Informatique S3 :
Structures de données en Python
Outils Python pour les data science
L2 MIASHS

Laurent Boyer

2024

Aujourd'hui – cours 3

Les types en Python

Modifiables vs Immuables

Tuples

Hiérarchie (partielle) des types en Python

https://docs.python.org/fr/3/reference/datamodel.html

- None
- numbers.Number
  - numbers.Integral
    - int
    - bool
  - numbers.Real = float
  - numbers.Complex = complex
- Sequences
  - listes
  - chaînes de caractères
  - tuples
- Ensembles
- Dictionnaires

Section 2

Modifiables vs Immuables

Types modifiables (un exemple)

- **Les listes sont modifiables :**

```
1 l = [1,3,5,7]
2 print(l)
3 l[3]=32
4 print(l)
```

▼

Types modifiables (un exemple)

- **Les listes sont modifiables :**

```
1 l = [1,3,5,7]
2 print(l)
3 l[3]=32
4 print(l)
```

▼

```
[1, 3, 5, 7]
[1, 3, 5, 32]
```

Types modifiables (un exemple)

- **Les listes sont modifiables :**

```
1 l = [1,3,5,7]
2 print(l)
3 l[3]=32
4 print(l)
```

▼

```
[1, 3, 5, 7]
[1, 3, 5, 32]
```

- Et il y a de nombreuses autres manières de modifier une liste : `append`, `insert`, etc.

Types immuables (un exemple)

- **Les entiers (`int`) sont immuables :**

Une fois un entier créé il n'est jamais modifié.

Types immuables (un exemple)

- **Les entiers (`int`) sont immuables :**

Une fois un entier créé il n'est jamais modifié.

Aucune fonction ne peut modifier un entier comme `append` ou `insert` modifient les listes.

Types immuables (un exemple)

- **Les entiers (`int`) sont immuables :**

Une fois un entier créé il n'est jamais modifié.

Aucune fonction ne peut modifier un entier comme `append` ou `insert` modifient les listes.

- **Questions/remarques ?**

Types immuables (un exemple)

- **Les entiers (`int`) sont immuables :**

Une fois un entier créé il n'est jamais modifié.

Aucune fonction ne peut modifier un entier comme `append` ou `insert` modifient les listes.

- **Questions/remarques ?**

Pourtant on peut faire cela :

```
1 x = 4
2 print(x)
3 x = 32
4 print(x)
```

▼

Types immuables (un exemple)

- **Les entiers (`int`) sont immuables :**

Une fois un entier créé il n'est jamais modifié.

Aucune fonction ne peut modifier un entier comme `append` ou `insert` modifient les listes.

- **Questions/remarques ?**

Pourtant on peut faire cela :

```
1 x = 4
2 print(x)
3 x = 32
4 print(x)
```

▼

```
4
32
```

- `x` est bien modifiée, alors qu'en est-il ?

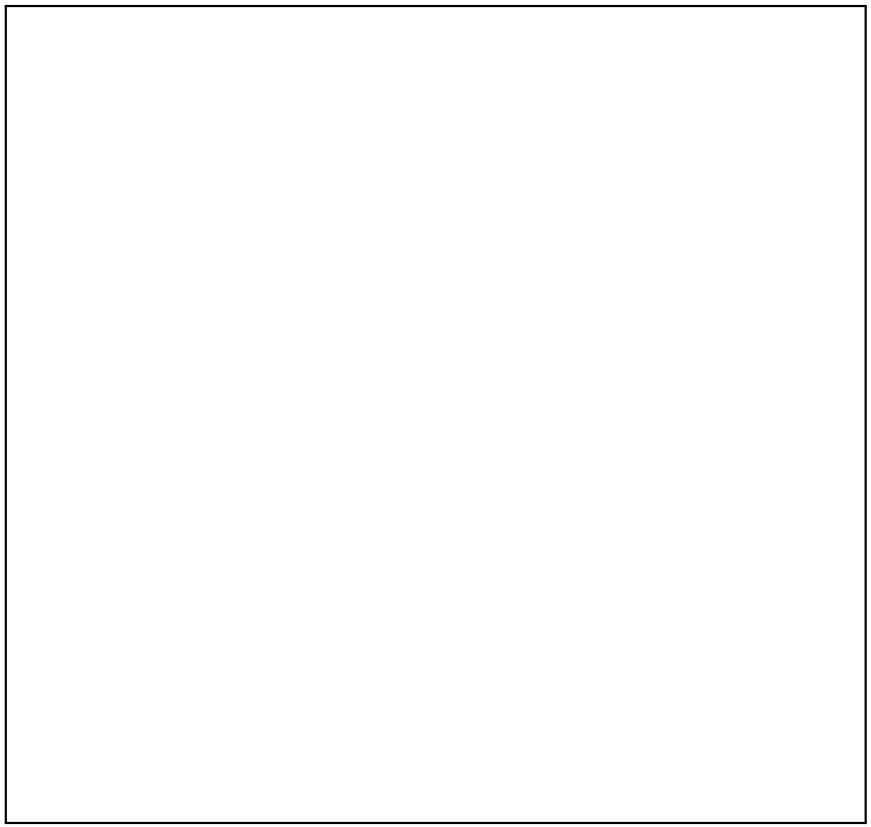
Qu'est-ce qu'une variable en Python ?

- **C'est une référence** à *quelque chose* en mémoire.

Qu'est-ce qu'une variable en Python ?

- **C'est une référence** à *quelque chose* en mémoire.

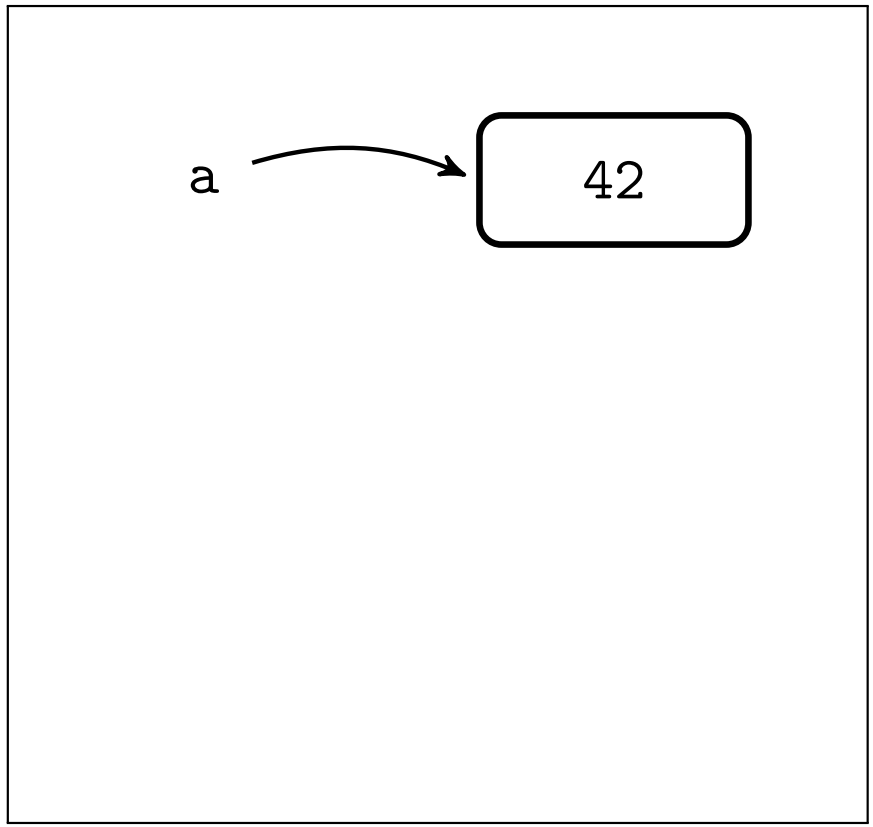
```
1 a = 42
```



Qu'est-ce qu'une variable en Python ?

- **C'est une référence** à *quelque chose* en mémoire.

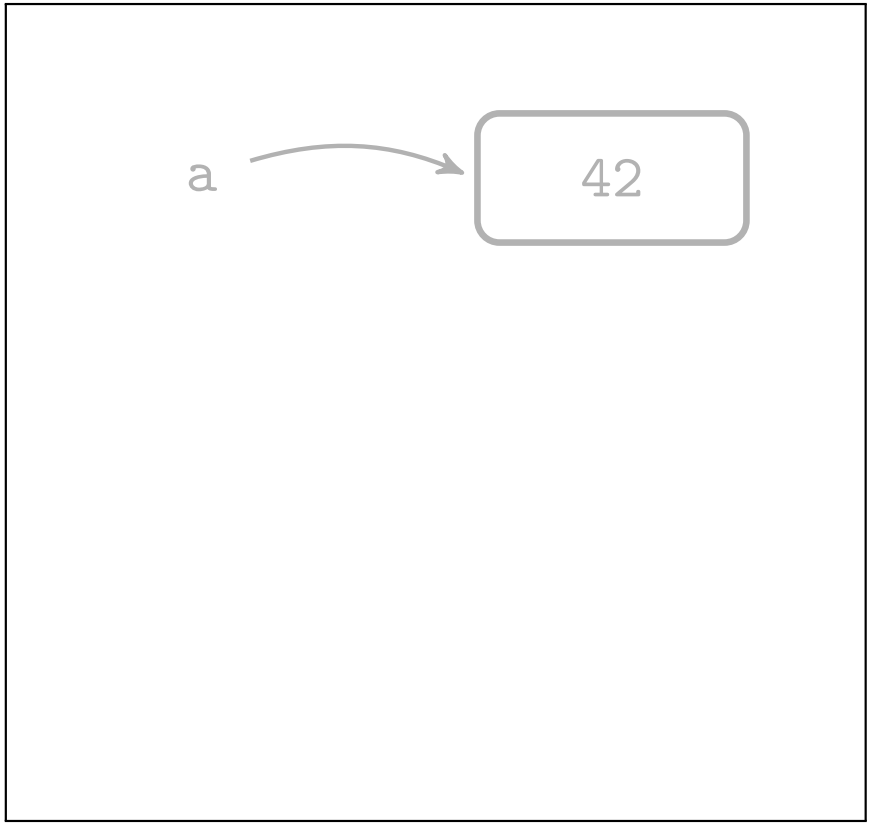
```
1 a = 42
```



Qu'est-ce qu'une variable en Python ?

- **C'est une référence** à *quelque chose* en mémoire.

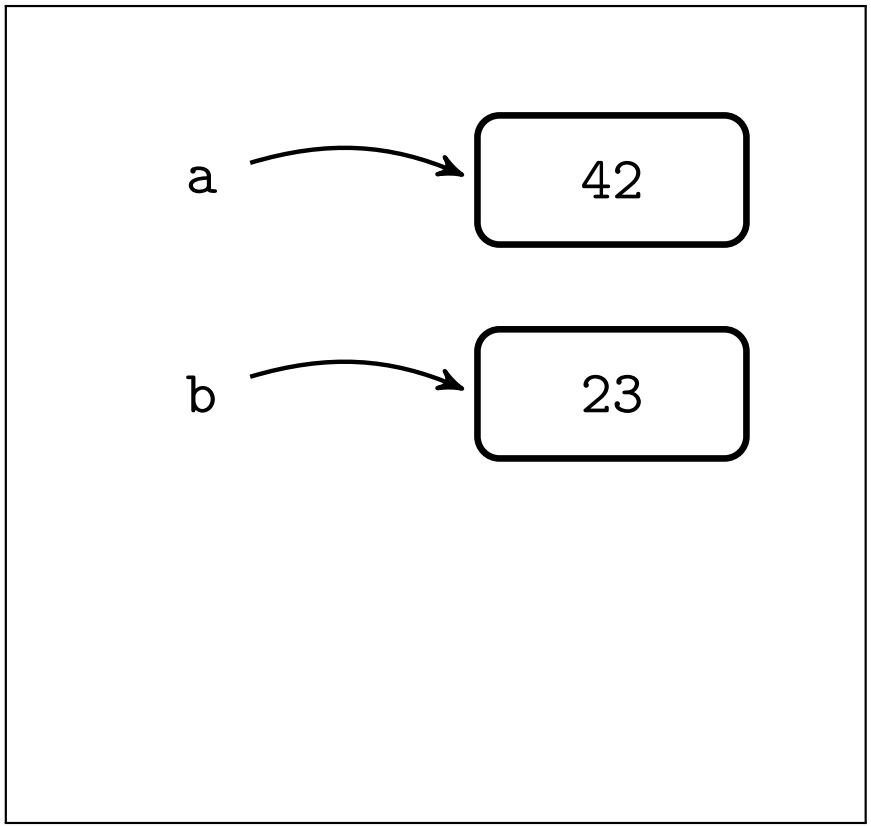
```
1 a = 42
2 b = 23
```



Qu'est-ce qu'une variable en Python ?

- **C'est une référence** à *quelque chose* en mémoire.

