Structure de données élémentaires en Python : listes, tuples, chaînes de caractères, dictionnaires

DIU Enseigner l'Informatique au Lycée

G. Huard, L. Mounier, C. Parent-Vigouroux, A. Rasse, B. Wack

UFR IM2AG, Université Grenoble Alpes

mai 2019

Remerciements : Stéphane Gonnord, Matthieu Moy, Marc de Falco

Plan

Tableaux et listes
- Manipulations de base
- Utilisation pour implémenter des structures de données
- Listes définies par compréhension
- Tranchage (slicing)

Itérations
- Objets itérables
- Autres structures itérables usuelles

À propos de références
- Aliasing
- Références vs. valeurs

Plan

Tableaux et listes
- Manipulations de base
- Utilisation pour implémenter des structures de données
- Listes définies par compréhension
- Tranchage (slicing)

Itérations
- Objets itérables
- Autres structures itérables usuelles

À propos de références
- Aliasing
- Références vs. valeurs

Les tableaux Python

- Hétérogènes
- Dimension donnée par `len`
- Accès aléatoire, **en arrière**
- **Modifiables** (on dit aussi *mutable*)

```
>>> L = [1, True, 3]
>>> len(L)
3
>>> L[0]
1
>>> L[-1]
3
>>> L[-2]
True
>>> L[0] = 42
>>> L
[42, True, 3]
```

Mais aussi...

Ajout en queue append

- Prend l'élément à rajouter en argument
- Modifie le tableau
- Ne renvoie rien

```
>>> L = [42, True, 3]
>>> L.append(4)
>>> L
[42, True, 3, 4]
```

Retrait en queue pop

- Ne prend pas d'argument
- Modifie le tableau
- Renvoie l'élément retiré

```
>>> L.pop()
4
>>> L
[42, True, 3]
```

Mais aussi...

Ajout en queue append

- Prend l'élément à rajouter en argument
- Modifie le tableau
- Ne renvoie rien

```
>>> L = [42, True, 3]
>>> L.append(4)
>>> L
[42, True, 3, 4]
```

Retrait en queue pop

- Ne prend pas d'argument
- Modifie le tableau
- Renvoie l'élément retiré

```
>>> L.pop()
4
>>> L
[42, True, 3]
```

En Python, on parle plutôt de listes que de tableaux.

Quelques explications

Une liste est un objet

- La plupart des opérations de manipulation sont donc des méthodes.
  `>>> nomdelaliste.nomdelamethode(...)`
- Comme pour les modules :

```
>>> dir(maliste)              # liste des methodes
>>> help(maliste.methode) # doc d'une methode
```

Attention : ce ne sont pas des listes chaînées

- accès et modification aléatoire en temps constant
- ajout et suppression en temps constant amorti

Concaténation

La **concaténation** + crée une nouvelle liste :

```
>>> a = [2, 4, 6]
>>> b = [8, 10, 12]
>>> c = a + b
>>> c
[2, 4, 6, 8, 10, 12]
```

Concaténation

La **concaténation** + crée une nouvelle liste :

```
>>> a = [2, 4, 6]
>>> b = [8, 10, 12]
>>> c = a + b
>>> c
[2, 4, 6, 8, 10, 12]
```

Concaténations multiples : `L * n` concatène *n* instances de `L`.

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

Autres opérateurs

- L'opérateur **in** permet de tester l'appartenance à une liste :

```
>>> 7 in [3, 5, 7, 11, 13]
True
>>> if v in premiers:
...     print(v, 'est un nombre premier.')
```

On dispose aussi de `v not in L`.

