Méthodes de programmation

Notion de mutabilité et effet de bord

Anicet E. T. Ebou, ediman.ebou@inphb.ci

01

Modification des listes et des chaînes de caractères

Notion de mutabilité

Dans votre console python essayez le code suivant:

```
liste = [0, 1, 2, 3, 4]

liste[2] = 0

liste
```

Que constatez vous? La modification a-t-elle pu être effectuée?

Notion de mutabilité

Ensuite, toujours dans votre console Python essayez le code suivant:

```
nom = “soro”
nom[2] = “y”
```

Que constatez vous? La modification a-t-elle pu être effectuée?

Notion de mutabilité

On constate donc que:

- **Les listes sont des objets modifiables**: on peut supprimer des éléments, modifier des éléments et rajouter des éléments.
- **Les chaînes de caractères ne sont pas des objets modifiables**. On ne peut pas modifier ou supprimer les éléments d'une chaîne de caractères. La seule possibilité pour modifier une chaîne de caractère est malheureusement de la recréer entièrement !

Notion de mutabilité

Définition *- Un objet est dit mutable si on peut le modifier.*

Explications

Lorsqu'on crée une variable, c'est-à-dire lorsqu'on écrit l'instruction suivante :

```
x = val
```

où `x` est le nom de notre variable et `val` est la valeur que l'on a attribuée à `x`, alors on crée des pointeurs entre `x` et les emplacements en mémoire contenant les valeurs données. Python *ne crée pas* une boîte, l'appelle `x` et met `val` dedans. L'opération est plus délicate et permet d'optimiser la mémoire.

Explications

En réalité, Python crée quelque part dans la mémoire une case dans laquelle il met la valeur `val` et il conserve en mémoire un tableau des variables du type: nom, type, adresse. `x` a donc pour nom `x`, pour type `type(x)` et pour adresse l'adresse dans laquelle est stockée `val`.

On a accès à cette adresse à travers la demande suivante :

```
id(x)
```

Explications

Définition - *On dit que x est un* ***pointeur****, il ne sait pas ce que vaut pas sa donnée mais connaît seulement l'adresse de sa donnée. Cette adresse s'appelle* ***l'identifiant*** *de la variable x.*

Adresse d’une donnée non mutable

Quand une donnée n’est pas mutable, *elle est à un seul endroit de la mémoire*. Si on recrée une variable avec la même valeur, *elle pointera vers la même adresse* (ce qui permet d’optimiser la mémoire).

Si l’on affecte deux variables à cette même valeur, les deux variables pointent alors vers la même adresse mémoire.

Adresse d’une donnée non mutable

Par exemple, essayez le code suivant dans votre console Python:

```
nom = “soro”
id(nom)
nom1 = nom
id(nom1)
```

Vous remarquerez alors que les variables `nom` et `nom1` ont les mêmes adresses (par exemple en faisant: `id(nom) == id(nom1)`).

Adresse d’une donnée mutable

Un objet mutable peut se retrouver à plusieurs endroits différents de la mémoire. Si l’on crée deux listes, elles auront des adresses différentes, même si elles contiennent initialement les mêmes éléments.

Si on recrée une variable avec la même valeur, elle pointera vers une nouvelle adresse. Lorsque l’on modifie la valeur d’une liste, on ne change pas son adresse.

Adresse d’une donnée mutable

```
x = [1, 2, 3]
id(x)
x.append(12)
id(x)
```

Vous remarquerez alors que l’adresse de la liste ne change pas.

Adresse d’une donnée mutable

Une liste étant mutable, on peut la modifier sans pour autant créer une nouvelle référence vers l’objet : l’adresse mémoire reste la même. Quand on copie une chaîne de caractères, on a autant d'adresses mémoire que de chaînes de caractères.

02

Aliasing

Aliasing

Le principal problème que l'on va rencontrer est lors des copies. Si x est une variable (mutable ou non), l'instruction :

```
y = x
```

se contente de créer un nouveau référencement vers le même emplacement en mémoire. Ainsi, si x a la valeur `var` alors en tapant y=x, y est une variable du même genre que x avec la même valeur que x et surtout la même adresse que x.

