# TP 2 : Types séquentiels (tableaux)

On complète dans ce TP les bases de la programmation avec Python en présentant les trois types séquentiels de données : `liste`, `tuple` et `str`.

## I. Les listes en Python

### I.1. Syntaxe

#### I.1.a) Écriture en extension

- Pour définir une liste, on encadre ses éléments par des crochets.

```
premiers = [2, 3, 5, 7]
print(type(premiers))
```

L'ordre des coefficients est important.

```
In [1] :    premiers_bis = [2, 3, 7, 5]
In [2] :    premiers == premiers_bis
Out [2] :   False
```

- La liste vide est celle qui n'a aucun élément.

```
vide = []
```

- Les éléments qui constituent une liste peuvent être de différents types.

```
differents = [3, 3.14, 'pi', 3==4]
```

#### I.1.b) Écriture en compréhension

**Définition d'ensemble en mathématiques**

- On a vu précédemment qu'on pouvait créer des listes en extension. Au lieu d'opter pour cette stratégie consistant à écrire un par un tous les éléments d'une liste, on peut construire cette liste en spécifiant les propriétés qui définissent ses valeurs. Cette différence entre l'écriture en compréhension et en extension est très classi que mathématiques lorsque l'on étudie des ensembles. Typiquement, l'ensemble des carrés d'entiers compris entre 1 et 5 peut s'écrire à l'aide des deux manières suivantes :
  - × **en extension** : $\{1, 4, 9, 16, 25\}$.
  - × **en compréhension** : $\{k^2 \mid k \in [\![1, 5]\!]\}$.

  En mathématiques, il est fréquent d'utiliser plutôt la deuxième manière de procéder. Elle est notamment beaucoup plus rigoureuse lorsque l'on s'intéresse à des ensembles contenant un nombre infini de valeurs. Typiquement, si on s'intéresse à l'ensemble des carrés d'entiers positifs, on écrira soit :
  - × $\{0^2, 1^2, 2^2, 3^2, 4^2, \ldots\}$,
  - × $\{k^2 \mid k \in \mathbb{N}\}$.

  On se sert ici de l'utilisation des points de suspension pour décrire tous les éléments de la liste en extension. Cette notation n'est pas très rigoureuse et on compte sur l'habitude du lecteur pour comprendre ce qui est signifié.
- En informatique, on ne s'intéresse pas à des listes de tailles infinies. Il n'en demeure pas moins qu'il pourrait être intéressant d'écrire des listes en compréhension. C'est l'une des fonctionnalités fantastiques (ne mâchons pas nos mots) offerte par **Python**.

## Les listes en compréhension

- La syntaxe pour créer une liste en compréhension (ou parle aussi de compréhension de liste) est assez proche de celle utilisée en mathématique. Il s'agit de spécifier la propriété que doivent vérifier les éléments que l'on souhaite faire apparaître dans la liste. Typiquement, on peut demander de créer la liste contenant toutes les valeurs entières comprises entre `0` et `10` (exclu) ou encore la liste des cinq premiers carrés d'entiers. Pour ce faire, on procède généralement en demandant l'itération (à l'aide d'un `for`) des valeurs d'un intervalle (défini par un `range`).

```
Out [3] : [k for k in range(10)]
In  [3] : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Out [4] : [k**2 for k in range(1, 6)]
In  [4] : [1, 4, 9, 16, 25]
```

- Il est à noter que ce procédé peut aussi permettre de construire des listes contenant toujours la même valeur.

```
Out [5] : [0 for k in range(10)]
In  [5] : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

  On constate ici que le `k` ne joue pas de valeur particulière pour la détermination de la valeur `0`. On peut donc s'en passer.