Autres opérateurs

- L'opérateur `in` permet de tester l'appartenance à une liste :

```
>>> 7 in [3, 5, 7, 11, 13]
True
>>> if v in premiers:
...     print(v, 'est un nombre premier.')
```

  On dispose aussi de `v not in L`.
- D'autres méthodes utiles : `count`, `index`, `sort`, `sum`...
- On peut aussi donner à pop l'indice de l'élément à supprimer :

```
>>> c.pop(2)
6
>>> c
[2, 4, 8, 10, 12]
```

Plan

Tableaux et listes
- Manipulations de base
- Utilisation pour implémenter des structures de données
- Listes définies par compréhension
- Tranchage (slicing)

Itérations
- Objets itérables
- Autres structures itérables usuelles

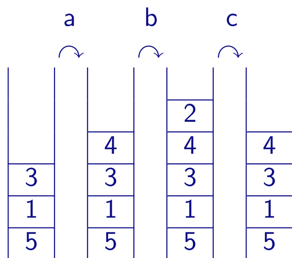
À propos de références
- Aliasing
- Références vs. valeurs

Implémentation d'une pile

Idée intuitive

Semblable à une pile d'assiettes.
Politique : **Dernier arrivé, premier servi** (LIFO).

a. insérer 4
b. insérer 2
c. extraire



Interface d'une pile

$$
\begin{array}{rcl}
\text{PileVide} & : & () \rightarrow \mathit{Pile} \\
\text{Empiler} & : & \mathit{Element} \times \mathit{Pile} \rightarrow \mathit{Pile} \\
\\
\text{EstVide} & : & \mathit{Pile} \rightarrow \mathit{bool} \\
\text{Sommet} & : & \mathit{Pile} \rightarrow \mathit{Element} \\
\text{Dépiler} & : & \mathit{Pile} \rightarrow \mathit{Pile}
\end{array}
$$

Interface d'une pile

$$
\begin{aligned}
\text{PileVide} &: () \to \mathit{Pile} \\
\text{Empiler} &: \mathit{Element} \times \mathit{Pile} \to \mathit{Pile} \\
\text{EstVide} &: \mathit{Pile} \to \mathit{bool} \\
\text{Sommet} &: \mathit{Pile} \to \mathit{Element} \\
\text{Dépiler} &: \mathit{Pile} \to \mathit{Pile}
\end{aligned}
$$

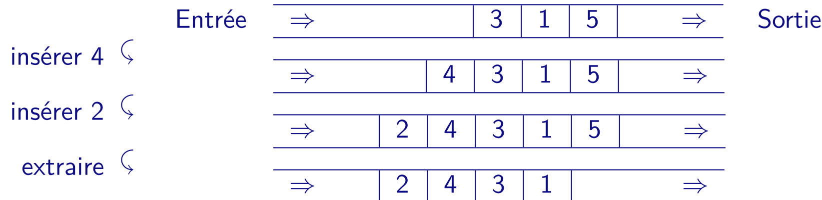
En Python avec des listes :

```
def PileVide(): return []
def Empiler(x,p): p.append(x)
def EstVide(p): return (p == [])
def Sommet(p): return p[-1]
def Depiler(p): p.pop()
```

Attention à la tentation : implémentation d'une file

Idée intuitive

Correspond à la « file d'attente » des supermarchés.
Politique : **Premier arrivé, premier servi** (FIFO).



Interface d'une file

$$\begin{aligned}
\text{FileVide} &: () \rightarrow \mathit{File} \\
\text{Enfiler} &: \mathit{Element} \times \mathit{File} \rightarrow \mathit{File} \\
\text{EstVide} &: \mathit{File} \rightarrow \mathit{bool} \\
\text{Tête} &: \mathit{File} \rightarrow \mathit{Element} \\
\text{Défiler} &: \mathit{File} \rightarrow \mathit{File}
\end{aligned}$$

Il est tentant de faire « comme » pour les piles :

```
def FileVide(): return []
def Enfiler(x,f): f.append(x)
def EstVide(f): return (f == [])
def Tete(f): return f[0]
def Defiler(f): f.pop(0)
```

Interface d'une file

$$\begin{aligned} \text{FileVide} &: () \to \mathit{File} \\ \text{Enfiler} &: \mathit{Element} \times \mathit{File} \to \mathit{File} \\ \text{EstVide} &: \mathit{File} \to \mathit{bool} \\ \text{Tête} &: \mathit{File} \to \mathit{Element} \\ \text{Défiler} &: \mathit{File} \to \mathit{File} \end{aligned}$$

Il est tentant de faire « comme » pour les piles :

```
def FileVide(): return []
def Enfiler(x,f): f.append(x)
def EstVide(f): return (f == [])
def Tete(f): return f[0]
def Defiler(f): f.pop(0)
```

Mais pop avec un argument a un coût linéaire en la taille de la liste, comme la plupart de ces opérateurs prédéfinis.

