

Les instructions += et autres revisit  es

Compl  ment - niveau interm  diaire


Nous avons vu en deuxi  me semaine (S  quence "Les types num  riques") une premi  re introduction aux instructions += et ses d  riv  es comme *=, **=, etc.

Ces constructions ont une d  finition    g  om  trie variable

En C quand on utilise += (ou encore ++) on modifie la m  moire en place – historiquement, cet op  rateur permettait au programmeur d'aider    l'optimisation du code pour utiliser les instructions assembleur idoines.

Ces constructions en python s'inspirent clairement de C, aussi dans l'esprit ces constructions devraient fonctionner en **modifiant** l'objet r  f  renc   par la variable.

Mais les types num  riques en python ne sont **pas mutables**, alors que les listes le sont. Du coup le comportement de += est diff  rent selon qu'on l'utilise sur un nombre ou sur une liste. Voyons cela sur des exemples tr  s simples.

```
 # Premier exemple avec un entier

# on commence avec une r  f  rence partag  e
a = b = 3
a is b

# on utilise += sur une des deux variables
a += 1

# ceci n'a pas modifi   b
# c'est normal, l'entier n'est pas mutable

print a
print b
print a is b

# Deuxi  me exemple, cette fois avec une liste

# la m  me r  f  rence partag  e
a = b = []
a is b
```

```
# pareil, on fait += sur une des variables
a += [1]

# cette fois on a modifié a *et* b
# car += a pu modifier la liste en place
print a
print b
print a is b
```

Vous voyez donc que la sémantique de += (c'est bien entendu le cas pour toutes les autres formes d'instructions qui combinent l'affectation avec un opérateur) **est différente** suivant que l'objet référencé par le terme de gauche est **mutable ou immuable**.

Pour cette raison, c'est là une opinion personnelle, cette famille d'instructions n'est pas le trait le plus réussi dans le langage, et je ne recommande pas de l'utiliser.

Précision sur la définition de +=

Nous avons dit en première semaine, et en première approximation, que

```
[-] x += y
```

était équivalent à

```
[-] x = x + y
```

Au vu de ce qui précède, on voit que ce n'est **pas tout à fait exact**, puisque :

```
[-] # si on fait x += y sur une liste
# on fait un effet de bord sur la liste
# comme on vient de le voir

a = []
print "avant", id(a)
a += [1]
print "après", id(a)

# alors que si on fait x = x + y sur une liste
# on crée un nouvel objet liste

a = []
print "avant", id(a)
a = a + [1]
print "après", id(a)
```

Vous voyez donc que vis-à-vis des références partagées, ces deux façons de faire mènent à un résultat différent.