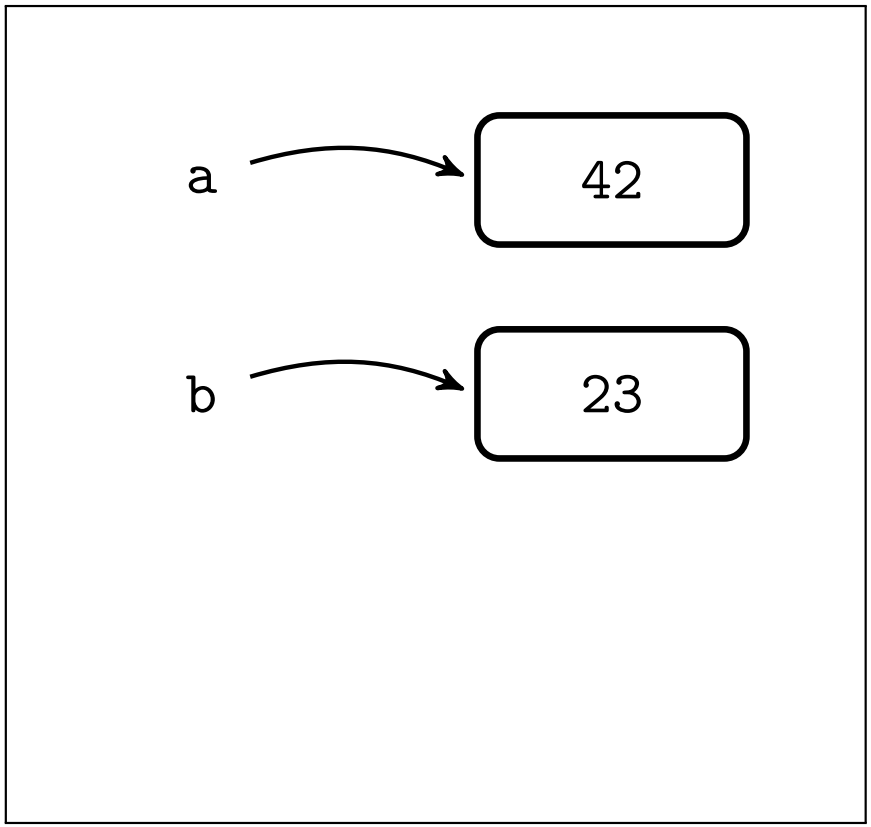
```
1 a = 42
2 b = 23
```



Qu'est-ce qu'une variable en Python ?

- **C'est une référence** à *quelque chose* en mémoire.

```
1 a = 42
2 b = 23
3 print(a)
4 print(b)
```

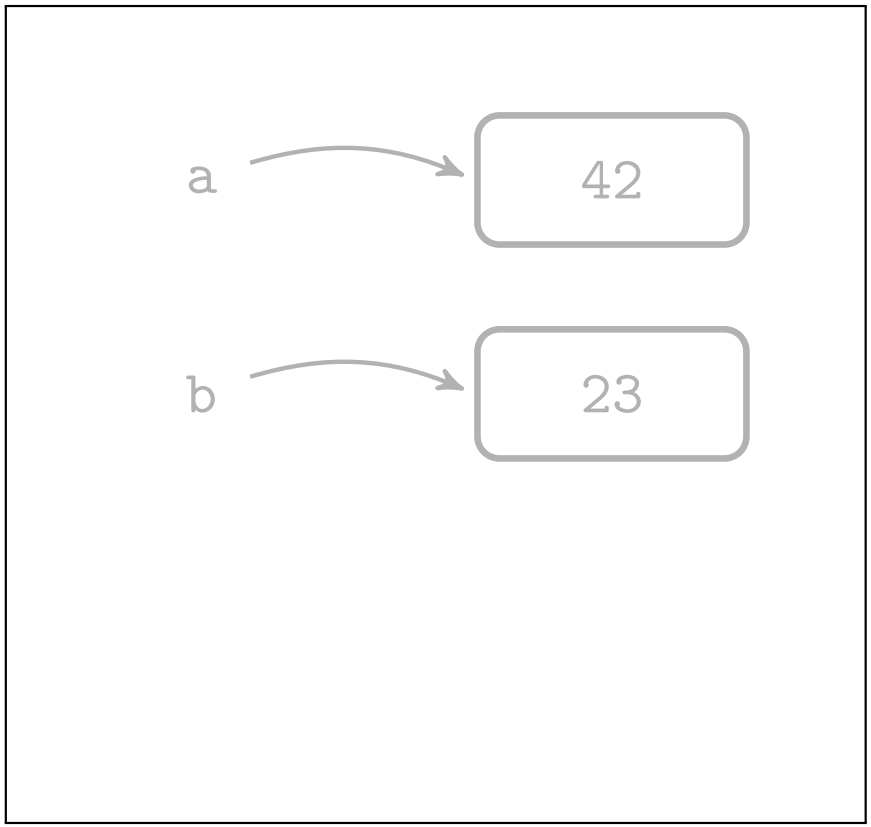


▼

```
42
23
```

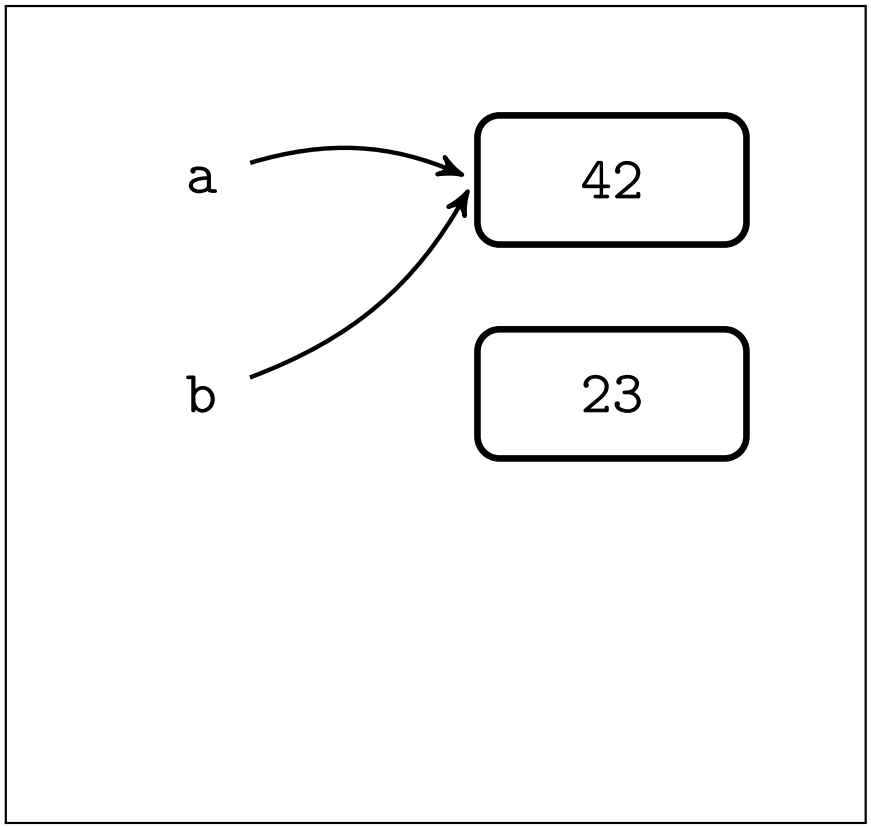
Modifions une variable immuable ??

```
1 a = 42
2 b = 23
3 print(a)
4 print(b)
5 b = a
```



Modifions une variable immuable ??

```
1 a = 42
2 b = 23
3 print(a)
4 print(b)
5 b = a
```

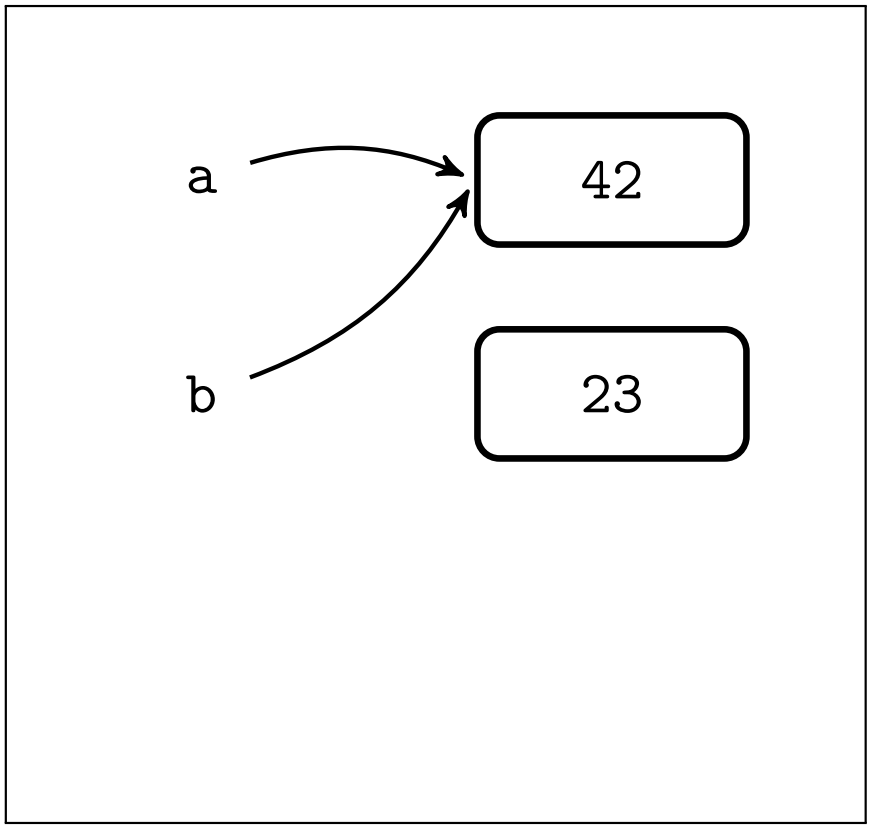


Modifions une variable immuable ??

```
1 a = 42
2 b = 23
3 print(a)
4 print(b)
5 b = a
6 print(a)
7 print(b)
```

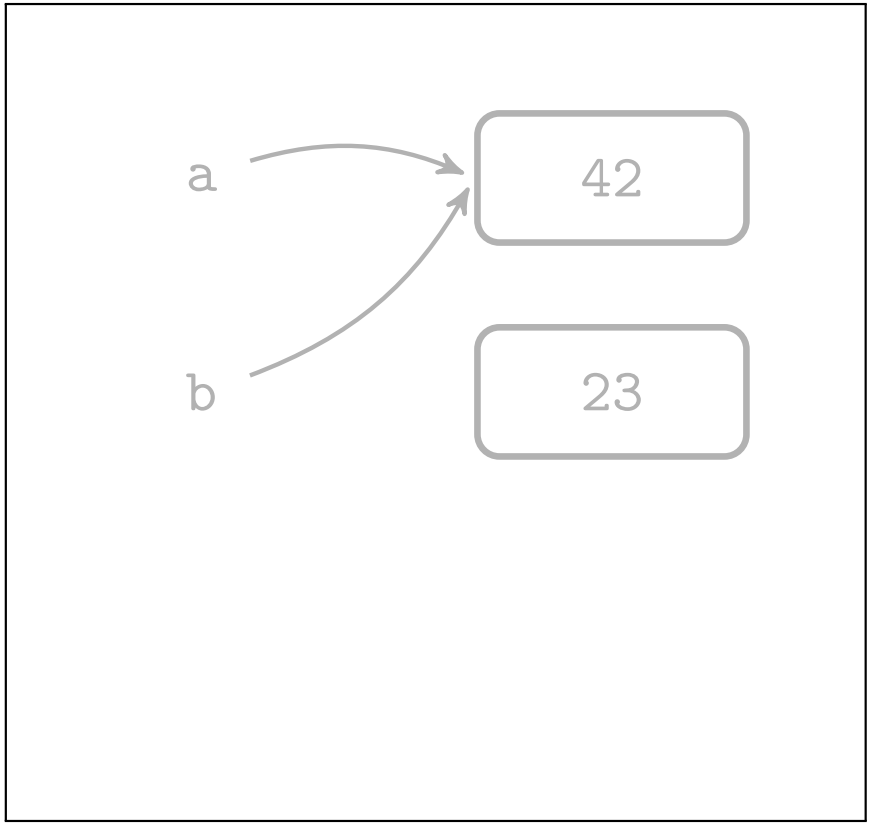
▼

```
42
23
42
42
```



Modifions une variable immuable ??

```
1 a = 42
2 b = 23
3 print(a)
4 print(b)
5 b = a
6 a = 12
7 print(a)
8 print(b)
```