Plan

Tableaux et listes
- Manipulations de base
- Utilisation pour implémenter des structures de données
- Listes définies par compréhension
- Tranchage (slicing)

Itérations
- Objets itérables
- Autres structures itérables usuelles

À propos de références
- Aliasing
- Références vs. valeurs

Listes définies «par compréhension»

- De manière analogique à la notation ensembliste $\{3i^2 + 1 \mid 1 \leq i \leq 9\}$ en mathématiques :

```
>>> [ 3*i**2 + 1 for i in range(1,10) ]
[4, 13, 28, 49, 76, 109, 148, 193, 244]
```

Listes définies «par compréhension»

- De manière analogique à la notation ensembliste $\{3i^2 + 1 \mid 1 \leq i \leq 9\}$ en mathématiques :

  ```
  >>> [ 3*i**2 + 1 for i in range(1,10) ]
  [4, 13, 28, 49, 76, 109, 148, 193, 244]
  ```

- Il est possible de combiner les compréhensions :

```
>>> [(i,j) for i in range(4) for j in range(i)]
[(1, 0), (2, 0), (2, 1), (3, 0), (3, 1), (3, 2)]
```

L'indice qui varie le plus vite est celui qui apparaît en dernier : le plus simple est de se souvenir que cette construction remplace deux boucles `for` imbriquées dans le même ordre :

```
for i in range(4):
  for j in range(i):
    ...
```

Pour des listes de listes

Attention, c'est différent si les compréhensions sont vraiment emboîtées :

```
>>> [[(i,j) for i in range(4)] for j in range(i)]
NameError: name 'i' is not defined
```

```
>>> [[(i,j) for j in range(i)] for i in range(4)]
[[], [(1, 0)], [(2, 0), (2, 1)], [(3, 0), (3, 1),
```

Pour des listes de listes

Attention, c'est différent si les compréhensions sont vraiment emboîtées :

```
>>> [[(i,j) for i in range(4)] for j in range(i)]
NameError: name 'i' is not defined
```

```
>>> [[(i,j) for j in range(i)] for i in range(4)]
[[], [(1, 0)], [(2, 0), (2, 1)], [(3, 0), (3, 1),
```

On peut également également rajouter un `if` :

- soit pour sélectionner :

```
>>> [i**2 for i in range(10) if i%3 != 0]
[1, 4, 16, 25, 49, 64]
```

- soit pour obtenir des valeurs conditionnelles :

```
>>> [i if i%2==0 else -i for i in range(10)]
[0, -1, 2, -3, 4, -5, 6, -7, 8, -9]
```

Plan

Tableaux et listes
- Manipulations de base
- Utilisation pour implémenter des structures de données
- Listes définies par compréhension
- Tranchage (slicing)

Itérations
- Objets itérables
- Autres structures itérables usuelles

À propos de références
- Aliasing
- Références vs. valeurs

Principes de base

On peut découper une liste en « tranches » (*slicing* en anglais).
Par exemple, pour obtenir les éléments d'indices 2 à 4 d'une liste :

```
>>> L
['do', 're', 'mi', 'fa', 'sol', 'la', 'si', 'do']
>>> L[2:4]
['mi, 'fa']
```

Attention : comme d'habitude en Python, intervalle fermé à gauche et ouvert à droite.
`L[i:j]` désigne les éléments d'indice compris dans $[\![i;j[\![$.

En particulier `L[i:i]` est une liste vide.

