Les instructions += et autres revisitées

Complément - niveau intermédiaire

Nous avons vu en deuxième semaine (Séquence "Les types numériques") une première introduction aux instructions `+=` et ses dérivées comme `*=`, `**=`, etc.

Ces constructions ont une définition à géométrie variable

En C quand on utilise `+=` (ou encore `++`) on modifie la mémoire en place - historiquement, cet opérateur permettait au programmeur d'aider à 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 référencé par la variable.

Mais les types numériques en python ne sont **pas mutables**, alors que les listes le sont. Du coup le comportement de `+=` est différent selon qu'on l'utilise sur un nombre ou sur une liste. Voyons cela sur des exemples très simples.

```
# Premier exemple avec un entier

# on commence avec une référence partagée
a = b = 3
a is b


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

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

print a
print b
print a is b


# Deuxième exemple, cette fois avec une liste

# la même référence partagée
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 avions 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.