Source Image

Quelle est le rôle de la fonction **id** en Python ?

Définition

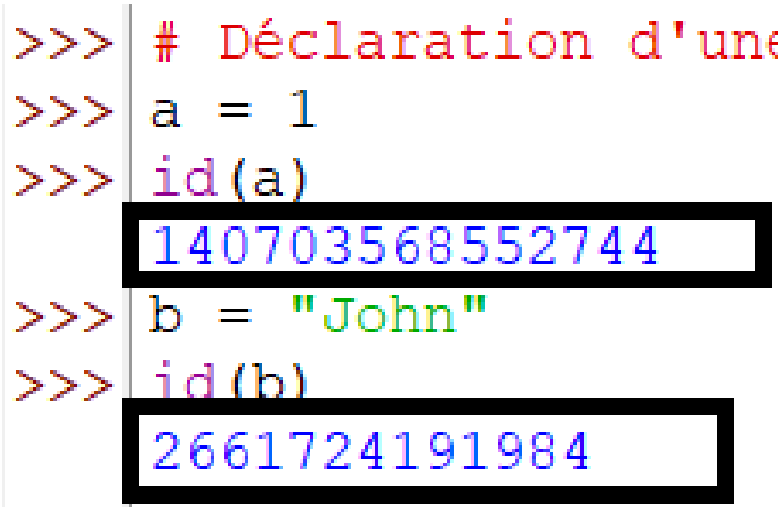
Par définition, les variables sont des façon imagée des cases dans lesquelles Python va stocker des valeurs identifiées par une étiquette appelée nom de variable.

Source Image

Si la mémoire de l'ordinateur était représentée comme une étagère avec plusieurs tiroirs numéroté, la valeur d'une variable sera stockée dans un des tiroirs.

Le rôle de *id* en Python permettra de renvoyer le numéro de la case mémoire où est stockée la valeur de la variable.

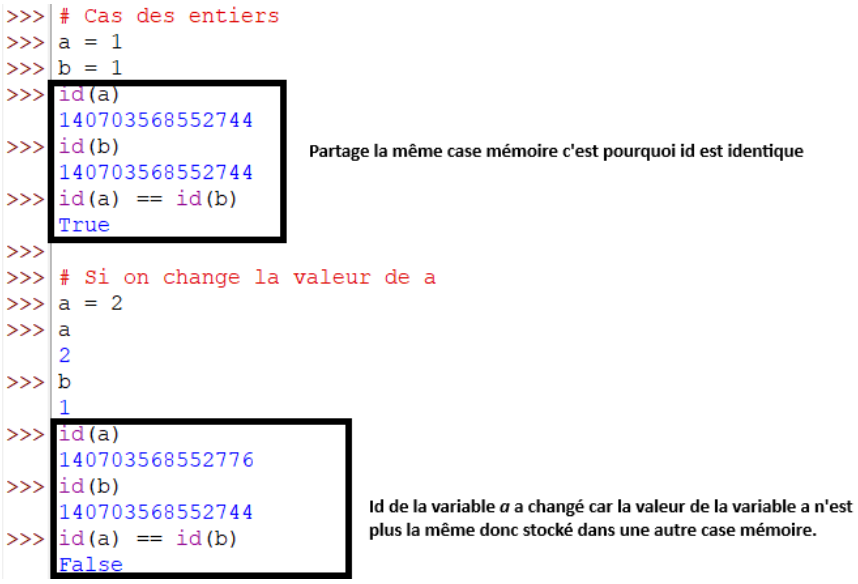
Pour utiliser **id** on utilise le format suivant: `id(nom_variable)`. Par exemple:



Dans l'exemple précédent on voit que les variable *a* et *b* ont un numéro de case mémoire différent ce qui signifie que les valeurs des variables sont stockées dans des cases mémoire différentes.

La fonction id ne renvoie pas tout le temps le même numéro de case mémoire. La valeur obtenue varie d'un ordinateur à l'autre.

Quand on ecrit un programme en pour les type primitifs comme **int**(entier), **float**(nombre à virgule), **bool**(booléen) et les types **str**(chaine) lorsque deux variables ont la même valeur, les deux variables partage la même case mêmoire tant que la valeur est identique. Comme dans les exemples ci dessous:



Le même résultat est obtenu avec les **strings** type **str**.

```
>>> # Cas des strings
>>> a = "hello"
>>> b = "hello"
>>> id(a)
3106135108912
>>> id(b)
3106135108912
>>> id(a) == id(b)
True
>>>
>>> # Si on change la valeur de b
>>> b = "hi"
>>> a
'hello'
>>> b
'hi'
>>> id(a) == id(b)
False
```

Ce mécanisme de partage de la même case mémoire n'augmente pas la mémoire occupée par une variable. 100 variables ayant la même valeur et 50 variables ayant la même valeur sont liée à une seule case mémoire.

Cependant celà n'est plus pareil pour les types **Objets** comme les **list**(liste), **tuple** et les **dict**(dictionnaires). Toute déclaration d'une variable de type objet sera stockée dans une nouvelle case mémoire peut importe qu'une autre variable existe ou non.

```
>>> # Déclaration d'une liste
>>> l1 = [1, 2]
>>> l2 = [1, 2]
>>> id(l1)
3106135014912
>>> id(l2)
3106128038592
>>> id(l1) == id(l2)
False
>>>
>>> # Déclaration de tuples
>>> t1 = 1, 2
>>> t2 = 1, 2
>>> id(t1) == id(t2)
False
>>>
>>> # Déclaration de dict
>>> d1 = {"nom": "toto"}
>>> d2 = {"nom": "toto"}
>>> id(d1) == id(d2)
False
```

Contrairement aux types primitifs ici on a une augmentation de la mémoire occupée. 100 variables de type objet occuperont 100 cases mémoires différentes.