À noter : une tranche est une copie d'une sous-liste (voir plus loin la section sur les références)

Syntaxe générale

`L[start:stop:step]`

- `start` est l'indice où commencer
- `stop` est l'indice avant lequel il faut s'arrêter
- `step` est l'incrément à donner à l'indice à chaque étape

Slicing avec incrément

`L[1:11:2]` désigne les éléments de rang 1,3,5,7,9.

Syntaxe générale

`L[start:stop:step]`

- `start` est l'indice où commencer
- `stop` est l'indice avant lequel il faut s'arrêter
- `step` est l'incrément à donner à l'indice à chaque étape

Slicing avec incrément

`L[1:11:2]` désigne les éléments de rang 1,3,5,7,9.

Par défaut `start` vaut 0, `stop` vaut `len(L)` et `step` vaut 1.

Bornes implicites

- `L[1::2]` désigne les éléments de rang impair
- `L[::2]` ceux de rang pair

Utilisation avancée

On peut également *remplacer* une tranche d'une liste :

```
>>> L[3:6] = [1, 2, 3, 4, 5]
>>> L
['do', 're', 'mi', 1, 2, 3, 4, 5, 'si', 'do']
```

Utilisation avancée

On peut également *remplacer* une tranche d'une liste :

```
>>> L[3:6] = [1, 2, 3, 4, 5]
>>> L
['do', 're', 'mi', 1, 2, 3, 4, 5, 'si', 'do']
```

Cela permet :

- d'insérer des éléments dans une liste

```
>>> a = [2, 4, 6, 8]
>>> a[2:2] = [18]
>>> a
[2, 4, 18, 6, 8]
```

- ou d'en supprimer

```
>>> a[2:4] = []
>>> a
[2, 4, 8]
```

Plan

Tableaux et listes
- Manipulations de base
- Utilisation pour implémenter des structures de données
- Listes définies par compréhension
- Tranchage (slicing)

Itérations
- Objets itérables
- Autres structures itérables usuelles

À propos de références
- Aliasing
- Références vs. valeurs

Rappel : itération simple

En Python, un objet est dit *itérable* s'il est capable de renvoyer un par un tous ses éléments. Quand on écrit `for x in a`, la variable x prend successivement toutes les valeurs des éléments de a.

```
>>> for couleur in L :
        print(couleur)
bleu
vert
rouge
```

On peut itérer sur les `range` mais aussi sur les listes, les chaînes...

Exemple : extraction d'une sous-liste

Seuil

Cette fonction prend une liste `L` d'entiers et renvoie la sous-liste des entiers dépassant un seuil `s`.

```
def seuil(L, s):
    M = []
    for x in L:
        if x > s:
            M.append(x)
    return M
```

Le Pythonneux écrira plutôt `[ x for x in L if x>s ]` !

Double parcours

Bégaiement

Cette fonction prend une liste de chaînes et affiche chaque caractère de chaque chaîne deux fois de suite.

```
def begaie(L):
    for chaine in L:
        for carac in chaine:
            print (carac, carac)
```

Itérateurs « avancés »

- **reversed**(it) renvoie les mêmes éléments en ordre inverse.

```
>>> L = [1, 17, 'bleu']
>>> for x in reversed(L): print(x)
bleu
17
1
```

- **enumerate**(it) renvoie les couples (i, it[i]).

```
s = 'python'
for i, x in enumerate(s):
    print('Le', i, 'ème caractère est', x)
```

Plan

Tableaux et listes
- Manipulations de base
- Utilisation pour implémenter des structures de données
- Listes définies par compréhension
- Tranchage (slicing)

Itérations
- Objets itérables
- Autres structures itérables usuelles

À propos de références
- Aliasing
- Références vs. valeurs

Les tuples (*p*-uplets)

- Pratiques pour regrouper des données
- Hétérogènes
- Accès aléatoire (ou direct) (indexé de 0 à n-1)
- Déconstruction
- **Non modifiables**

```
>>> t = (1, 2, 3)
>>> (True, 42)
(True, 42)
>>> t[0]
1
>>> (x, y) = (2, 3)
>>> x, y = 2, 3
>>> x, y = y, x
>>> x
3
>>> t[0] = 0
TypeError: 'tuple'object does not support item assignment
```

Les chaînes de caractères

- Les chaînes de caractères sont aussi **non modifiables**
- Quelques opérations usuelles :

```
>>> a = 'Lycée'
>>> len(a)
5
>>> a.upper()  # nouvelle chaine
'LYCÉE'
>>> a[0]
'L'
>>> a[2:4]
'cé'
>>> '-'.join(['Un', 'mot', 'et', 'un', 'autre'])
'Un-mot-et-un-autre'
```

Mais aussi...

