Lycée Les 3 Sources — **Bourg-Lès-Valence**

NSI 1re – Thème 2 — Année **2025-26**

Portée des variables et données (im)muables

Portée des variables

Lors de l'exécution d'un programme, la valeur de chacune des variables est stockée en mémoire. Lors d'une affectation, on associe une adresse mémoire au nom de la variable. Lorsqu'on utilise la variable dans une expression, c'est la valeur qui se trouve à l'adresse mémoire associée qui est utilisée. Lorsqu'on réaffecte une nouvelle valeur à une variable, elle peut rester associée à la nouvelle adresse ou s'en voir attribuer une nouvelle. Mais que se passe-t-il lorsqu'on utilise le même nom de variable dans le programme principal et dans une fonction ?

```python
def double(x):
    x = 2*x
    print(x)
```

```
>>> x = 5
>>> double(x)
10
>>> x
5
```

On peut remarquer que la valeur de `x` n'a pas été modifiée après l'exécution de la fonction. Pourtant `x` valait bien `10` dans la fonction. C'est parce qu'il y a en fait 2 variables `x`. Une qui est **globale** et l'autre qui est **locale** à la fonction. Lors de l'exécution de la fonction, de nouvelles variables sont créées en mémoire, correspondant aux paramètres et aux variables utilisées par la fonction. Ces variables sont locales et leur **portée** est limitée à l'exécution de la fonction. Une fois celle-ci finie, la variable globale `x` est retrouvée.

```
global
  x  5
  double
    x  10
```

Dans l'exemple ci-contre, on définit une variable globale `N` qui est utilisée dans la fonction. Il y a une variable locale créée pour chaque paramètre et pour chaque variable dont la valeur est modifiée dans le corps de la fonction. Ici, il s'agit de `x` et `y`. C'est donc la valeur globale de `N` qui est utilisée. C'est pour cela qu'il est conseillé de n'utiliser que des constantes au niveau global et de donner en paramètre les autres valeurs.

Par contre, il faut faire attention à ne pas mélanger valeur globale et locale :

```python
N = 2
def plus_N(x):
    y = x + N
    return y
```

```
>>> x = 3
>>> plus_N(x)
5
>>> x
3
```

```
global
  N  2
  x  3
  plus_N
    x  3
    y  5
```

```python
N = 2
def attention():
    N = N + 1
    return N
```

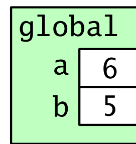
```
>>> attention()
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
  File "...", line 31, in attention
    N = N + 1
UnboundLocalError: local variable 'N' referenced before assignment
```

Puisque la valeur de `N` est modifiée dans la fonction, elle est considérée comme une valeur locale et n'a donc pas de valeur initiale. On ne peut donc pas mélanger les valeurs globale et locale d'une variable, sauf si celle-ci correspond à un objet mutable.

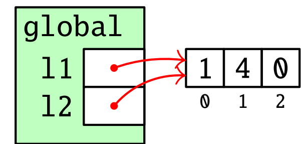
Mutable ou pas

On considère les 2 exemples suivants :

```
>>> a = 5
>>> b = a
>>> a = a + 1
>>> b
5
```




```
>>> l1 = [1, 4]
>>> l2 = l1
>>> l1.append(0)
>>> l2
[1, 4, 0]
```

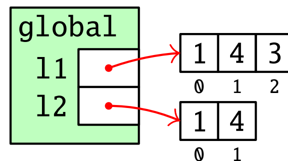


Dans le premier exemple, on peut remarquer que même si b est définie à partir de a, sa valeur est inchangée lorsqu'on modifie a. Dans le second exemple, au contraire, modifier la liste l1 modifie également la liste l2. La différences entre ces deux exemples vient de la façon dont les objets sont stockés. Un objet simple est directement stocké dans la case mémoire. Au contraire, pour des objets complexes, comme les listes ou les dictionnnaires, la case mémoire stocke l'adresse où se trouve le reste de l'objet. On parle de **pointeur** dans d'autres langages. Ainsi, l1 et l2 pointent sur le même objet. Si on modifie l'un, on modifie l'autre. C'est l'objet lui même qui est modifié et pas l'adresse stockée par les variables qui pointent sur lui.

En Python, pour modifier la valeur associée à une variable, soit on fait une affectation, ce qui change la valeur ou l'adresse stockée, soit on modifie directement l'objet pointé. Seuls certains objets peuvent être modifiés sans affectation et sont dits **mutables**. C'est le cas des listes et des dictionnaires qui ont des opérations spécifiques pour ajouter ou modifier les valeurs qu'ils contiennent.

Au contraire, des objets comme les n-uplets ou les textes sont **immuables**. Il n'y a qu'une affectation qui permet de modifier la valeur d'une variable associée à un objet immuable. Cela crée un nouvel objet et ne change pas l'objet initial, ni les éventuelles variables qui pointent dessus. Pour copier un objet mutable, on peut utiliser la méthode copy() :

```
>>> l1 = [1, 4]
>>> l2 = l1.copy()
>>> l1.append(3)
>>> l2
[1, 4]
```



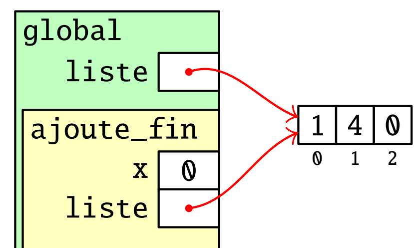

```
>>> dico1 = {"a": 1, "b": 2}
>>> dico2 = dico1.copy()
>>> dico2["b"] = 7
>>> dico1
{'a': 1, 'b': 2}
```

Conséquences sur les fonctions

Comme nous l'avons vu précédemment, les variables correspondant à des nombres ne sont pas modifiées lorsqu'elles sont passées en paramètre d'une fonction. Ce n'est pas le cas pour les types mutables comme les listes.

```
def ajoute_fin(liste, x):
    liste.append(x)
```

```
>>> liste = [1, 4]
>>> ajoute_fin(liste, 0)
>>> liste
[1, 4, 0]
```



C'est parce que la variable locale utilisée dans la fonction pointe vers le même objet que la variable globale. Les modifications de cet objet persistent donc après l'appel à la fonction. On dit que la liste est modifiée **en place**. Il faut faire attention à l'utilisation d'objets mutables dans des fonctions. Si on ne veut pas modifier la valeur de départ, il faut utiliser une copie de l'objet lors de l'appel ou dans la fonction.