```
Out [6] : [0 for _ in range(10)]
In  [6] : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- Il est possible d'enchaîner les itérations. Cela permet par exemple d'obtenir tous les triplets d'éléments de $\{0,1\}$.

```
Out [7] : [(i, j, k) for i in range(2)
           for j in range(2) for k in range(2)]

In  [7] : [(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1),
           (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
```

- Il est aussi possible d'ajouter des conditions à l'aide de `if`. Il est par exemple possible de créer la liste de tous les carrés multiples de 6 d'entiers plus petits que 20.

```
Out [8] : [k**2 for k in range(20) if k**2 % 6 == 0]
In  [8] : [0, 36, 144, 324]
```

  Il est aussi possible d'obtenir tous les triplets d'éléments différents de $\{1,2,3\}$ (autrement dit toutes les permutations de l'ensemble $\{1,2,3\}$).

```
Out [9] : [(i, j, k) for i in range(1, 3)
           for j in range(1, 3) for k in range(1, 3)
           if i != j and j != k and k !=i]

In  [9] : [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1),
           (3, 1, 2), (3, 2, 1)]
```

### Exercice 1

1. Construire la liste `L` des entiers de 1 à 500.
2. Importer le module `random` avec la commande `import random as rd`.
   Construire une liste `A` de 1000 nombres aléatoires à l'aide de la fonction `rd.random`.

## I.2. Opérations sur les listes

### I.2.a) Premières opérations

- la fonction `len` (de l'anglais *length*) permet de calculer leur longueur,

```
In [10] : len([2, 'Hello', 7.])
Out[10] : 3
```

- on accède aux éléments individuels avec la syntaxe `[k]` où `k` est un indice positif valide,

```
In [11] : [2, 'Hello', 7.][0]
Out[11] : 2
```

**Python** numérote les éléments d'un tableau à partir de 0.

On peut également accéder aux éléments d'un tableau en partant de la fin.

```
In [12] : [2, 'Hello', 7.][-1]
Out[12] : 7.
```

**Exercice 2**

1. Écrire un script calculant le maximum des éléments (nombres réels) d'une liste `A`.
2. Modifier la fonction précédente pour qu'elle affiche également la première position de ce maximum dans la liste.

- on peut en calculer une tranche avec la syntaxe `[i:j]` (*slicing*) qui extrait tous les éléments dont les indices sont compris entre `i` et `j` exclus,

```
In [13] : premiers[1:2]
Out[13] : [3, 5]
```

- ces structures peuvent être concaténées avec l'opérateur `+` et répétées avec l'opérateur `*`.

```
In [14] : [2, 4] + [0] * 3
Out[14] : [2, 4, 0, 0, 0]
```

### I.2.b) Autres opérations

- On peut ajouter un élément à une liste à l'aide de la méthode `.append`.

```
In [15] : premiers.append(11)
In [16] : print(premiers)
Out[16] : [2, 3, 5, 7, 11]
```

**Exercice 3**

Écrire un script qui retourne la liste des 100 premiers termes de la suite de Fibonacci.
*On rappelle que cette suite $(u_n)_{n\in\mathbb{N}}$ est définie de la façon suivante :*

$$\begin{cases} u_0 = 1 \\ u_1 = 1 \\ \forall n \in \mathbb{N},\ u_{n+2} = u_n + u_{n+1} \end{cases}$$

- On peut supprimer un élément d'une liste en utilisant la commande `del`.

```
In [17] : premieres = [2, 3, 5, 42, 7, 11, 13, 17]
In [18] : del premiers[3]
In [19] : print(premiers)
Out[19] : [2, 3, 5, 7, 11, 13, 17]
```

- Pour supprimer le dernier élément d'une liste `L`, on utilise la méthode `.pop` qui supprime et retourne la valeur du dernier élément de la liste.

```
In [20] : a = premiers.pop()
In [21] : print(premiers)
Out[21] : [2, 3, 5, 7, 11, 13]
In [22] : print(a)
Out[22] : 17
```

- On peut sommer les éléments d'une liste à l'aide de la fonction `sum`.

```
In [23] : sum(premiers)
Out[23] : 28
```

**Exercice 4**

*1.* Écrire un script **Python** calculant la moyenne d'une liste `L` :

*a)* sans la fonction `sum`.

*b)* avec la fonction `sum`.

*2.* Écrire un script **Python** calculant la variance d'une liste `L`.

