

# Fondements informatiques I

## Cours 6: Listes

Sorina Ionica `sorina.ionica@uvsq.fr`

Sandrine Vial `sandrine.vial@uvsq.fr`

# Les listes

Un **tableau** est une structure de données contenant une série d'éléments, où la position de chaque élément compte.

## Syntaxe

On peut accéder aux éléments d'une liste par leur **indice**.

```
tableau : ["bus", "voiture", "vélo", "trotinette"]  
indice :      0           1           2           3
```

# Les listes

Un **tableau** est une structure de données contenant une série d'éléments, où la position de chaque élément compte.

## Syntaxe

On peut accéder aux éléments d'une liste par leur **indice**.

```
tableau : ["bus", "voiture", "vélo", "trotinette"]
indice :   0         1         2         3
```

En Python, on utilise le type `list` pour les représenter.

```
# Exemple: Liste des capitales européennes
capitales = ["Paris", "Berlin", "Madrid", "Athènes", "Zagreb", "Rome"]

# Exemple: Liste d'entiers
entiers = [5, 0, 17, 42, 23]
```

# Les listes

La construction de listes en Python est très *flexible*.

- Il est possible de créer des listes contenant des valeurs de **types différents**.

```
# Liste mixte  
mixte = [7, "tennis", True, 5.3]
```

Cette liste contient à la fois des valeurs de type `int`, `string`, `boolean` et `float`.

- La fonction `len()` nous permet de connaître le nombre d'éléments dans une liste.

```
liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
print("La liste contient", len(liste), "éléments.")
```

# Longueur d'une liste

- Une liste a une longueur variable. La méthode `append()` permet d'ajouter un élément à la fin d'une liste.

```
mixte = [7, "tennis", True, 5.3]
```

```
mixte.append("natation")  
print(mixte)
```

- La méthode `remove()` permet d'enlever un élément de la liste.

```
mixte.remove("tennis")  
mixte.remove(7)  
print(mixte)
```

Attention : les méthodes `append` et `remove` sont propres à l'objet (`mixte`). Chose à clarifier plus tard ...

# Accéder aux éléments d'une liste

Pour une liste de longueur  $n$  :

- L'indice est la **position** d'un élément dans la liste.
- Pour une liste de  $n$  éléments, les indices vont de 0 à  $n - 1$ .
- Les indices doivent être des nombres **entiers**.

## Un premier exemple

```
liste = ["bus", "voiture", "velo", "trotinette"]  
  
print(liste[0])  
print(liste[2])
```

# Indiçage négatif

Chaque élément d'une liste a deux indices : un indice positif  $i$  et un indice négatif  $i - n$ .

liste	:	["bus", "voiture", "velo", "trotinette"]			
indice positif	:	0	1	2	3
indice négatif	:	-4	-3	-2	-1

## Indiçage négatif

```
liste = ["bus", "voiture", "vélo", "trotinette"]  
print(liste[-1])  
print(liste[-2])  
print(liste[0],liste[-4])
```

- L'indiçage négatif revient à compter de la fin.
- **Avantage** : Accéder au dernier ou avant-dernier élément de la liste, sans connaître sa longueur (indices -1 et -2 respectivement)

# Tranches

On peut accéder à une partie (une tranche) d'une liste  $L$  de longueur  $n$  en utilisant des indices :

- $L[m:k]$  donne accès aux éléments  $L[m]$ ,  $L[m+1]$ ,  $\dots$ ,  $L[k-1]$
- $L[m:]$  donne accès aux éléments  $L[m]$ ,  $\dots$ ,  $L[n-1]$
- $L[:k]$  donne accès aux éléments  $L[0]$ ,  $L[1]$ ,  $\dots$ ,  $L[k-1]$

## Exemple

```
liste = [2, 3, 5, 7, 11, 13, 17, 19, 21, 23]
print(liste[2:6])
print(liste[-8:-4])
liste[3:]
liste[:5]
liste[:-3]
```



# Modifier ou ajouter des éléments dans une liste

**Rappel.** Les listes sont des objets **mutables**. Leurs éléments peuvent donc être modifiés.

- En écrivant `liste[i] = nouvel_element`, l'élément de la liste à l'indice `i` est remplacé par `nouvel_element`.

```
capitales = ["Paris", "Berlin", "Madrid", "Athènes", "Zagreb"]
capitales[2:5] = ["Dublin", "Budapest", "Vienne"]
print(capitales)
```

# Listes en compréhension

Imaginons, qu'on veut créer une liste contenant la racine carrée des entiers  $0, 1, \dots, 9$ .