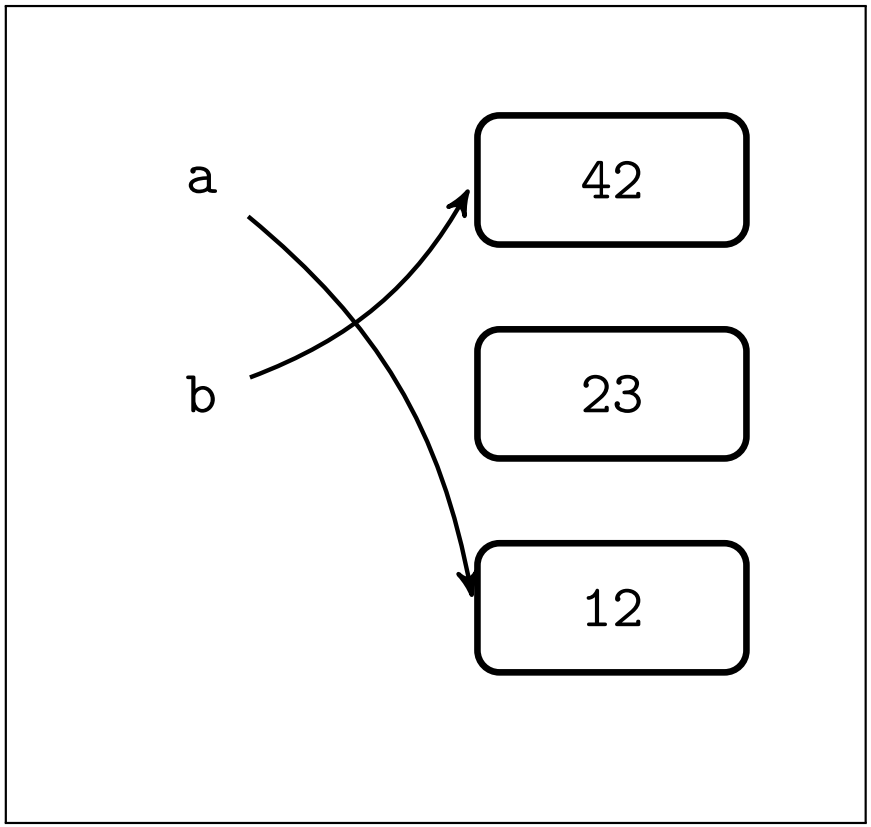


Modifions une variable immuable ??

```
1 a = 42
2 b = 23
3 print(a)
4 print(b)
5 b = a
6 a = 12
7 print(a)
8 print(b)
```

▼

```
42
23
12
42
```

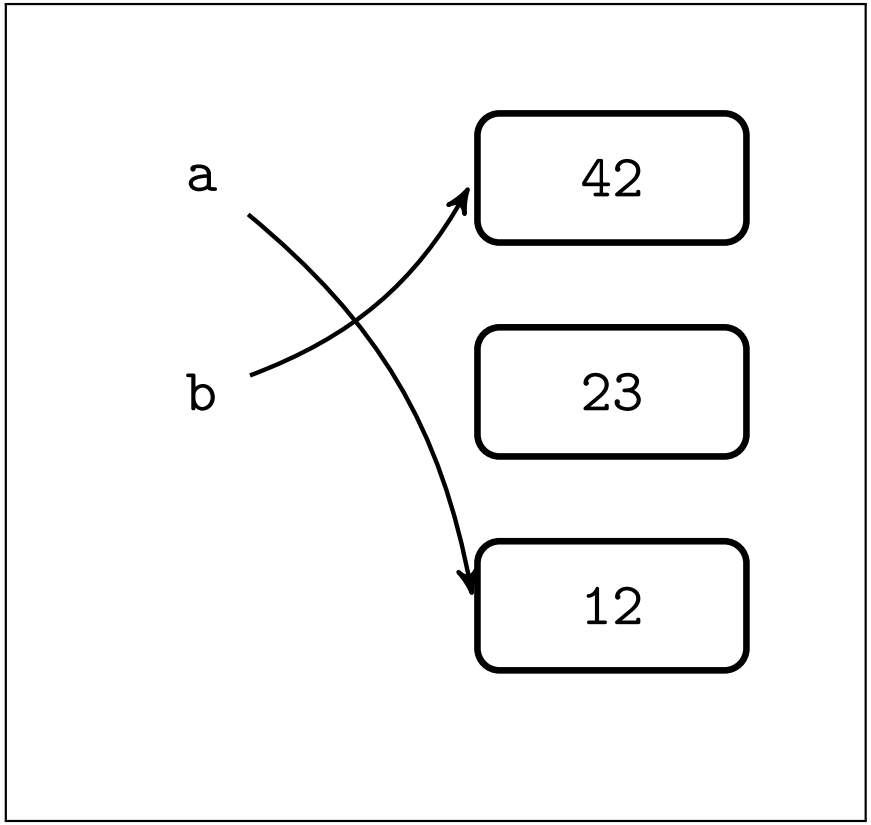


Modifions une variable immuable ??

```
1  a = 42
2  b = 23
3  print(a)
4  print(b)
5  b = a
6  a = 12
7  print(a)
8  print(b)
```

▼

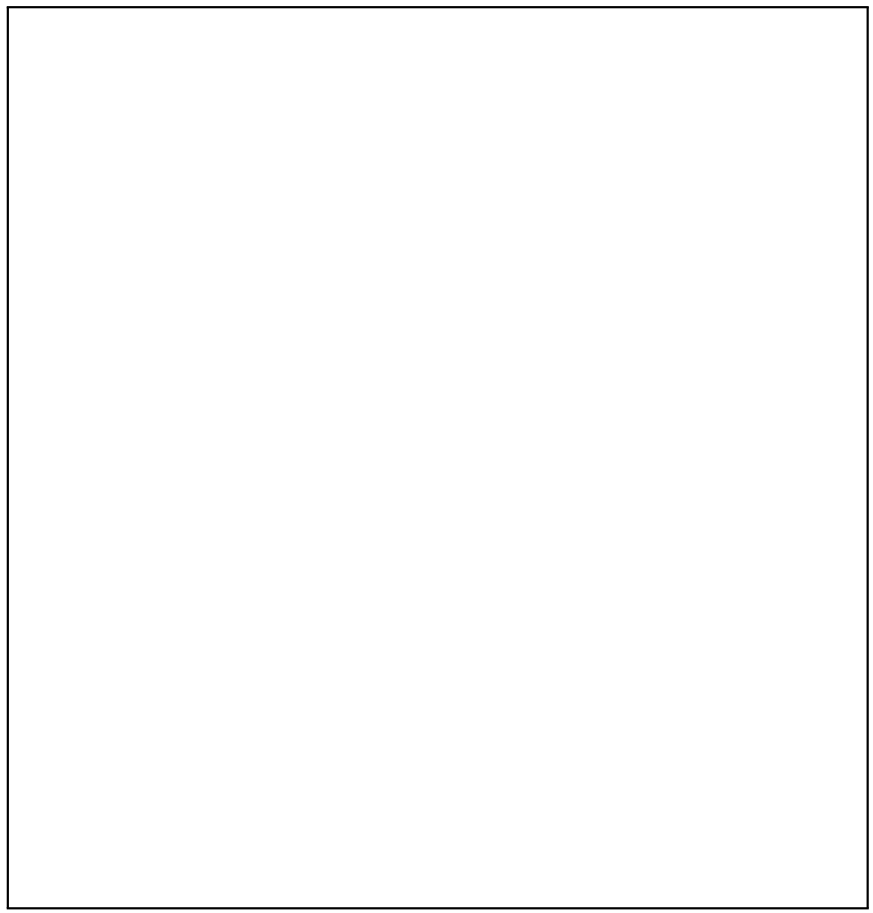
```
42
23
12
42
```



- **Immuable** = le contenu de *la case* n'est jamais modifié.
- Mais on peut *changer la flèche* = faire une nouvelle affectation.

Le cas modifiable (exemple des listes) (1/2)

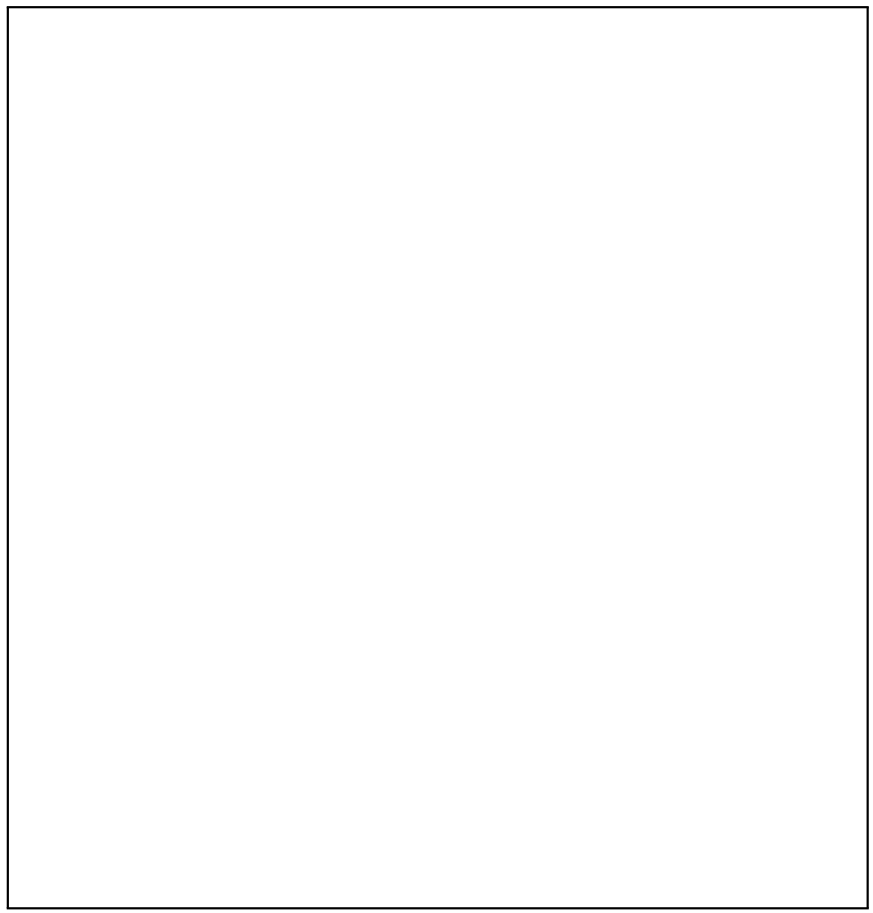
- On peut faire la même chose = changer la référence.



Le cas modifiable (exemple des listes) (1/2)

- On peut faire la même chose = changer la référence.

```
1 l1 = [1,3]
2 l2 = [4,5,6]
3 l3 = [2]
4
5 l1 = l2
6 l3 = [1,6,7]
7 print(l1)
8 print(l2)
9 print(l3)
```



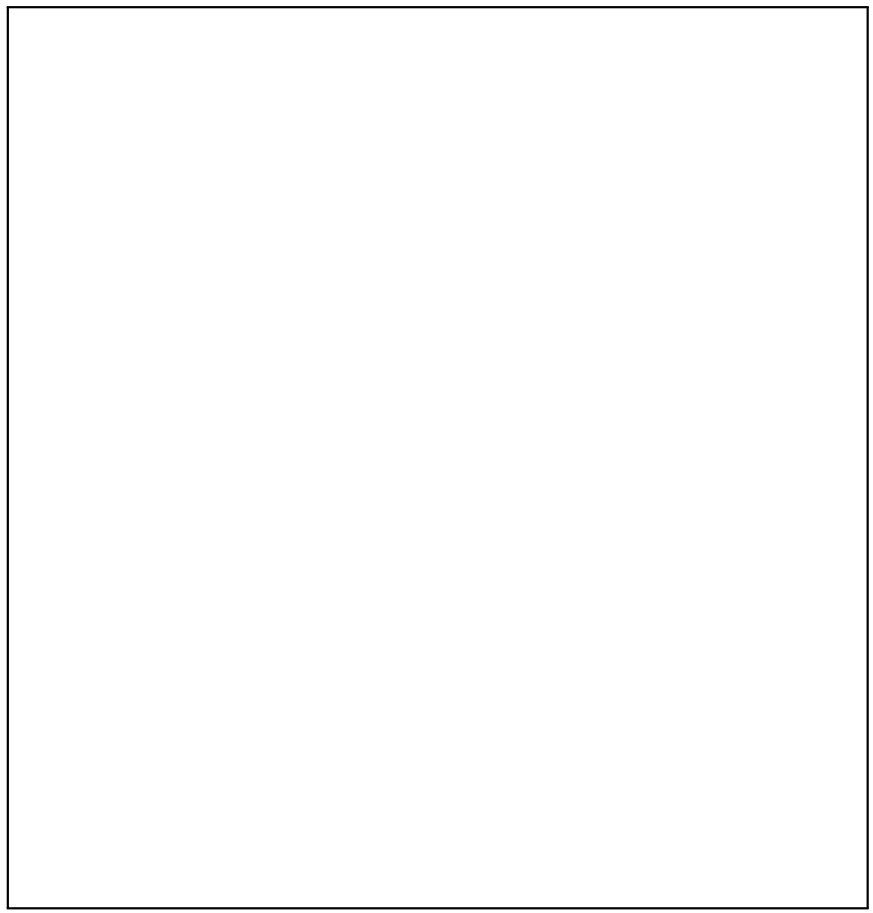
Le cas modifiable (exemple des listes) (1/2)

