2 Variables informatiques

La variable est la base de l'informatique. En tout cas de la programmation dans un langage impératif. On appelle de la sorte un langage fonctionnant sur le principe de modifications successives de l'état de mémoire de la machine (i.e. modification du contenu des variables, donc de leurs valeurs), suivant les instructions fournies par le code du programme.
Beaucoup de langages fonctionnent de façon impérative, mais ce n'est pas le seul paradigme de programmation possible : par exemple, le langage Caml, étudié en option informatique, n'est pas un langage impératif.
Python est un langage impératif. Dans ce contexte, la notion de variable est centrale. Il est donc important de savoir précisément ce qui se cache derrière cette terminologie.

I Notion de variable informatique

Définition 2.1.1 (Variable informatique)

Une variable est la donnée de :

- une localisation en mémoire, représentée par son adresse (identifiant de la variable)
- un nom donné à la variable pour la commodité d'utilisation (appels, affectations)

Ainsi, une variable est à voir comme une correspondance entre un nom et un emplacement en mémoire. Une variable x va pointer vers un certain emplacement en mémoire, en lequel il pourra être stocké des choses.

Définition 2.1.2 (Contenu d'une variable)

Le contenu d'une variable est la valeur (ou l'objet) stockée à l'emplacement mémoire réservé à la variable. Le déchiffrage du code binaire de ce contenu dépendra du `type` de la variable.

Ainsi, dire qu'une variable x est égale (à un moment donné) à 2, signifie qu'on a une variable, dont le nom est x, et qu'à l'emplacement mémoire réservé est stockée la valeur 2.

Remarque 2.1.3

Toutes les informations transitent en binaire, et sont stockées en binaire (donc uniquement avec des chiffres 0 et 1). Ceci provient de la nature physique des espaces de stockages, réalisés par des éléments qui peuvent être chargés (1) ou déchargés (0). Le transit de l'information se fait sous forme d'impulsions électriques nécessitant aussi un codage binaire. C'est donc une traduction binaire de l'objet à stocker qui est mise en mémoire, et non l'objet lui-même. La traduction et le décodage dépendent du type de l'objet.

Définition 2.1.4 (Affectation)
L'affectation est l'action consistant à définir le contenu d'une variable, c'est-à-dire, explicitement, à stocker une valeur donnée à l'emplacement mémoire associé.

L'affectation se fait toujours en associant (suivant une certaine syntaxe qui dépend du langage) la valeur à stocker et le nom de la variable. Ainsi, en Python la syntaxe est :

```
x = 3
```

Attention, l'égalité d'affectation n'est pas commutative : on place le nom de la variable à gauche, et la valeur à droite. Par ailleurs, il ne faut pas confondre l'égalité d'affectation, du test d'égalité entre deux valeurs, souvent noté différemment (par exemple, en Python, le test d'égalité est noté `a == b`).

Définition 2.1.5 (Lecture, ou appel)
La lecture, ou l'appel d'une variable est le fait d'aller récupérer en mémoire le contenu d'une variable, afin de l'utiliser dans une instruction.

La lecture se fait le plus souvent en donnant le nom de la variable :

```
>>> a = 2
>>> a
2
>>> b = a + 3
>>> b
5
```

Définition 2.1.6 (État d'une variable)
L'état momentané d'une variable est la donnée de cette variable (donc son nom et son adresse) et de son contenu.

Définition 2.1.7 (Type d'une variable)
Les variables peuvent avoir différents types, prédéfinis, ou qu'on peut définir. Le type représente la nature du contenu (un réel, un entier, un complexe...).

Le type détermine la taille à réserver en mémoire pour la variable, ainsi que l'algorithme de traduction de l'objet en code binaire et inversement.