## I.3. Listes et gestions de la mémoire

### I.3.a) Alias d'une liste

- Pour comprendre les différentes notions de copies d'une liste, on s'intéresse tout d'abord au programme suivant.

```
1  a = [1, 2]
2  b = a
3  a[0] = 5
4  print(a)
5  print(b)
```

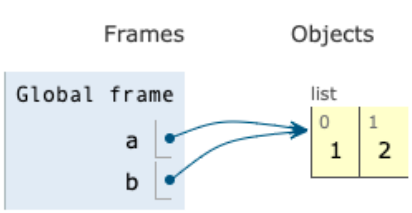
Ce programme permet d'afficher le contenu des listes `a` et `b`. Notons que :

× la liste `a` est initialisée à `[1, 2]`. Son premier terme est ensuite mis à jour en ligne 3. Ainsi, en ligne 4, on s'attend à l'affichage : `[5, 2]`.
C'est bien le cas.

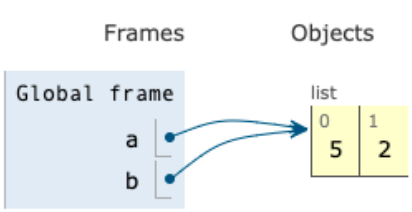
× la liste `b` est initialisée en ligne 2. On lui affecte la valeur de `a`, à savoir `[1, 2]`. Elle n'est pas mise à jour ensuite. Ainsi, en ligne 5, on s'attend à l'affichage : `[1, 2]`.
Ce n'est pas le cas ! En réalité, c'est `[5, 2]` qui est encore affiché.

Que s'est-il passé ? Pour répondre à cette question, il faut s'interroger sur la gestion de la mémoire.

- Pour visualiser plus précisément ce que réalise le code précédent, on va utiliser **Python Tutor**. Il suffit alors de recopier le code est de cliquer sur *Visualize Execution*. Après exécution de la ligne 3 du code, on obtient le graphique variables / objets suivant :




On s'aperçoit alors que `a` et `b` sont 2 noms différents mais qu'ils « pointent » vers le même objet. Cela permet de comprendre le fonctionnement du code précédent. Après exécution de la ligne 5, voici ce qu'on obtient :



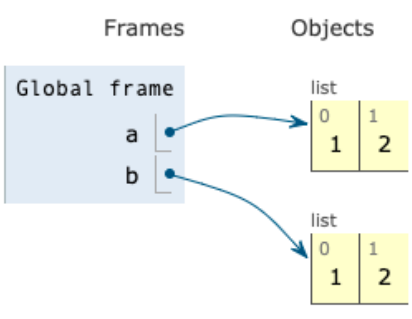

On dit alors que `b` est un **alias** de la liste `a`.

### I.3.b) Copie superficielle d'une liste

- Dans l'exemple précédent, on a vu que créer un alias ne correspond pas à créer une nouvelle liste mais simplement créer un nouveau nom qui pointe vers le même objet liste. Si on veut créer une nouvelle liste, on peut procéder comme suit.

```
1  a = [1, 2]
2  b = a[:]
3  a[0] = 5
4  print(a)
5  print(b)
```

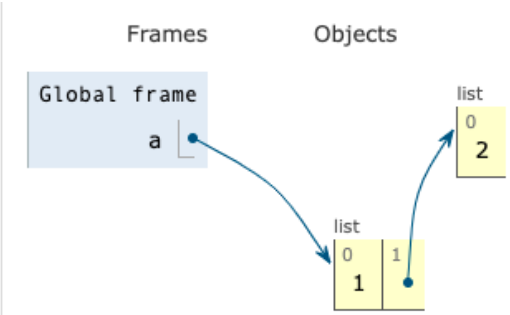
- En ligne 2, on crée une variable `b`. Celle-ci est affectée à la tranche `a[:]` qui correspond à tout le contenu de la liste `a`. On l'a vu au moment du **slicing** : les tranches (= slices) créées sont de nouveaux objets. On a donc apparemment bien réussi à faire ce que l'on souhaitait, à savoir créer une copie de la liste initiale qui est un objet différent de la liste initiale.
Après exécution de la ligne 2, voici le schéma variables / objets :