- On peut faire la même chose = changer la référence.

```
1 l1 = [1,3]
2 l2 = [4,5,6]
3 l3 = [2]
4
5 l1 = l2
6 l3 = [1,6,7]
7 print(l1)
8 print(l2)
9 print(l3)
```

▼

```
[4, 5, 6]
[4, 5, 6]
[1, 6, 7]
```



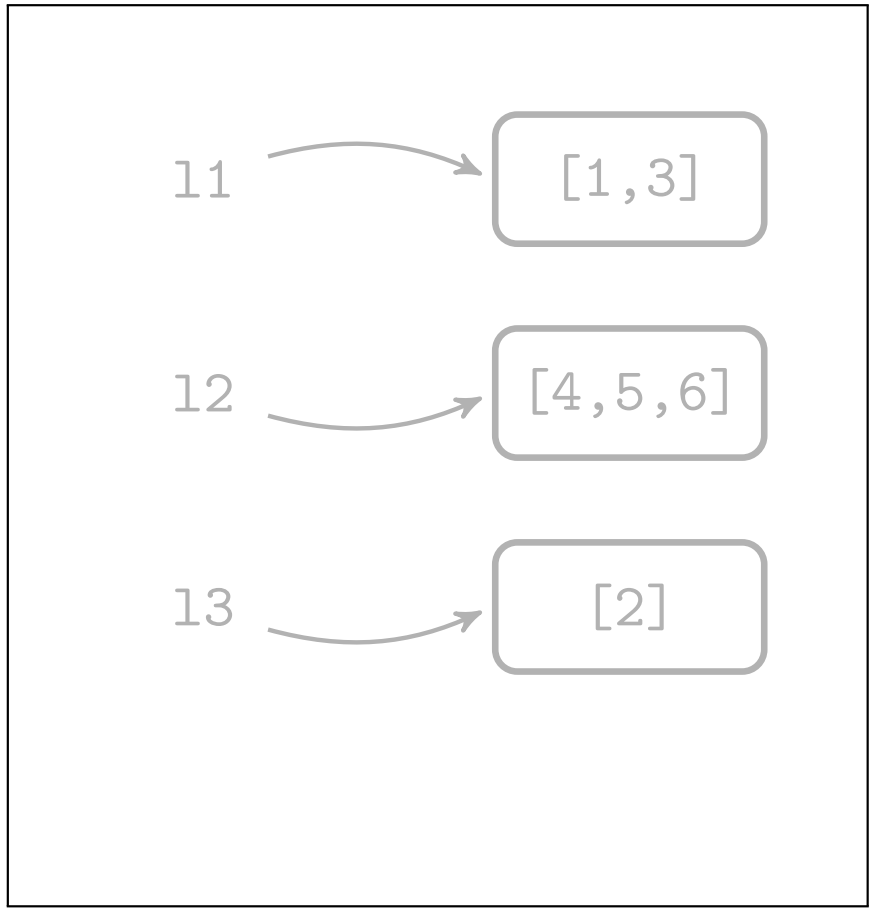
Le cas modifiable (exemple des listes) (1/2)

- On peut faire la même chose = changer la référence.

```
1  l1 = [1,3]
2  l2 = [4,5,6]
3  l3 = [2]
4
5  l1 = l2
6  l3 = [1,6,7]
7  print(l1)
8  print(l2)
9  print(l3)
```

▼

```
[4, 5, 6]
[4, 5, 6]
[1, 6, 7]
```



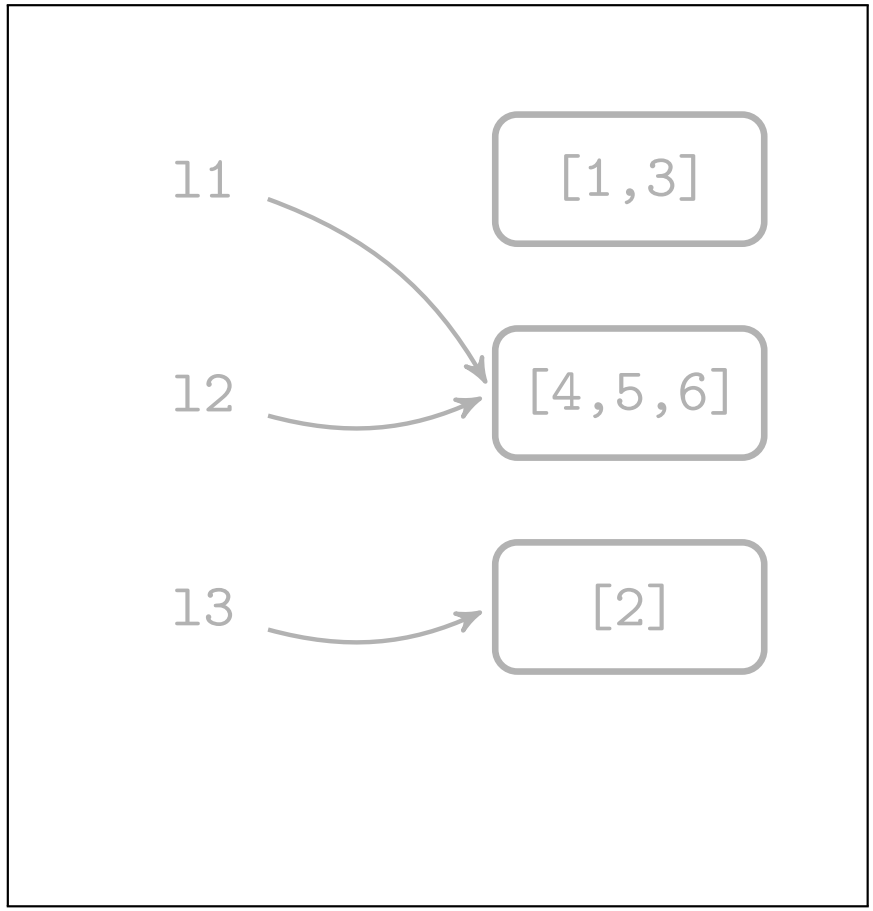
Le cas modifiable (exemple des listes) (1/2)

- On peut faire la même chose = changer la référence.

```
1  l1 = [1,3]
2  l2 = [4,5,6]
3  l3 = [2]
4
5  l1 = l2
6  l3 = [1,6,7]
7  print(l1)
8  print(l2)
9  print(l3)
```

▼

```
[4, 5, 6]
[4, 5, 6]
[1, 6, 7]
```



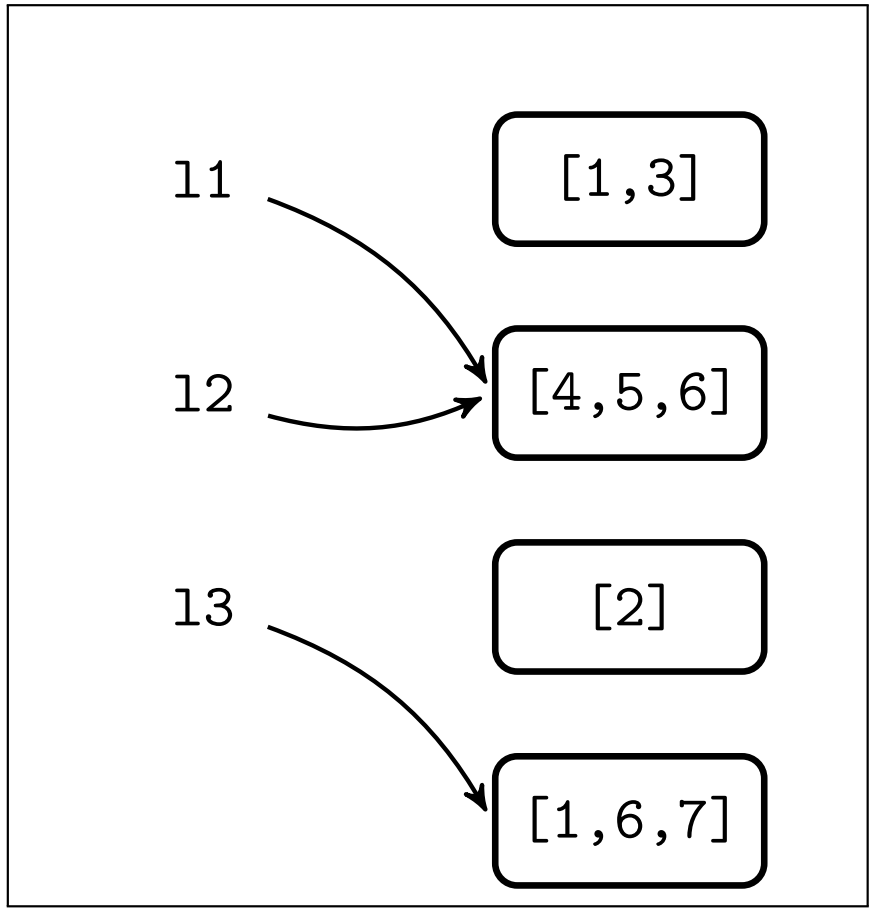
Le cas modifiable (exemple des listes) (1/2)

- On peut faire la même chose = changer la référence.

```
1 l1 = [1,3]
2 l2 = [4,5,6]
3 l3 = [2]
4
5 l1 = l2
6 l3 = [1,6,7]
7 print(l1)
8 print(l2)
9 print(l3)
```

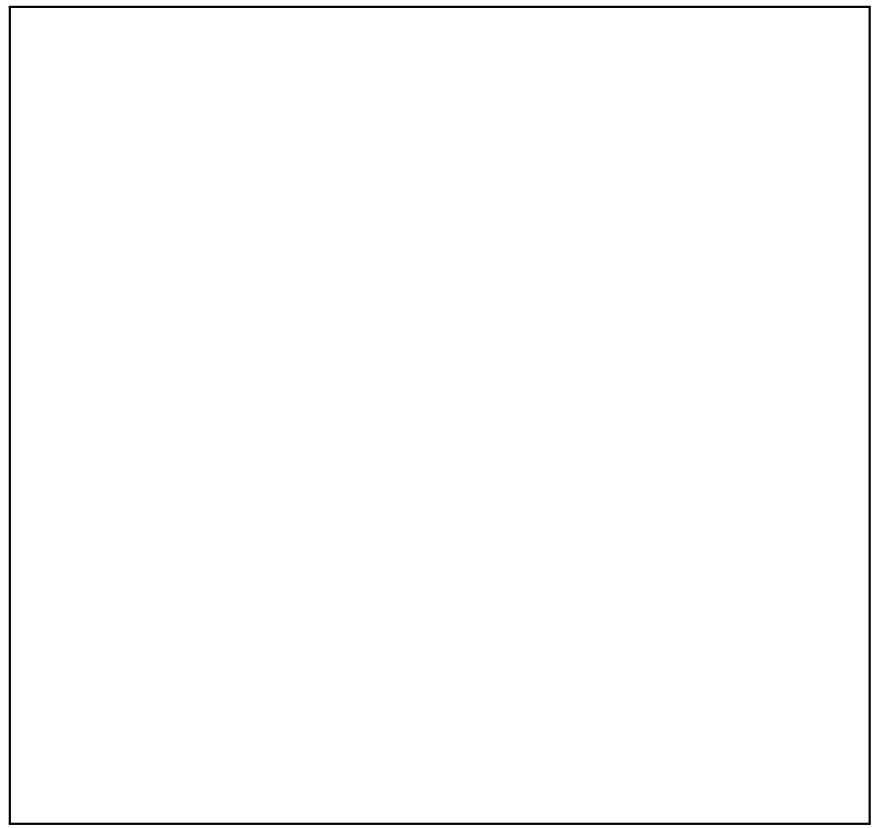
▼

```
[4, 5, 6]
[4, 5, 6]
[1, 6, 7]
```



Le cas modifiable (exemple des listes) (2/2)

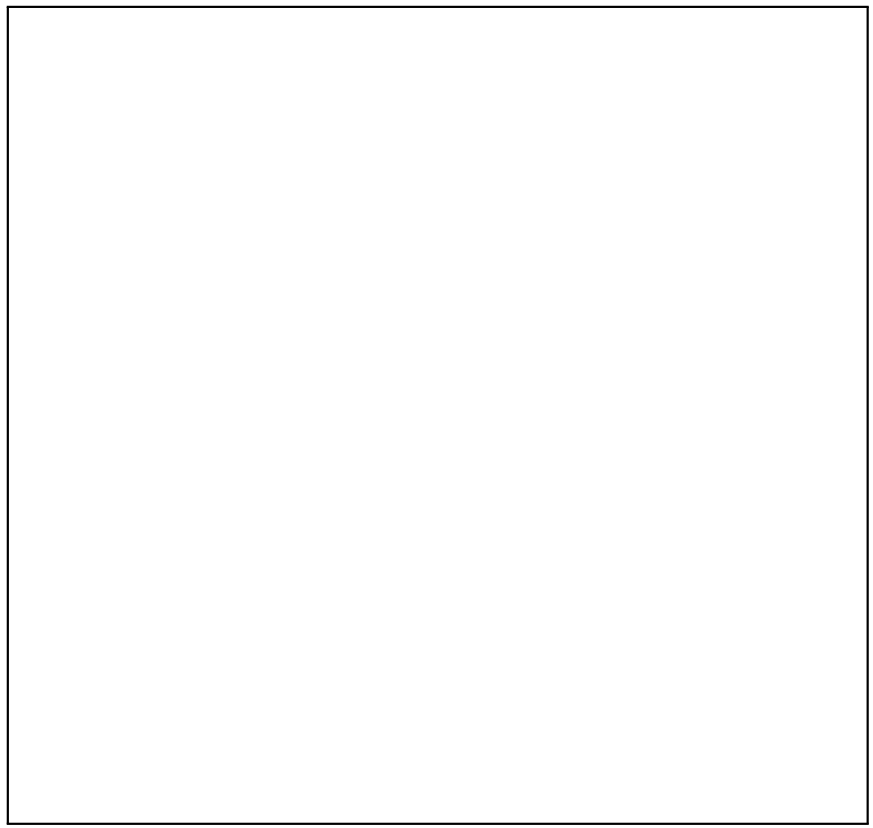
- Mais on peut aussi modifier *le contenu des cases*.



