu'on est alors très proche de la boucle déjà vue `for i in range(a,b+1)` qui se comporte comme si on parcourait le tableau `[a,a+1,...,b]`.

Si le tableau comporte n éléments et que chaque itération du corps de boucle s'exécute au plus en c itérations, la boucle s'exécutera en nc itérations. En considérant que c ne dépend pas de n , on dit que la **complexité temporelle en pire cas** est linéaire dans le nombre d'éléments du tableau. Quand c est majoré par une constante, on dit que la complexité temporelle en pire cas est en $O(n)$.

1.1.3 Opération élémentaires

Ajout d'un élément Il est possible d'ajouter un élément à la fin d'un tableau en $O(1)$ (temps constant) avec la **méthode** `.append` on ne détaillera pas ce que signifie une méthode au delà du fait que c'est une fonction qui appartient au tableau :

```
[4]: t = [ 1, 2 ]
    t.append(3)
    t # maintenant t contient [1, 2, 3]
```

```
[4]: [1, 2, 3]
```

On peut notamment se servir de cela pour initialiser un tableau, par exemple, pour faire le tableau `[1, 2, 3, ... ]` :

```
[5]: # Création d'un tableau vide
t1 = []
# Ajout d'éléments au sein du tableau via la méthode 'append'
for i in range(0,5):
    t1.append(i)
print(t1)
```

```
[0]
[0, 1]
[0, 1, 2]
[0, 1, 2, 3]
[0, 1, 2, 3, 4]
```

```
[6]: # On peut ainsi faire une fonction qui renvoie un tel tableau
def mon_range(n):
    t = []
    for i in range(n):
        t.append(i)
    return t

mon_range(10)
```

```
[6]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Concaténation de deux tableaux On peut utiliser `+` pour coller deux tableaux à la suite :

```
[7]: [1, 2, 3] + [4, 5, 6]
```

```
[7]: [1, 2, 3, 4, 5, 6]
```

On prendra garde au fait que cette opération est linéaire par rapport au nombre d'éléments des deux listes. Ainsi écrire `l = l + [1]` prendra **beaucoup** plus longtemps que `l.append(1)`.

Duplication ou multiplication de tableaux Maintenant qu'on a défini une opération `+` on peut en déduire une opération `*` sur le modèle des entiers ainsi `l * 4` est juste `l + l + l + l`.

On se sert très rarement de cette opération, sauf pour un cas spécifique : initialiser un tableau avec une valeur constante donnée dans chaque case.

```
[8]: t = [ 0 ] * 5
t # contient 5 zeros
```

```
[8]: [0, 0, 0, 0, 0]
```

ATTENTION si on l'utilise pour des valeurs plus compliquées, notamment pour des tableaux de tableaux, ça ne marchera pas car, comme on le verra en détail, la copie est superficielle.

1.2 Tableaux des entiers pairs

```
[1]: def pair(p):  
    '''Prend en argument un entier p et renvoie un tableau constitué  
    des entiers pairs allant de 0 (inclus) à 2*p (inclus) '''  
    ### BEGIN SOLUTION  
    L = [0]*(p+1)  
    for k in range(p+1):  
        L[k] = 2*k  
    return L  
    ### END SOLUTION
```

```
[5]: from civ import init, valide, test  
init()  
test(pair,3,[0,2,4,6])  
test(pair,0,[0])  
valide()
```

OK!: pair(3) vaut [0, 2, 4, 6]

OK!: pair(0) vaut [0]

Tous les tests sont valides

1.3 Moyenne des éléments

```
[11]: def moyenne(t):  
    '''Prend en argument un tableau t et renvoie la moyenne des éléments  
    de ce tableau. Cette fonction renverra 0 dans le cas d'un tableau vide'''  
    ### BEGIN SOLUTION  
    n = len(t)  
    if n == 0:  
        return 0  
    else :  
        m = 0  
        for x in t:  
            m = m+x  
        return m/n  
    ### END SOLUTION
```

```
[12]: from civ import test  
test(moyenne,[],0)  
test(moyenne,[1,1,2,-1,3],1.2)
```

OK!: moyenne([]) vaut 0

OK!: moyenne([1, 1, 2, -1, 3]) vaut 1.2

1.4 Entrelacement de tableaux

```
[13]: def entrelace(t1,t2):  
    '''Prend en argument deux tableaux de même longueur et renvoie  
    un tableau de la forme [t1[0],t2[0],t1[1],t2[1],...]'''  
    ### BEGIN SOLUTION  
    L = []  
    n = len(t1)  
    for k in range(n):  
        L.append(t1[k])  
        L.append(t2[k])  
    return L  
    ### END SOLUTION
```