Avec une boucle :

```
import math
l = []
for i in range(10) :
    l.append(math.sqrt(i))

print(l)
```

De manière compacte, en une seule instruction :

```
l = [math.sqrt(i) for i in range(10)]

print(l)
```

# Listes en compréhension

On peut créer des listes de façon **simple**, **concise** et **compacte**.

## Syntaxe

```
L = [expr for x in t]
```

- Dans cette syntaxe : les crochets [], les mots clés for et in sont obligatoires.
- t est souvent une liste ou une collection de type range.
- x est la variable de contrôle
- expr est une expression qui dépend en général de x et dont la valeur est placée dans la liste.

Exercice : Créer avec une seule instruction une liste qui contient la table de multiplication de 5.

# Listes en compréhension

On peut créer des listes de façon **simple**, **concise** et **compacte**.

## Syntaxe

```
L = [expr for x in t]
```

- Dans cette syntaxe : les crochets [], les mots clés for et in sont obligatoires.
- t est souvent une liste ou une collection de type range.
- x est la variable de contrôle
- expr est une expression qui dépend en général de x et dont la valeur est placée dans la liste.

Exercice : Créer avec une seule instruction une liste qui contient la table de multiplication de 5.

```
mult5 = [5*i for i in range(1,11)]
```

# Test d'appartenance

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

```
nombres = [1, 3, 5, 7, 9, 11]
print(3 in nombres)
print(2 in nombres)
print(5 in nombres and 7 in nombres)
```

En combinant avec le mot-clé `not` on peut vérifier si un élément est absent d'une liste.

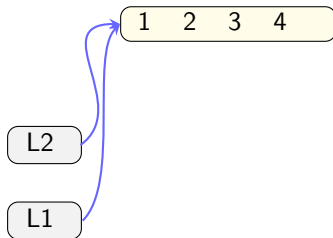
```
4 not in nombres
```

# Copie d'une liste

Qu'affiche-t-on à l'exécution ?

```
L1 = [1, 2, 3, 4]
L2 = L1
print(L2)

L2.append(5)
print(L2)
print(L1)
```



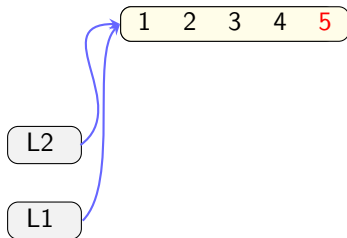
- Il ne faut jamais utiliser l'opérateur = pour copier une liste.
- Les deux objets L1 et L2 partagent la **même zone mémoire**. Une modification de l'un entraîne une modification de l'autre.

# Copie d'une liste

Qu'affiche-t-on à l'exécution ?

```
L1 = [1, 2, 3, 4]
L2 = L1
print(L2)

L2.append(5)
print(L2)
print(L1)
```

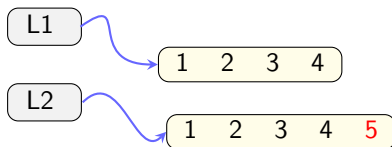


- Il ne faut jamais utiliser l'opérateur = pour copier une liste.
- Les deux objets L1 et L2 partagent la **même zone mémoire**. Une modification de l'un entraîne une modification de l'autre.

# Copie d'une liste

```
L1 = [1, 2, 3, 4]  
L2 = L1[:]  
print(L2)
```

```
L2.append(5)  
print(L2)  
print(L1)
```



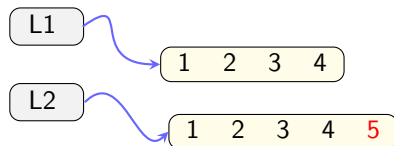


# Copie d'une liste

On peut utiliser la fonction `list()` qui permet de créer une nouvelle liste.

```
L1 = [1, 2, 3, 4]
L2 = list(L1)
print(L2)

L2.append(5)
print(L2)
print(L1)
```



# Concatener deux listes

Qu'affiche-t-on à l'exécution ?

## Exemple (rappel)

```
capitales1 = ["Paris", "Berlin", "Madrid", "Athènes", "Zagreb", "Rome"]  
capitales2 = ["Bruxelles", "Oslo"]  
  
capitale1=capitales1 + capitales2  
print(capitales1)
```

Comment faire pour créer une nouvelle liste contenant la concatenation ?

