20200409-biost1_cent1_est1_synthese_portee_variables

April 9, 2020

1 Explication détaillée de la gestion des variables en Python

- En Python, le nom d'une variable peut être vu comme une étiquette pointant vers une zone mémoire. On peut récupérer l'identifiant d'une zone mémoire en appelant la fonction `id(nom_variable)`.
- Il existe deux grandes familles de variables :
  - Celles de type "non mutables" : il s'agit des types de base (str, int, float, bool) plus quelques autres (tuples, frozen set...)
  - Celles de type "mutables" : concerne les types "complexes" (list, dict, set, ...) qui peuvent être modifiés intrinsèquement (par l'indice, la clé...).

1.1 Portée des variables

L'endroit où est déclarée une variable détermine son niveau d'accessibilité dans le programme, c'est ce qu'on appelle la portée.

1.2 Variables non mutables

Si on affecte une valeur à une variable, puis qu'on affecte une autre valeur à cette même variable, on va créer deux zones mémoire distincte et le nom de la variable (l'étiquette) va pointer d'abord vers la première zone puis ensuite vers la seconde zone (la fonction `id` nous permet ici d'identifier la zone mémoire).

Exemple :

```
a = 2
print(id(a))  # ex. 139902503606208
a = 3
print(id(a))  # ex. 139902503606240
```

A noter que si le contenu est identique, l'id reste le même :

```
a = 2
print(id(a))  # ex. 139902503606208
a = 2
print(id(a))  # c'est le même
```

Notons aussi le comportement avec deux noms de variable différents pour un même contenu :

```
a = 2
print(id(a))  # ex. 139902503606208
b = a
print(id(b))  # c'est le même
```

1.3 Variables mutables

On les appelle variables mutables car on peut les modifier intrinsèquement sans changer de zone mémoire. Par exemple :

```
lst = [1, 2, 3]
print(id(lst))  # ex. 139902398225728
lst[0] = 8
print(id(lst))  # C'est le même
lst.append(20)
print(id(lst))  # C'est le même
```

En revanche, comme pour les variables non mutables, une nouvelle affectation provoque un changement d'id :

```
lst = [1, 2, 3]
print(id(lst))  # ex. 139902398225088
lst = [7, 2, 4]
print(id(lst))  # ex. 139902398225536
```

A noter que si le contenu est identique, l'id est tout de même changé (à la différence des variables non mutables) :

```
lst = [1, 2, 3]
print(id(lst))  # ex. 139902398224960
lst = [1, 2, 3]
print(id(lst))  # ex. 139902398225408
```

Cependant comme pour les variables non mutables, deux noms de variables peuvent pointer vers le même contenu :

```
lst = [1, 2, 3]
print(id(lst))  # ex. 139902398224576
lst2 = lst
print(id(lst2))  # C'est le même
```

Corrolaire :

```
lst = [1, 2, 3]
lst2 = lst  # Ce n'est pas une copie, mais un "pointage" identique
lst2[0] = 6
print(lst2)
print(lst)  # lst a aussi été modifié
```

1.4 Lecture du contenu d'une variable en fonction de sa portée

1.4.1 Déclaration hors fonction : variable globale

Lorsqu'une variable est déclarée en dehors de toute fonction, c'est une **variable globale**, c'est à dire qu'elle est accessible (on peut lire son contenu) dans l'intégralité du programme.

```
def ma_fonction1():
    # La variable est accessible
    print(var_globale)

# Programme principal
var_globale = 3
ma_fonction1()
```

1.4.2 Déclaration dans une fonction : variable locale

Lorsqu'une variable est déclarée dans une fonction (ou figurant en paramètre de la fonction), la variable n'est accessible que dans la fonction, c'est ce qu'on appelle une **variable locale**.

```
def ma_fonction2():
    # La variable var_f2 est accessible uniquement dans la fonction
    var_f2 = 1
    print(var_f2)

def ma_fonction3(mon_param):
    # La variable mon_param est accessible uniquement dans la fonction
    print(mon_param)

# Programme principal
ma_fonction2()
print(var_f2)  # Erreur, variable inconnue !

ma_fonction3(4)
print(mon_param)  # Erreur, variable inconnue !
```

