Tableaux — Première NSI

"Objectifs"

À la fin du chapitre, on doit savoir :

- créer un tableau en Python, en extension ou en compréhension ;
- accéder à ses éléments, les modifier et extraire une tranche ;
- parcourir un tableau par élément ou par indice ;
- distinguer copie et référence partagée (aliasing) ;
- prévoir les modifications d'un tableau passé en argument à une fonction ;
- utiliser les principales méthodes des listes Python et connaître le coût des opérations usuelles.

Crédits

Ce cours est largement inspiré des chapitres 5 et 8 du manuel NSI de la collection Tortue (Ballabonski, Conchon, Filliâtre, N'Guyen). Le *Prépabac Première NSI* de Guillaume Connan (Hatier) a également été consulté.

Définition d'un tableau

"Définition 1"

Un **conteneur** regroupe plusieurs objets. Une **séquence** est un conteneur dont les éléments sont ordonnés et repérés par leur position, appelée **indice** (ou *index*).

Un **tableau** est une séquence permettant d'accéder directement à chacun de ses éléments à partir de son indice.



Tableaux en Python

"Propriété 1"

En Python, nous représentons les tableaux par des objets de type `list`, appelés aussi **listes**. Ils s'écrivent entre crochets, avec des éléments séparés par des virgules.

- Les indices commencent à **0** : un tableau de longueur `n` a pour indices valides `0` à `n - 1`.
- `len(tab)` donne sa longueur et `[]` désigne le tableau vide.
- Les éléments peuvent être modifiés : les listes Python sont **mutables**.
- Leur taille peut varier : les tableaux/listes sont **dynamiques**. S'ils peuvent contenir des types différents, nous privilégierons des tableaux homogènes avec des éléments de même type.

"Exemple 1"

```
tab = [4, 20, 10]
len(tab)        # 3
tab[0]          # 4
tab[2] = 6      # tab vaut désormais [4, 20, 6]
```

L'accès ou la modification d'un élément par son indice a un coût en **temps constant**, noté $O(1)$.

Accès aux éléments d'un tableau

"Méthode 1"

Pour un tableau/liste `tab` :

Expression	Effet
`tab[k]`	Lire l'élément d'indice `k`
`tab[k] = valeur`	Modifier cet élément

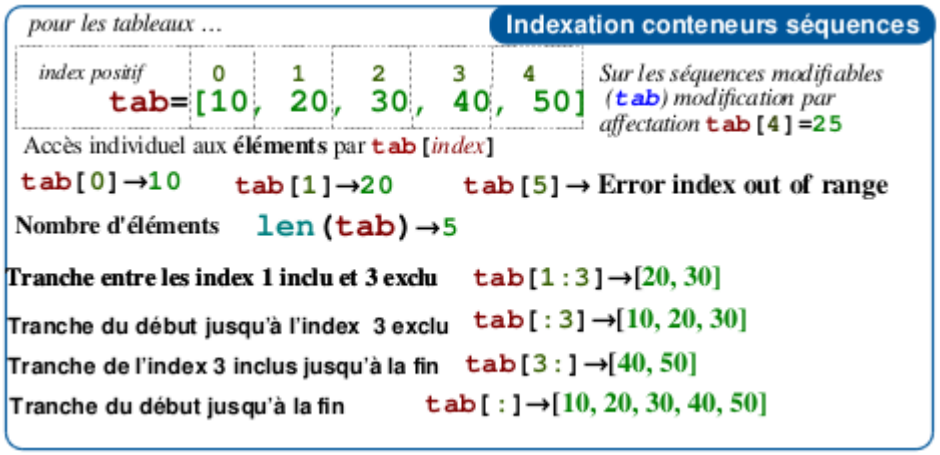
Expression	Effet
`tab[a:b]`	Créer un tableau avec les éléments d'indices `a` inclus à `b` exclu (*slicing*)
`x in tab`	Tester la présence de `x`
`x not in tab`	Tester l'absence de `x`

Un indice hors limites provoque une exception `IndexError`. Python accepte les indices négatifs, comptés depuis la fin ; nous utiliserons principalement les indices positifs.

"Exemple 2"

```
tab = [10, 20, 30, 40, 50]
tab[1:4]       # [20, 30, 40]
tab[:2]        # [10, 20]
tab[3:]        # [40, 50]
30 in tab      # True
```

La recherche d'une valeur avec `in` ou `not in` a un coût **temps linéaire** dans le pire cas (valeur en dernière position ou absente) noté $O(n)$.



