Chapitre 5 : Portée d'une variables, passage d'argument à une fonction

Spé ISN

24 septembre 2019

Introduction

Lorsqu'on utilise une variable, on a besoin de contrôler les valeurs qu'elle peut prendre et des effets de bord qu'une modification de sa valeur peut engendrer dans le reste du programme. Cela va dépendre de la portée de la variable.
On doit souvent passer des variables comme argument à des fonctions, là aussi on a besoin de comprendre ces mécanismes pour maîtriser les effets de bord : autoriser ceux qui sont désirés et empêcher les autres.

I Portée d'une variable

Définition 1

- Dans un code, la **portée d'une variable** définit les endroits du code où la variable est accessible.
- En `Python`, la portée d'une variable est lexicale c'est-à-dire qu'elle est définie par l'endroit où la variable est définie.
- En `Python`, une variable est **définie** dès qu'elle reçoit une valeur par une instruction d'**affectation**.

  C'est la dernière instruction d'affectation associée à un nom de variable qui détermine la portée de cette variable.
- Il existe deux grandes catégories de variables en `Python` :
  - Une variable définie dans une fonction est une **variable locale** à cette fonction, elle est accessible dans le bloc d'instruction de cette fonction et dans toutes les éventuelles fonctions qu'elle peut englober mais elle n'est pas visible dans tous les blocs qui englobent la fonction (programme principal ou fonctions englobantes).
  - Une variable définie dans le programme principal (pas dans une fonction) est une **variable globale**, elle est accessible dans l'ensemble du code.
- Un même nom de variable peut être utilisé dans une définition de variable par affectation à plusieurs niveaux d'imbrication de fonctions dans un même code.

  Pour déterminer la portée d'une variable utilisée à un endroit fixé du code, on applique la règle **LEGB** pour **Locale Englobante Globale Builtins**. On recherche d'abord la variable dans la portée de la fonction locale, puis dans la portée d'une fonction englobante, puis dans le programme principale à l'extérieur de toute fonction et enfin dans le module `builtins` qui est importé par défaut.

- Il est possible d'enfreindre la règle **LEGB** en utilisant le mot clef **global** si on veut définir une variable dans une fonction avec une portée globale ou avec le mot clef **nonlocal** si on veut définir une variable dans une fonction englobée avec une portée dans la fonction englobante.
- La portée des variables en `Python` est très bien expliquée dans cette video :

https://d381hmu4snvm3e.cloudfront.net/videos/qYPPwG7tu2eU/SD.mp4

Exercice 1

1. Que se passe-t-il lorsqu'on exécute les codes ci-dessous ? Commenter.

.....

.....

.....

.....

.....

Programme 1

```
a = 1

def f():
    a = 734
    print(a)

f()
print(a)
```

Programme 2

```
a = 1

def f():
    a = a + 1
    print(a)

f()
print(a)
```

2. Que se passe-t-il lorsqu'on exécute les codes ci-dessous ? Commenter en précisant la portée des variables `a`, `b` et `c` lors de chaque exécution de l'instruction `print(a, b, c)`.

.....

.....

.....

.....

Programme 3

```
a, b, c = 731, 734, 735
def f():
    b, c = 736, 737
    def g():
        c = 738
        print(a, b, c)
    g()
    print(a, b, c)
f()
print(a, b, c)
```

Exercice 2

Que se passe-t-il lorsqu'on exécute les codes ci-dessous ? Commenter en précisant la portée de la variable `a` lors des appels successifs des fonctions `incremente1(a)`, `incremente2()` puis `incremente3()`.

.....

.....

.....