Remarque 2.1.8 (Déclaration de variables)

- Certains langages imposent de *déclarer* les variables avant de les utiliser. C'est lors de cette déclaration que l'ordinateur attribue une localisation en mémoire à la variable. Pour le faire, il doit connaître le type de la variable. Ainsi, on déclare toujours une variable en précisant son type.
- Certains langages (dont Python) dispensent l'utilisateur de cette déclaration préalable des variables. Dans ce cas, un emplacement en mémoire est alloué à une variable lors de l'affectation. Le type (nécessaire pour savoir quelle place attribuée) est déterminé automatiquement, suivant la nature de la valeur stockée lors de cette affectation.

```
>>> x=3
>>> type(x)
<class 'int'>
>>> x=3.
>>> type(x)
<class 'float'>
>>> x=(3,4)
>>> type(x)
<class 'tuple'>
```

Définition 2.1.9 (Typage dynamique, typage statique)

On dit qu'un langage de programmation a un typage statique s'il impose la déclaration préalable. Dans le cas contraire, on parle de typage dynamique.

L'attribution automatique du type peut parfois être ambiguë. Par exemple, si l'utilisateur veut stocker la valeur 3, comme indiqué plus haut, Python va le stocker sous un type entier. Mais l'utilisateur pouvait avoir envie d'en faire une utilisation réelle. Cela est possible du fait que le type d'une variable peut changer :

```
>>> x = 3
>>> type(x)
<class 'int'>
>>> x /= 2
>>> x
1.5
>>> type(x)
<class 'float'>
```

La variable `x` est passée du type entier au type réel.

Définition 2.1.10 (Typage faible)

On dit qu'un langage a un typage faible (ou est faiblement typé) s'il autorise une variable à changer de type au cours de son existence.

Ainsi, notre essai ci-dessus montre que Python est faiblement typé.
Mais physiquement, comment gérer le fait qu'une variable réelle prend plus de place en mémoire qu'une variable entière ? La réponse est simple :

```
>>> x =3
>>> id(x)
211273705504
>>> x /= 2
>>> id(x)
140635669676560
```

L'emplacement de x a changé. En fait, toute affectation modifie l'emplacement mémoire (donc l'identifiant) d'une variable, même si elle conserve le même type.

Certaines méthodes modifient certaines variables en place (donc sans changement d'adresse) : il ne s'agit dans ce cas pas d'affectations.

II Structures de données

Il est fréquent de regrouper plusieurs valeurs dans une variable ayant une structure plus complexe capable de mémoriser plusieurs valeurs selon une hiérarchie déterminée, par exemple sous forme d'une liste ou d'un ensemble.

Définition 2.2.1 (Structure de données, attribut)

Une *structure de données* est une organisation de la mémoire en vue de stocker un ensemble de données (n'étant pas nécessairement de même type). Les données qui constituent la structure sont appelées *attributs*.

Une structure de données est déterminée par la façon dont sont rangés les attributs les uns par rapport aux autres, et la façon à laquelle on accède à ces données. Ainsi, il existe plusieurs types de structures de données.
Une structure de données est définie par une *classe*, possédant un type (le nom de la classe, déterminant la structure en mémoire à adopter), et un ensemble de méthodes associées à la classe (des fonctions permettant de modifier d'une façon ou d'une autre un objet)

Définition 2.2.2 (Instantiation)

Une instantiation d'une classe est la création d'une variable dont le type est celui de la classe. Il s'agit donc de la création :

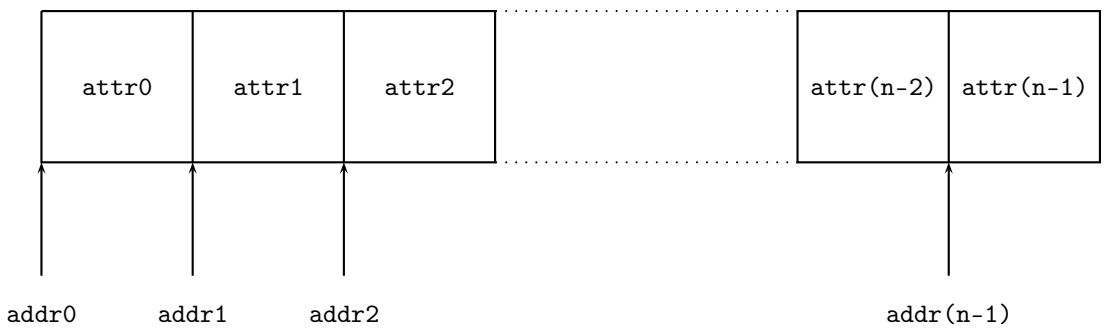
- d'un nom de variable, couplé à une adresse mémoire et un type
- d'un contenu, à l'adresse mémoire en question, dont la structure de stockage dépend de la classe.

