Tableaux (type `list`)

Cours

Un **tableau** est une suite **ordonnée** d'éléments qui peuvent être **modifiés** (muables[8]).

En Python les tableaux sont du type `list`[9].

Création

Un tableau est déclaré par une série de valeurs séparées par des virgules, et le tout **encadré par des crochets** "`[ ]`". Il contient des éléments du même type (selon le programme de 1ere[1]) .

```
>>> t = [1, 2, 3, 4]
>>> t
[1, 2, 3, 4]
>>> type(t)
<class 'list'>
```

⚠️ Comme pour les p-uplets, ne pas confondre la virgule de séparateur d'éléments avec le point de séparateur décimal.

Il est possible de créer un tableau vide :

```
t_vide = []      # Creation d'un tableau vide
```

ou un tableau contenant un seul élément :

```
t_1_elem = [5] # Creation d'un tableau avec un seul element
```

Fonction `len()`

La fonction `len()` renvoie la longueur d'un tableau, c'est-à-dire le nombre d'éléments qu'il contient.

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> len(animaux)
4
>>> len([1, 2, 3, 4, 5, 6, 7, 8])
8
```

Accès aux éléments

Comme pour les chaines de caractères et les p-uplets, les éléments d'un tableau `t` sont indexés à partir de `0` jusqu'à `len(t)` **exclus**, c'est-à-dire le dernier élément est en position `len(t) - 1`. Il est possible d'accéder aux

éléments par leur index entre crochets.

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1]
'b'
```

⚠️ Le premier élément est à l'index 0.

Comme pour les p-uplets, on peut utiliser des indices négatifs, le dernier élément à l'indice `-1`, l'avant-dernier `-2`, etc.

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[-1]
'f'
>>> t[-3]
'd'
```

L'accès à une partie d'un tableau (une « tranche ») se fait sur le modèle `t[début:fin]` [2] pour récupérer tous les éléments, entre les positions `debut` (inclus) et `fin` (**exclu**).

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[1:-2]
['b', 'c', 'd']
```

Lorsqu'aucun indice n'est indiqué à gauche ou à droite du symbole deux-points, Python prend par défaut tous les éléments depuis le début ou tous les éléments jusqu'à la fin respectivement.

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:]
['b', 'c', 'd', 'e', 'f']
>>> t[:2]
['a', 'b']
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Il est aussi possible de disperser, ou « déballer », un tableau en affectant tous ses éléments dans plusieurs variables :

```
>>> a, b, c, d = [1, 2, 3, 4]
>>> a
1
```

Le mot clé `in` permet de vérifier si un élément `elem` est présent dans un tableau `t`, `elem in t` renvoie un booléen :

```
>>> t = [1, 3 ,8]
>>> 3 in t
True
>>> 4 in t
False
```

De façon très similaire aux p-uplets, le mot clé `in` permet aussi d'écrire une boucle pour parcourir (ou «itérer») toutes les valeurs d'un tableau. Comparons à nouveau différentes méthodes pour parcourir un tableau `t` :

Avec une boucle non bornée `while`

Il faut gérer l'indice de boucle `i` pour qu'il parcourt toutes les positions des valeurs de `t`, c'est-à-dire l'initialiser à `0` puis l'incrémenter à chaque passage dans la boucle (`i = i + 1`) jusqu'à ce qu'il dépasse `len(t) - 1`. `t[i]` permet d'accéder à la valeur du tableau qui se trouve à la position `i`.

```
>>> t = [1, 3, 8]
>>> i = 0
>>> while i < len(t):
...     print(t[i])
...     i = i + 1
...
1
3
8
```

Avec une boucle bornée `for`

Avec `for i in range(len(t)):` , l'indice de boucle `i` prend automatiquement les valeurs allant de `0` à `len(t) - 1`. `t[i]` permet d'accéder à la valeur du p-uplet qui se trouve à la position `i`.

```
>>> t = [1, 3, 8]
>>> for i in range(len(t)):
...     print(t[i])
...
1
3
8
```

Avec une boucle bornée `for` et le mot clé `in`

`for elem in t` permet d'accéder directement à toutes les du tableau les unes après les autres, sans connaître leurs positions.

```
>>> t = [1, 3, 8]
>>> for elem in t:
...     print(elem)
...
1
3
8
```

La boucle `for elem in t` est plus simple pour parcourir les valeurs d'un tableau, par exemple pour rechercher la plus petite ou la plus grande valeur dans ce tableau, mais elle ne permet pas d'accéder à sa position. Pour accéder à la position d'une valeur que l'on recherche, il faut utiliser les deux autres méthodes.

