Modèle d'affectation en Python

Laurent Signac – CC-BY-SA – 07-07-20 0852 ee792a5ceb7a34b50ea5

Le modèle de la variable comme tiroir, ou boîte, contenant une valeur ne reflète pas les mécanismes de Python. Comprendre les mécanismes réels permet de limiter les surprises et de prévoir le comportement de certains programmes : Python ne stocke pas des valeurs dans des variables, mais **donne des noms à des objets**.

1 Affectation

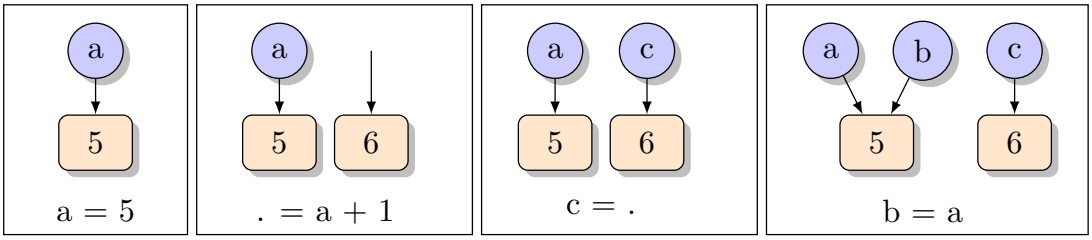
Lorsqu'on écrit `a=5` cela signifie qu'on donne le nom `a` à `5`. Autrement dit, on place une étiquette `a` sur l'objet.

En règle générale, lorsqu'on écrit `a = quelquechose`, ce qui est à droite du `=` est évalué et créé en mémoire, s'il n'existe pas déjà, puis on lui donne un nom : `a`.

Voici un exemple :

```
a = 5
c = a + 1
b = a
```

Ce qui correspond aux mécanismes suivants :



Dans le cas `b = a`, puisque l'objet (à droite du `=`) de nom `a` existe déjà, on lui donne simplement un nouveau nom. L'objet de valeur 5 a donc deux noms : `a` et `b`.

2 Passage de paramètres non mutables

Lorsqu'on passe des paramètres à une fonction ou une procédure, tout se passe *comme si* on réalisait une affectation (en Python, on dit que le passage des paramètres se fait par affectation).

Par exemple :

```
def mafonction(x):
    ...

y = 3
print(mafonction(y))
```

Lors de l'appel à `mafonction(y)`, tout se passe comme si on avait écrit : `x = y` puis qu'on avait ensuite exécuté le corps de la fonction.

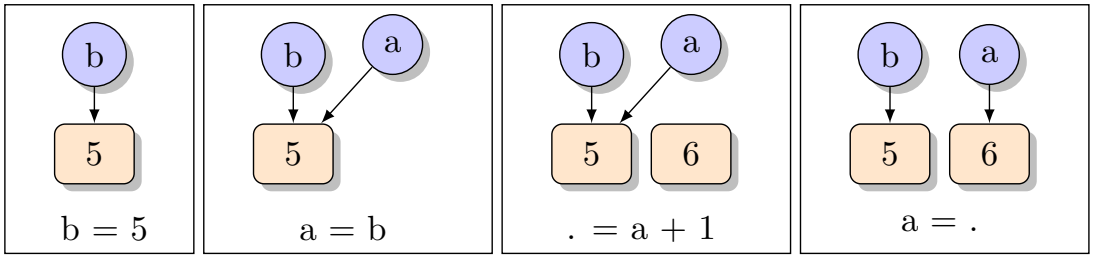
Ce mécanisme, et celui de l'affectation décrit plus haut, permettent d'expliquer le comportement suivant :

```python
def ajoute(a):
    a = a + 1

b = 5
ajoute(b) # passage du paramètre : a = b
print(b)
```

Qu'affichera se programme ? 5 ou 6 ? Essayez de répondre à la question avant de lire la suite (70 % des personnes interrogées se trompent)

Si on décompose les opérations d'affectation, on obtient ce mécanisme :



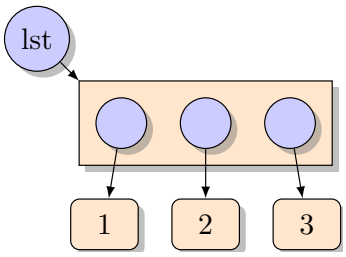
Il devient alors clair que le programme affiche 5. Car la valeur de `b` n'est modifiée à aucun moment.