"Exercice 1"

1. Soit la liste suivante :
   `liste_pays = ["France","Allemagne","Italie","Belgique"]`.
   Que renvoie : `liste_pays[2]` ?
2. Quelle est la valeur référencée par le tableau L après exécution du programme ci-dessous ?

```
L = [731, 732, 734]
L[0], L[1] = L[1], L[0]
M = L
M[1] = 732
```

- **Réponse 1 :** [732, 731, 734]
- **Réponse 2 :** [731, 732, 734]
- **Réponse 3 :** [732, 732, 734]

3. On définit : `L = [10,9,8,7,6,5,4,3,2,1]`.
   Quelle est la valeur de `L[L[3]]` ?
   - **Réponse 1 :** 3
   - **Réponse 2 :** 4
   - **Réponse 3 :** 7
   - **Réponse 4 :** 8

Construction d'un tableau

"Méthode 2"

Trois constructions courantes :

- **En extension** : on écrit les éléments, par exemple `[8, 10, 11, 14]`.
- **Par conversion** : `list(range(5))` produit `[0, 1, 2, 3, 4]` ; `list("abc")` produit `['a', 'b', 'c']`.
- **En compréhension** : `[expression for variable in iterable]`, éventuellement suivie d'une condition `if`.

"Exemple 3"

```
carres = [k ** 2 for k in range(6)]
# [0, 1, 4, 9, 16, 25]
positifs = [x for x in [-2, 5, 0, -1, 3] if x >= 0]
# [5, 0, 3]
```

Une compréhension produit un **nouveau tableau**. La condition `if` sert à filtrer les éléments.

"Exercice 2"

1. *Auteur : Germain Becker, question n°326 Genumsi*.
   Quel est le tableau `t` construit par les instructions suivantes ?

```
tab = [1, 2, -3, 7, 4, 10, -1, 0]
t = [e for e in tab if e >= 0]
```

   - **Réponse 1 :** `t = [1, 2, 7, 4, 10, 0]`
   - **Réponse 2 :** `t = [e, e, e, e, e, e]`
   - **Réponse 3 :** `t = [1, 2, 7, 4, 10]`
   - **Réponse 4 :** `t = [-3, -1, 0]`

2. Quel est le résultat de l'évaluation de l'expression Python suivante ? *Auteur : Eric Rougier*

```
[2 ** n for n in range(4)]
```

   - **Réponse 1 :** `[0, 2, 4, 6, 8]`
   - **Réponse 2 :** `[1, 2, 4, 8]`
   - **Réponse 3 :** `[0, 1, 4, 9]`
   - **Réponse 4 :** `[1, 2, 4, 8, 16]`

3. On exécute le script suivant :

```
L = [12,0,8,7,3,1,5,3,8]

a = [elt for elt in L if elt < 4]
```

   Quelle est la valeur de `a` à la fin de son exécution ?

   - **Réponse 1 :** `[12,0,8]`
   - **Réponse 2 :** `[12,0,8,7]`
   - **Réponse 3 :** `[0,3,1,3]`
   - **Réponse 4 :** `[0,3,1]`

"Exercice 3"

1. Pour chacun des tableaux ci-dessous définis en compréhension, déterminez la séquence d'éléments qu'il contient :
   - `t1 = [k ** 2 for k in range(6)]`
   - `t2 = [k ** 2 for k in range(-5, 6) if k **2 >= 4]`
   - `t3 = [i * j for i in range(1, 3) for j in range(2, 4)]`
2. Qu'affiche le programme Python ci-dessous ?

```
L=[0,1,2]
M=[3,4,5]
N=[L[i]+M[i] for i in range(len(L))]
print(N)
```

Opérations sur les tableaux

"Méthode 3"

Les opérateurs `+` et `*` permettent de construire de nouveaux tableaux :

```
[1, 2] + [3, 4]    # [1, 2, 3, 4]
[0] * 4            # [0, 0, 0, 0]
```

`+` **concatène** deux tableaux ; `*` **répète** leurs éléments. Ces opérations ne modifient pas les tableaux d'origine.

"Attention"

Répéter une valeur calculée une seule fois ne répète pas le calcul :

```
from random import randint
t1 = [randint(1, 6) for _ in range(5)]   # cinq tirages
t2 = [randint(1, 6)] * 5                 # un tirage répété cinq fois
```

Pour construire des éléments indépendants, privilégier une **compréhension**, notamment pour des tableaux d'objets mutables.