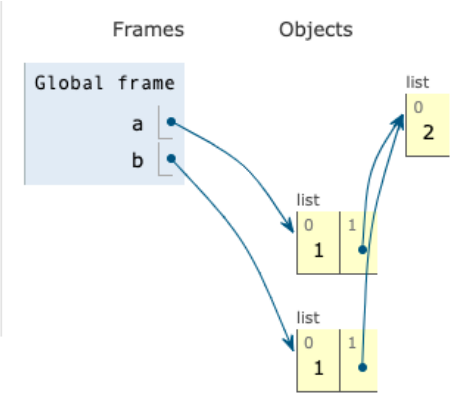
- Dans le point précédent on a créé une **copie superficielle** de la liste `a` (on parle de **shallow copy** en anglais). Le qualificatif **superficielle** peut sembler étonnant. Pour mieux le comprendre, considérons le programme suivant.

```
1  a = [1, [2]]
2  b = a[:]
3  a[1][0] = 5
4  print(a)
5  print(b)
```

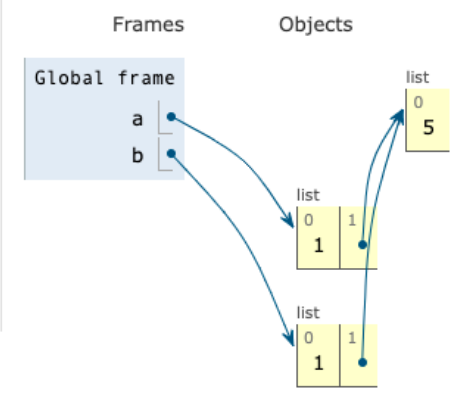
Après exécution de la ligne 1, on a le schéma variables / objets suivant :




Après exécution de la ligne 2, ce schéma est mis à jour comme suit.




On comprend mieux le qualificatif « superficielle ». Une nouvelle liste est bien créée. Les objets simples (entiers et flottants) sont bien copiés dans cette nouvelle liste. Le problème survient avec les objets composés. On ne crée pas un nouvel objet pour la liste `[2]` avec une référence qui pointerait vers ce nouvel objet. On crée au contraire, dans la liste `b`, une référence vers l'objet initial `[2]`. De sorte qu'en sortie de ligne 3, on obtient le schéma suivant :



- On peut également obtenir une copie superficielle d'une liste à l'aide de la méthode `.copy`.

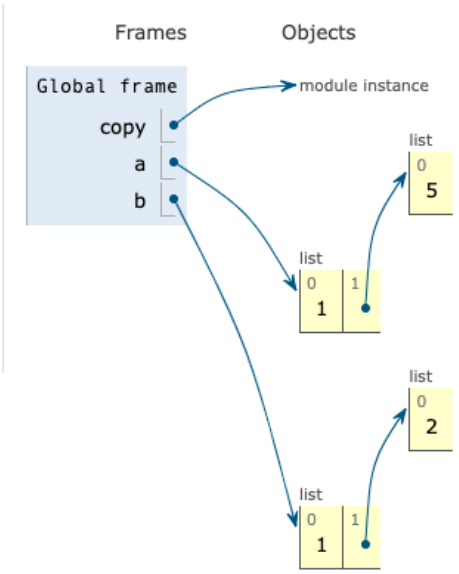
```
1  a = [1, 2]
2  b = a.copy()
3  a[0] = 5
4  print(a)
5  print(b)
```

### I.3.c) Copie profonde d'une liste

- On a vu dans le paragraphe précédent que la notion de copie superficielle crée une copie qui « s'arrête au premier niveau ». Si l'on souhaite que la liste copiée ne partage aucune référence avec la liste initiale, il faudrait que les objets composés de `a` donnent naissance à de nouveaux objets lors de la copie. Attention, car ces objets composés (il s'agissait de `[2]` dans l'exemple précédent) peuvent eux-mêmes contenir des objets composés qui peuvent eux-mêmes contenir des objets composés et ainsi de suite (par exemple `[2, [1, [3, 5]]]`). Si l'on veut empêcher qu'il y ait des références partagées par `a` et `b`, il faut que la copie se fasse à tous les niveaux de profondeur. C'est pourquoi l'on parle de `copie profonde` (on parle de **deepcopy** en anglais). Si l'on effectue une copie profonde de `a`, un nouvel objet est créé. On insère alors récursivement dans ce nouvel objet des copies des objets trouvés dans la liste original. C'est pourquoi on parle aussi de **copie récursive**.
- Afin de réaliser une copie profonde, on utilise généralement la bibliothèque `copy` de **Python**.

```
1  import copy
2  a = [1, [2]]
3  b = copy.deepcopy(a)
4  a[1][0] = 5
5  print(a)
6  print(b)
```

Voici le schéma après exécution de la ligne 3.