Le cas modifiable (exemple des listes) (2/2)

- Mais on peut aussi modifier *le contenu des cases*.

```
1 l1 = [1,3]
2 l2 = [4,5,6]
3
4 l1[0] = l2[2]
5 l1.append(4)
6 print(l1)
7 print(l2)
```



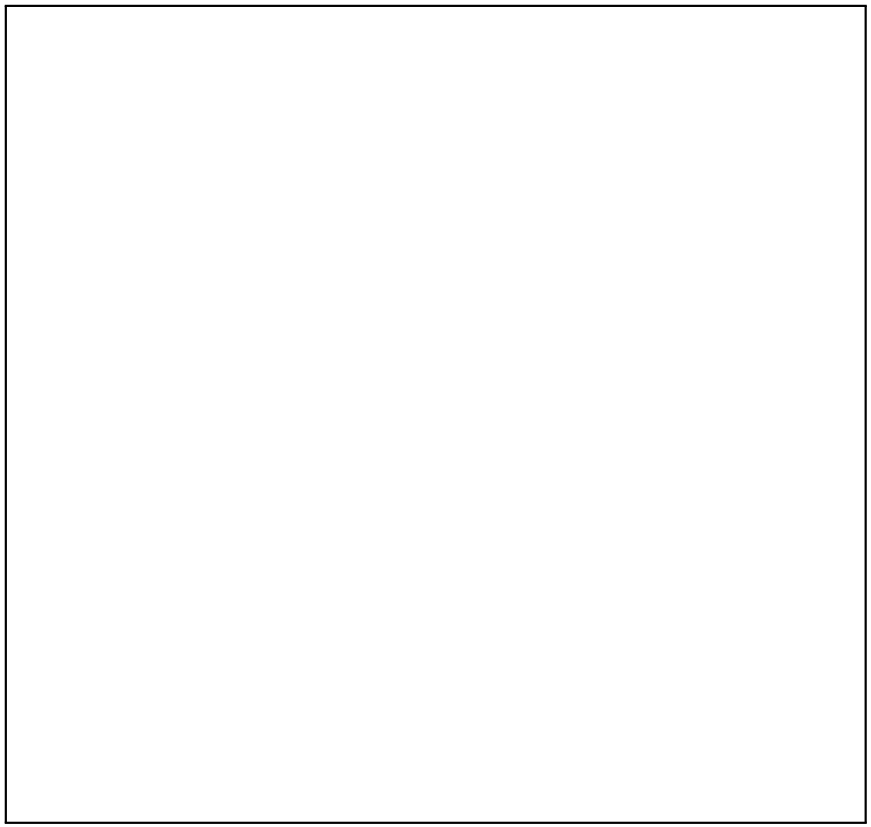
Le cas modifiable (exemple des listes) (2/2)

- Mais on peut aussi modifier *le contenu des cases*.

```
1  l1 = [1,3]
2  l2 = [4,5,6]
3
4  l1[0] = l2[2]
5  l1.append(4)
6  print(l1)
7  print(l2)
```

▼

```
[6, 3, 4]
[4, 5, 6]
```



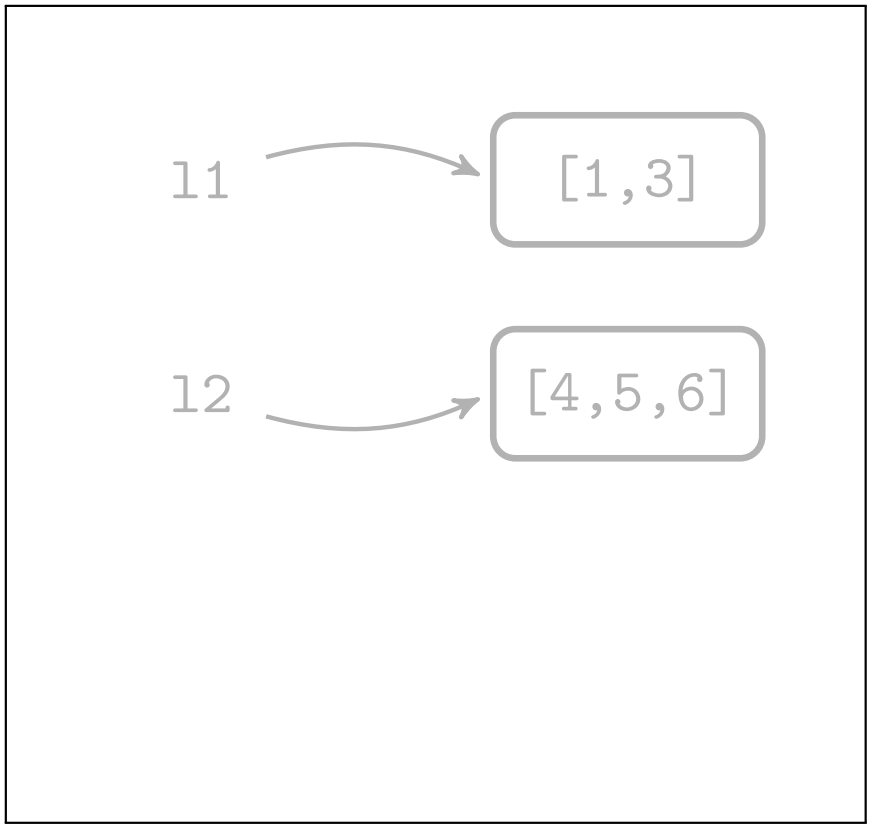
Le cas modifiable (exemple des listes) (2/2)

- Mais on peut aussi modifier *le contenu des cases*.

```
1  l1 = [1,3]
2  l2 = [4,5,6]
3
4  l1[0] = l2[2]
5  l1.append(4)
6  print(l1)
7  print(l2)
```

▼

```
[6, 3, 4]
[4, 5, 6]
```



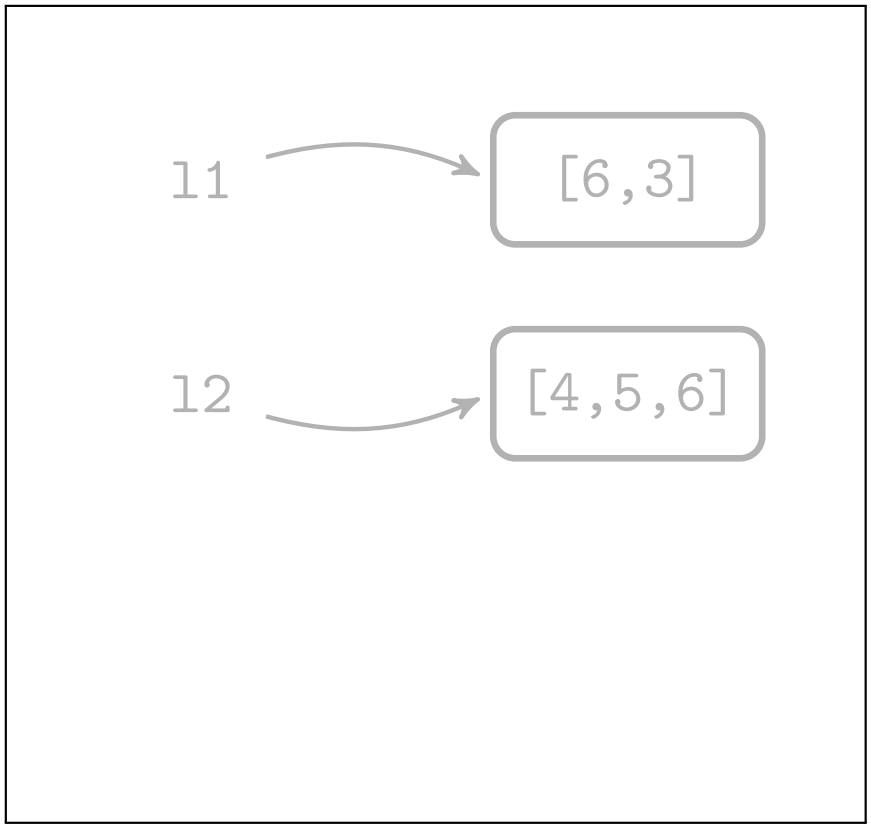
Le cas modifiable (exemple des listes) (2/2)

- Mais on peut aussi modifier *le contenu des cases*.

```
1  l1 = [1,3]
2  l2 = [4,5,6]
3
4  l1[0] = l2[2]
5  l1.append(4)
6  print(l1)
7  print(l2)
```

▼

```
[6, 3, 4]
[4, 5, 6]
```



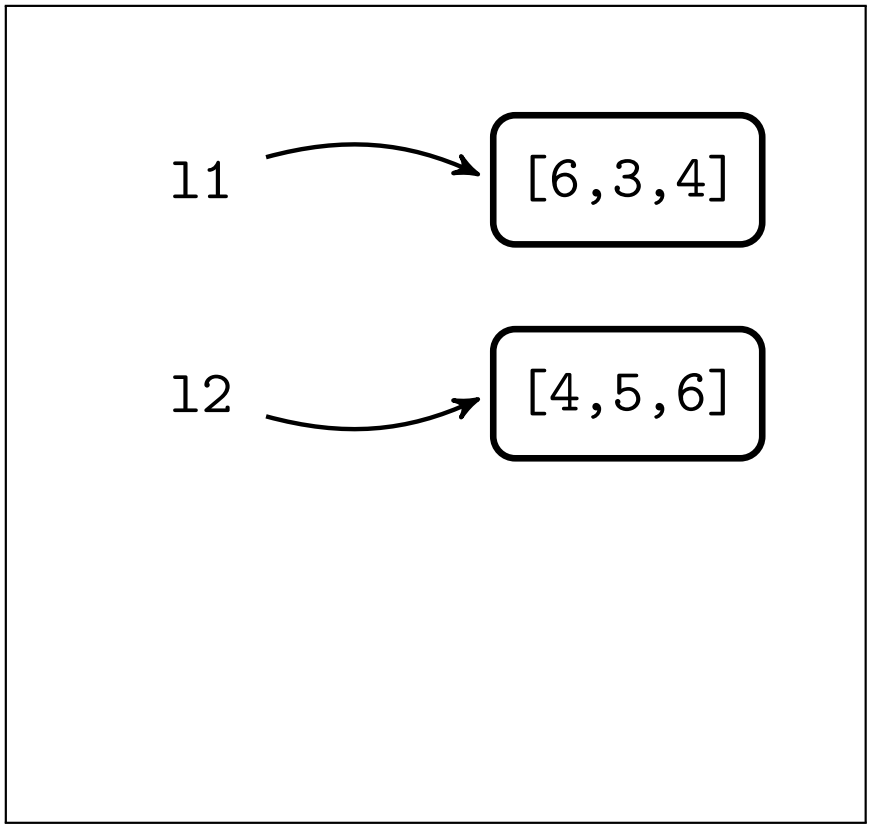
Le cas modifiable (exemple des listes) (2/2)

- Mais on peut aussi modifier *le contenu des cases*.

```
1  l1 = [1,3]
2  l2 = [4,5,6]
3
4  l1[0] = l2[2]
5  l1.append(4)
6  print(l1)
7  print(l2)
```

▼

```
[6, 3, 4]
[4, 5, 6]
```



Modifiable ou immuable, récapitulons

Un type de variable est :

- **soit immuable**
  = on ne peut jamais modifier la valeur des variables de ce type après leur création.
- **soit modifiable**
  = on peut modifier la valeur des variables de ce type.

