Objets et noms

La lecture de cette page n'est pas nécessaire en première approche, mais elle permet de comprendre la relation entre les objets et les noms de variable, ce qui peut être utile dans certains contextes.

Un **objet** désigne une entité stockée en mémoire permettant de représenter un certain type de données. Les types `int`, `float`, `bool`, `list`, `tuple`, `str`, `function` sont des types d'objet. Par exemple, un nombre entier est représenté par un objet de la classe `int`.

Certains objets sont **mutables** (types `list` et `dict` par exemple). D'autres objets son **non mutables** (`int`, `float`, `bool`, `str`, `tuple`, `function`). Le contenu d'un objet mutable peut être modifié, par exemple on peut changer un élément d'une liste. En revanche, un objet non mutable ne peut être modifié.

On garde le mot anglais (mutable) utilisé dans la documentation officielle de Python, qui pourrait se traduire par *modifiable*, car *mutable* existe en langue française, bien que très rarement employé.

Pour identifier un objet, on doit lui associer un **nom** (appelé aussi **identifiant**), ce qui se fait par une opération appelée **affectation**. Par exemple :

```
>>> a = 10
```

crée un entier de valeur 10 et affecte cet entier au nom `a`. Il s'agit en réalité d'un nom associé à une référence de l'objet, en pratique l'adresse en mémoire de cet objet.

Le nom `a` est appelé couramment **variable** car on peut changer d'objet référencé (ce qui ne signifie pas changer l'objet lui-même). Par exemple :

```
>>> a = 15
```

crée un nouvel objet (entier de valeur 15) et affecte ce nouvel objet au nom `a`. Le terme de *variable* est emprunté à d'autres langages, par exemple le langage C, dans lesquels les objets sont effectivement variables. En toute rigueur, on ne devrait pas l'employer pour le langage Python, car il pourrait laisser croire que l'objet est variable, alors que les objets non mutables ne sont justement pas modifiables. En Python, c'est la référence d'objet associée au nom qui est variable, pas l'objet lui-même.

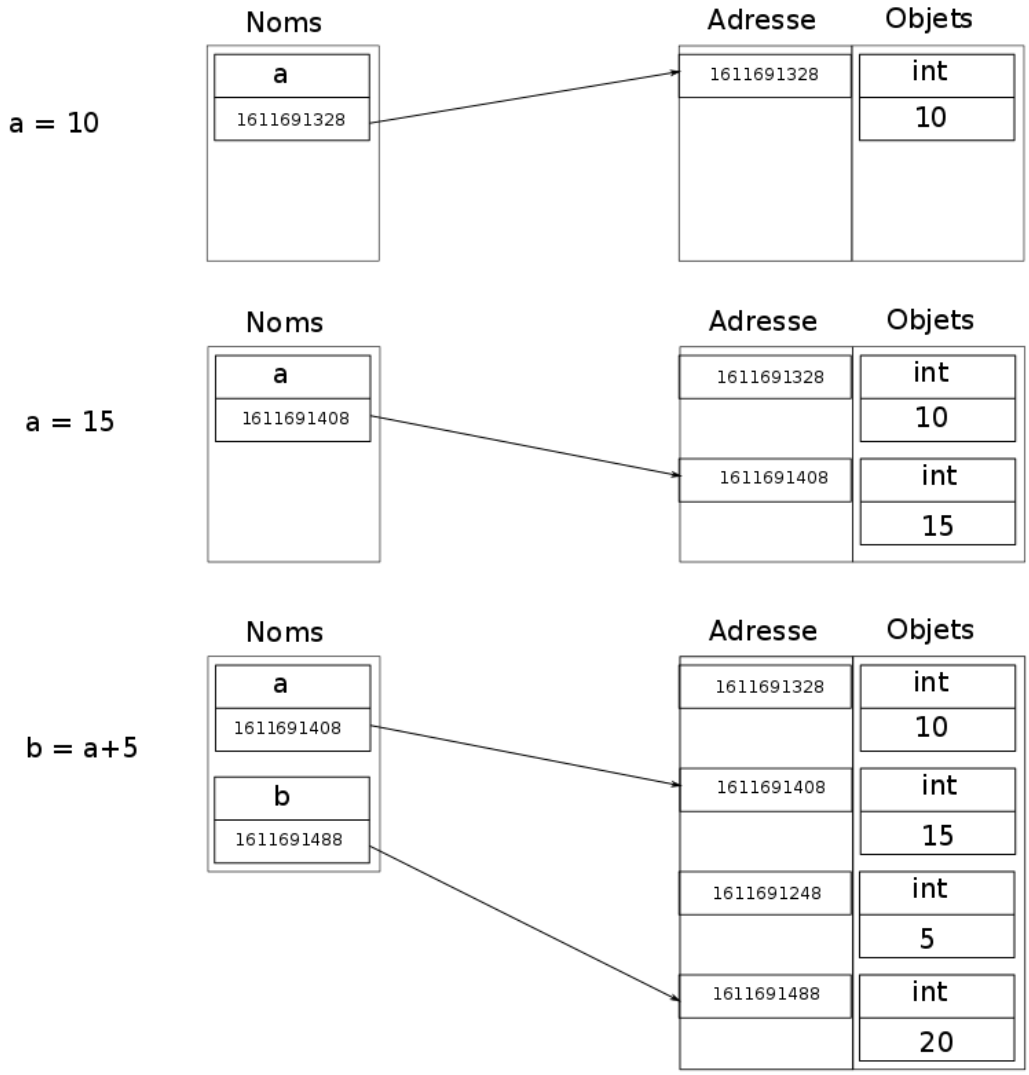
Dans les situations courantes, on peut cependant concevoir l'instruction `a = 15` précédente comme un changement de la valeur contenue dans la variable de nom `a` (alors qu'il s'agit en réalité d'un changement d'objet).

