Analyse approfondie des différences d'identifiants d'objets (`id`) entre REPL et script Python : Une perspective senior

ali.zainoul.az@gmail.com

18 juin 2025

1 Abstract

En Python, des différences subtiles mais fondamentales peuvent apparaître entre le comportement interactif de l'interpréteur (REPL) et l'exécution d'un script source. Ces différences, souvent invisibles pour les débutants, sont cruciales à comprendre pour tout développeur expérimenté. Cet article examine ces comportements à travers le prisme de la fonction `id()` et des objets immuables comme les `tuple`, en mettant en lumière les mécanismes internes de CPython : l'interning, le constant folding, et la gestion des objets constants.

2 Comprendre `id()` en profondeur

La fonction `id(obj)` retourne un identifiant unique de l'objet pendant sa durée de vie. Dans CPython, cet identifiant correspond à l'adresse mémoire en RAM de l'objet. Cela a des conséquences directes pour l'opérateur `is`, qui teste l'identité, et non l'égalité de valeur :

```
a = (1, 2, 3)
b = (1, 2, 3)
print(a == b)  # True, les valeurs sont egales
print(a is b)  # False ou True selon le contexte
print(id(a), id(b))  # Affiche les adresses memoire
```

2.1 REPL : un environnement interactif non prédictible

Dans le REPL, Python applique parfois des optimisations dynamiques. Contrairement au script, chaque ligne peut être évaluée indépendamment, rendant l'interning imprévisible. CPython ne garantit PAS que deux tuples identiques saisis l'un après l'autre soient stockés au même endroit :

```
>>> a = (1, 2, 3)
>>> b = (1, 2, 3)
>>> a is b
False   # comportement tres courant dans le REPL
```

2.2 Script source : compilation et optimisation à froid

Lorsqu'un script est exécuté depuis un fichier, l'interpréteur compile d'abord tout le module et effectue des optimisations dites "à froid", comme la fusion des constantes (*constant folding*). Dans ce contexte, Python est capable de réutiliser un seul objet pour plusieurs affectations identiques.

```
# script.py
a = (1, 2, 3)
b = (1, 2, 3)
print(a is b)   # Tres souvent True
```

Ce comportement est dû à l'évaluation des constantes et à leur stockage dans le champ `co_consts` du bytecode.

3 Les optimisations internes de CPython

3.1 1. Small Integer Caching

Python conserve en mémoire les entiers de `-5` à `256` pour éviter de recréer ces objets fréquemment. Ceci est vrai dans tous les contextes (REPL ou script) :

```
>>> a = 100
>>> b = 100
>>> a is b
True
```

3.2 2. String Interning

Les chaînes de caractères courtes, ou utilisées comme identifiants, peuvent être internées automatiquement. Cela rend les comparaisons plus rapides (puisqu'elles se font par adresse). Exemple :

```
>>> a = "python"
>>> b = "python"
>>> a is b
True
```

3.3 3. Constant Folding

CPython détecte à la compilation les expressions évaluables et les précompile. Cela vaut pour :

— les littéraux numériques : `2 + 3` devient `5`
— les tuples de constantes : `(1, 2, 3)` est replié dans `co_consts`

3.4 4. Tuple Interning (partiel et contextuel)

Contrairement aux chaînes ou entiers, les tuples ne sont pas systématiquement internés. Toutefois, dans un script, deux variables recevant exactement le même tuple littéral sont souvent affectées à la même instance mémoire :

```
a = (1, 2, 3)
b = (1, 2, 3)
print(a is b)  # True si optimisation active
```

4 Inspection avec le module `dis`

Le module `dis` permet d'examiner le bytecode de CPython. Il est essentiel pour un développeur senior souhaitant comprendre les optimisations internes.

```
import dis

def generate_tuple():
    t = (1, 2, 3)
    return t

print(generate_tuple.__code__.co_consts)
dis.dis(generate_tuple)
```

Sortie typique en Python 3.11+ :

```
(None, (1, 2, 3))
  3           0 RESUME                   0
  4           2 LOAD_CONST               1 ((1, 2, 3))
              4 STORE_FAST               0 (t)
  5           6 LOAD_FAST                0 (t)
              8 RETURN_VALUE
```

Cela montre que le tuple est compilé comme une constante et chargé via `LOAD_CONST`.

5 Recommandations pratiques

— Ne jamais s'appuyer sur l'identité d'objets (`is`) pour les objets immuables comme les tuples ou les chaînes, sauf si vous testez explicitement le singleton `None`.
— Toujours préférer `==` à `is` pour tester les équivalences de valeur.
— Comprendre que l'interning et le folding sont des optimisations d'implémentation, non garanties par la spécification du langage.
— Utiliser `dis` pour diagnostiquer et auditer des performances ou comportements inattendus.

6 Références et documentation

— Documentation Python – `id()`
— `sys.intern` pour l'interning des chaînes
— Stack Overflow : Why do identical tuples have different ids?
— Stack Overflow : Console assigns different IDs to identical immutable objects
— RealPython : Understanding the `id()` function in Python
— Source CPython - Implémentation des tuples (GitHub)

7 Conclusion

La gestion des identifiants d'objet en Python est intimement liée à l'implémentation sous-jacente de l'interpréteur (souvent CPython). Les différences entre l'exécution en REPL et en script ne sont pas des incohérences mais des choix d'optimisation délibérés.

Un développeur senior Python doit connaître ces subtilités pour éviter des erreurs logiques dans les tests d'identité, comprendre les mécanismes de mémoire, et optimiser des performances sans tomber dans les pièges des micro-optimisations.