Modifiable ou immuable, récapitulons

Un type de variable est :

- **soit immuable**
  = on ne peut jamais modifier la valeur des variables de ce type après leur création.
- **soit modifiable**
  = on peut modifier la valeur des variables de ce type.

Que la variable soit de type immuable ou modifiable, on peut faire une nouvelle affectation (= *changer les flèches*).

Avantages et inconvénients (1/2)

Exemple pour les listes

Avantages et inconvénients (1/2)

Exemple pour les listes

- **Avantage :** on peut faire des modifications sans recopier tous les éléments de la liste.

```
1 l = [x for x in range(10**45)]
2 l[3] = 5
3 l.append(0)
```

Avantages et inconvénients (1/2)

Exemple pour les listes

- **Avantage :** on peut faire des modifications sans recopier tous les éléments de la liste.

```
1 l = [x for x in range(10**45)]
2 l[3] = 5
3 l.append(0)
```

- **Inconvénient :** attention aux modifications involontaires.

```
1 l = [x for x in range(10)]
2 m = l
3 l[3] = 5
4 print(m) # m est modifiee aussi
```

\+ même problème dans les fonctions, qui peuvent modifier une liste reçue comme paramètre.

Avantages et inconvénients (2/2)

Exemple pour les `int`.

- **Avantage principal (garantie d'immuabilité):**

```
1 x = 3
2 y = x
```

Même si `x` et `y` pointent vers le même objet, aucune modification de la valeur de `x` ne risque d'entrainer une modification de la valeur de `y`. Et inversement.

- C'est aussi **l'inconvénient principal**.

Avantages et inconvénients (2/2)

Exemple pour les `int`.

- **Avantage principal (garantie d'immuabilité):**

```
1 x = 3
2 y = x
```

  Même si `x` et `y` pointent vers le même objet, aucune modification de la valeur de `x` ne risque d'entrainer une modification de la valeur de `y`. Et inversement.

- C'est aussi **l'inconvénient principal**.

▶ **Modifiables et immuables correspondent à des usages différents.**

Avantages et inconvénients (2/2)

Exemple pour les `int`.

- **Avantage principal (garantie d'immuabilité):**

```
1 x = 3
2 y = x
```

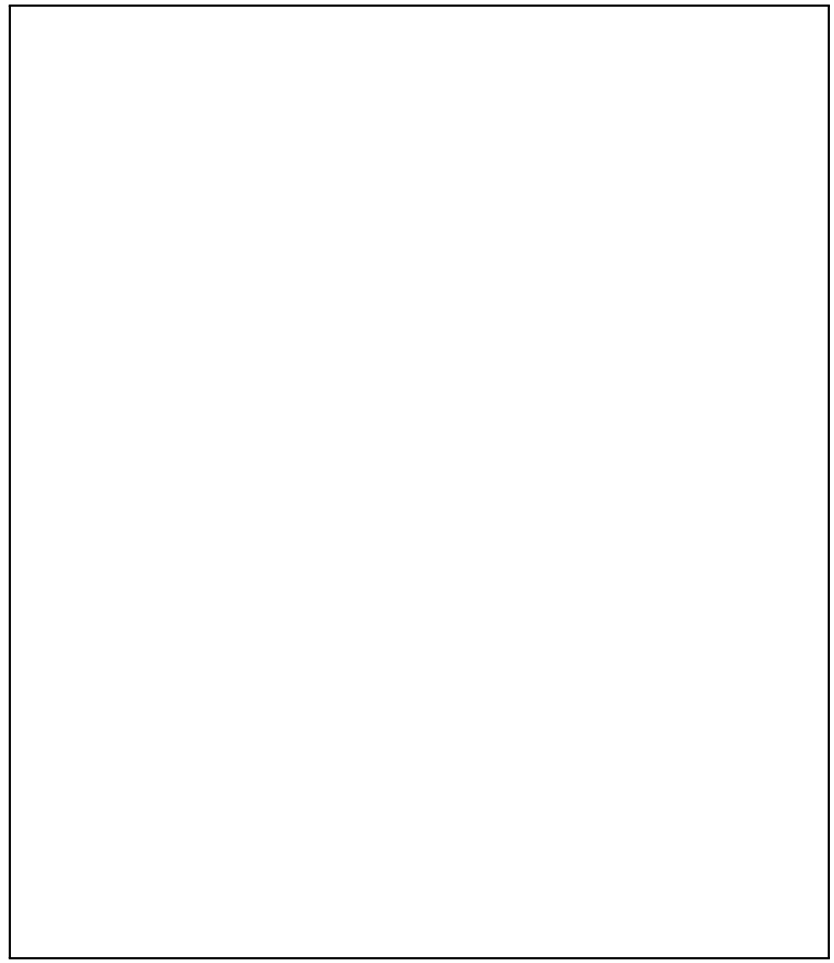
Même si `x` et `y` pointent vers le même objet, aucune modification de la valeur de `x` ne risque d'entrainer une modification de la valeur de `y`. Et inversement.

- C'est aussi **l'inconvénient principal**.

▶ **Modifiables et immuables correspondent à des usages différents.**

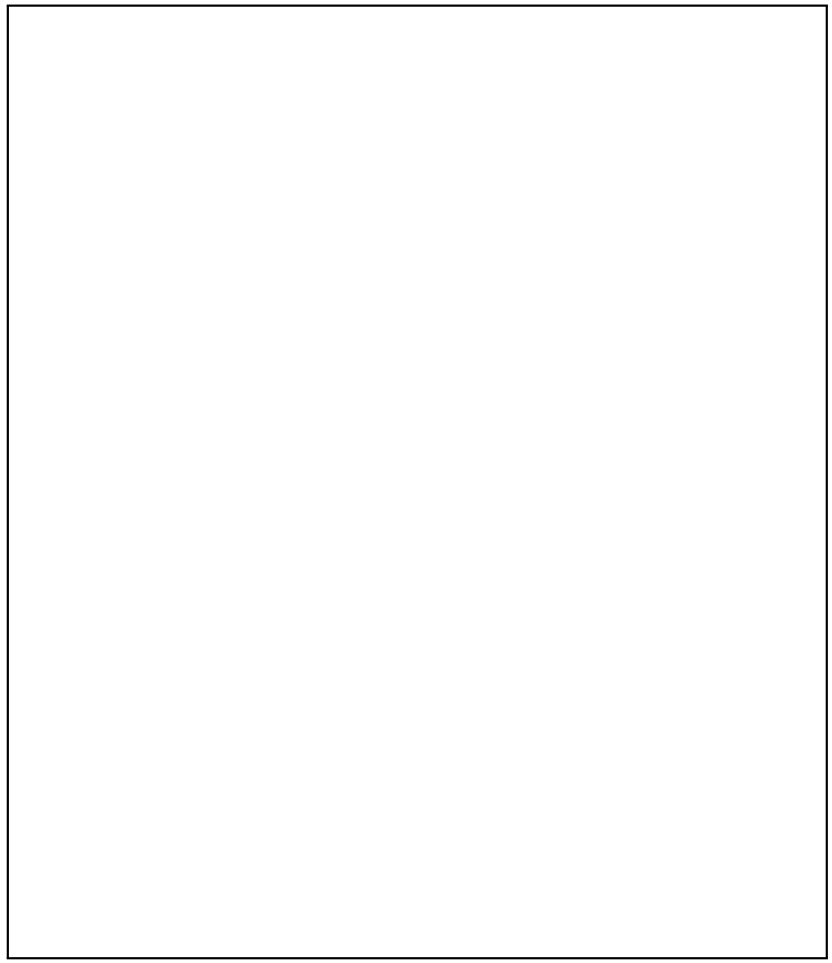
Exemple : Si on fabrique un très grand compteur qui incrémente un très gros entier, on aurait intérêt à utiliser un type modifiable fait de manière ad-hoc.

Implication pour les fonctions



Implication pour les fonctions

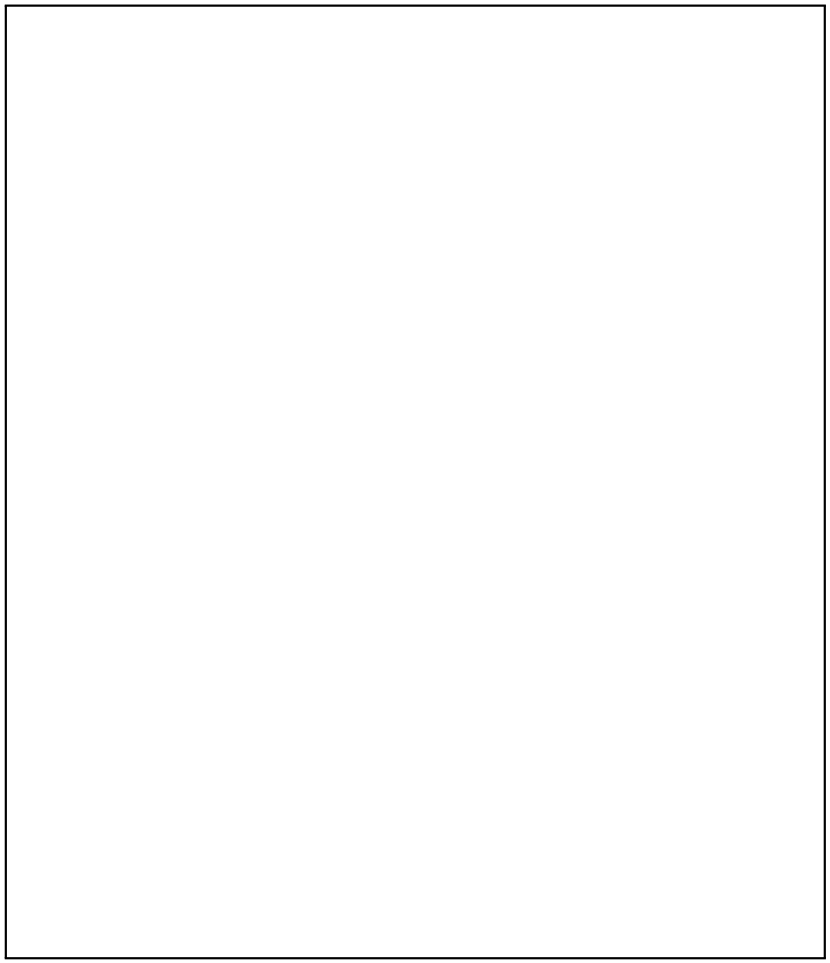
```
1  def ouh(l,m,x) :
2      l.remove(2)
3      m = l
4      x +=1
5      return m,x
6
7  a = [1,2,3]
8  b = [0,0]
9  y = 1
10 d,z = ouh(a,b,y)
11
12 print(a,b,y)
13 print(d,z)
```



Implication pour les fonctions

```
1  def ouh(l,m,x) :
2      l.remove(2)
3      m = l
4      x +=1
5      return m,x
6
7  a = [1,2,3]
8  b = [0,0]
9  y = 1
10 d,z = ouh(a,b,y)
11
12 print(a,b,y)
13 print(d,z)
```

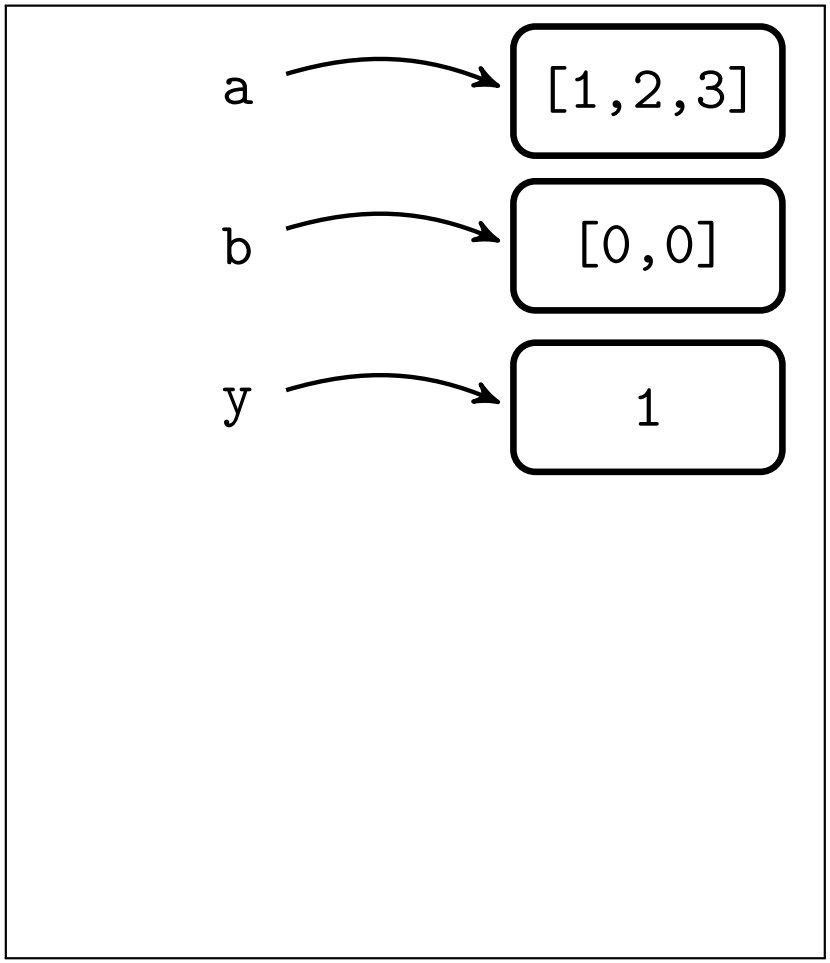
```
[1, 3] [0, 0] 1
[1, 3] 2
```



Implication pour les fonctions

```
1  def ouh(l,m,x) :
2      l.remove(2)
3      m = l
4      x +=1
5      return m,x
6
7  a = [1,2,3]
8  b = [0,0]
9  y = 1
10 d,z = ouh(a,b,y)
11
12 print(a,b,y)
13 print(d,z)
```