Modifier un tableau

Modifier un élément

À la différence des chaines de caractères et p-uplets, il est possible de modifier la valeur d'un élément dans un tableau :

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> animaux[2]= "lion"
>>> animaux
['girafe', 'tigre', 'lion', 'souris']
```

Opérations sur tableaux

Deux opérations sont possibles, l'addition et la multiplication :

- L'opérateur d'addition « `+` » **concatène** (assemble) deux tableaux.
- L'opérateur de multiplication « `*` » entre un nombre entier et une tableau **duplique** (répète) plusieurs fois les éléments dans un nouveau tableau.

```
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]
>>> 3 * [1, 2]
[1, 2, 1, 2, 1, 2]
```

Ajouter de nouveaux éléments

Il existe plusieurs méthodes pour ajouter des éléments à un tableau `t` :

- `t.append(x)` ajoute un élément `x` à la fin d'un tableau `t`.

  ```
  >>> t = [1, 2, 3]
  >>> t.append(4)
  >>> t
  [1, 2, 3, 4]
  ```

- `t.insert(i, x)` insère un élément `x` à la position donnée par l'indice `i`. `i` est la position de l'élément courant avant lequel l'insertion doit s'effectuer.

  ```
  >>> t = ['a', 'b', 'd']
  >>> t.insert(2, 'c')
  >>> t
  ['a', 'b', 'c', 'd']
  ```

- `t.extend(autretableau)` étend un tableau `t` en lui ajoutant tous les éléments de `autretableau`.

  ```
  >>> t = [1, 2, 3]
  >>> t.extend([4, 5, 6])
  >>> t
  [1, 2, 3, 4, 5, 6]
  ```

⚠️ Ne pas confondre `append` (ajouter un élément) et `extend` (étendre un tableau). Si on utilise `append` avec un tableau on obtient un tableau de tableaux !

```
>>> t.append([4, 5, 6])
>>> t
[1, 2, 3, [4, 5, 6]]
```

Supprimer des éléments

Il existe plusieurs méthodes pour supprimer des éléments à un tableau `t` :

- `t.remove(x)` supprime le premier élément dont la valeur est égale à `x`. Si le tableau contient plusieurs fois la valeur `x`, seule la première occurrence trouvée est supprimée :

  ```
  >>> t = [12, 13, 14, 15]
  >>> t.remove(13)
  >>> t
  [12, 14, 15]
  ```

- `t.pop(i)` supprime l'élément situé à la position `i` et le renvoie en valeur de retour. Si aucune position n'est spécifiée, `t.pop()` supprime et renvoie le dernier élément du tableau :

  ```
  >>> t = ['a', 'b', 'c', 'd', 'e']
  >>> t.pop()
  'e'
  >>> t
  ['a', 'b', 'c', 'd']
  ```

- L'instruction `del`[10] permet aussi de supprimer un élément du tableau :

  ```
  >>> t = [1, 2, 3, 4, 5]
  >>> del t[3]
  >>> t
  [1, 2, 3, 5]
  ```

  Ou encore le tableau entier avec l'instruction `del t`, alors la variable `t` n'existe plus.

  ```
  >>> del t
  >>> t
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    NameError: name ' tu ' is not defined
  ```

D'autres méthodes bien utiles

Deux méthodes pour trier un tableau :

- `t.sort()` ordonne les éléments dans le tableau.

  ```
  >>> t = [5, 8, 2, 1]
  >>> t.sort()
  ```

```
>>> t
[1, 2, 5, 8]
```

- `t.reverse()` inverse l'ordre des éléments du tableau.

```
>>> t.reverse()
>>> t
[8, 5, 2, 1]
```

Pour trouver un élément dans un tableau :

- `t.index(x)` renvoie la position du premier élément du tableau dont la valeur égale `x`.

```
>>> t = [5, 0, 3, 2, 8, 6]
>>> t.index(2)
3
>>> ['a', 'c', 'd', 'e'].index('c')
1
```

- `t.count(x)` renvoie le nombre d'éléments ayant la valeur `x` dans le tableau.

```
>>> [1, 1, 2, 2, 2, 3, 4, 4, 3].count(3)
2
```

Enfin, `dir(list)` permet d'obtenir la liste exhaustive des méthodes disponibles pour les tableaux.

> Les méthodes telles que `insert()`, `remove()` ou `sort()`, qui ne font que modifier le tableau, ne renvoient pas de valeur (ou plutôt elles renvoient `None`).

Conversion de type (cast)

Comme la fonction `tuple()`, la fonction `list()` prend en argument un objet séquentiel (une chaine de caractère par exemple) et renvoie le tableau correspondant :

```
>>> list('abc')
['a', 'b', 'c']
>>> list((1, 2, 3))
[1, 2, 3]
```

On peut créer un tableau vide avec la fonction list() sans argument.