# Listes imbriquées

Une liste où chaque élément est à nouveau une liste s'appelle **liste imbriquée**, **double liste** ou **liste bidimensionnelle**.

```
L = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]  
print(L[0])  
print(L[1])  
print(L[2][1])
```

## Listes imbriquées

On veut calculer la somme de tous les éléments de la liste L.

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

```
somme = 0
```

```
for i in range(len(L)) :  
    for j in range(len(L[i])) :  
        somme += L[i][j]
```

```
print("La somme de tous les éléments de la liste vaut :", somme)
```

## Listes imbriquées

On veut calculer la somme de tous les éléments de la liste L.

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

```
somme = 0
for i in range(len(L)) :
    for j in range(len(L[i])) :
        somme += L[i][j]

print("La somme de tous les éléments de la liste vaut :", somme)
```

ou encore

```
somme = 0
for i in L :
    for j in i :
        somme += j

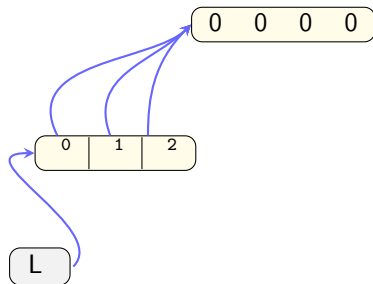
print("La somme de tous les éléments de la liste vaut :", somme)
```

# Création de listes imbriquées

```
m = 3  
n = 4  
L = [[0] * n] * m  
print(L)
```

m listes sont créées mais qui font toutes référence à la même liste !

```
L[0][1] = 1  
print(L)
```

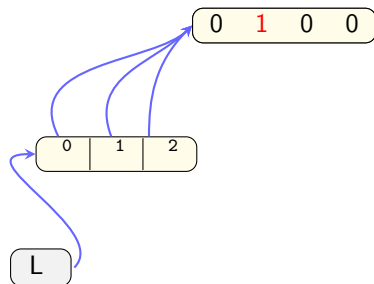


# Création de listes imbriquées

```
m = 3  
n = 4  
L = [[0] * n] * m  
print(L)
```

m listes sont créées mais qui font toutes référence à la même liste !

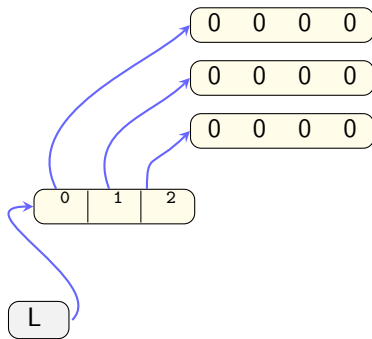
```
L[0][1] = 1  
print(L)
```



# Création de listes imbriquées

Plusieurs façons correctes de faire. La plus compacte est sûrement la suivante :

```
m = 3  
n = 4  
L = [[0] * n for i in range(m)]  
print(L)
```



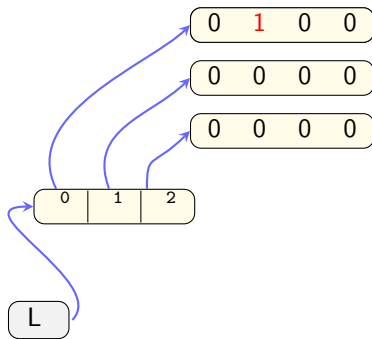
```
L[0][1] = 1  
print(L)
```



# Création de listes imbriquées

Plusieurs façons correctes de faire. La plus compacte est sûrement la suivante :

```
m = 3  
n = 4  
L = [[0] * n for i in range(m)]  
print(L)
```



```
L[0][1] = 1  
print(L)
```

# Création de listes imbriquées

**Exercice** : Créer une liste en deux dimensions, contenant la matrice suivante :

$$\begin{bmatrix} 1 & 2 & 2 & 2 \\ 0 & 1 & 2 & 2 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Création de listes imbriquées

**Exercice** : Créer une liste en deux dimensions, contenant la matrice suivante :

$$\begin{bmatrix} 1 & 2 & 2 & 2 \\ 0 & 1 & 2 & 2 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
L = [[0] * 4 for i in range(4)]
for i in range(4) :
    for j in range(4) :
        if i == j :
            L[i][j] = 1
        elif i < j :
            L[i][j] = 2

print(L)
```

# Création de listes imbriquées

Solution plus compacte :

```
L = [0] * 4
for i in range(4) : # pour chaque ligne
    L[i] = [0] * i + [1] + [2] * (n - i - 1)

print(L)
```