# II. Les chaînes de caractères

## II.1. Syntaxe

- Les chaînes de caractères (le type `str`) sont des suites de caractères alphanumériques délimités par des guillemets simples ou doubles.

```
1 L = 'La horde du contrevent'
2 print(type(L))
```

- La chaîne vide est celle qui ne contient aucun caractère.

```
1 vide = ''
```

## II.2. Opérations sur les chaînes de caractères

### II.2.a) Premières opérations

- la fonction `len` (de l'anglais *length*) permet de calculer leur longueur,

```
In [24] : len('La horde du contrevent')
Out[24] : 22
```

- on accède aux éléments individuels avec la syntaxe `[k]` où `k` est un indice positif valide,

```
In [25] : 'La horde du contrevent'[4]
Out[25] : 'o'
```

On peut également accéder aux éléments d'un tableau en partant de la fin.

```
In [26] : 'La horde du contrevent'[-4]
Out[26] : 'v'
```

**Exercice 5**
Écrire une fonction `troncature(chaîne: str, k: int)` qui renvoie une nouvelle chaîne de caractères obtenue en retirant les `k` derniers caractères de la chaîne `chaîne`.

**Exercice 6**
Un palindrome est un mot qui se lit de la même manière dans les deux sens. Écrire et tester une fonction `est_palindrome(mot: str)` qui teste si la chaîne de caractères `mot` est un palindrome.

- on peut en calculer une tranche avec la syntaxe `[i:j]` (*slicing*) qui extrait tous les éléments dont les indices sont compris entre `i` et `j` exclus,

```
In [27] : 'La horde du contrevent'[3:8]
Out[27] : 'horde'
```

- ces structures peuvent être concaténées avec l'opérateur `+` et répétées avec l'opérateur `*`.

```
In [28] : 'Alain' + ' Damasio'
Out[28] : 'Alain Damasio'
```

**Exercice 7**
Écrire une fonction `bonjour(nom: str)` qui renvoie la chaîne de caractères `'Bonjour *** ¡` où les `***` sont remplacées par le `nom` placé en argument.
*(on ne demande pas d'afficher cette chaîne)*

**Exercice 8**

1. Écrire une fonction `affiche_avec_cadre(message: str)` qui affiche dans le shell le `message` placé en argument, encadré par les caractères `'*'` comme dans l'exemple ci-dessous :

```
**********************************************
* Le nombre d'étoiles dépend de la longueur ! *
**********************************************
```

2. À l'aide des fonctions des exercices 7 et 8, afficher votre message de salutation dans un cadre.

**Commentaire**

Notons que ces premières opérations sont exactement les mêmes que celles définies pour les listes. Cela est cohérent car ces deux structures sont dites de **type séquentiel** (on parle aussi de structure de données **indicées**). Autrement dit, ce sont des suites d'éléments.
Les types séquentiels regroupent trois types de tableaux :

- les listes (le type `list`, qui sont des structures de données **mutables** en plus d'être indicées. Cela signifie que l'on peut modifier ses éléments individuellement.
- les chaînes de caractères (le type `str`), qui sont des structures de données **immuables** en plus d'être indicées. Cela signifie qu'on ne peut modifier ses éléments individuels.
- les tuples (le type `tuple`) ou $n$-uplets qui sont des structures de données indicées et immuables.

Les opérations définies en ***I.2.a)***, ***II.2.a)*** et ***III.2.a)*** sont donc identiques. On les rappellera très brièvement en ***III.2.a)*** pour les tuples.