```
t_vide = list()
```

Nous avons déjà vu l'utilisation de la fonction `range()`. Lorsqu'elle est utilisée en combinaison avec la fonction list(), on obtient une tableau d'entiers. Par exemple :

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

La méthode `.split()` permet de découper une chaîne en tableau avec un séparateur : `chaine.split(separateur)`.

```
>>> chaine = 'Bonjour tout le monde !'
>>> chaine.split(' ')
['Bonjour', 'tout', 'le', 'monde', '!']
```

Réciproquement la méthode `.join()` permet de convertir un tableau en chaîne de caractères en insérant le paramètre entre les éléments du tableau: `elementSeparateur.join(t)```.

```
>>> t = ['Bonjour', 'tout', 'le', 'monde', '!']
>>> ' '.join(t)
'Bonjour tout le monde !'
```

Création par compréhension

Il est possible de créer un tableau par compréhension en utilisant l'une de ces syntaxes :

- avec une expression sur les valeurs dans un `ancien_tableau` ou d'une fonction `range` :

  ```
  nouveau_tableau = [expression(i) for i in ancien_tableau]
  nouveau_tableau = [expression(i) for i in range(...)]
  ```

  Exemple :

  ```
  >>> carres = []
  >>> for x in range(10):
  ...     carres.append(x**2)
  >>> carres
  [0, 1, 4, 9, 16, 25, 36, 49, 64, 81
  ```

  est equivalent à :

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

  ou avec un autre tableau :

  ```
  >>> mult_3 = [3 * i for i in [1, 2, 3]]
  >>> mult_3
  [3, 6, 9]
  >>> mult_ = [3 * i for i in [1, 2, 3]]
  >>> mult_3
  [3, 6, 9]
  ```

- avec une fonction : `nouveau_tableau = [f(x) for x in ... ]`

  Exemple :

  ```
  >>> def f(x):
  ...     return 2*x + 3
  >>> absisses = [1, 2, 3, 5, 10]
  ```

```
>>> ordonnees = [f(x) for x in absisses]
>>> ordonnees
[5, 7, 9, 13, 23]
```

- avec une condition `if` : `nouveau_tableau = [expression(i) for i in ... if condition]`

  Exemple :

```
>>> carre_mult_3 = [x**2 for x in range(10) if x%3 == 0]
>>> carre_mult_3
[0, 9, 36, 81]
```

- avec une condition `if...else` : `nouveau_tableau = [expression(i) if condition else autre_expression(i) for i in ...]` (attention l'ordre est différent)

  Exemple :

```
>>> carre_mult_3_or_0 = [x**2 if x%3 == 0 else 0 for x in range(10)]
>>> carre_mult_3_or_0
[0, 0, 0, 9, 0, 0, 36, 0, 0, 81]
```

- avec plusieurs paramètres : `nouveau_tableau = [expression(i, j) for i in ... for j in ... if condition]`

  Exemples:

```
>>> [x + y for x in [10, 30, 50] for y in [20, 40, 60]]
[30, 50, 70, 50, 70, 90, 70, 90, 110]
```

Tableaux de tableaux

Pour finir, il est tout à fait possible de construire des tableaux de p-uplets ou des tableaux de tableaux. Cette fonctionnalité peut parfois être très pratique.

Création d'un tableau de tableaux

On peut créer un tableau qui contient des tableaux :

```
>>> t0 = [0, 0, 0]
>>> t1 = [1, 1, 1]
>>> t2 = [2, 2, 2]
>>> t = [t0, t1, t2]
>>> t
[[0, 0, 0], [1, 1, 1], [2, 2, 2]]
```

Il était possible d'écrire directement :

```
>>> t = [[0, 0, 0], [1, 1, 1], [2, 2, 2]]
>>> t
[[0, 0, 0], [1, 1, 1], [2, 2, 2]]
```

Il est aussi possible de construire le tableau de tableaux ligne par ligne.

```
>>> t = []
>>> for i in range(3):
...     t_i = [i for j in range (3)]
...     t.append(t_i)
...
>>> t
[[0, 0, 0], [1, 1, 1], [2, 2, 2]]
```

Ou par compréhension.