3 Listes, mutations, affectations

Les listes sont des objets qui contiennent des références vers d'autres objets. Par exemple, la liste `lst`, créée ainsi :

```python
lst = [1, 2, 3]
```

pourrait être représentée par ce schéma :

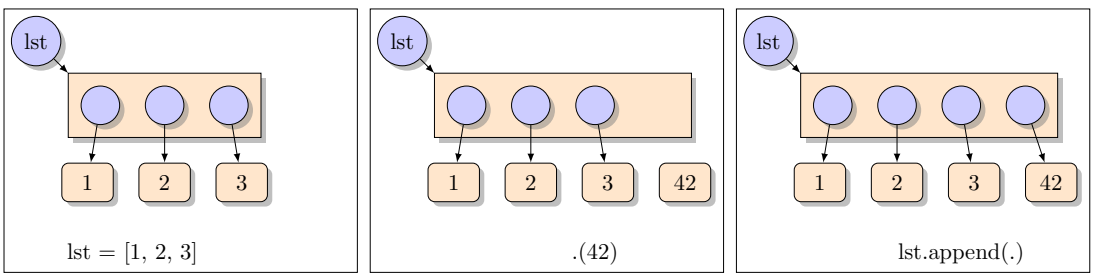


La liste contient 3 références (les cercles) qui désignent 3 objets, stockés en dehors de la liste.

Il existe deux moyens qui permettent, à partir de la liste `[1, 2, 3]`, d'obtenir la liste `[1, 2, 3, 42]`. Ces deux moyens sont souvents confondus, mais ils sont très différents.

La première solution consiste à faire *muter* la liste :

```python
lst = [1, 2, 3]
lst.append(42)
```