### II.2.b) Spécificités des chaînes de caractères

Contrairement aux listes, les chaînes de caractères sont des structures *immuables*. On ne peut donc pas modifier l'un de ses éléments individuellement. La commande d'affectation suivante renvoie donc une erreur.

```
In [29] :    L[18] = 't'
TypeError :  'str' object does not support item assignment
```

# III. Les tuples

## III.1. Syntaxe

- les tuples (le type `tuple`) ou $n$-uplets sont des suites finies de valeurs séparées par des virgules (si besoin enclos par des parenthèses)

```
1 fibo = (1, 1, 2, 3, 5, 8)
2 print(type(L))
```

- Le tuple vide est celui qui ne contient aucun élément.

```
1 vide = ()
```

- Pour définir un tuple de longueur 1, on utilise la syntaxe suivante.

```
1 T = (3,)
```

## III.2. Opérations sur les tuples

### III.2.a) Premières opérations

- la fonction `len` (de l'anglais *length*) permet de calculer leur longueur,

```
In [30] : len((2, 3, 5, 7, 11))
Out[30] : 5
```

- on accède aux éléments individuels avec la syntaxe `[k]` où `k` est un indice positif valide,

```
In [31] : (2, 3, 5, 7, 11)[3]
Out[31] : 7
```

  On peut également accéder aux éléments d'un tableau en partant de la fin.

```
In [32] : (2, 3, 5, 7, 11)[-2]
Out[32] : 7
```

- on peut en calculer une tranche avec la syntaxe `[i:j]` (*slicing*) qui extrait tous les éléments dont les indices sont compris entre `i` et `j` exclus,

```
In [33] : fibo[2:4]
Out[33] : (2, 3, 5)
```

- ces structures peuvent être concaténées avec l'opérateur `+` et répétées avec l'opérateur `*`.

```
In [34] : (1,) * 10 + (2, 3)
Out[34] : (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 3)
```

### III.2.b) Spécificités des tuples

Les tuples, comme les chaînes de caractères sont des structures *immuables*. On ne peut donc pas modifier l'un de ses éléments individuellement. La commande d'affectation suivante renvoie donc une erreur.

```
In [35] : fibo[3] = 7
TypeError : 'tuple' object does not support item assignment
```

# IV. Deux exercices plus corsés

**Exercice 9 Suite de Conway**

Les premiers termes de la suite de Conway sont 1, 11, 21, 1211, 111221… Chaque terme est obtenu en lisant à haute voix le terme précédent (c'est pourquoi Conway avait baptisé cette suite « Look and say »).

Par exemple, le terme 1211 se lit : « un 1, un 2, deux 1 ». Le terme suivant est donc 111221.

1. Afficher les 10 premiers termes de la suite de Conway. Quelle est la complexité de la fonction qui vous avez codée ?
   *On remarquera qu'il est ici plus facile de manipuler les entiers sous la forme de chaînes de caractères.*
2. Soit $n \in \mathbb{N}^*$. Il a été démontré que, si l'on note $u_n$ le nombre de chiffre du $n^{\text{ème}}$ terme de cette suite, alors le rapport $\dfrac{u_{n+1}}{u_n}$ tend vers une constante, appelée *constante de Conway*. Calculez-en une valeur approchée.
3. Une autre propriété remarquable de cette constante est qu'elle ne dépend pas de la valeur initiale (à l'exception de 22 qui produit une suite constante). Le tester expérimentalement en choisissant différentes valeurs initiales.

**Exercice 10 Permutation**

Une *permutation* est un arrangement ordonné d'objets. Par exemple, 3124 est une permutation des chiffres 1, 2, 3 et 4.

Si on classe toutes les permutations des chiffres 0, 1 et 2 dans l'ordre croissant, on obtient :

012 021 102 120 201 210

1. Écrire une fonction `facto(k: int)` calculant `k`!.
2. Quelle est la millionième permutation des chiffres 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9 lorsqu'on les classe dans l'ordre croissant ?
   *Indication : on pourra remarquer que la $n^{ème}$ permutation commence par le chiffre $k = \lfloor \frac{n}{9!} \rfloor$.*