"Exercice 4"

1. On exécute le script suivant :

   ```
   a = [1, 2, 3]
   b = [4, 5, 6]
   c = a + b
   ```

   Que contient la variable `c` à la fin de cette exécution ?

- **Réponse 1 :** `[5,7,9]`
- **Réponse 2 :** `[1,4,2,5,3,6]`
- **Réponse 3 :** `[1,2,3,4,5,6]`
- **Réponse 4 :** `[1,2,3,5,7,9]`

2. Soit la fonction `mystere` ci-dessous.
   Quelle est la valeur retournée par `mystere([831, 832, 833, 841, 842, 843])` ?

```
def mystere(u):
  v = []
  n = len(u)
  for k in range(n):
    v = [u[k]]+v
  return v
```

Parcours d'un tableau

"Méthode 4"

Il existe deux parcours principaux.

Par valeur — quand on a seulement besoin des éléments :

```
tab = [3, 1, 4]
for valeur in tab:
    print(valeur)
```

Par indice — quand on doit connaître la position d'un élément ou modifier le tableau :

```
for k in range(len(tab)):
    print(k, tab[k])
```

Un parcours complet de `n` éléments a un coût **linéaire**, $O(n)$.

"Exercice 5"

1. On donne deux codes de fonctions qui prennent en paramètre un tableau d'entiers et renvoient la somme de tous ses éléments. Pour chaque fonction, déterminer si on utilise un parcours par index ou par valeur. Quel code vous semble le plus lisible ?

```
def somme1(tab):
  s = 0
  for e in tab:
    s = s + e
  return s
```

```
def somme2(tab):
  s = 0
  for k in range(0, len(tab)):
    s = s + tab[k]
  return s
```

2. On veut écrire une fonction qui prend en paramètres un tableau d'entiers `t` et un entier `v`. Si `v` est dans `t`, la fonction doit renvoyer l'index de la première occurrence de `v` en parcourant le tableau par index croissant à partir de 0. Si `v` n'est pas `t` la fonction doit renvoyer -1. On propose deux codes, un seul est correct, lequel ?

```
  def premiere_occ1(t, v):
    for e in t:
      if e == v:
        return e
    return -1
```

```
  def premiere_occ2(t, v):
    for k in range(0, len(t)):
      if t[k] == v:
        return k
    return -1
```

3. Écrire le code d'une fonction qui prend en paramètres un tableau d'entiers `t` et un entier `v`. Si `v` est dans `t`, la fonction doit renvoyer l'index de la dernière occurrence de `v` en parcourant le tableau par index croissant à partir de 0. Si `v` n'est pas `t` la fonction doit renvoyer -1.
4. En voulant programmer une fonction qui calcule la valeur minimale d'un tableau d'entiers, on a écrit :

```
def minimum(L):
    mini = 0
    for e in L:
        if e < mini:
            mini = e
    return mini
```

Cette fonction a été mal programmée. Pour quel tableau ne donnera-t-elle pas le résultat attendu, c'est-à-dire son minimum ?

- **Réponse 1 :** `[-1,-8,12,2,23]`
- **Réponse 2:** `[0,18,12,2,3]`
- **Réponse 3:** `[-1,-1,12,12,23]`
- **Réponse 4:** `[1,8,12,2,23]`

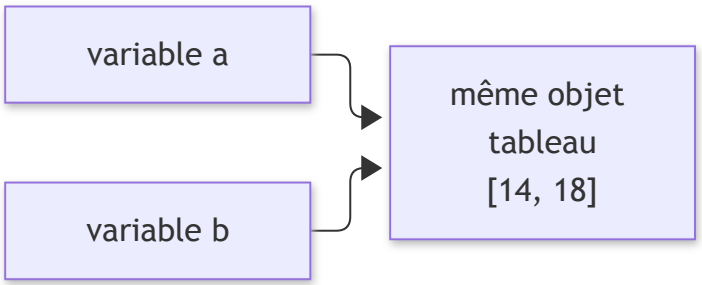
5. Écrire une fonction `max_tab(tab)` qui renvoie le maximum d'un tableau de nombres passé
en paramètre. La fonction renvoie `None` si le tableau est vide.

Aliasing

Aliasing et copie de tableau

"Définition 2"

Une variable de type `list` référence un **objet tableau**. L'affectation `b = a` ne copie pas cet objet : les deux noms désignent le **même tableau**. Ce partage de référence est appelé **aliasing**.



