Listes

Le type `list` permet de stocker des valeurs dans un ordre précis.

Python

```
# on crée une liste avec 3 valeurs
lst = ['bonjour', 3.14, True]
```

Une valeur de type `list` est itérable :

- on peut accéder à un élément de la liste, par exemple `lst[0]` ;
- on peut parcourir une liste.

Pour accéder à un élement d'une liste situé à un endroit précis, on doit connaître son *indice* : le premier élément d'une liste a l'indice zéro, le deuxième l'indice 1, *et cætera*.

Python

```
# on crée une liste avec 3 valeurs
>>> lst = ['bonjour', 3.14, True]
# premier élément : indice 0
>>> lst[0]
'bonjour'

# deuxième élément
>>> lst[1]
3.14
```

1. Opérations de base

1.1. Créer une liste

- `lst = list()` crée une liste vide ;

- `lst = []` fait la même chose ;
- `lst = ['a', 7, True]` crée une liste composée de 3 éléments.

Une liste peut contenir des éléments de plusieurs types mais en pratique on évite cela.

1.2. Modifier un élément

Le type `list` est *mutable* : on peut changer un ou des éléments d'une liste *sans changer la liste elle-même*.

Python

```
>>> lst = [2, 3, 4, 1]
# on change le deuxième élément
>>> lst[1] = 10
>>> lst
[2, 10, 4, 1]
```

1.3. Ajouter un élément en fin de liste

On reprend l'exemple précédent

Python

```
>>> lst.append(7) # ajoute 7 à la fin de la liste
>>> lst
[2, 10, 4, 1, 7]
```

Remarque

`lst = lst + [7]` a le même effet que `lst.append(7)` : on crée une « mini-liste » `[4]`, on concatène les 2 listes et on remet le résultat dans `lst`.
En pratique la première méthode est la plus simple et aussi la plus rapide.

1.4. Insérer un élément à une place donnée

Pour une liste `lst` valant `[2, 10, 4, 1]`, si on veut insérer la valeur 5 à l'indice 1 on écrira :

```
lst.insert(1, 5)
```

et `lst` vaudra `[2, 5, 10, 4, 1]`

La syntaxe est `lst.insert(indice, valeur)`

1.5. Retirer un élément à une position donnée

Si une liste `lst` a pour valeur `[3 ,7, 1]` et qu'on veut supprimer son deuxième élément alors on écrit :

```
del lst[1]
```

Ensuite, `lst` aura la valeur `[3,1]`.

1.6. Retirer une valeur précise

Pour retirer une valeur *qui appartient à une liste* on procède ainsi :

Si `lst` a la valeur `[1, 2, 5, 4, 2, 3]` alors l'instruction

```
lst.remove(2)
```

Supprime la *première occurrence* de `2` dans `lst`.
Après cela, `lst` a la valeur `[1, 5, 4, 2, 3]`.

1.7. Concaténer des listes

On peut procéder de 2 manières :

- `lst1.extend(lst2)` ajoute les éléments de la liste `lst2` à la fin de `lst1`;
- `lst1 = lst1 + lst2` crée une liste avec les éléments de `lst1` et ceux de `lst2`, puis replace le résultat dans `lst1`.

En pratique la première méthode est plus rapide.

1.8. Longueur d'une liste

La fonction `len`

- prend en entrée une liste `lst`;

– renvoie la longueur de cette liste.

Ainsi `len([2, 3, 4])` vaut 3.

1.9. Divers

`lst.sort()` trie la liste dans l'ordre croissant.

`lst.reverse()` met les éléments dans l'ordre inverse.

2. Opérations avancées

2.1. Copier une liste (mauvaise méthode)

Python

```
>>> lst1 = [3, 5, 2]
>>> lst2 = lst1
>>> lst1[0] = 2
>>> lst1
[2, 5, 2]

>>> lst2
[2, 5, 2]  # problème : lst2[0] a changé aussi !
```

Ce comportement « étrange » vient du fait que le type `list` est *mutable*. Nous allons expliquer cela plus tard dans ce chapitre.