Une des structures les plus utilisées est la structure de tableau (les listes de Python par exemple en sont une variante). Nous l'évoquons ci-dessous un peu plus en détail.

Définition 2.2.3 (Structure de tableau)

Un tableau (statique) informatique est une structure de données séquentielle (un rangement linéaire des attributs), dont tous les attributs sont de même type. Un tableau a une taille fixe, déterminée au moment de sa création.

En mémoire, les différents attributs d'un tableau sont stockés de façon contiguë. L'homogénéité du type des attributs permet alors de stocker chaque attribut sur un nombre déterminé d'octets. Ainsi, la place prise en mémoire ne dépend pas du type d'objet stocké, et ne risque pas de varier au cours du temps :



Les données étant rangées de façon contiguë, la connaissance de l'adresse initiale `addr0` détermine complètement l'adresse de début de l'attribut i. Plus explicitement, si les données sont d'un type nécessitant une place N de stockage, on aura une relation du type : `addr(i)= addr(0) + Ni`.

Ainsi :

Proposition 2.2.4 (Complexité des opérations élémentaires sur un tableau)

(i) L'accès à un élément d'un tableau se fait en temps constant (ne dépendant ni de la taille du tableau, ni de la valeur de l'indice)

(ii) La recherche d'un élément se fait en $O(n)$, c'est-à-dire au plus en un temps proportionnel à la taille du tableau (il faut parcourir tout le tableau dans l'ordre)

(iii) L'insertion d'un élément se fait en $O(n)$ (il faut décaler tous les suivants, par réécritures)

(iv) La suppression d'un élément se fait en $O(n)$ (il faut décaler aussi)

(v) La recherche de la longueur se fait en temps constant (c'est une donnée initiale).

On peut améliorer cette structure, en autorisant des tableaux de taille variable. On parle alors de tableaux dynamiques.

Définition 2.2.5 (Tableaux dynamiques)

Un tableau dynamique est une structure de données séquentielle de taille variable, pour des données ayant toutes le même type.

Définition 2.2.6 (Capacité, taille)

- La capacité d'un tableau est le nombre de cases momentanément disponibles en mémoire pour le tableau
- La taille d'un tableau (ou la longueur) est le nombre de cases réellement utilisées (les autres étant considérés comme vides).

La capacité d'un tableau dynamique peut évoluer suivant les besoins ; en particulier, insérer des éléments peut faire dépasser la capacité. Dans ce cas, la capacité est doublée : il est donc créé un nouveau tableau de taille double. On ne peut pas se contenter de créer les cases manquantes, car comme pour un tableau, les cases doivent être contiguës en mémoire, et il n'est pas certain que les cases suivant les cases déjà existantes soient disponibles.
Ainsi, une augmentation de capacité nécessite en principe un changement d'adresse donc une réécriture complète (on recopie les données). C'est pour cette raison que lorsqu'on augmente la capacité, on le fait franchement (on double), quitte à avoir des cases inutilisée, afin de ne pas avoir à faire ces réécritures à chaque opération. C'est la raison de la distinction entre capacité et taille.
La capacité et la taille du tableau sont stockés en des emplacements directement accessibles. Ainsi, n désignant la taille d'un tableau, on peut étendre les résultats obtenus pour les tableaux standards :

Proposition 2.2.7 (Complexité des opérations élémentaires sur un tableau dynamique)

(i) L'accès à un élément d'un tableau dynamique se fait en temps constant (ne dépendant ni de la taille du tableau, ni de la valeur de l'indice)

(ii) La recherche d'un élément se fait en $O(n)$,

(iii) L'insertion d'un élément se fait en $O(n)$ (sauf lorsque cela induit un dépassement de capacité)

