L'opérateur `is`

Complément - niveau basique

Les opérateurs `is` et ==

Nous avons déjà parlé de l'opérateur == qui compare la valeur de deux objets.

Python fournit aussi un opérateur `is` qui permet de savoir si deux valeurs correspondent au même objet en mémoire.

Nous allons illustrer la différence entre ces deux opérateurs.

Pour bien comprendre cette différence, il faut se souvenir que

- == compare **la valeur** des deux objets, alors que
- `is` vérifie si les deux valeurs correspondent **au même objet**

Voyons ceci sur quelques exemples simples

```
# deux listes identiques
a = [1, 2]
b = [1, 2]
print '==', a == b
print 'is', a is b


# la même liste
a = [1, 2]
b = a
print '==', a == b
print 'is', a is b
```

Utilisez `is` plutôt que == lorsque c'est possible

La pratique usuelle est d'utiliser `is` lorsqu'on compare avec un objet qui est un singleton, comme typiquement `None`.

Par exemple on préfèrera écrire :

```
if a is None:
    print 'a est indéfini'
```

plutôt que

```
if a == None:
    print 'a est indéfini'
```

qui se comporte de la même manière (à nouveau, parce qu'on compare avec `None`), mais est légèrement moins lisible, et franchement moins pythonique :)

Notez aussi et surtout que `is` est **plus efficace** que ==. En effet `is` peut être évalué en temps constant, puisqu'il s'agit essentiellement de comparer les deux adresses. Alors que pour == il peut s'agir de parcourir toute un structure de données possiblement très complexe.

Complément - niveau intermédiaire

La fonction `id`

Pour bien comprendre le fonctionnement de `is` nous allons voir la fonction `id` qui retourne un identificateur unique pour chaque objet; un modèle mental acceptable est celui d'adresse mémoire.

```
id(True)
```

Comme vous vous en doutez, l'opérateur `is` peut être décrit formellement à partir de `id` comme ceci

(`a is b`) $\Longleftrightarrow$ (`id(a) == id(b)`)

Certains types de base sont des singletons

Un singleton est un objet qui n'existe qu'en un seul exemplaire dans la mémoire. Un usage classique des singletons en python est de minimiser le nombre d'objets immuables en mémoire. Voyons ce que cela nous donne avec des entiers

```
a = 3
b = 3
print 'a', id(a), 'b', id(b)
```

Tiens, c'est curieux, nous avons ici deux objets, que l'on pourrait penser différents, mais en fait ce sont les mêmes; `a` et `b` désignent le même objet python, et on a

```
a is b
```

Il se trouve que, dans le cas des petits entiers, python réalise une optimisation de l'utilisation de la mémoire. Quel que soit le nombre de variables dont la valeur est `3`, un seul objet correspondant à l'entier `3` est alloué et créé, pour éviter d'engorger la mémoire. On dit que l'entier `3` est implémenté comme un singleton; nous reverrons ceci en exercice.

On trouve cette optimisation avec quelques autres objets python, comme par exemple

```
a = ""
b = ""
a is b
```

Ou encore, plus surprenant:

```
a = "foo"
b = "foo"
a is b
```

Conclusion cette optimisation ne touche aucun type mutable (heureusement); pour les types immuables, il n'est pas extrêmement important de savoir en détail quels objets sont implémentés de la sorte.

Ce qui est par contre extrêmement important est de comprendre la différence entre `is` et `==`, et de les utiliser à bon escient au risque d'écrire du code fragile.

Pour en savoir plus

Aux étudiants de niveau avancé, nous recommandons la lecture de la section "Objects, values and types" dans la documentation python

https://docs.python.org/2/reference/datamodel.html#objects-values-and-types
(https://docs.python.org/2/reference/datamodel.html#objects-values-and-types)

qui aborde également la notion de "garbage collection", que nous n'aurons pas le temps de traiter dans ce MOOC.