Quand on utilise l'opération **=** pour assigner une variable à une autre c'est la copie de la case mémoire qui est faite. **a=b** signifie assigne aux variables **a** et **b** la même case mémoire qui est celle de **b**. Vue que la case mémoire est pareille dans le cas des listes celà revient à manipuler la même liste.

```
>>> # Utilisation de l'opérateur = sur les int
>>> a = 1
>>> id(a)
140703568552744
>>> b = a
>>> id(b)
140703568552744
>>> id(a) == id(b)
True
>>>
>>> # Utilisation de l'opérateur = sur les listes
>>> l1 = [1, 2]
>>> id(l1)
3106135159744
>>> l2 = [1, 2]
>>> id(l2)
3106135102400
>>> id(l1) == id(l2)
False
>>> # Les deux listes sont dans deux cases mémoire différentes si on modifie une liste l'autre n'est pas impactée
>>> l1.append(3)
>>> l1
[1, 2, 3]
>>> # utilisation de = change la case mémoire
>>> l2 = l1
>>> id(l2)
3106135159744
>>> id(l1)
3106135159744
>>> id(l1) == id(l2)
True
>>> # Si on modifie une liste l'autre sera modifiée car c'est la même case mémoire qui est modifiée
>>> l1
[1, 2, 3]
>>> l2
[1, 2, 3]
>>> l1.append(4)
>>> l1
[1, 2, 3, 4]
>>> l2
[1, 2, 3, 4]
```

Subtilité avec input

Quand l'utilisateur entre dans le jeu celà change.

Précédemment pour les types primitifs, **id** renvoyait la même valeur parce les variables contenaient les mêmes valeurs. Quand on utilise la fonction **input** qui permet à l'utilisateur de saisir une valeur, la valeur saisie de l'utilisateur est stockée dans une nouvelle case mémoire.

```
>>> # Création d'une variable avec une valeur
>>> a = 1
>>> id(a)
140703568552744
>>> # Demander une valeur à l'utilisateur qui va être 1
>>> b = input()
1
>>> id(b)
140703568596096
>>>
>>> # Si on convertit la saisie de l'utilisateur en entier on aura le même id
>>> b = int(b)
>>> b
1
>>> id(b)
140703568552744
>>>
```

Quand l'utilisateur doit saisir une nombre la saisie est stockée dans une nouvelle case mémoire car on ne sait pas si ce que va fournir l'utilisateur est une valeur déjà stockée ou non.

Après saisie de l'utilisateur si on convertit et qu'on a la même valeur qu'une déjà stockée, l'identifiant devient celui de la valeur actuellement stockée on parle de **mutation**.

Ce mécanisme de mutation ne fonctionne pas avec les chaines de caractères ni avec les types objets.

Quand il s'agit du type **str** la case mémoire où la valeur de l'utiliseur est stockée ne changera pas même si la même valeur existe déjà c'est pourquoi on dit que les chaine de caractères **str** sont **immuables** car leur **id** ne change plus quand déjà alloué.

```
>>> # Définition d'une variable chaine
>>> nom1 = "John"
>>> id(nom1)
    3106135084912
>>> # Demander à l'utilisateur de saisir une chaine qui sera la même valeur John
>>> nom2 = input()
    John
>>> nom2
    'John'
>>> id(nom2)
    3106135156272
>>> id(nom1) == id(nom2)
    False
>>> # Même si on essaie de convertir çà ne change rien
>>> nom1 = str(nom1)
>>> nom2 = str(nom2)
>>> id(nom1) == id(nom2)
    False
```

Subtilité des comparaisons avec *is* et *is not*

Dans les versions récentes de Python (par exemple dans la 3.10) on a les mots clés **is not** et **is** qui sont utilisés par fois à tort pour vérifier si une deux variables ont la même valeur.

Cependant cette méthode n'est pas du tout la même chose quand on parle des types complexes comme les **string** (chaines de caractères).

En réalité les mots clés **is** et **is not** font une comparaison des cases mémoire c'est à dire du résultat des valeurs de **id** pas des valeurs des variables.

Ces mots clés vérifient que deux variables sont stockées dans la même case mémoire. Pour comparer les valeurs on utilisera **==** et **!=** quand on parle des chaines.

```
>>> # is et is not sans input sur les entiers
>>> a = 1
>>> b = 1
>>> a is b
True
>>> id(a)
140703568552744
>>> id(b)
140703568552744
>>>
>>> # Utilisation de inputs
>>> a = 1
>>> b = input()
1
>>> a is b
False
>>> b = int(b)
>>> a is b
True
```

== et *!*= sur les types primitifs sont identiques que *is* et *is not*.

Sur les chaines c'est plus pareil.

```
>>> # is sur les chaines sans input
>>> a = "Hello"
>>> b = "Hello"
>>> a is b
True
>>> id(a) == id(b)
True
>>>
>>> # Quand on utilise input c'est plus pareil
>>> a = "Hi"
>>> b = input()
Hi
>>> a is b
False
>>> id(a) == id(b)
False
>>> # Pour comparer les valeurs utiliser ==
>>> a == b
True
```

Pour résumer:

- **is** et **is not** donne le même résultat que **!=** et **==** pour les variables de types simples car ces valeurs sont mutables, les identifiants de case mémoire change quand les valeurs sont identiques. Plusieurs variables avec la même valeur auront la même case mémoire.
- **is** et **is not** sont différents de **!=** et **==** pour les variables de types complexes comme les chaines. Deux chaines de mêmes valeurs n'ont pas forcément les mêmes identifiants de case mémoire. Pour les chaines de caractères on utilisera **==** pour vérifier l'égalité des valeurs des variables de types chaine et **!=** pour vérifier la différence entre les valeurs.