(iv) La suppression d'un élément se fait en $O(n)$

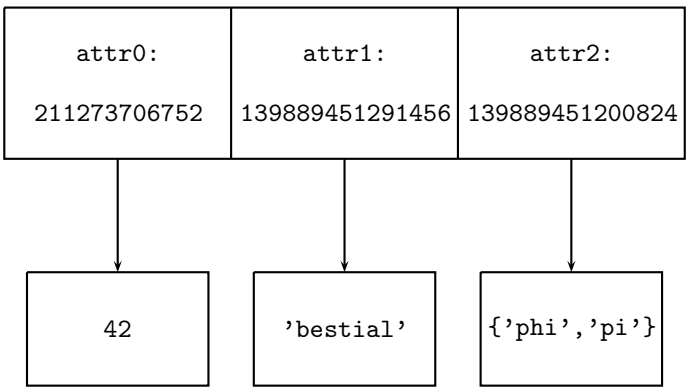
(v) La recherche de la longueur se fait en temps constant (il faut juste récupérer la donnée en mémoire).

Définition 2.2.8 (Listes Python)
Une liste en Python est assimilable à une structure de tableau dynamique, dont les attributs sont des pointeurs (adresses) vers des emplacements mémoires où sont stockées les données de la liste.

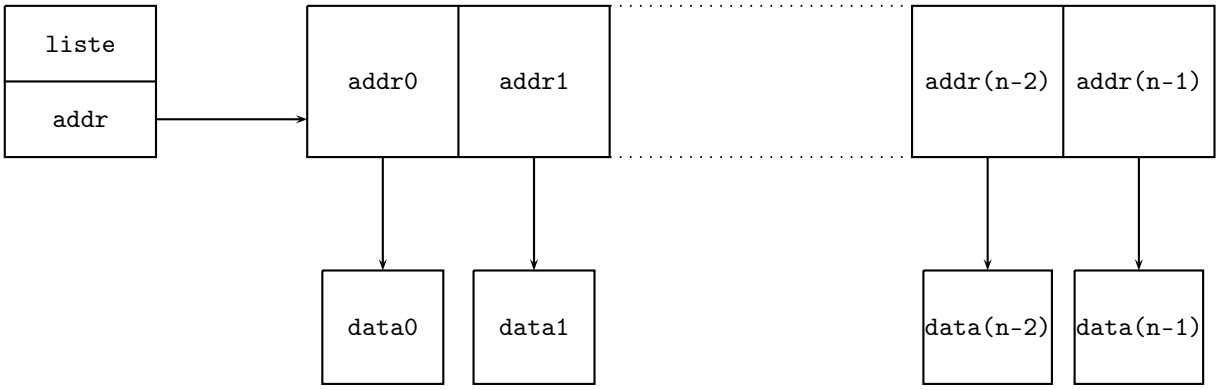
Ainsi les attributs réels d'une liste sont tous de même type (des adresses), comme cela doit être le cas pour un tableau. Mais les objets cibles (qu'on appellera aussi attributs par abus, ou données) peuvent être de type quelconque, et différents les uns des autres.
Par exemple, créons la liste suivante :

```
>>> liste = [42,'bestial',{'phi','pi'}]
>>> id(liste)
139889451112064
>>> id(liste[0])
211273706752
>>> id(liste[1])
139889451291456
>>> id(liste[2])
139889451200824
>>> type(liste[0])
<class 'int'>
>>> type(liste[1])
<class 'str'>
>>> type(liste[2])
<class 'set'>
```

La structure de cette liste en mémoire sera donc :



Plus généralement, on aura une structure en mémoire du type suivant :



Remarque 2.2.9

Cela ne correspond pas à la terminologie adoptée dans beaucoup d'autres langages, ou une liste possède une structure différente (liste chaînée, ou doublement chaînée) : dans une structure de ce type, les données ne sont pas nécessairement stockées de façon contiguë. À l'emplacement de la donnée k, on trouve la donnée à mémoriser, ainsi que l'adresse de la donnée de rang $k+1$ (c'est le chaînage), et éventuellement de rang $k-1$ (en cas de double-chaînage). La donnée intiale de la liste est l'adresse du premier terme. Ainsi, pour accéder au k-ième élément, on est obligé de parcourir tous les premiers éléments de la liste, en suivant les liens successifs : l'accès à un élément n'est plus en temps constant ; ni le calcul de la longueur qui nécessite de parcourir toute la liste. En revanche, l'insertion et la suppression se font en temps constant (une fois la position repérée).