```
[14]: from civ import init, valide, test  
init()  
test(entrelace,[0,1,-1,12],[54,2,-100,3],[0,54,1,2,-1,-100,12,3])  
test(entrelace,[],[],[])  
valide()
```

OK!: entrelace([0, 1, -1, 12], [54, 2, -100, 3]) vaut [0, 54, 1, 2, -1, -100, 12, 3]

OK!: entrelace([], []) vaut []

Tous les tests sont valides

1.5 Produit des éléments

```
[15]: def produit(t):  
    '''Renvoie le produit des éléments du tableau d'entiers t.  
    Par convention, si le tableau est vide on renvoie 1'''  
    ### BEGIN SOLUTION  
    p = 1  
    for x in t:  
        p = p * x  
    return p  
    ### END SOLUTION
```

```
[16]: from civ import init, valide, test  
init()  
test(produit, [], 1)  
test(produit,[1,2,3],6)  
test(produit,[2],2)  
valide()
```

OK!: produit([]) vaut 1

OK!: produit([1, 2, 3]) vaut 6

OK!: produit([2]) vaut 2

Tous les tests sont valides

Il est possible de convertir un `range` en un tableau à l'aide de la fonction `list` : `list(range(3))` renvoie ainsi `[0,1,2]`.

En déduire une fonction `fact` qui calcule la factorielle d'un entier.

Rappel : $n! = 1 \times 2 \times \dots \times n$.

```
[17]: def fact(n):  
    '''Renvoie la factorielle de l'entier n'''  
    ### BEGIN SOLUTION  
    return produit(range(1,n+1))  
    ### END SOLUTION
```

```
[18]: from civ import test, init, valide  
init()  
test(fact,0,1)  
test(fact,3,6)  
valide()
```

OK!: `fact(0)` vaut 1

OK!: `fact(3)` vaut 6

Tous les tests sont valides

2 Recherche séquentielle d'un élément dans un tableau

```
[19]: def recherche(t, cle):  
    '''Recherche dans le tableau t le plus petit indice correspondant à la_  
→valeur cle.  
Si la valeur n'est pas présente dans le tableau, on produira une erreur avec  
raise ValueError'''  
    ### BEGIN SOLUTION  
    for i in range(len(t)):  
        if t[i] == cle:  
            return i  
    raise ValueError  
    ### END SOLUTION
```

```
[20]: from civ import test, test_erreur, init, valide  
init()  
test(recherche,[1,2,3], 2,1)  
test(recherche,[1,2,3], 1,0)  
test(recherche,[1], 1,0)  
test(recherche,[1,1,1], 1,0)  
# ces tests vérifient que vous faites bien des erreurs quand il n'y a pas_  
→l'élément  
test_erreur(recherche,[1], 2, ValueError)  
test_erreur(recherche, [], 2, ValueError)  
valide()
```

```

OK!: recherche([1, 2, 3], 2) vaut 1
OK!: recherche([1, 2, 3], 1) vaut 0
OK!: recherche([1], 1) vaut 0
OK!: recherche([1, 1, 1], 1) vaut 0
OK!: recherche([1], 2) a bien fait l'erreur ValueError
OK!: recherche([], 2) a bien fait l'erreur ValueError
Tous les tests sont valides

```

2.1 Premier et second maximum

```

[10]: def maximum(t):
        '''Renvoie le plus petit indice du maximum du tableau t sans utiliser la_
        →fonction max
        Si le tableau est vide, on produira une erreur avec raise ValueError'''
        ### BEGIN SOLUTION
        if t == []:
            raise ValueError
        m = 0
        for i in range(1, len(t)):
            if t[i] > t[m]:
                m = i
        return m
        ### END SOLUTION

```

```

[11]: from civ import test, test_erreur, init, valide
init()
test(maximum,[1,2,3,4],3)
test(maximum,[4,3,2,1],0)
test(maximum,[-3,-2], 1)
test(maximum,[1],0)
test(maximum,['a','b'],1)
test_erreur(maximum, [], ValueError)
valide()

```

```

OK!: maximum([1, 2, 3, 4]) vaut 3
OK!: maximum([4, 3, 2, 1]) vaut 0
OK!: maximum([-3, -2]) vaut 1
OK!: maximum([1]) vaut 0
OK!: maximum(['a', 'b']) vaut 1
OK!: maximum([]) a bien fait l'erreur ValueError
Tous les tests sont valides

```