2.2. Copier une liste (bonne méthode)

Python

```
>>> lst1 = [3, 5, 2]
>>> lst2 = lst1[:] # on copie tous les éléments de lst1
 ↪  dans lst2
>>> lst1[0] = 2
>>> lst1
[2, 5, 2]

>>> lst2
[3, 5, 2] # Ouf !
```

2.3. Extraire une sous-chaîne

Soit `lst` une liste de longueur n et p et q deux entiers tels que $0 \leqslant p < q \leqslant n$. Alors

- `lst[p:q]` est la liste composée des éléments `lst[p]`, ..., `lst[q-1]`;
- `lst[:q]` signifie `lst[0:q]`;
- `lst[p:]` signifie `lst[p:n]`.

Exemple

Si `lst` vaut `[2, 5, 3, 4, 9, 2, 5]` alors

- `lst[2:6]` vaut `[3, 4, 9, 2]`;
- `lst[3:]` vaut `[4, 9, 2, 5]`;
- `lst[:2]` vaut `[2, 5]`.

3. Parcourir une liste

3.1. Parcours selon les indices

Définition : parcours selon les indices

Soit `lst` une liste de longueur `n`.

Alors ses éléments sont `lst[0]`, ..., `lst[n-1]` et on parcourt la liste en

- considérant un entier `i` qui jouera le rôle d'*indice*;
- faisant parcourir à `i` la plage de valeurs `range(n)`;
- considérant les `lst[i]`.

Exemple : un parcours selon les indices

On affiche les élements d'une liste grâce à un parcours par les indices.

```
lst = [54, 65, 123]
n = len(lst) # n vaut 3
for i in range(n): # range(3), c'est 0, 1, 2
    print(lst[i])
```

Le parcours d'une liste par les indices est *crucial* lorsque lors du parcours, on veut savoir à quelle place on se trouve dans la liste. C'est le cas quand on veut déterminer si une liste est triée dans l'ordre croissant ou non : il faut regarder si chaque élement est plus petit que le suivant dans la liste.
C'est aussi le cas quand on veut par exemple déterminer l'indice de la première apparition d'une valeur dans une liste.

3.2. Parcours selon les éléments

C'est plus simple que le parcours selon les indices mais on pert un peu d'information car pendant le parcours, on ne sait pas à quelle place on se trouve dans la liste.

Définition : parcours selon les éléments

Le parcours des éléments d'une liste `lst` s'effectue à l'aide d'une simple boucle `for x in lst`. `x` prend alors successivement les valeurs de chacun des éléments de `lst`, dans l'ordre.

Exemple : un parcours selon les éléments

```
lst = [54, 65, 123]
for x in lst:
    print(x)
```

3.3. Bilan

On peut toujours utiliser un parcours de liste selon les indices. On peut toujours transformer un parcours selon les éléments en un parcours selon les indices. Le contraire est faux si l'on a *absolument* besoin de savoir quels sont les indices des éléments que l'on examine lors du parcours.

À éviter absolument

Les codes comme celui-ci :

```
for i in lst:
    print(i)
```

On a l'impression que `i` est un indice mais c'est une valeur, et l'expérience prouve que dans 90% des cas, une erreur du type `lst[i]` survient, alors que…`i` n'est pas un indice ici !

De même les codes comme celui-là :

```
for x in range(len(lst)):
    print(lst[x])
```

Là encore il y a beaucoup de chances que l'indice `x` soit pris pour une valeur.

Conseil

Réserver les noms de variables `i`, `j` et `k` pour les indices et `x`, `y` et `z` pour les éléments.

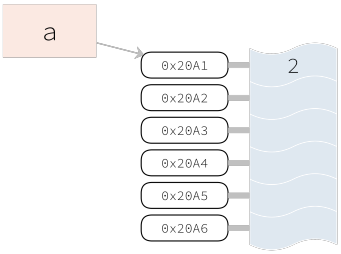
4. Mutabilité

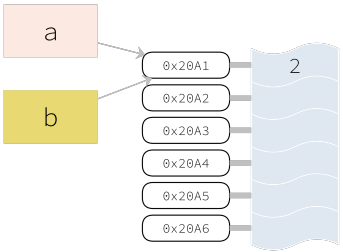
Examinons la différence entre un type *non mutable* tel que `int` et le type `list`, qui est *mutable*.