Nous restons assez vagues sur la structure d'ensemble qui est plutôt du ressort du programme d'option.

Définition 2.2.10 (Structure d'ensemble, set)

Un ensemble informatique est une structure de données, donc les attributs sont stockés à un emplacement en mémoire dépendant de leur valeur (plus rigoureusement, un ensemble est une structure arborescente).

Le positionnement en mémoire en fonction de la valeur donne une structure non ordonnée (les éléments ne sont pas rangés dans un certain ordre), pour lequel le test d'appartenance se fait en temps quasi-constant en cherchant une valeur à la place où elle doit être (il s'agit d'un temps logarithmique en fait, car on doit explorer un arbre le long d'une branche)
En Python, les ensembles sont du type `set`, et représentés entre accolades : `{1,2,3}` par exemple.

III Mutabilité ; cas des listes

Définition 2.3.1 (Mutabilité)

Un type d'objets est dit *mutable* si les objets de ce type sont modifiables (sans changement d'adresse, donc pas sous forme d'une réaffectation). Il est dit *non mutable* ou *immuable* sinon.

```
>>> liste = [1,2]
>>> id(liste)
140098803414224
>>> liste[1]+=1
>>> liste
[1, 3]
>>> id(liste)
140098803414224
>>> couple = (1,2)
>>> couple[1]+=1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

L'exemple ci-dessus montre que les listes sont mutables : on peut changer leurs attributs, sans changer l'adresse de la liste elle-même. En revanche, les tuples ne semblent pas mutables.

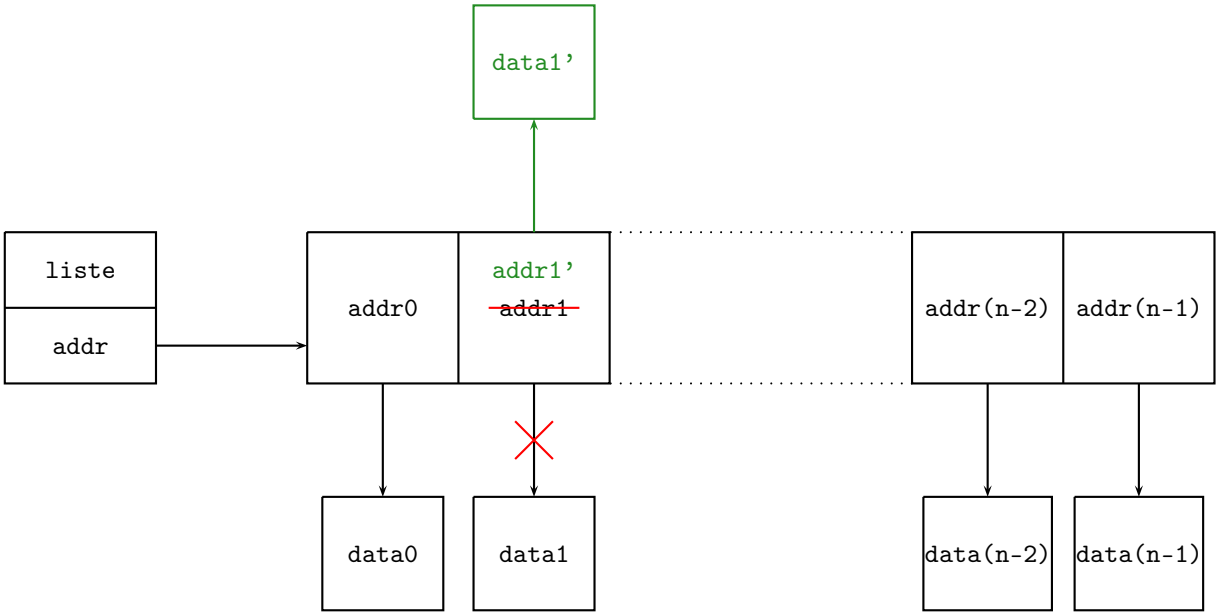
Exemple 2.3.2 (Mutabilité des principales classes en Python)

- Les `list` et les `set` sont mutables.
- Les types numériques (`int`, `float`, `boolean`, `complex`...), les `tuples`, les `str` (chaînes de caractères) ne sont pas mutables.

Voyons quel est l'effet de la modification sur les attributs réels de la liste, donc sur l'adresse des données :