- Les ensembles :
  - non ordonnés
  - ne contiennent pas de doublons

```
>>> {'zero', 1, 1, 'deux'}
{1, 'zero', 'deux'}
```

- Un générateur est une fonction qui peut
  - s'interrompre et renvoyer un résultat avec le mot-clé `yield`,
  - puis reprendre là où elle s'était arrêtée.

On peut utiliser un générateur comme itérable :

```
def naturels():
    n=0
    while True:
        yield n
        n += 1
```

```
for i in naturels():
    print(i)
0
1
2
3...
```

Les dictionnaires

- Associent des valeurs à des clés.

```
>>> d = { 'cle1':17, 42:'val2' }
```

- Les clés doivent être de type non modifiable (souvent entiers et/ou chaînes de caractères)

```
>>> d['abc'] = 'def'
```

- Une même clé ne peut apparaître qu'en un exemplaire

```
>>> d[42] = 'toto'
>>> d
{'cle1':17, 42:'toto', 'abc':'def'}
```

Exemples d'utilisations :

- Tableaux non contigüs, ou remplis dans le désordre
- Enregistrements nommés

Opérateurs sur les dictionnaires

- L'accès est rapide (temps constant en moyenne)

```
>>> d['cle1']
17
>>> 'abc' in d
True
```

- Parcours :

```
>>> for k in d:  # dans l'ordre d'insertion
        print(k, '--->', d[k])
cle1 ---> 17
42 ---> toto
abc ---> def
```

Plan

Tableaux et listes
- Manipulations de base
- Utilisation pour implémenter des structures de données
- Listes définies par compréhension
- Tranchage (slicing)

Itérations
- Objets itérables
- Autres structures itérables usuelles

À propos de références
- Aliasing
- Références vs. valeurs

Les cauchemars commencent : Références implicites

Les listes, comme tous les autres types modifiables, sont manipulées par référence.

Version int

```
>>> x = 3
>>> y = x
>>> x = x + 1
>>> x
4
>>> y
3
```

Version liste

```
>>> a = [3]
>>> b = a
>>> a.append(1)
>>> a
[3, 1]
>>> b
[3, 1]
```

Les deux variables `a` et `b` *pointent* vers la même liste.

Attention au passage de paramètres

Puisque *tout* est référence :

```
def f(t):
    t[2] = 0

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

Une fonction est susceptible de modifier ses arguments pour tous les types de données mutables.

Regardons sous le capot

- Un objet python est représenté en machine par une suite d'octets dans la mémoire vive.
- Lorsqu'on veut faire référence à un objet on le fait par son adresse en mémoire.
- On peut obtenir l'adresse en mémoire d'un objet avec la fonction `id`.

```
>>> a = [11, 22, 33]
>>> id(a)
3072380108
```

(NB : l'adresse en question dépend de beaucoup de facteurs, vous n'obtiendrez pas la même que ci-dessus)

Regardons sous le capot

- Un objet python est représenté en machine par une suite d'octets dans la mémoire vive.
- Lorsqu'on veut faire référence à un objet on le fait par son adresse en mémoire.
- On peut obtenir l'adresse en mémoire d'un objet avec la fonction **id**.