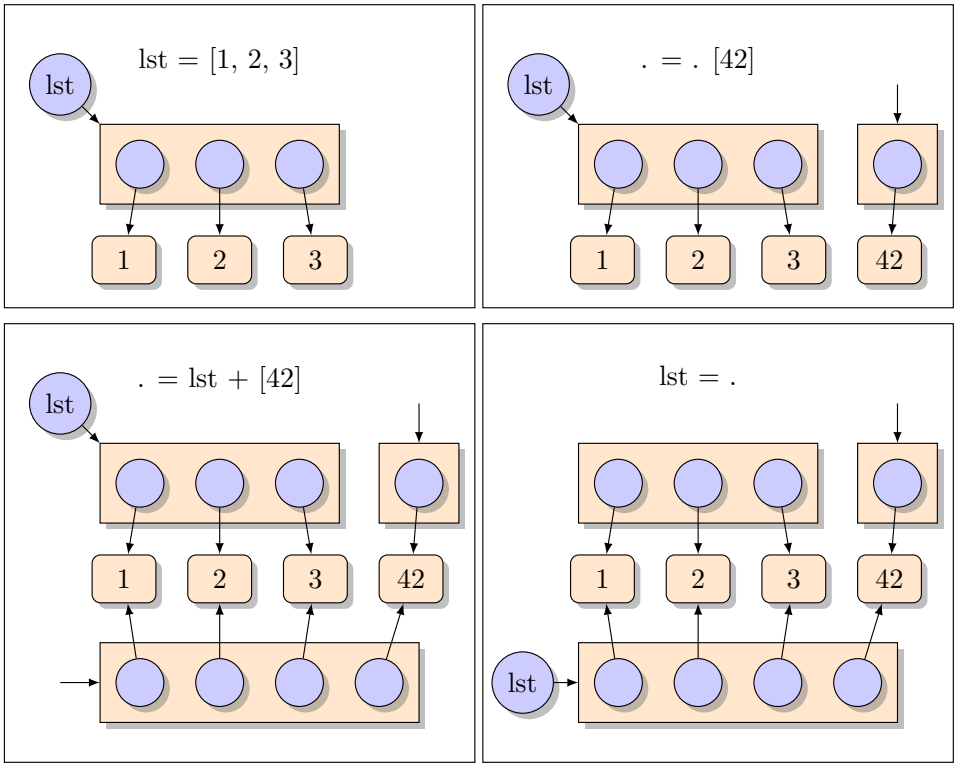


La méthode **append** fait *muter* la liste. Ce n'est **pas** une affectation. Il n'y a **pas** de symbole `=`, donc pas de modification d'étiquette (de nom).

L'autre solution consiste à écrire :

```
lst = [1, 2, 3]
lst = lst + [42]
```

Dans ce cas, on a une affectation (`lst = lst + [42]`). Le mécanisme est le même que celui déjà rencontré : on crée la liste `lst + [42]` qui est la concaténation de deux listes (`lst` et `[42]`). Une fois cette nouvelle liste créée, on lui donne un nom : `lst`. L'étiquette `lst` est donc déplacée. Le mécanisme correspondant est :



On voit bien que dans ce cas, `lst` est une **nouvelle** liste, et non pas l'ancienne dans laquelle on aurait ajouté une valeur.

En Python, réaliser une affectation, c'est **donner un nouveau nom à un objet** et non pas copier l'objet.

Si on met de côté des considérations d'efficacité (utiliser `append` est bien plus rapide), l'utilisation de l'une ou l'autre des deux solutions (`lst.append(42)` ou `lst = lst + [42]`) à l'intérieur d'une fonction donne des comportements très différents, mais tout à fait consistants avec les mécanismes évoqués.

4 Passage de paramètres mutables

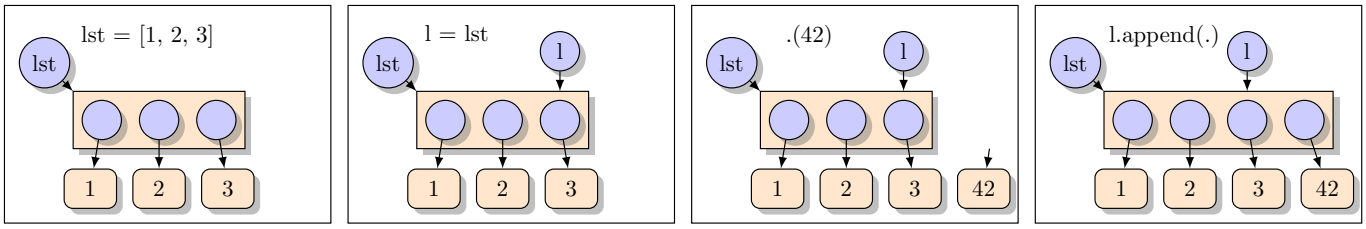
Considérons le programme suivant :

```
def complete_liste(l):
    l.append(42)

lst = [1, 2, 3]
complete_list(lst)
print(lst)
```

Que va afficher ce code ? `[1, 2, 3]`, ou `[1, 2, 3, 42]` ? Essayez de répondre à la question avant de lire la suite.