Programme 4

```
def incremente1(a):
    a = a + 1

def incremente2():
    global a
    a = a + 1

def incremente3():
    a = a + 1

a = 734
for k in range(10):
    incremente1(a)
print(a)
for k in range(10):
    incremente2()
print(a)
for k in range(10):
    incremente3()
print(a)
```

II Passage d'argument à une fonction

Définition 2

```
def fonction(parametre1, parametre2):
    bloc de la fonction
```

Lorsque la définition d'une fonction comporte des paramètres, lors de l'appel de cette fonction on doit assigner à chaque **paramètre** une valeur obtenue par évaluation d'une expression calculée à partir de littéraux et de variables. Les valeurs effectives transmises à la fonction lors de l'appel sont appelées des **arguments**. On crée alors pour chaque argument variable locale à la fonction référencée par le nom du paramètre formel.
Lorsqu'on passe comme argument une variable, on distingue habituellement deux types de passage d'argument :

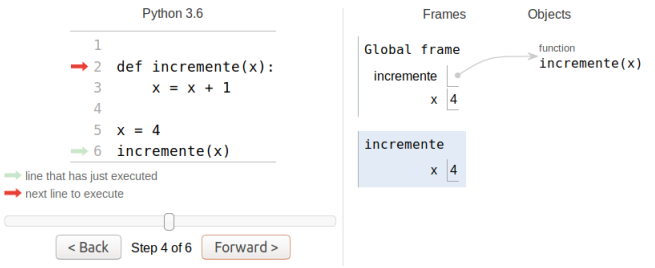
- Le **passage par valeur** : le paramètre formel reçoit une copie de la variable passée en argument. Dans ce cas une modification subie dans la fonction par la variable locale créée lors de l'appel ne se répercute pas sur la variable passée en argument.
- Le **passage par référence** : le paramètre formel reçoit une référence vers la valeur référencée par la variable passée en argument. Dans ce cas une modification subie dans la fonction par la variable locale créée lors de l'appel se répercute sur la variable passée en argument. Le passage par référence permet d'économiser de l'espace mémoire mais peut se traduire par des effets de bord.

En `Python`, on a vu qu'il existait des **objets non modifiables** comme les types de base (`int`, `float`, `bool`) ou les collections de type `tuple` et des **objets modifiables** comme les collections de type `list` ou `dict`.
En `Python` tous les passages d'argument se font par référence, même si pour les objets non modifiables, le comportement est le même que pour un passage par valeur. Lors du passage d'une variable en argument, le paramètre formel reçoit la référence de cette variable. Si l'objet référencé est modifiable (une liste, un dictionnaire), il pourra être modifié par la fonction, sinon (pour un entier, un flottant ...) il ne pourra pas être modifié.

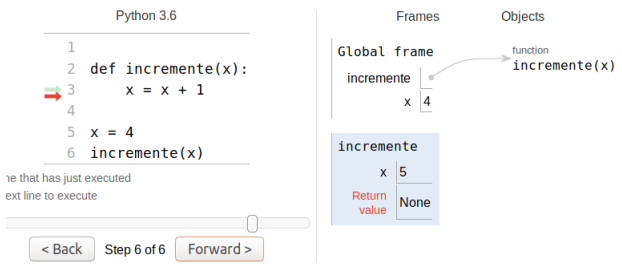
Exemple 1 *Passage en argument d'une variable référençant un objet non modifiable*

On donne ci-dessous un exemple de passage d'une variable référençant un entier non modifiable à une fonction, on peut observer que même si la variable globale et la variable locale à la fonction portent le même nom `x`, la variable locale disparaît à la fin de l'exécution de la fonction sans laisser de trace sur la variable globale. Testez en ligne sur PythonTutor.