4.1. Variables de type non-mutable

`a = 2`

La valeur 2 est stockée en mémoire et une variable `a` est créée, associée à cette valeur.



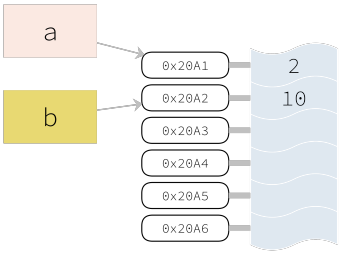


`b = a`

Une deuxième variable `b` est créée, avec pour valeur 2 également. Elles partagent la même adresse-mémoire.

```
b = 10
```

La valeur 10 est stockée dans une autre adresse mémoire (car la valeur 2 sert toujours pour `a`) et associée à `b`.

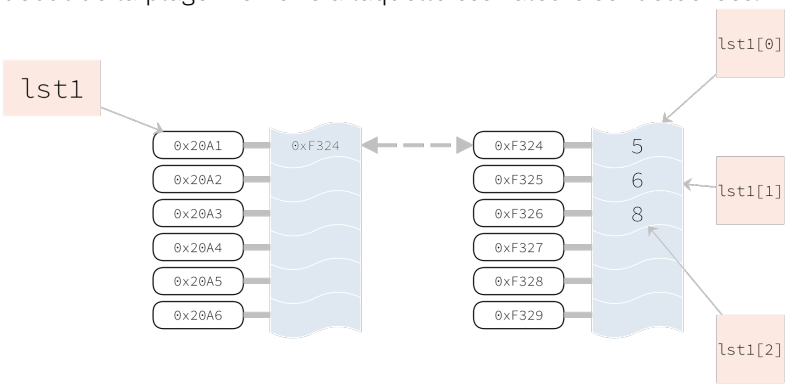


4.2. Variables de type mutable

Copier une liste (mauvaise méthode)

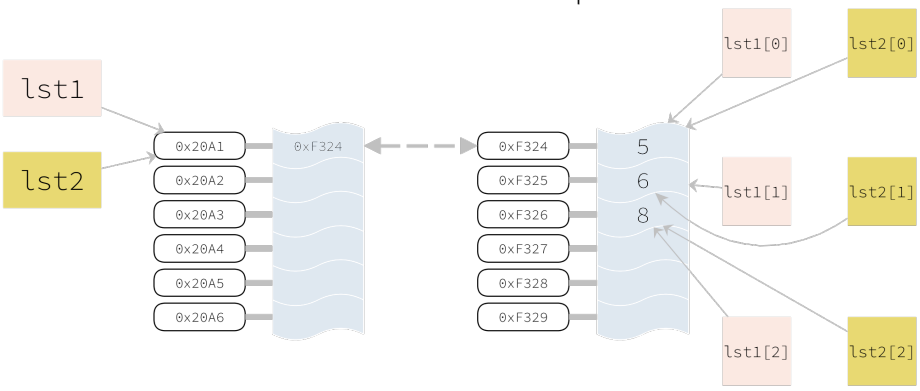
```
lst1 = [5, 6, 8]
```

Les élements 5, 6 et 8 sont stockés en mémoire et `lst1` contient l'adresse du début de la plage mémoire à laquelle ces valeurs sont stockées.