```
>>> liste = [1,2]
>>> id(liste[0])
211273705440
>>> id(liste[1])
211273705472
>>> liste[1]+=1
>>> id(liste[1])
211273705504
```

Ainsi, comme toute réaffectation sur une variable, l'opération effectuée sur la donnée modifie son adresse (en fait, les types numériques sont non mutables). Cette nouvelle adresse est stockée dans la liste.
Une affectation sur un des attributs peut donc être représenté de la façon suivante :

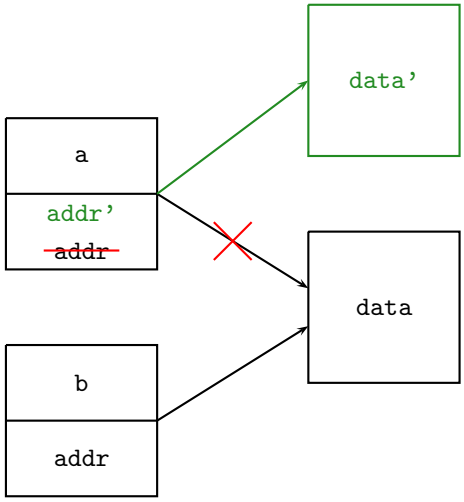


Intéressons-nous maintenant à la copie de listes. Pour commencer, étudions ce qu'il se passe quand on copie un entier, et qu'on modifie une des deux copies :

```
>>> a = 1
>>> b = a
>>> id(a)
211273705440
>>> id(b)
211273705440
>>> a += 1
>>> a
2
>>> b
1
```

```
>>> id(a)
211273705472
>>> id(b)
211273705440
```

Au moment de la copie, la donnée n'a pas été recopiée : il a été créé une variable `b` pointant vers la même adresse que `a`. Ainsi, la donnée n'est stockée qu'une fois en mémoire. La modification de `a` par réaffectation se fait avec modification d'adresse, donc avec copie ailleurs du résultat. Comme `b` pointe toujours vers l'ancienne adresse, le contenu de `b` n'est pas modifié. Cela peut se représenter ainsi :

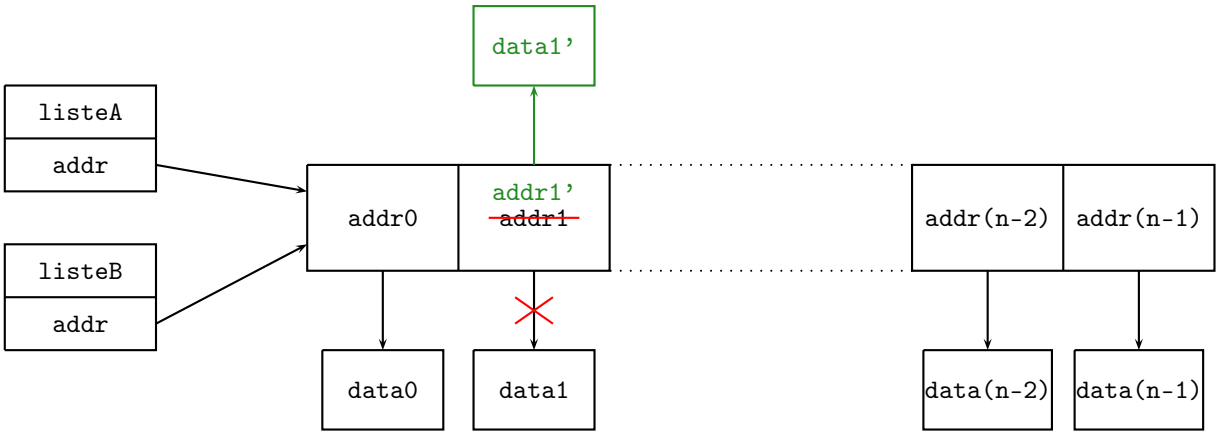


De façon générale, on a toujours indépendance entre une variable et sa copie lorsqu'on travaille avec un objet non mutable (car toute modification se fait avec changement d'adresse).

Voyons maintenant ce qu'il en est des listes. En particulier, que se passe-t-il sur une copie d'une liste lorsqu'on modifie un attribut de la liste initiale ?

