Fonctions et mutables

1 - Problématique

1.1 Paramètre référençant un type simple

LECTURE POSSIBLE

On peut accèder en lecture à un contenu placé dans un paramètre. Sinon, tout serait un peu compliqué...

```python
def lecture(entree) :
    """La fonction lit le paramètre"""
    return entree

a = 12
print( lecture(a) )  # Affiche 12
```

MODIFICATION IMPOSSIBLE par nouvelle affectation depuis la fonction

Une fonction ne peut pas modifier une variable globale (c'est à dire du programme principal). Si on réalise une affectation dans la fonction, on crée simplement une **variable locale** dans l'espace (local) des noms de la fonction.

```python
def modification(entree) :
    """La fonction ne parvient pas à MODIFIER l'argument stocké dans le paramètre entree"""
    entree = 0     # Affectation donc nouvelle variable locale !
    return entree  # Renvoi du contenu local

a = 12
print( modification(a) )
print( a )
```

On obtient :

```
0
12
```

On renvoie bien le 0 référencé par la variable locale de la ligne 3 mais la variable globale **a** n'a pas été modifiée.

Modification par nouvelle affectation dans le programme lui-même

La seule manière de tenir compte de la réponse de la fonction serait de faire cela :

```python
def modification(entree) :
    """La fonction ne parvient pas à MODIFIER l'argument stocké dans le paramètre entree"""

    entree = 0     # Affectation donc nouvelle variable locale !
    return entree  # Renvoi du contenu local

a = 12
a = modification(a)
print( a )
```

On affiche bien `0`

Remarquez bien que cela fonctionne car il y a une affectation dans le programme principal. Ce n'est pas la fonction "qui agit" : elle ne fait que transmettre une réponse et c'est le programme qui réalise l'affectation. **Il ne s'agit donc pas d'une modification depuis la fonction**.

1.2 Paramètre référençant un tableau

LECTURE POSSIBLE

On peut accèder en lecture à un contenu placé dans un paramètre-tableau. Sinon, tout serait un peu compliqué...

```python
def lecture(tableau :list[int], i :int) -> int :
    """Lit la case i du tableau"""
    return tableau[i]

a = [20, 10, 30]
print( lecture(a, 2) )   # Affiche 30
```

L'argument a est envoyé dans le paramètre tableau. La fonction parvient à lire le contenu de la case 2 du tableau, à savoir 30.

MODIFICATION POSSIBLE du contenu

Plus étonnant : la modification du contenu d'un type construit mutable reçu dans un paramètre est possible. Notez bien qu'**on modifie le contenu t[i]** des cases, pas le conteneur t lui-même.

```python
def mise_a_0(tableau :list[int], i :int) -> None :
    """Modifie la case i en y plaçant 0"""
    tableau[i] = 0      # Modification de la case i, pas du tableau !
    # Aucun retour, donc équivalent à return None

a = [20, 10, 30]
mise_a_0(a, 2)
print(a)
```

On obtient :

```
[20, 10, 0]
```

Le tableau a bien été modifié dans la fonction et la modification est bien effective puisqu'elle apparaît encore dans le programme principal.

Pourquoi est-ce que cela fonctionne ?

La seule manière de tenir compte de la réponse de la fonction serait de faire cela :

L'explication arrivera plus loin. Pour le moment, retenons qu'en Python :

— Une fonction ne peut pas modifier une variable globale passée en paramètre.
— Une fonction peut modifier le CONTENU d'une variable mutable passée en paramètre.
— Bien entenu, une fonction ne peut pas modifier le CONTENU d'une variable immutable passée en paramètre !

2 - Exemple : tableau modifié ou copie modifiée

Pour répondre aux exercices, il est fondamental de bien chercher cette information : la fonction doit-elle renvoyer une copie modifiée du tableau (on utilisera alors une création par compréhension) ou doit-elle modifier directement le tableau ?

Modification x2

On doit utiliser le parcours par indices pour réaliser la modification.

```python
def fois2(t :list) -> None :
    for i in range(len(t)) :
        t[i] = t[i] * 2
```

Copie x2

Cette fois, on doit utiliser par exemple la création par compréhension.

```python
def fois2(t :list) -> list :
    copie = [v for v in t]
    for i in range(len(copie)) :
        copie[i] = copie[i] * 2
    return copie
```

On peut même faire plus court :

```python
def fois2(t :list) -> list :
    return [v*2 for v in t]
```

3 - Effet de bord : l'explication sur les modifications d'un mutable

3.1 Espace des noms : variables 6.3.4

3.2 Tableau statique : type construit 5.3.1

3.3 Fonction native id()

La fonction native `id()` renvoie l'identifiant de la zone-mémoire qui est associée à une variable dans l'espace des noms.