Copie d’objet non mutable

La copie n’est pas un problème lorsque les objets ne sont pas mutables (par exemple pour les réels, les entiers, les booléens, et les chaînes de caractères). Si on veut modifier le premier objet, il faudra tout réécrire et **ceci n’affectera pas la copie** (car en réécrivant le premier objet, il prend une nouvelle adresse mémoire).

Copie d’objet non mutable

Lors d’une copie, les deux variables pointent initialement vers le même emplacement mémoire, mais elles ne sont pas associées définitivement. En effet, si l’on modifie la valeur de la première variable, son adresse changera mais la seconde variable ne sera pas affectée.

Veuillez essayer successivement les lignes de codes Python suivantes dans votre console pour visualiser ce qui est expliqué.

Copie d’objet non mutable

```
ch1 = '123'
id(ch1)
ch2 = ch1
id(ch2)
id(ch1) == id(ch2)
ch1 = '456'
id(ch1)
ch2
id(ch1) == id(ch2)
```

Copie d’objet mutable

La copie pose souci lorsque la variable est mutable car, quand x est un objet mutable, le y=x signifie faire une référence : x et y sont le même objet et auront donc toujours les mêmes éléments. Ainsi, **modifier la liste** x **va également modifier** y puisque x et y référencent la même adresse.

Copie d’objet mutable

Saisissons le code suivant dans la console Python

```
x = [1, 2, 3]
y = x
y
x[1] = 5
y
```

L’instruction x[1] = 5 modifiant x, modifie, au passage, y puisque ces objets sont les mêmes.

Copie d’objet mutable

Définition - Ce phénomène porte le nom d’**aliasing**: les deux variables x et y sont deux alias différents de la même liste.

Astuces pour lutter contre l'aliasing

Pour avoir une copie indépendantes d'une liste x sans être gêné par l'aliasing, les tactiques les plus classiques sont :

- **Astuce 1 : slicing**. On écrit ceci :

```
y = x[:]
```

- **Astuce 2 : la méthode copy.** On écrit ceci :

```
y = x.copy()
```

Astuces pour lutter contre l’aliasing

Attention, ces deux techniques ne fonctionnent pas lorsque les listes contiennent elles-mêmes d’autres listes. On utilise alors la fonction `deepcopy()` du module `copy` pour s’en sortir.

Effets de bord

Définition - Une variable globale non mutable ne peut pas être modifiée à l'intérieur d'une fonction. En revanche, un objet mutable peut être modifié directement par l'application d'une fonction, c'est l'**effet de bord**.

Une fonction est à effet de bord si elle modifie une variable en dehors de son environnement local. En Python, cela se rencontre souvent avec les structures mutables comme les tableaux (de type list) ou les dictionnaires (de type dict).

Effets de bord

```
tab = [1, 1, 2, 3, 5, 8, 13]
def func():
    tab.append(tab[-1]+tab[-2])
func()
print(tab)
```

À chaque appel de `func()`, la variable `tab` est modifiée alors qu'elle est extérieure au code qui définit `func()`. Cela ne génère pas d'erreur mais peut entraîner des erreurs de conceptions plus globales puisqu'il devient difficile de maîtriser le contenu de la variable `tab`.

Effets de bord

Les variables de type immuable (`int`, `float`, `str`, `tuple`, etc.) ne permettent pas d'utiliser nativement les effets de bord. Cela peut entraîner des erreurs du type `UnboundLocalError`, qui signifie que l'on veut utiliser une variable **globale** comme locale.

Effets de bord

Essayez le code suivant et constatez l'erreur qui est retournée.

```
mot = “Python”
def func():
    mot = mot + “3”
func()
print(mot)
```

Effets de bord

En programmation fonctionnelle il est considéré un avantage de programmer sans effets de bord. S'il n'y a pas d'effets de bord, on est sûr qu'une fonction renvoie toujours la même valeur, si elle est appelée avec les mêmes arguments. Ceci permet un raisonnement plus approfondi sur la correction du programme.