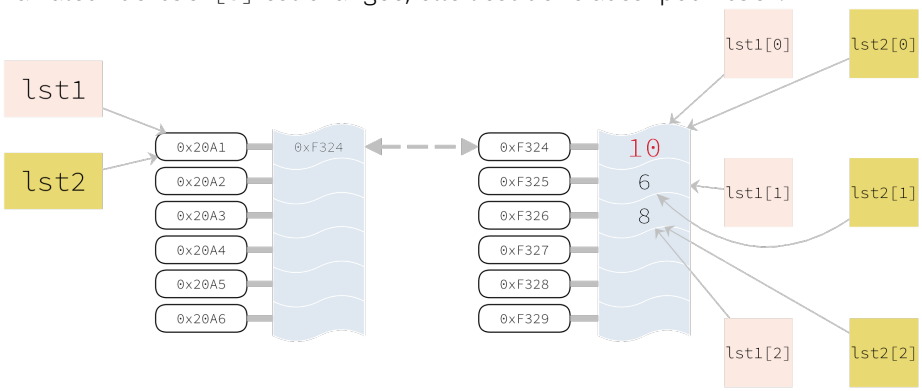

```
lst2 = lst1
```

La variable `lst2` est associée à la même valeur que `lst1`.



```
lst1[0] = 10
```

La valeur de `lst1[0]` est changée, elle l'est donc aussi pour `lst2`.

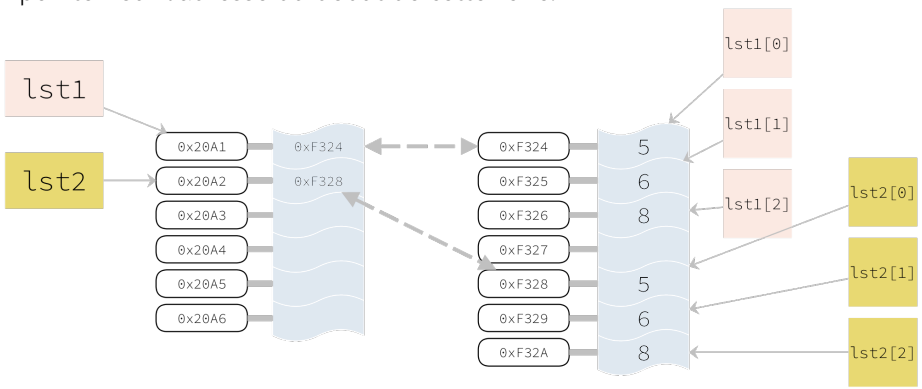


Voilà donc pourquoi lorsqu'on écrit `lst2 = lst1`, tout changement dans `lst1` se reflète aussi dans `lst2` et vice-versa.

Copier une liste (bonne méthode)

```
lst2 = lst1[:]
```

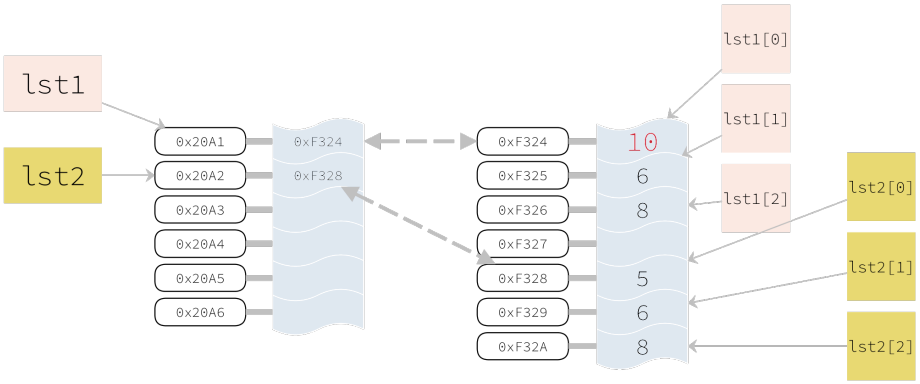
Les éléments de `lst1` sont recopiés dans une autre zone mémoire, et `lst2` « pointe » sur l'adresse du début de cette zone.




```
lst1[0] = 10
```

Le changement n'affecte pas `lst2`.

10


lst1
lst2
0x20A1
0x20A2
0x20A3
0x20A4
0x20A5
0x20A6
0xF324
0xF328
0xF324
0xF325
0xF326
0xF327
0xF328
0xF329
0xF32A
10
6
8
5
6
8
lst1[0]
lst1[1]
lst1[2]
lst2[0]
lst2[1]
lst2[2]