```
>>> t = [[i for j in range (3)] for i in range(3)]
>>> t
[[0, 0, 0], [1, 1, 1], [2, 2, 2]]
```

par exemple, cette compréhension de tableaux combine les éléments de deux tableaux s'ils ne sont pas égaux :

```
>>> table = [[x, y] for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
>>> table
[[1, 3], [1, 4], [2, 3], [2, 1], [2, 4], [3, 1], [3, 4]]
```

Exercice corrigé

Construire le tableau de tableaux suivant par compréhension :

```
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
 [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
 [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
 [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
 [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
 [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
 [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
 [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
 [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
 [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]]
```

Réponse

Accès aux éléments

Pour accéder à un élément du tableau de tableaux, on utilise l'indiçage habituel :

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

Pour accéder à un élément d'un sous-tableau, on utilise un double indiçage :

```
>>> t[1][2]
6
```

```
>>> t[2][1]
8
```

Dans le cas d'un tableau de tableaux avec des sous-tableaux de même taille, on parle parfois de matrice.

On dit que cette matrice a 2 dimensions et est de taille 3 x 3. Les éléments sont donc identifiés par `t[no de ligne][no de colonne]`.

Exercice corrigé

Écrire une fonction `lucas(a, b, c)` prenant en paramètres 3 entiers relatifs `a`, `b` et `c`, vérifie par asserts que `0 < a < b < c - a` et `b ≠ 2a` puis renvoie le carré magique 3x3 en utilisant la méthode d'Édouard Lucas sous forme d'un tableau de tableaux :

c + a	c – a – b	c + b
c – a + b	c	c + a – b
c – b	c + a + b	c – a

« En mathématiques, un carré magique d'ordre n est composé de n^2 entiers strictement positifs, écrits sous la forme d'un tableau carré. Ces nombres sont disposés de sorte que leurs sommes sur chaque rangée, sur chaque colonne et sur chaque diagonale principale soient égales. » source : https://fr.wikipedia.org/wiki/Carré_magique_(mathématiques)

Réponse

Exercice corrigé

Écrire une fonction qui vérifie qu'un carré est magique (ou pas). « En mathématiques, un carré magique d'ordre n est composé de n^2 entiers strictement positifs, écrits sous la forme d'un tableau carré. Ces nombres sont disposés de sorte que leurs sommes sur chaque rangée, sur chaque colonne et sur chaque diagonale principale soient égales. » source : https://fr.wikipedia.org/wiki/Carré_magique_(mathématiques)

Réponse

Tableaux muables

Les tableaux sont muables c'est-à-dire qu'on peut modifier chaque élément d'un tableau individuellement, supprimer ou ajouter des éléments. Mais on peut aussi modifier les variables de type `int`, `float`, `tuple`, `str` ou `bool` pourtant dit « immuables », alors qu'elle est la différence ?

Observons la différence quand on modifie la valeur d'une variable en utilisant la fonction `id()` qui renvoie l'identifiant de la variable en mémoire.

avec une variable de type "immuable"

```
>>> a = 1
>>> id(a)
2366593132848
>>> a = a + 1
>>> id(a)
2366593132880
```

Une nouvelle variable `a` est créée en mémoire quand on change sa valeur.

avec une variable de type "muable"

```
>>> t = [1]
>>> id(t)
2366637916800
>>> t.append(2)
>>> id(t)
2366637916800
```

C'est la même variable `t` qui reste en mémoire avec une valeur différente.

Copie de tableau

Comparons ce qu'il se passe quand on copie une variable immuable, par exemple de type `int`, et une variable de type `list` muable.

avec une variable de type "immuable"

```
>>> a = 1
>>> b = a
```

Modifions `b`.

```
>>> b = 2
>>> b
2
```

Qu'est-il arrivé à `a` ?

```
>>> a
1
```

`a` n'a pas changé.

avec une variable de type "muable"

```
>>> t = [1, 2, 3]
>>> u = t
```

Modifions `u`.

```
>>> u[2] = 4
>>> u
[1, 2, 4]
```

Qu'est-il arrivé à `t` ?