```
>>> a = [11, 22, 33]
>>> id(a)
3072380108
```

(NB : l'adresse en question dépend de beaucoup de facteurs, vous n'obtiendrez pas la même que ci-dessus)

Une variable Python ne contient jamais un objet mais seulement l'adresse d'un objet (on dit que la variable pointe vers l'objet).

Une copie paresseuse

En cas de copie de variable, Python ne copie en fait que l'adresse :

```
>>> id(a)
3072380108
>>> b = a
>>> id(b)
3072380108
```

Une copie paresseuse

En cas de copie de variable, Python ne copie en fait que l'adresse :

```
>>> id(a)
3072380108
>>> b = a
>>> id(b)
3072380108
```

Deux conséquences majeures :

- La copie se fait en temps constant.
  C'est en particulier le cas lors d'un appel de fonction.
- À la suite de la copie, les deux variables pointent vers le même objet (elles sont alias l'une de l'autre).
  Toute modification de l'une sera visible de l'autre !

Les questions d'aliasing sont récurrentes en informatique, et loin d'être spécifiques à Python.

Cas concrets qu'on rencontrera tôt ou tard

- Considérons cet exemple :

```
>>> m = [[0]*2]*2
>>> m
[[0, 0], [0, 0]]
```

Cas concrets qu'on rencontrera tôt ou tard

- Considérons cet exemple :

```
>>> m = [[0]*2]*2
>>> m
[[0, 0], [0, 0]]
```

- Si, maintenant, on affectait la case (0,0) :

```
>>> m[0][0] = 1
>>> m
[[1, 0], [1, 0]]
```

  - Argh ! Comment expliquer ça ?

Cas concrets qu'on rencontrera tôt ou tard

- Considérons cet exemple :

```
>>> m = [[0]*2]*2
>>> m
[[0, 0], [0, 0]]
```

- Si, maintenant, on affectait la case (0,0) :

```
>>> m[0][0] = 1
>>> m
[[1, 0], [1, 0]]
```

  - Argh ! Comment expliquer ça ?
- Python va d'abord évaluer l'expression `[0]*2`, et comme c'est une liste va utiliser la référence du résultat pour l'opération suivante

Cas concrets qu'on rencontrera tôt ou tard (2)

- Que valent a et b après l'exécution du code suivant :

```
>>> a = [11, 22, 33]
>>> b = [7, 14, a]
>>> b
[7, 14, [11, 22, 33]]
>>> b[2][1]=17
>>> a[0] = 19
```

Cas concrets qu'on rencontrera tôt ou tard (2)

- Que valent a et b après l'exécution du code suivant :

```
>>> a = [11, 22, 33]
>>> b = [7, 14, a]
>>> b
[7, 14, [11, 22, 33]]
>>> b[2][1]=17
>>> a[0] = 19

>>> a
[19, 17, 33]
>>> b
[7, 14, [19, 17, 33]]
```

Cas concrets qu'on rencontrera tôt ou tard (2)

- Que valent `a` et `b` après l'exécution du code suivant :

```
>>> a = [11, 22, 33]
>>> b = [7, 14, a]
>>> b
[7, 14, [11, 22, 33]]
>>> b[2][1]=17
>>> a[0] = 19

>>> a
[19, 17, 33]
>>> b
[7, 14, [19, 17, 33]]
```

- Pour obtenir une *vraie* copie :
`y = copy.copy(x)` après un `import copy`

Plan

Tableaux et listes
- Manipulations de base
- Utilisation pour implémenter des structures de données
- Listes définies par compréhension
- Tranchage (slicing)

Itérations
- Objets itérables
- Autres structures itérables usuelles

À propos de références
- Aliasing
- Références vs. valeurs

Tout est référence, mais ...

En Python, tout est référence :

- Après `x = y`, les variables `x` et `y` représentent le même objet (même **`id`**)
- Après `x = y + z`, on crée un nouvel objet de valeur `y + z`, dont l'**`id`** est stocké dans `x`.
- Après `x.methode(...)` ou `x[...] = ...`
  la valeur de `x` peut être modifiée
  mais `x` ne change pas d'**`id`**.
- Avec des non-mutables (nombres, tuples, chaînes), il n'y a que l'affectation pour modifier une variable !
  Donc le partage de mémoire est "transparent".

Modification de référence ou modification de valeur ?

Les deux morceaux de code suivants sont-ils équivalents ?