"Exemple 4"

```
a = [14, 18]
b = a
b[0] = 7
print(a)        # [7, 18] : modification visible par a
```

Contrairement à l'affectation d'une nouvelle valeur à une variable de type `int`, une modification *en place* du tableau est visible par tous ses alias.

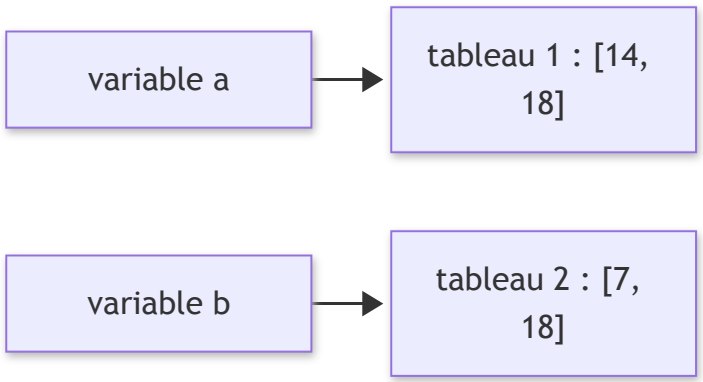
"Méthode 5 : copier un tableau"

Pour créer une **copie superficielle**, on utilise `b = a[:]` ou `b = a.copy()`. Les deux tableaux sont alors distincts, mais les objets qu'ils contiennent peuvent encore être partagés.

```
a = [14, 18]
b = a[:]
b[0] = 7
print(a)         # [14, 18]
```

Pour des tableaux **imbriqués** (tableaux de tableaux), une copie totalement indépendante peut nécessiter `copy.deepcopy(a)` du module `copy`.

Dans cet exemple, les deux noms référencent deux tableaux distincts :



"Exercice 6"

1. Écrire une fonction `copie(tab)` qui renvoie une copie superficielle du tableau `tab` sans utiliser de *slicing*.
2. *Auteur : Nicolas Revéret*
   On considère le code suivant :

```
def f(tab):
  for i in range(len(tab)//2):
    tab[i],tab[-i-1] = tab[-i-1],tab[i]
```

Après les lignes suivantes :

```
tab = [2,3,4,5,7,8]
f(tab)
```

Quelle est la valeur de la variable `tab` ?

Réponses :

- **Réponse 1** `[2,3,4,5,7,8]` **Réponse 2** `[5,7,8,2,3,4]`
- **Réponse 3** `[8,7,5,4,3,2]` **Réponse 4** `[4,3,2,8,7,5]`

Aliasing et passage de tableau en paramètre d'une fonction

"Propriété 2"

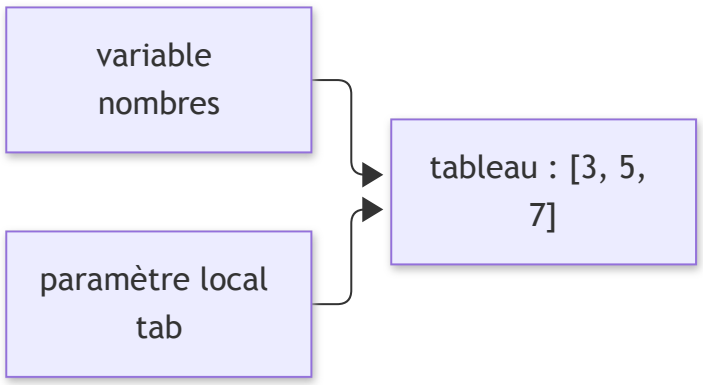
Lorsqu'on passe un tableau en argument d'une fonction, le **paramètre local référence le même tableau** que l'argument. Une modification *en place* dans la fonction est donc visible après son appel : c'est un **effet de bord**.

"Exemple 5"

```
def incremente_tab(tab: list) -> None:
    """Augmente de 1 chaque élément du tableau."""
    for k in range(len(tab)):
        tab[k] = tab[k] + 1

nombres = [2, 4, 6]
incremente_tab(nombres)
print(nombres)      # [3, 5, 7]
```

Pendant l'exécution de `incremente_tab(nombres)`, le paramètre local `tab` et la variable `nombres` désignent le même tableau (ici après les modifications).