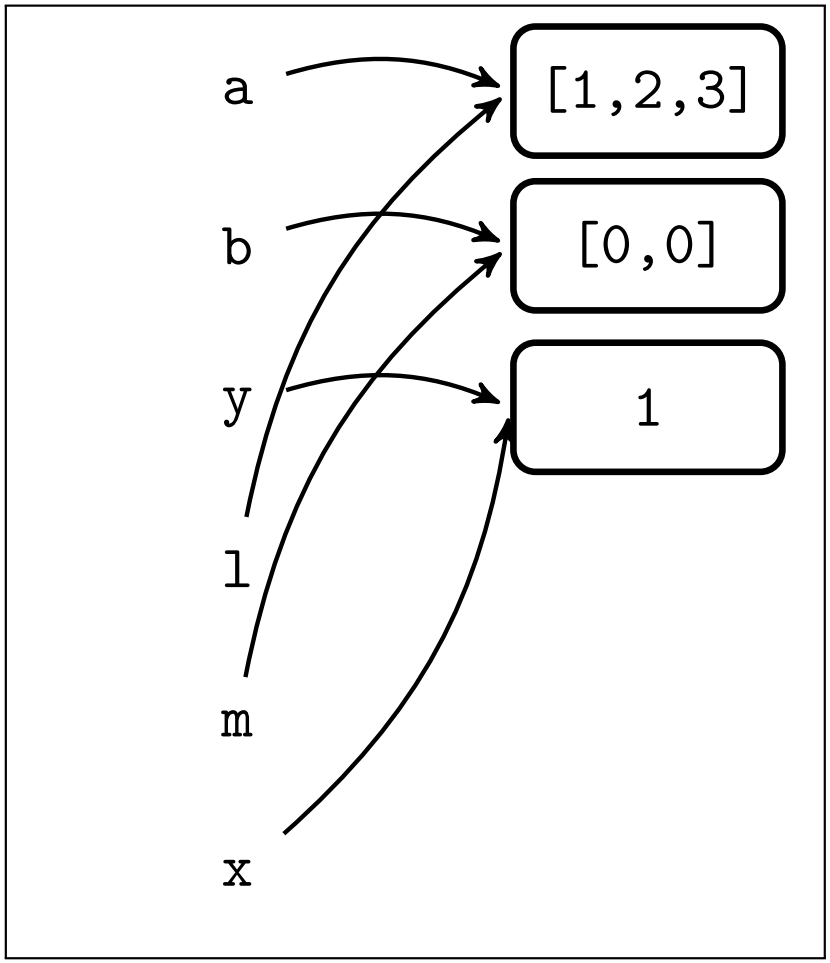
```
[1, 3] [0, 0] 1
[1, 3] 2
```



Implication pour les fonctions

```
1  def ouh(l,m,x) :
2      l.remove(2)
3      m = l
4      x +=1
5      return m,x
6
7  a = [1,2,3]
8  b = [0,0]
9  y = 1
10 d,z = ouh(a,b,y)
11
12 print(a,b,y)
13 print(d,z)
```

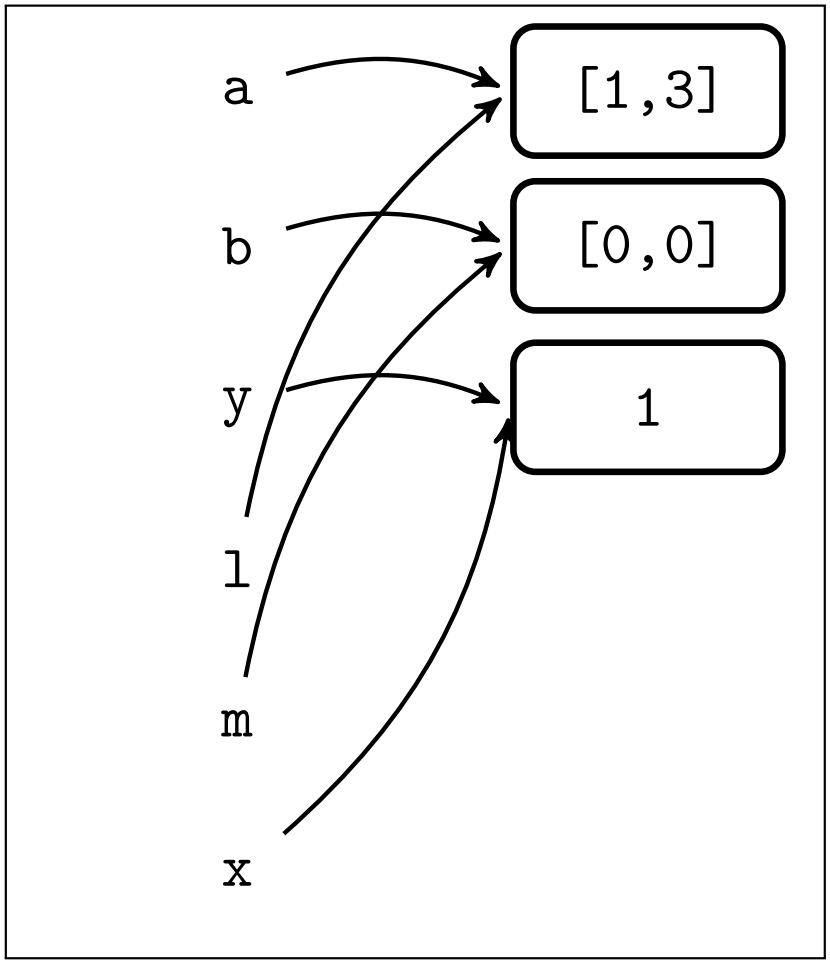
```
[1, 3] [0, 0] 1
[1, 3] 2
```



Implication pour les fonctions

```
1  def ouh(l,m,x) :
2      l.remove(2)
3      m = l
4      x +=1
5      return m,x
6
7  a = [1,2,3]
8  b = [0,0]
9  y = 1
10 d,z = ouh(a,b,y)
11
12 print(a,b,y)
13 print(d,z)
```

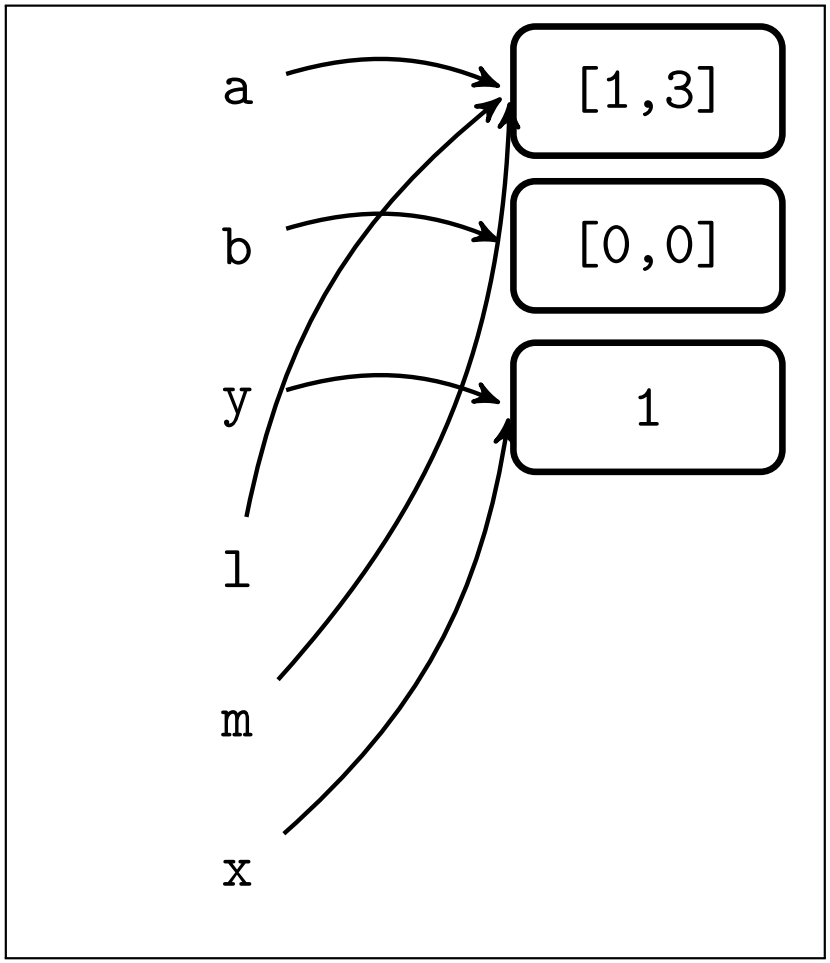
```
[1, 3] [0, 0] 1
[1, 3] 2
```



Implication pour les fonctions

```
1  def ouh(l,m,x) :
2      l.remove(2)
3      m = l
4      x +=1
5      return m,x
6
7  a = [1,2,3]
8  b = [0,0]
9  y = 1
10 d,z = ouh(a,b,y)
11
12 print(a,b,y)
13 print(d,z)
```

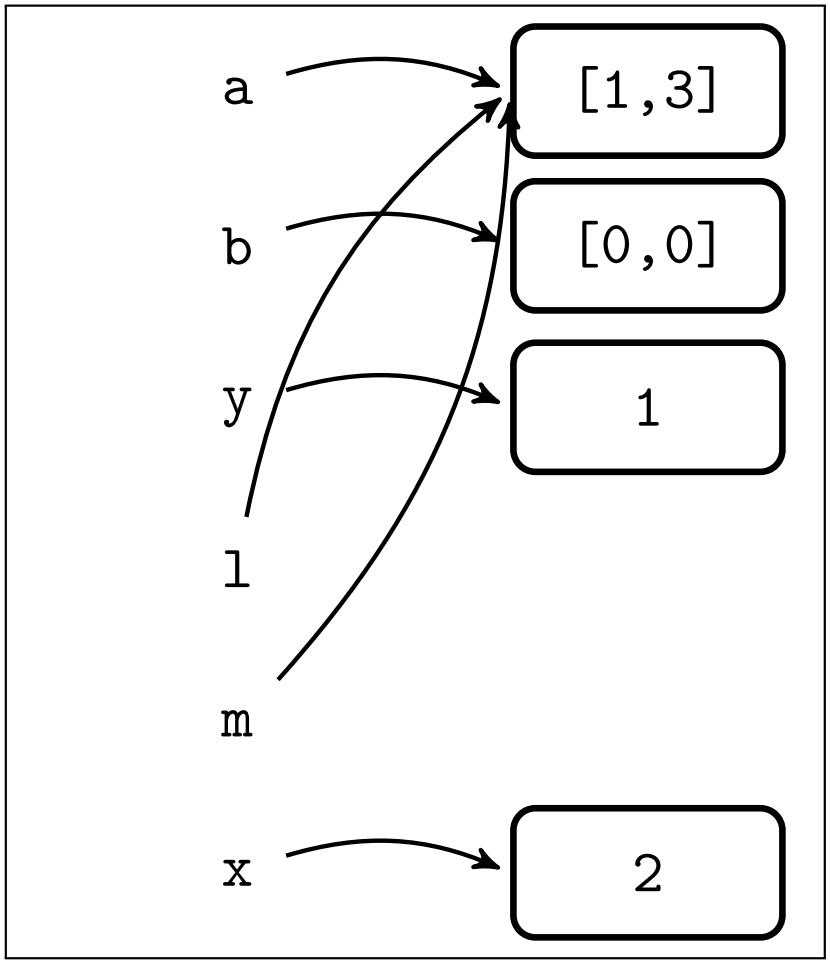
```
[1, 3] [0, 0] 1
[1, 3] 2
```



Implication pour les fonctions

```
1  def ouh(l,m,x) :
2      l.remove(2)
3      m = l
4      x +=1
5      return m,x
6
7  a = [1,2,3]
8  b = [0,0]
9  y = 1
10 d,z = ouh(a,b,y)
11
12 print(a,b,y)
13 print(d,z)
```