```
liste1 = [1, 2, 3]

liste1.append(42)
```

```
liste2 = [1, 2, 3]

liste2 = liste2 + [42]
```

Modification de référence ou modification de valeur ?

Les deux morceaux de code suivants sont-ils équivalents ?

```
liste1 = [1, 2, 3]

liste1.append(42)
```

```
>>> id(liste1)
4105280840
>>> id(liste1)
4105280840
```

```
liste2 = [1, 2, 3]

liste2 = liste2 + [42]
```

```
>>> id(liste2)
4104487368
>>> id(liste2)
4104487432
```

Modification de référence ou modification de valeur ?

Les deux morceaux de code suivants sont-ils équivalents ?

```
liste1 = [1, 2, 3]

liste1.append(42)
```

```
>>> id(liste1)
4105280840
>>> id(liste1)
4105280840
```

```
liste2 = [1, 2, 3]

liste2 = liste2 + [42]
```

```
>>> id(liste2)
4104487368
>>> id(liste2)
4104487432
```

Voir aussi le déroulement sur `http://pythontutor.com/`

Et pour des types non modifiables

```
>>> a = 3

>>> b = a

>>> b = b + 1

>>> b
4
>>> a
3
```

Alors, lorsqu'on écrit `a=b`,
alias ou pas alias ?

Et pour des types non modifiables

```
>>> a = 3

>>> b = a

>>> b = b + 1

>>> b
4
>>> a
3
```

```
>>> id(a)
10771584
>>> id(b)
10771584
>>> id(b)
10771616
```

Alors, lorsqu'on écrit a=b, alias ou pas alias ?

- Oui, ce sont des alias à la copie
- Mais b+1 est un nouvel objet
- Les entiers sont immuables donc toute affectation modifie la référence

Passage d'arguments aux fonctions

Version int

```
>>> def f(x):
        x = x + 1
```

```
>>> a = 3
>>> f(a)
>>> a
3
```

Version liste

```
>>> def g(L):
        L.append(0)
```

```
>>> M = [1, 2, 3]
>>> g(M)
>>> M
[1, 2, 3, 0]
```

Passage d'arguments aux fonctions

Version int

```
        # id(x) == id(a)

>>> def f(x):
        x = x + 1



>>> a = 3
>>> f(a)
>>> a
3
```

Version liste

```
        # id(L) == id(M)

>>> def g(L):
        L.append(0)



>>> M = [1, 2, 3]
>>> g(M)
>>> M
[1, 2, 3, 0]
```

Passage d'arguments aux fonctions

Version int

```
>>> def f(x):
        x = x + 1

    # modifie id(x)
    # ET sa valeur

>>> a = 3
>>> f(a)
>>> a
3
```

Version liste

```
>>> def g(L):
        L.append(0)

    # ne modifie pas id(L)
    # MAIS modifie sa valeur

>>> M = [1, 2, 3]
>>> g(M)
>>> M
[1, 2, 3, 0]
```

Un exemple perturbant : la concaténation

Avec les listes, `a+b` construit une nouvelle liste :

```
>>> a = [2, 4, 6]
>>> b = [8, 10, 12]
>>> c = a
>>> id(a)
3072535468
>>> id(c)
3072535468
>>> a = a + b      # id(a) change !
>>> id(a)
3072535472
>>> id(c)
3072535468
>>> c
[2, 4, 6]
```

Concaténation “abrégée”

mais on peut aussi écrire `a += b` qui modifie a en place.
(On parle d'**extension** plutôt que de concaténation.)

```
>>> a = [2, 4, 6]
>>> b = [8, 10, 12]
>>> c = a
>>> id(a)
3072535468
>>> id(c)
3072535468
>>> a += b       # id(a) ne change pas !
>>> id(a)
3072535468
>>> c
[2, 4, 6, 8, 10, 12]
```

Ce n'est donc pas équivalent à `a = a + b`...