```
>>> t
[1, 2, 4]
```

`t` a aussi été modifié quand on a modifié `u` !

Les deux variables `t` et `u` ne sont pas deux objets différents mais **deux noms qui font référence au même objet en mémoire**. Pour s'en convaincre on peut vérifier les adresses des variables

avec une variable de type "immuable"

```
>>> id(a)
2366593132848
>>> id(b)
2366593132880
```

`a` et `b` sont bien deux variables différentes.

avec une variable de type "muable"

```
>>> id(t)
2366638078720
>>> id(u)
2366638078720
```

`t` et `u` sont deux noms pour la même variable.

Pour copier un tableau , il faut créer une **copie explicite** du tableau initial. Cela peut se faire de plusieurs manières[3] :

- Avec `t[:]` qui créé une copie *des données* du tableau `t` (en opposition à une copie du tableau `t`) :

  ```
  >>> t = [1, 2, 3]
  >>> u = t[:]
  ```

- Avec la fonction `list(t)` qui renvoie un tableau formé des éléments de la variable `t` :

  ```
  >>> u = list(t)
  ```

- Ou encore utiliser la méthode `.copy()` :

  ```
  >>> u = t.copy()
  ```

Tableau passé en paramètre de fonction

Passer des arguments à une fonction d'un type muable comme `list` [4] génère des effets de bord[5] .

Nous avons vu précédemment qu'une fonction ne modifie pas la valeur d'une variable passée en paramètre en dehors de son exécution, les paramètres sont passés par valeur. C'est en effet le cas avec des variables de type immuable mais ce n'est pas le cas pour les variables de type muable comme le type `lis`t. Dans ce cas, la fonction reçoit l'adresse en mémoire de la variable passée en argument et peut donc en modifier le contenu.

Illustrons cela des fonctions `f(x)` et `g(x)` qui modifient simplement la valeur d'un paramètre `x`.

avec une variable de type "immuable"

```
def f(x):
    x = 2
```

avec une variable de type "muable"

```
def g(x):
    x.append(2)
```

Appelons ces fonctions en passant des variables `a` et `t` en paramètre [6]:

avec une variable de type "immuable"

```
>>> a = 1
>>> f(a)
>>> a
1
```

La valeur de `a` n'a pas été modifiée par la fonction `f`.

avec une variable de type "muable"

```
>>> t = [1]
>>> g(t)
>>> t
[1 ,2]
```

La valeur de `t` a été modifiée par la fonction `g` !

Autres effets

On peut initialiser un tableau avec une valeur par défaut pour tous les éléments, par exemple des zéros, avec

```
>>> t = [0] * 3
>>> t
[0, 0, 0]
```

Mais attention à ne pas utiliser cette méthode pour des tableaux de tableaux :

```
>>> t = [[0] * 3] * 3
>>> t
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> t[0][0] = 1
>>> t
[[1, 0, 0], [1, 0, 0], [1, 0, 0]]
```

On préfèrera donc : `t = [0 for i in range(3)]` et `t = [[0 for i in range(3)] for i in range(3)]`

De la même façon, on ne doit pas définir une valeur par défaut de paramètre de fonction avec un tableau de type `list`[7], par exemple dans la fonction suivante :

```
def ajouter(a, L = []):
    L.append(a)
    return L
```

La valeur par défaut n'est évaluée qu'une seule fois puis la fonction accumule les arguments au fil des appels :

```
>>> M = ajouter(1)
>>> M
[1]
>>> N = ajouter(2)
>>> N
[1, 2]
```

Pour finir, il revient au même de faire par exemple `n += 1` et `n = n + 1` pour des entiers mais par pour des tableaux. Comparons :

```
def f(L, n) :
    for k in range(n) :
        L += [k]  # equivalent à L.append(), modifie la valeur de l'argument L

def g(L, n) :
    for k in range(n) :
        L = L + [k]   # crée une nouvelle variable 'locale' L et ne modifie par l'argument L
```

---

1. Le type `list` de Python offre plus de possibilité qu'un tableau et notamment peut contenir des éléments de types différents (y compris d'autres tableaux). ↩
2. Il est aussi possible de préciser un `pas` sur le modèle `t[début:fin:pas]`. ↩
3. ⚠️Aucune de ces approches ne fonctionne pour un tableau de tableaux, il faut par exemple utiliser la méthode `.deepcopy()` du module `copy`. ↩
4. Ou de type `dict` ou `set` qui sont aussi des types muables. ↩
5. Un effet de bord se produit quand une fonction modifie le contenu d'une variable qui appartient au contexte appelant. ↩
6. Les variables `a` et `t` pourraient s'appeler aussi `x` ce qui donnerait le même résultat. ↩
7. Voir https://docs.python.org/fr/3/tutorial/controlflow.html#default-argument-values. ↩
8. ou *mutable* en anglais. ↩
9. Par abus de langage on francise parfois le terme en « liste » pour désigner un tableau. ↩
10. `del` est une instruction Python, pas une fonction, elle s'écrit donc sans parenthèse. ↩