```
>>> listeA=[1,2,3]
>>> listeB = listeA
>>> id(listeA)
140293216095552
>>> id(listeB)
140293216095552
>>> listeA[1]+=2
>>> listeA
[1, 4, 3]
>>> listeB
[1, 4, 3]
```

Les modifications faites sur la liste initiale sont aussi visibles sur la copie ! Ainsi, la copie n'est pas du tout indépendante de l'original. Ceci s'explique bien par le diagramme suivant :



La modification n'affecte par directement l'adresse de `listeA`, mais une adresse d'un des attributs, adresse stockée en un espace mémoire vers lequel pointe également `listeB`.

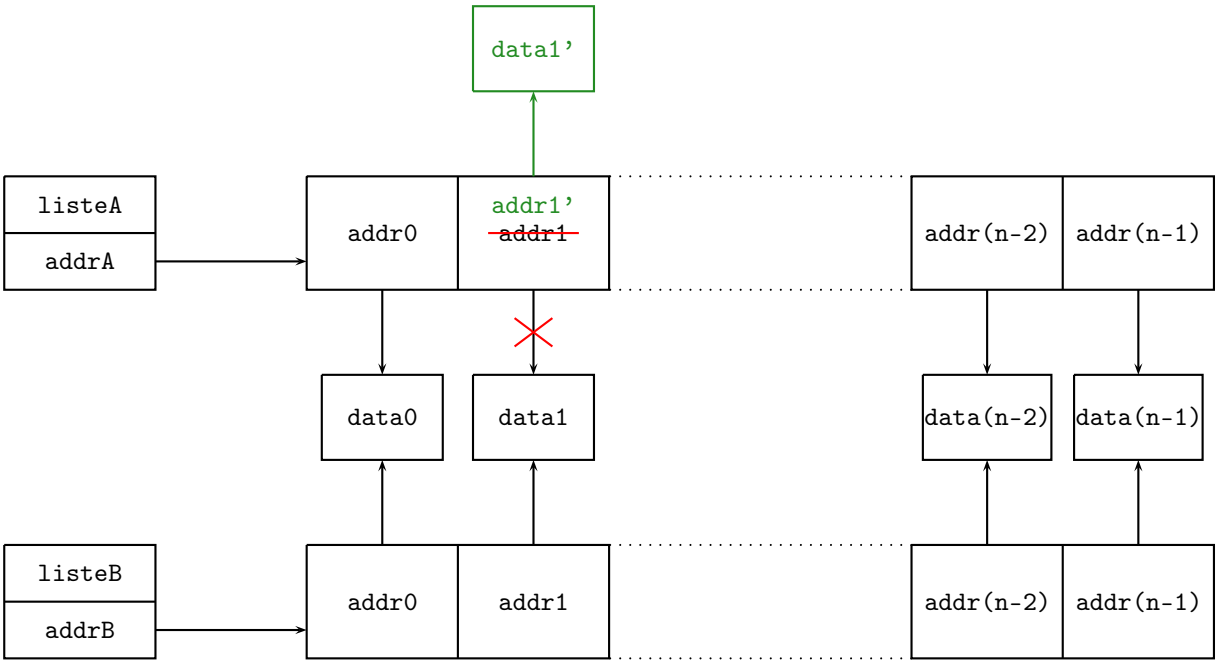
Pour rendre les copies « plus » indépendantes, il faut recopier les adresses des données ailleurs. Cela peut se faire par exemple par un slicing (couper une tranche d'une liste), qui se fait par copie des adresses en un autre emplacement mémoire. En revanche, les données elles-mêmes conservent la même adresse (donc les identifiants des attributs de `listeA` et `listeB` sont les mêmes) :