Une telle fonction, qui agit principalement par effet de bord et ne renvoie pas de valeur utile (`None`), est souvent appelée **procédure**.

"À distinguer"

Réaffecter le paramètre (`tab = [1, 2]`) ne modifie pas le tableau initial : cela fait simplement référencer **un autre objet** au nom local `tab` .

"Exercice 7"

1. On veut écrire une procédure `echange(tab, i, j)` qui échange les éléments d'index `i` et `j` d'un tableau `tab` . On fournit le modèle ci-dessous :

```
def echange(tab, i, j):
  assert 0 <= i < len(tab) and 0 <= j < len(tab), "message d'erreur"
  # à compléter
```

   - Remplacer le message d'erreur de l'assertion par un message pertinent.
   - Compléter la fonction.

2. Écrire une procédure `permutation_circulaire(tab)` qui modifie la position des éléments du tableau `tab` par permutation circulaire de gauche à droite :

```
>>> t = list(range(6))
>>> t
[0, 1, 2, 3, 4, 5]
>>> permutation_circulaire(t)
>>> t
[5, 0, 1, 2, 3, 4]
>>> permutation_circulaire(t)
>>> t
[4, 5, 0, 1, 2, 3]
```

Méthodes de tableau dynamique en Python

"Méthode 6"

Les listes Python sont **dynamiques** : leur taille peut changer. Leurs méthodes s'appellent avec la notation `tab.methode(...)`.

Méthode	Rôle	Coût usuel ou au pire
`tab.append(x)`	Ajouter `x` à la fin	Amorti constant $O(1)$
`tab.insert(k, x)`	Insérer `x` à l'indice `k`	Linéaire $O(n)$
`tab.pop()`	Retirer et renvoyer le dernier élément	Constant $O(1)$
`tab.pop(k)`	Retirer et renvoyer l'élément d'indice `k`	Linéaire $O(n)$
`tab.remove(x)`	Supprimer la première occurrence de `x`	Linéaire $O(n)$
`tab.count(x)`	Compter les occurrences de `x`	Linéaire $O(n)$

Les insertions ou suppressions au début peuvent entraîner le décalage des éléments qui suivent. Ces méthodes (à l'exception de `count`) **modifient le tableau existant**.

"Exemple 6"

```
t = [1, 3]
t.append(4)        # [1, 3, 4]
t.insert(1, 2)     # [1, 2, 3, 4]
x = t.pop()        # x vaut 4 ; t vaut [1, 2, 3]
t.remove(2)        # [1, 3]
t.count(1)         # 1
```

`remove(x)` lève `ValueError` si `x` n'est pas présent ; `pop()` lève `IndexError` si le tableau est vide.

"Exercice 8"

1. Écrire une fonction fonction `filtre_notes(tab)` qui extrait toutes les notes inférieures à 10 d'un tableau de notes entre 0 et 20 et les renvoie dans un autre tableau.
2. Écrire une fonction renvoyant le maximum d'un tableau passé en paramètre et le tableau des positions où ce maximum est atteint. La fonction renvoie `[None, []]` si le tableau est vide.
3. On définit en Python la fonction suivante :

```
def f(L):
    S = []
    for i in range(len(L)-1):
        S.append(L[i] + L[i+1])
    return S
```

Quel est le tableau renvoyé par `f([1, 2, 3, 4, 5, 6])` ?

- **Réponse 1** `[3, 5, 7, 9, 11, 13]`
- **Réponse 2** `[1, 3, 5, 7, 9, 11]`
- **Réponse 3** `[3, 5, 7, 9, 11]`
- **Réponse 4** cet appel de fonction déclenche un message d'erreur

Synthèse

"À retenir"

- Un tableau est une **séquence indexée** ; en Python, nous utilisons le type `list` et des indices à partir de `0`.
- `tab[k]` accède à un élément en coût constant en $O(1)$; `len(tab)` donne la longueur. Une recherche séquentielle et un parcours complet ont un coût linéaire en $O(n)$.
- On construit un tableau **en extension**, par `list(...)` ou **en compréhension** ; on peut le parcourir par **valeur** ou par **indice**.
- Une affectation entre variables de type `list` crée des **alias** ; `tab[:]` réalise une **copie superficielle**.
- Le passage d'un tableau à une fonction partage sa référence : une modification en place produit un **effet de bord**.
- Les méthodes `append`, `insert`, `pop`, `remove` modifient les listes dynamiques ; leurs coûts dépendent notamment des décalages nécessaires.