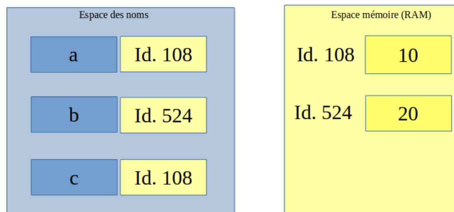

FIGURE 12.1 – espace des noms

```
>>> a = 10
>>> id(a)
__108

>>> id(10)
__108
```

3.4 Tableaux : copie et alias

A - Alias

Des **alias** sont des variables qui **pointent exactement vers le même contenu** en mémoire.

Si on modifie le contenu pointé en utilisant l'un des alias, l'utilisation de l'autre alias va également pointer vers le contenu modifié.

Pour réaliser des alias en Python, il suffit de réaliser une affectation : les deux variables vont pointer vers le même contenu mutable.

```
>>> t1 = [10, 20, 30]
>>> t2 = t1
# t2 et t1 référencent le même contenu

>>> id(t1)
139936748427648

>>> id(t2)
139936748427648

>>> t1[0] = 1000  # On modifie t1
>>> t1
[1000, 20, 30]
>>> t2            # t2 est modifié aussi
[1000, 20, 30]
```

B - Copie

Une **copie** pointe vers **un contenu-mémoire qui a exactement le même contenu qu'une autre zone-mémoire**.

Si on modifie le contenu de la copie, on ne modifie pas le contenu initial.

Si on modifie le contenu initial après copie, on ne modifie pas la copie.

Pour réaliser des copies de tableaux contenant des types simples en Python, il suffit de réaliser une création par compréhension

```
>>> t1 = [10, 20, 30]
>>> t2 = [v for v in t1]
# contenu similaire à t1 mais ailleurs

>>> id(t1)
139936748427648

>>> id(t2)
139936751349056

>>> t1[0] = 1000  # On modifie t1

>>> t1
[1000, 20, 30]

>>> t2            # t2 n'est PAS modifié
[10, 20, 30]
```

C - Visuel des cas précédents

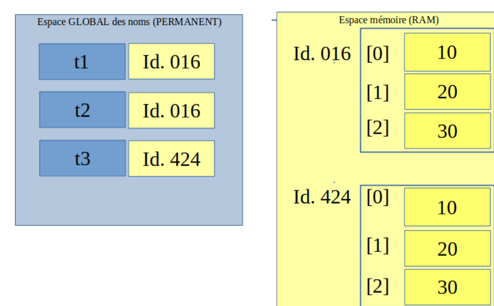

FIGURE 12.2 – alias et copie

`t1` et `t2` sont des alias puisqu'elles référencent la même zone mémoire.

`t3` est une copie du contenu précédent : valeurs identiques mais dans une zone mémoire différente.

4 - Bilan fonction et type constuit mutable

4.1 Paramètre référençant un mutable

Une **variable Python** n'est pas une boîte contenant un contenu.

1. En Python, l'espace des noms permet de comprendre la variable comme une traduction entre un nom et une référence-mémoire (dont on peut connaître l'identifiant avec `id()`).

2. En particulier, une variable référençant un type construit symbolise l'adresse du conteneur et pas les valeurs contenues

3. En conséquence, si on transmet à une fonction un argument de type construit et mutable, le paramètre correspondant va mener à la même zone mémoire que l'argument : l'argument et le paramètre sont des alias du même conteneur.

4. Conclusion : modifier le contenu du conteneur en utilisant le paramètre (local) va donc également modifier le contenu de l'argument envoyé.

4.2 Fonction : lire un tableau

Pour lire le contenu d'un tableau depuis une fonction : deux méthodes possibles

1. L'accès par indices

```
def afficher(t) :
    for i in range( len(t) ) :
        print(t[i])

notes = [15, 18, 8, 10, 12, 15, 20, 5, 12, 1
afficher(notes)
```

2. L'accès par valeurs

```
def afficher(t) :
    for v in t :
        print(v)
```

4.3 Fonction : modifier un tableau

Pour modifier un tableau depuis une fonction : **il faut passer par les indices**.

```
def modifier(t) :
    for i in range( len(t) ) :
        t[i] = t[i] ** 2
```

La modification est possible par effet de bord : le paramètre reçoit en réalité l'identifiant du tableau initial et devient ainsi un alias du tableau initial.

4.4 Fonction : renvoyer une copie d'un tableau

Pour renvoyer un nouveau tableau basé sur un tableau-paramètre depuis une fonction : une seule méthode utilisable :

- Créer une copie du tableau initial
- Modifier la copie en utilisant les indices
- Renvoyer la copie (et la stocker dans une variable !)

```
def copier_et_modifier_la_copie(t) :
    copie = [v for v in t]
    for i in range( len(copie) ) :
        copie[i] = copie[i] * 2
    return copie
```

Dans la plupart des cas, la création par compréhension peut résoudre le problème posé en une ligne :

```
def copier_et_modifier_la_copie(t) :
    return [v*2 for v in t]
```

www.infoforall.fr

CC BY NC SA