En utilisant ce qui a été vu, le mécanisme est :



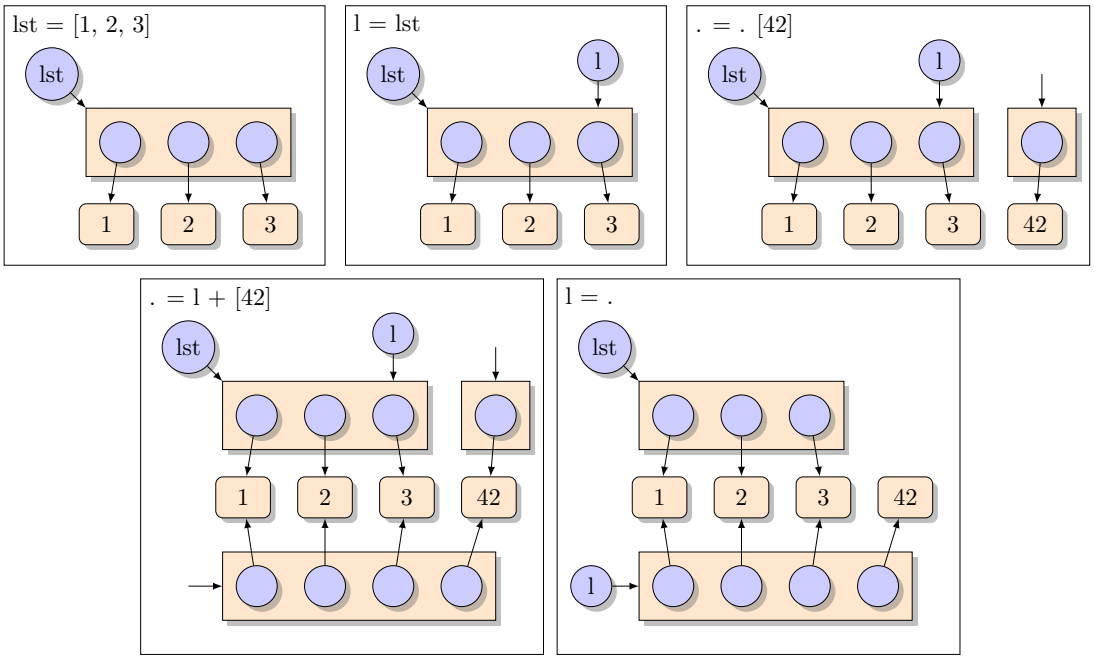
Le programme affichera donc `[1, 2, 3, 42]`. Contrairement à ce qui se produisait avec des nombres précédemment, on a ici fait *muter* l'objet passé en paramètre. Il est donc réellement modifié.

Mais si en revanche nous remplaçons `l.append(42)` par `l = l + [42]`, on obtient :

```
def complete_liste(l):
    l = l + [42]

lst = [1, 2, 3]
complete_list(lst)
print(lst)
```

Et dans ce cas, la fonction contient une affectation, le mécanisme est le suivant :



Et le code précédent affichera donc `[1, 2, 3]` : on n'a **pas** fait muter la liste.

5 Mode de passage des paramètres

De nombreux langages possèdent plusieurs modes de passage des paramètres (souvent appelés par valeur et par référence). Dans un cas, l'objet passé en paramètre est recopié, et le modifier dans la fonction n'aura pas d'influence sur l'original. Dans l'autre cas, c'est l'original qui est passé à la fonction, qui pourra donc le modifier.

En Python, les paramètres sont systématiquement passés par affectation (comme nous l'avons déjà vu), ce qui est ici à peu près équivalent à un passage par référence. C'est le caractère mutable ou non mutable, et l'emploi de l'affectation ou pas dans la fonction, qui fera que l'objet d'origine est modifié ou non. En d'autres termes, le **seul moyen** de s'assurer qu'un appel de fonction ne va pas modifier ses paramètres, c'est de lui envoyer des arguments **non mutables** (nombres, tuples… mais pas listes ou dictionnaires).

6 Références

Ne pas maîtriser cette particularité de Python est un problème pour de nombreux programmeurs. Aussi, les références sur le sujet sont nombreuses. La plus pertinente et complète semble être la conférence de Ned Batchelder en 2015. La vidéo est en ligne ainsi que la retranscription écrite : Facts and Myths about Python names and values[1]

[1] https://nedbatchelder.com/text/names1.html