Lorsqu'une expression contient un nom, celui-ci est recherché dans l'espace de noms courant. S'il est trouvé, il est remplacé par l'objet référencé. Par exemple, lorsque la ligne

```
>>> b = a + 5
```

est exécutée, l'expression à droite du signe `=` est tout d'abord évaluée. Le nom `a` est bien celui d'une variable de l'espace de noms courant (définie précédemment); il est donc remplacé par l'objet correspondant. Un objet de type `int` et de valeur 5 est créé puis la somme des deux entiers est calculée et le résultat est mis dans un nouvel objet de type `int`. Pour finir, l'affectation consiste à affecter ce nouvel objet au nom `b`, qui est créé puisqu'il n'existait pas.

La figure suivante montre l'évolution de la table des noms et des objets lors des trois affectations. L'adresse est l'emplacement en mémoire de l'objet.



Il faut remarquer que l'affectation consiste à affecter un objet à un nom et non pas affecter un nom à un objet. En effet, un objet peut avoir plusieurs noms, par exemple :

```
>>> a = b = 15.5
```

affecte le même objet (un `float` de valeur 15.5) aux deux noms `a` et `b`. Pour le vérifier, on peut utiliser la fonction `id`, qui renvoie l'identifiant d'un objet dont le nom est donné (son adresse en mémoire) :

```
>>> id(a),id(b)
(1445164352, 1445164352)
```

Pour savoir si deux noms font référence au même objet, on utilise le mot-clé `is` :

```
>>> a is b
True
>>> x = 10
>>> y = 10
>>> x is y
True
```

Sur ce dernier exemple, on constate que l'interpréteur n'a créé qu'un seul entier de valeur 10. Cependant :

```
>>> z = round(10.5)
>>> type(z)
<class 'int'>
```

```
>>> x is z
False
>>> x == z
True
```

La variable `z` fait bien référence à un entier de valeur 10, mais un nouvel objet a été créé.

Il est possible de supprimer un nom :

```
>>> del(a)
>>> a
Traceback (most recent call last):
  File "<pyshell#386>", line 1, in <module>
    a
NameError: name 'a' is not defined
>>> b
10
```

Comme on le voit sur cet exemple, l'objet référencé existe toujours et on peut encore y accéder via le nom `b`. Si on supprime aussi ce dernier :

```
>>> del(b)
```

l'objet existe toujours en mémoire mais il n'y a plus aucun nom qui lui est associé. Lorsqu'un objet n'est plus référencé par aucun nom, il devient impossible de l'utiliser. Le moteur de Python comporte un **ramasse miette** (garbage collector) qui se charge de libérer l'espace mémoire occupé par ces objets non utilisés.