```
>>> listeA = [1,2,3]
>>> listeB = listeA[:]
>>> id(listeA)
140379366079808
>>> id(listeB)
140379366057168
>>> id(listeA[0])
211273705440
>>> id(listeB[0])
211273705440
```

Modifions maintenant un attribut de `listeA`, et voyons l'effet sur `listeB` :

```
>>> listeA[0]+=3
>>> listeA
[4, 2, 3]
>>> listeB
[1, 2, 3]
>>> id(listeA[0])
211273705536
>>> id(listeB[0])
211273705440
```

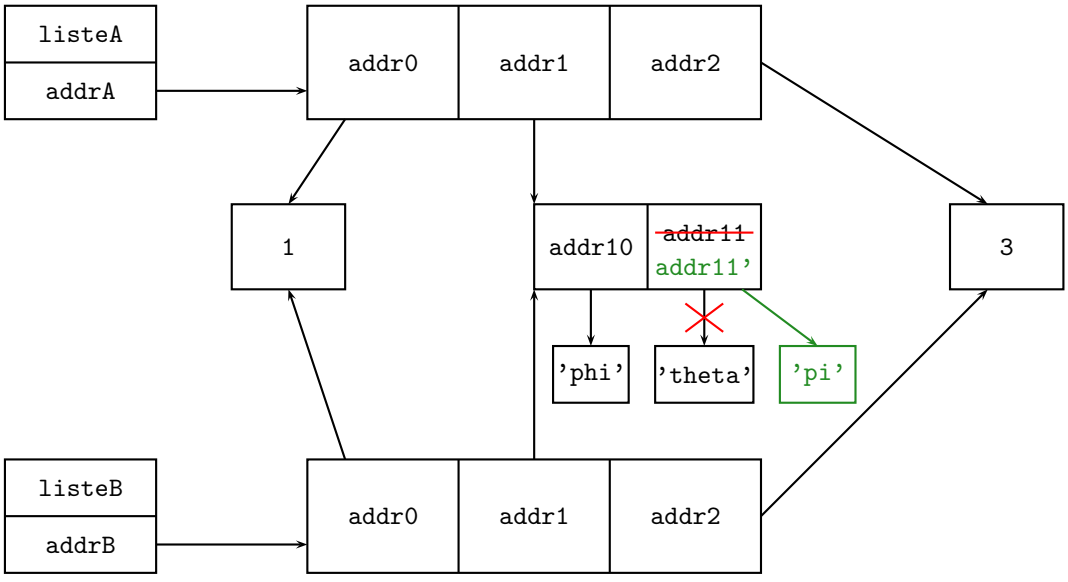
Cette fois, la modification n'est pas faite sur `listeB`. Ceci s'explique aisément par le diagramme suivant :



Mais cela ne règle pas entièrement le problème. En effet, si les données elles-mêmes sont mutables, elles peuvent être modifiées sans changement d'adresse. Dans ce cas, la copie pointe encore vers la même adresse que l'objet modifié : la modification est à nouveau visible sur la copie. Nous illustrons cela dans le cas où un des attributs est une liste :

```
>>> listeA=[1,['phi','theta'],3]
>>> listeB = listeA[:]
>>> listeA[1][1] = 'pi'
>>> listeA
[1, ['phi', 'pi'], 3]
>>> listeB
[1, ['phi', 'pi'], 3]
```

Cela s'illustre par le diagramme suivant :



Pour régler complètement le problème, on peut faire une copie récursive, qui va explorer la liste en profondeur, afin de recopier à des adresses différentes les sous-listes, sous-sous-listes etc. Une copie récursive

se fait avec la fonction `deepcopy` disponible dans le module `copy` :

```
>>> listeA = [1, ['phi','theta'],3]
>>> import copy
>>> listeB = copy.deepcopy(listeA)
>>> listeA[1][1] = 'pi'
>>> listeA
[1, ['phi', 'pi'], 3]
>>> listeB
[1, ['phi', 'theta'], 3]
```

Ainsi, l'original et la copie sont complètement indépendants.