1.5 Modification du contenu d'une variable en fonction de sa portée

1.5.1 Variables de type non mutables (immuables)

Pour rappel, ce sont les variables ayant un type de base Python (str, int, float, bool) ainsi que quelques autres (tuples, frozen set…). Elles ne peuvent être "modifiées" qu'au niveau où elles ont été déclarées. Notons que le terme "modifiées" est quelque peu abusif, car comme nous l'avons vu précédemment, l'affectation d'une nouvelle valeur provoque le stockage dans une nouvelle zone mémoire et l'étiquette (le nom) de la variable pointera vers cette nouvelle zone.

- Une variable globale n'est modifiable que dans une zone hors fonction (dans le "programme principal"). Notons qu'il est tout de même possible de contourner cette limitation en spécifiant le mot-clé `global` suivi du nom de la variable globale dans une fonction. Cela est toutefois déconseillé car cela nuit à la lisibilité du programme.

- Une variable locale à une fonction n'est modifiable que dans cette fonction (c'est logique puisqu'elle n'est accessible en lecture que dans cette fonction).

```
def ma_fonction4():
    # L'instruction ci-dessous crée une nouvelle variable
    # également nommée ma_var, différente de la variable globale
    ma_var = 3

## PP
ma_var = 4
ma_fonction4()
print(ma_var)  # Le résultat est 4 !
```

Autre exemple avec un comportement similaire :

```
def ma_fonction4b(ma_var):
    # L'instruction ci-dessous crée une nouvelle variable
    # également nommée ma_var, différente de la variable passée en paramètre
    # (qui est en fait la variable globale)
    ma_var = 3

## PP
ma_var = 4
ma_fonction4b(ma_var)
print(ma_var)  # Le résultat est 4 !
```

1.5.2 Variables de type mutables

Pour rappel, il s'agit d'objets Python plus complexes (listes, dictionnaires, sets...).

- Une variable globale est non seulement accessible dans la fonction, mais modifiable (de façon intrinsèque) directement par cette dernière.
- Une variable locale à une fonction n'est accessible et modifiable que dans cette fonction (idem cas précédents).

```
def ma_fonction5():
    g_lst1.append(8)

def ma_fonction6(param_lst):
    param_lst.append(3)  # On modifie la liste initiale

## PP
g_lst1 = [1, 2, 3]
ma_fonction5()
print(g_lst1)  # [1, 2, 3, 8]

g_lst2 = [1]
ma_fonction6(g_lst2)
print(g_lst2)  # [1, 3]
```

- Corrolaire 1 : attention aux modifications des variables de type mutables dans les fonctions, ils le seront après l'appel de la fonction.
- Corrolaire 2 : si on souhaite qu'une fonction modifie une variable mutable définie au niveau global, il n'est pas nécessaire de la passer en paramètre (cf. la fonction `ma_fonction5`).

Remarque: si on réaffecte la variable mutable, alors on se retrouve comme dans le cas des variables immuables.

```
def ma_fonction6(param_lst):
    # L'instruction ci-dessous crée une nouvelle variable
    # également nommée param_lst mais différente de celle passée en paramètre
    param_lst = []

## PP
g_lst1 = [1, 2, 3]
ma_fonction6()
print(g_lst1)  # [1, 2, 3]
```

Même comportement ici :

```
def ma_fonction6():
    # L'instruction ci-dessous crée une nouvelle variable
    # également nommée g_lst1 mais différente de celle qui est globale
    g_lst1 = []

## PP
g_lst1 = [1, 2, 3]
ma_fonction6()
print(g_lst1)  # [1, 2, 3]
```