Comme déjà précisé, les objets de type `int, float, bool, str, tuple, function` sont non mutables. Lorsqu'un objet de ce type a été créé en mémoire, il est impossible de le modifier. Il peut seulement être détruit par le ramasse miette lorsqu'il n'y a plus aucun nom qui fait référence à cet objet.

Dans les langages C/C++, il en est tout autrement : lorsqu'on crée une variable, on lui associe obligatoirement un type, et un espace mémoire est réservé correspondant à ce type. Il est bien sûr possible de modifier le contenu d'une variable et les changements se font directement dans l'espace mémoire correspondant.

Pourquoi alors certains objets de Python sont-ils non mutables ? Ce choix fait par les concepteurs du langage est lié au fait que les variables sont non typées, c'est-à-dire qu'un nom peut être utilisé pour faire référence à un objet de type quelconque. Par exemple :

```
a = 10 # créé un entier de valeur 10 et un nom y faisant référence
a = 1.1 # créé un flottant de valeur 1.1 et utilise le même nom que précédemment pour y faire référence
```

Le fait que les variables soient non typées offre une grande souplesse pour la programmation des algorithmes (il y a aussi des inconvénients). En réalité, il ne s'agit pas à proprement parler de variables mais de **noms** qui font référence à des objets statiques.

Comme on le voit sur l'exemple précédent, la seconde affectation nécessite la création d'un nouvel objet. Dès lors, si on avait écrit :

```
a = 10 # créé un entier de valeur 10 et un nom y faisant référence
a = 5 # créé un entier de valeur 5 et utilise le même nom que précédemment pour y faire référence
```

il n'y pas de raison que l'entier 5 ne soit pas un nouvel objet, bien qu'il soit de même type que le précédent. Si l'affectation tentait de modifier l'objet lui-même, elle aurait toute les chances d'échouer !

Le caractère non mutable des objets est donc une conséquence logique du caractère non typé des variables, c'est-à-dire des noms.

Pourquoi les listes sont-elles mutables ? Une liste est en réalité un conteneur de références à des objets. Par exemple, la liste :

```
>>> L = [0,1.1,"mot"]
```

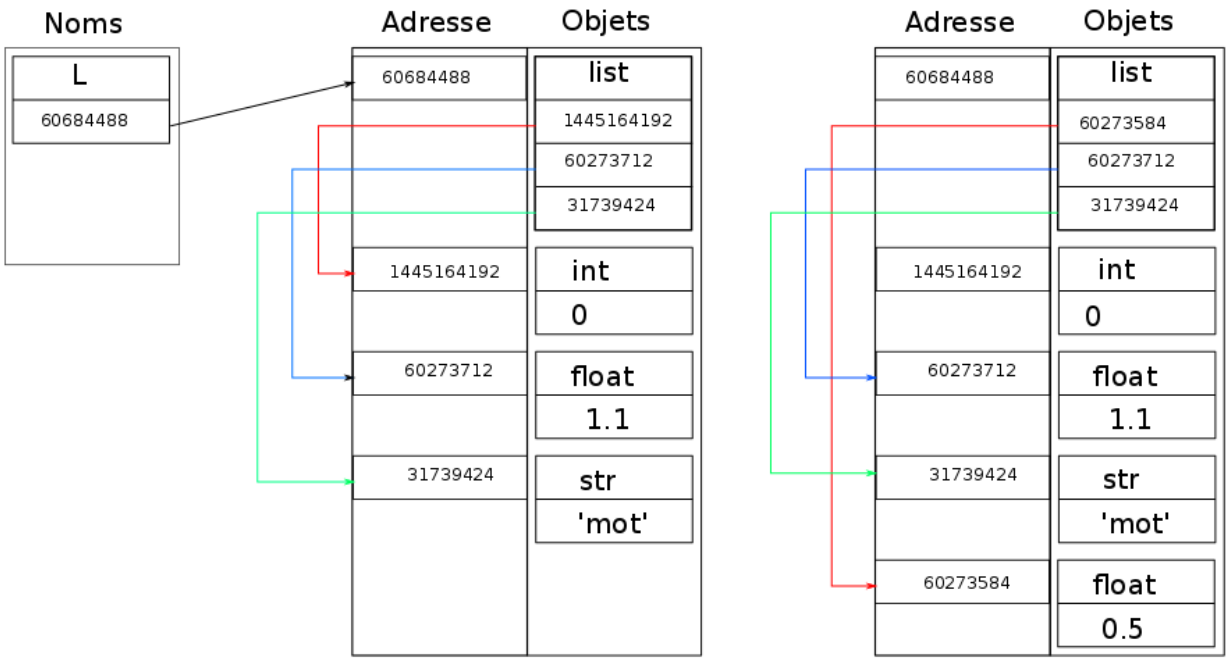
contient trois objets de type différents. Elle contient en réalité les trois références de trois objets, le premier de type `int`, le deuxième de type `float` et le troisième de type `str`.