```

[13]: def liste_maximum(t):
        '''Renvoie un tableau des indices du maximum du tableau t sans utiliser la_
        →fonction max
        Si le tableau est vide, on produira une erreur avec raise ValueError'''
        ### BEGIN SOLUTION

```

```

if t == []:
    raise ValueError
m = 0
res = [m]
for i in range(1, len(t)):
    if t[i] > t[m]:
        m = i
        res = [m]
    elif t[i] == t[m]:
        res.append(i)
return res
### END SOLUTION

```

```

[14]: from civ import test, test_erreur, init, valide
init()
test(liste_maximum,[1,2,3,4],[3])
test(liste_maximum,[4,3,2,1],[0])
test(liste_maximum,[4,3,4,1],[0,2])
test(liste_maximum,[-3,-2,-3,-2],[1,3])
test(liste_maximum,[1],[0])
test_erreur(liste_maximum, [], ValueError)
valide()

```

OK!: liste_maximum([1, 2, 3, 4]) vaut [3]
 OK!: liste_maximum([4, 3, 2, 1]) vaut [0]
 OK!: liste_maximum([4, 3, 4, 1]) vaut [0, 2]
 OK!: liste_maximum([-3, -2, -3, -2]) vaut [1, 3]
 OK!: liste_maximum([1]) vaut [0]
 OK!: liste_maximum([]) a bien fait l'erreur ValueError
 Tous les tests sont valides

Si t est un tableau contenant au moins deux valeurs comparables distinctes, on appelle second maximum de t la valeur dans t directement plus petite que sa valeur maximale.

```

[25]: def second_maximum(t):
    '''Renvoie le plus petit indice du second maximum du tableau t.
    Si le tableau contient moins de deux éléments distincts, on produira une
    ↪ erreur avec raise ValueError'''
    ### BEGIN SOLUTION
    if t == []:
        raise ValueError
    m = 0
    n = 1
    while n < len(t) and t[n] == t[m]:
        n = n + 1
    if n == len(t):
        raise ValueError
    i = n+1

```

```

if t[m] < t[n]:
    m, n = n, m
for x in range(i, len(t)):
    if t[x] > t[m]:
        m, n = x, m
    elif t[x] != t[m] and t[x] > t[n]:
        n = x
return n

```

END SOLUTION

```

[26]: from civ import test, test_erreur, init, valide
init()
test(second_maximum, [1,2,3,4], 2)
test(second_maximum, [1,2,3,3], 1)
test(second_maximum, [1,2,1,2], 0)
test_erreur(second_maximum, [], ValueError)
test_erreur(second_maximum, [1,1,1], ValueError)
valide()

```

OK!: second_maximum([1, 2, 3, 4]) vaut 2
 OK!: second_maximum([1, 2, 3, 3]) vaut 1
 OK!: second_maximum([1, 2, 1, 2]) vaut 0
 OK!: second_maximum([]) a bien fait l'erreur ValueError
 OK!: second_maximum([1, 1, 1]) a bien fait l'erreur ValueError
 Tous les tests sont valides

2.2 Sous-tableau de somme maximale

Étant donné un tableau t on appelle sous-tableau de t la donnée d'un couple d'indices (i, j) où $i < j$ représentant les éléments de t compris entre les indices i (compris) et j (non compris). On le notera $t[i:j]$ ce qui correspond à la syntaxe Python pour cette notion même si on ne l'utilisera pas pour répondre à cette question.

On considère ici un tableau t d'entiers et on cherche la plus grande somme d'un sous-tableau :

$$\max_{i < j} \sum_{x \in t[i:j]} x$$

```

[27]: def somme_maximale(t):
    '''Prend en entrée un tableau t d'entiers et renvoie la valeur de la somme_
    ↪ maximale
    d'un sous-tableau'''
    ## BEGIN SOLUTION
    m = 0
    n = len(t)
    for i in range(n):
        for j in range(i+1, n+1): # ATTENTION au n+1 ici !

```

```
s = 0
for k in range(i,j):
    s = s + t[k]
    if s > m:
        m = s
return m
## END SOLUTION
```

```
[28]: from civ import test, init, valide
init()
test(somme_maximale,[-5,3,7,15,-17,8,12,1,-6,8],31)
valide()
```

OK!: somme_maximale([-5, 3, 7, 15, -17, 8, 12, 1, -6, 8]) vaut 31
Tous les tests sont valides