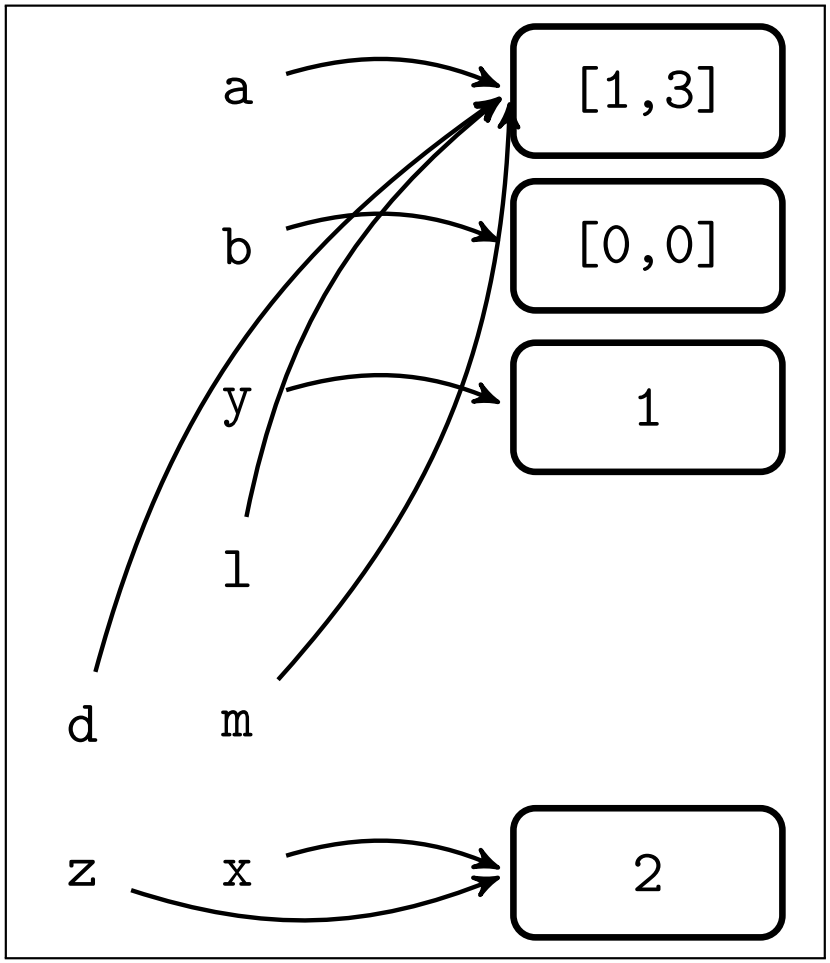
```
[1, 3] [0, 0] 1
[1, 3] 2
```



Implication pour les fonctions

```
1  def ouh(l,m,x) :
2      l.remove(2)
3      m = l
4      x +=1
5      return m,x
6
7  a = [1,2,3]
8  b = [0,0]
9  y = 1
10 d,z = ouh(a,b,y)
11
12 print(a,b,y)
13 print(d,z)
```

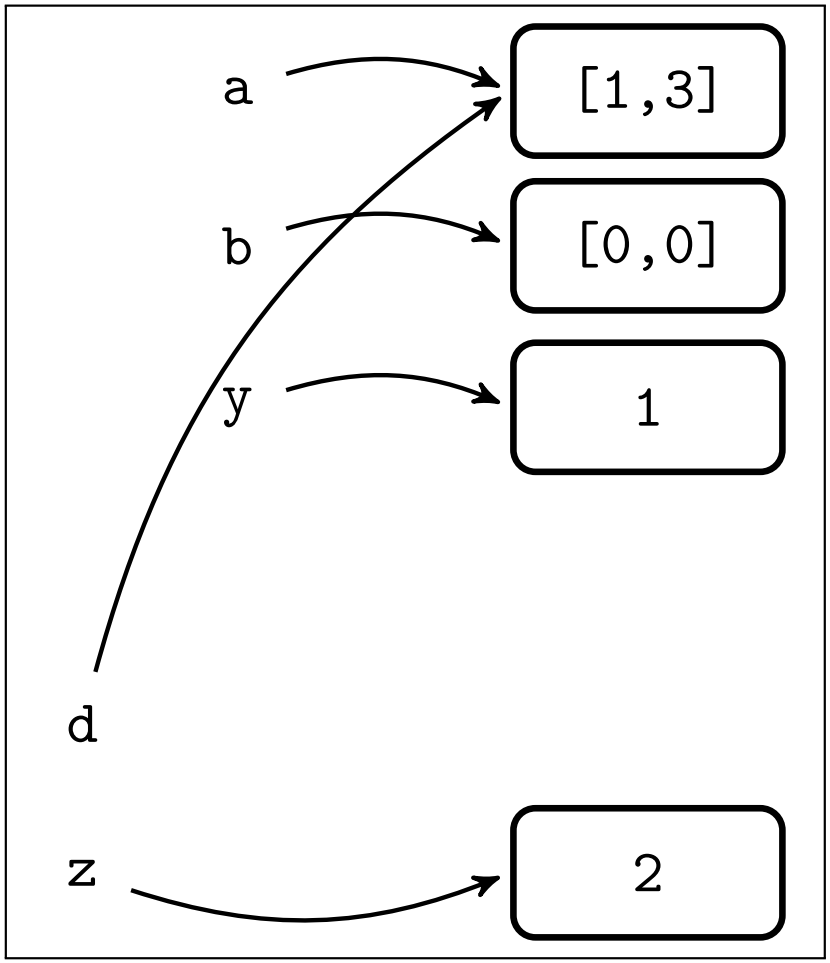
```
[1, 3] [0, 0] 1
[1, 3] 2
```



Implication pour les fonctions

```
1  def ouh(l,m,x) :
2      l.remove(2)
3      m = l
4      x +=1
5      return m,x
6
7  a = [1,2,3]
8  b = [0,0]
9  y = 1
10 d,z = ouh(a,b,y)
11
12 print(a,b,y)
13 print(d,z)
```

```
[1, 3] [0, 0] 1
[1, 3] 2
```



Hashable, immuable, clés de dictionnaires

- Les dictionnaires n'acceptent pas les listes comme clés :

Hashable, immuable, clés de dictionnaires

- Les dictionnaires n'acceptent pas les listes comme clés :

:

  - cela n'aurait aucun sens :

```
1 l = [1,2]
2 d = { [1,3] : 2, l : 3}
3 l[1] = 3
4 print(d) # ???
```

Hashable, immuable, clés de dictionnaires

- Les dictionnaires n'acceptent pas les listes comme clés :

:

  - cela n'aurait aucun sens :

```
1 l = [1,2]
2 d = { [1,3] : 2, l : 3}
3 l[1] = 3
4 print(d) # ???
```

  - en pratique Python refuse :

```
TypeError: unhashable type: 'list'
```

Hashable, immuable, clés de dictionnaires

- Les dictionnaires n'acceptent pas les listes comme clés :

:

  - cela n'aurait aucun sens :

```
1 l = [1,2]
2 d = { [1,3] : 2, l : 3}
3 l[1] = 3
4 print(d) # ???
```

  - en pratique Python refuse :
    `TypeError: unhashable type: 'list'`
  - La manière dont Python stocke les éléments des dictionnaires, pour être efficace, nécessite des objets fixés.

Hashable, immuable, clés de dictionnaires

- Les dictionnaires n'acceptent pas les listes comme clés :

:

  - cela n'aurait aucun sens :

```
1 l = [1,2]
2 d = { [1,3] : 2, l : 3}
3 l[1] = 3
4 print(d) # ???
```

  - en pratique Python refuse :
    `TypeError: unhashable type: 'list'`
  - La manière dont Python stocke les éléments des dictionnaires, pour être efficace, nécessite des objets fixés.

- `hashable` Tous les types immuables sont hashable et donc acceptés comme clés.

Hiérarchie (partielle) des types en Python

https://docs.python.org/fr/3/reference/datamodel.html

- None **immuable**
- numbers.Number
  - numbers.Integral
    - int **immuable**
    - bool **immuable**
  - numbers.Real = float **immuable**
  - numbers.Complex = complex **immuable**
- Sequences
  - listes **modifiable**
  - chaînes de caractères **immuable**
  - tuples **immuable**
- Ensembles set **modifiable**
- Dictionnaires **modifiable**

Tuples (1/3)

- **Un tuple est une séquence immuable.**

Tuples (1/3)

- **Un tuple est une séquence immuable.**

C'est l'équivalent non modifiable (immuable) de la liste.

Tuples (1/3)

- **Un tuple est une séquence immuable.**

C'est l'équivalent non modifiable (immuable) de la liste.

Illustration de la syntaxe :

```
1 x = (1, 2, 3)
2 print(x)
3 for y in x:
4     print(y)
5 for t in range(len(x)):
6     print(x[t])
```

- remarquez les `(` et `)`, au lieu des `[` et `]`,

Tuples (1/3)

- **Un tuple est une séquence immuable.**

C'est l'équivalent non modifiable (immuable) de la liste.

Illustration de la syntaxe :

```
1 x = (1, 2, 3)
2 print(x)
3 for y in x:
4     print(y)
5 for t in range(len(x)):
6     print(x[t])
```

- remarquez les `(` et `)`, au lieu des `[` et `]`,
- peuvent donc être utilisé comme clés des dictionnaires (cf TD).

Tuples (2/3)

- Toute tentative de modification de la valeur d'un tuple...

```
1 x = (1, 2, 3)
2 x[1] = 0
```

...se traduit par une erreur :

```
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
TypeError: 'tuple' object does not support item assignment
```

Tuples (2/3)

- Toute tentative de modification de la valeur d'un tuple...

```
1 x = (1, 2, 3)
2 x[1] = 0
```

...se traduit par une erreur :

```
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
TypeError: 'tuple' object does not support item assignment
```

- Par contre on peut bien sûr faire :

```
1 x = (1, 2, 3)
2 x = (1, 0, 3)
```

Tuples (3/3)

- Partage des fonctions avec les listes :
  - fonction `len` pour la longueur
  - slicing et accès par la fin avec les indices négatifs
  - parcours avec `for val in tuple`
  - `index` pour trouver la position d'un élément
  - `count` pour trouver le nombre d'occurrences
  - les opérateurs `in` et `not in` pour tester la présence (ou l'absence) d'un élément dans tuple
  - `min` et `max`
  - la concaténation par l'opérateur +

= celles qui ne modifient pas le tuple...

Tuples vs listes : opérateur +=

```
1 t = [1, 2]
2 g = t
3 t += [4, 5]
4 print(t)
5 print(g)
```

```
[1, 2, 4, 5]
[1, 2, 4, 5]
```

Tuples vs listes : opérateur +=

```
1 t = [1, 2]
2 g = t
3 t += [4, 5]
4 print(t)
5 print(g)
```

```
[1, 2, 4, 5]
[1, 2, 4, 5]
```

```
1 t = (1, 2)
2 g = t
3 t += (4, 5)
4 print(t)
5 print(g)
```

```
(1, 2, 4, 5)
(1, 2)
```

Tuples vs listes : opérateur +=

```
1 t = [1, 2]
2 g = t
3 t += [4, 5]
4 print(t)
5 print(g)
```

```
[1, 2, 4, 5]
[1, 2, 4, 5]
```

```
1 t = (1, 2)
2 g = t
3 t += (4, 5)
4 print(t)
5 print(g)
```

```
(1, 2, 4, 5)
(1, 2)
```

```
1 t = (1, 2)
2 g = t
3 t = t+ (4, 5)
4 print(t)
5 print(g)
```

```
(1, 2, 4, 5)
(1, 2)
```

Tuples vs listes : opérateur +=

```
1 t = [1, 2]
2 g = t
3 t += [4, 5]
4 print(t)
5 print(g)
```

```
[1, 2, 4, 5]
[1, 2, 4, 5]
```

```
1 t = (1, 2)
2 g = t
3 t += (4, 5)
4 print(t)
5 print(g)
```

```
(1, 2, 4, 5)
(1, 2)
```

```
1 t = [1, 2]
2 g = t
3 t = t+ [4, 5]
4 print(t)
5 print(g)
```

```
[1, 2, 4, 5]
[1, 2]
```

```
1 t = (1, 2)
2 g = t
3 t = t+ (4, 5)
4 print(t)
5 print(g)
```

```
(1, 2, 4, 5)
(1, 2)
```

Tuples vs listes : opérateur +=

- `+=` **n'a pas le même sens pour les listes et les tuples.**
  - pour les *tuples* :
    `a += b` est équivalent à `a = a + b`
  - pour les *listes* :
    `a += b` est équivalent à `a.append(b)`
    qui *n'est pas* équivalent à `a = a + b`

Tuples vs listes : opérateur +=

- `+=` **n'a pas le même sens pour les listes et les tuples.**
  - pour les *tuples* :
    `a += b` est équivalent à `a = a + b`
  - pour les *listes* :
    `a += b` est équivalent à `a.append(b)`
    qui *n'est pas* équivalent à `a = a + b`

Au bilan

- Deux familles de types différents :
  - **modifiables**,
  - **immuables**,
- des utilisations différentes,
- les **tuples** = version immuable des listes.