Ces objets sont eux-mêmes non mutables, d'où une apparente contradiction avec le fait que la liste soit mutable. Le test suivant apporte la réponse, en affichant les identifiants grace à la fonction `id`.

```
>>> id(L),id(L[0]),id(L[1]),id(L[2])
(60684488, 1445164192, 60273712, 31739424)
>>> L[0]=0.5
>>> id(L),id(L[0]),id(L[1]),id(L[2])
(60684488, 60273584, 60273712, 31739424)
```

Le changement du premier élément ne change pas la liste elle-même (son identifiant reste 60684488) mais change bien l'élément. Il n'y a donc aucune contradiction entre la non mutabilité des types élémentaires et la mutabilité de la liste, car celle-ci n'est rien d'autre qu'un conteneur de références à des objets, qui sont bien non mutables.

La figure suivante montre l'état des objets avant le changement du premier élément de la liste et après ce changement.



Il est d'ailleurs aisé de constater que les éléments d'une liste existent toujours après destruction de la liste, et qu'ils sont encore accessibles pourvu qu'on ait pris soin de définir un nom pour ces éléments :

```
>>> a=L[0]
>>> del(L)
>>> a
0.5
>>> id(a)
60273584
```

L'identifiant de l'objet référencé par `a` est bien le même que l'identifiant du premier élément de la liste. La liste n'existe plus, mais son premier élément existe toujours.

Lorsqu'on créé un nom (c.a.d. une variable), on ajoute dans la table de noms de **l'espace de nom** courant ce nom associé à la référence de l'objet auquel ce nom permet d'accéder. Lorsqu'on fait appel à une fonction, les instructions de cette fonction opèrent dans un espace de noms propre à la fonction, c'est-à-dire que les noms créés dans la fonction sont créés dans une table particulière appelée espace de noms de la fonction. Ces noms internes à la fonction sont appelées **variables locales**. Cette table est détruite lorsque la fonction a terminé son exécution.

Ainsi, les noms créés à l'intérieur d'une fonction ont une **portée** limitée à cette fonction. En revanche, l'espace des noms (c.a.d. la table des noms) utilisée par le code qui appelle la fonction est bien accessible à l'intérieur de la fonction et on parle alors de variables globales.

Considérons le code suivant, qui utilise trois espaces de noms :

```
def f(x,y):
    # x et y sont dans l'espace de noms de la fonction f
    a = 10 # espace de noms de la fonction f
    def g(x):
        a = 20 # espace de noms de la fonction g
        return a*x
    return g(x)+a*y+c # c est une variable globale

a = 1 #espace de nom du script contenant ce code (variables globales)
b = 0
c = 1
x = 1
y = 1
print(f(x,y)) # affiche 31
```

Dans cet exemple, on a volontairement choisi les mêmes noms pour des variables qui ne sont pas définies dans le même espace de noms, afin de bien souligner le fait que ces espaces de noms sont parfaitement indépendants. En pratique, il est bon d'utiliser les mêmes noms seulement si les objets référencés ont la même signification dans l'algorithme. Par exemple, dans les trois dernières lignes du code ci-dessus, il est pertinent d'employer les mêmes noms pour les variables globales `x,y` que pour les paramètres de la fonction.

Les noms de l'espace de noms global sont consultables avec la fonction `globals()`, ceux d'un espace de noms local sont consultables avec la fonction `locals()`. Ces deux fonctions renvoient un dictionnaire. Pour savoir si un nom appartient à l'espace de nom global, il suffit d'écrire le test suivant :

```
if 'nom' in globals():
```

Ces deux fonctions sont utiles pour le débogage, car la confusion entre variables locales et globales est une erreur fréquente.