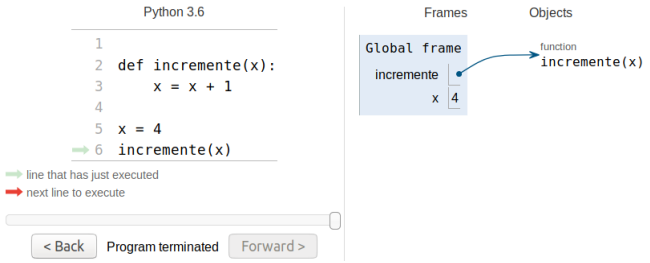
Étape 1



Étape 2



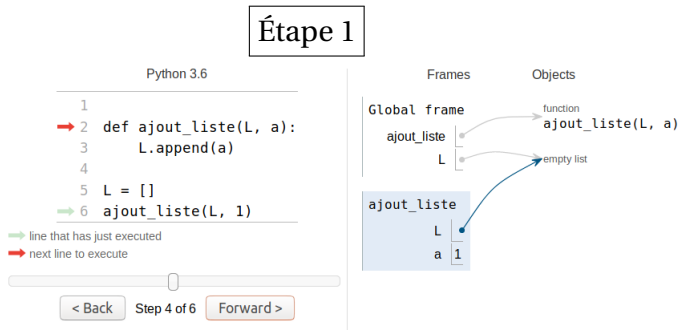
Étape 3



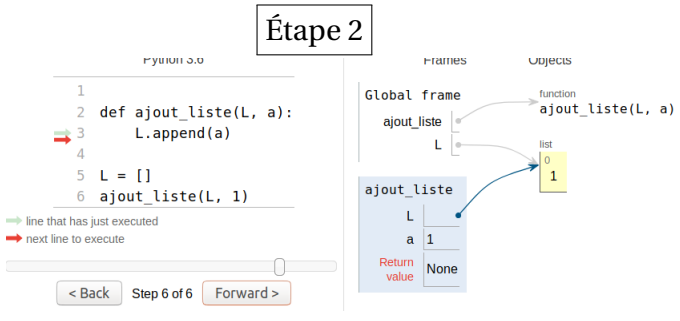
Exemple 2 *Passage en argument d'une variable référençant un objet modifiable*

On donne ci-dessous un exemple de passage d'une variable référençant une liste modifiable à une fonction, on peut observer l'effet de bord produit par chaque exécution de la fonction sur la liste passée en paramètre. Testez en ligne sur PythonTutor.

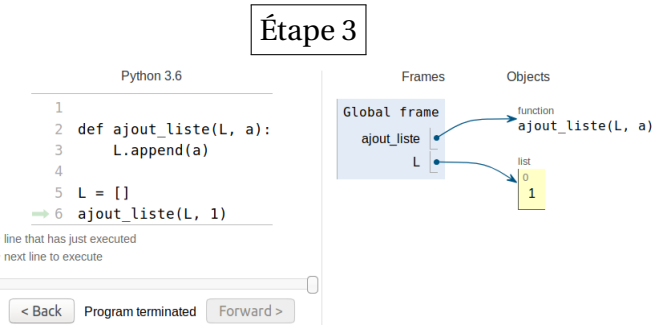
Étape 1



Étape 2



Étape 3



Exercice 3

1. Écrire une fonction `echange_valeurs(L, a, b)` qui prend en paramètre une liste `L` et deux entiers `a` et `b` et qui permute par effet de bord les valeurs de la liste `L` en positions `a` et `b`. On commencera par vérifier que `a` et `b` sont bien des index valides pour la liste.

2. Écrire une fonction `carre_liste(L)` qui prend en paramètre une liste `L` et qui modifie par effet de bord la liste `L` en remplaçant chacun de ses éléments par leur carré.

3. Écrire une fonction `carre_liste2(L)` qui prend en paramètre une liste `L` et qui retourne une nouvelle liste constituée des carrés des éléments de `L`, sans modifier la liste `L`.

4. Écrire une fonction `pair_impair(L)` qui prend en paramètre une liste `L` d'entiers et qui retourne une nouvelle liste dont constituée d'abord des éléments pairs de `L` puis des éléments impairs de `L`.

5. Écrire une fonction `pair_impair2(L)` qui prend en paramètre une liste `L` d'entiers et qui modifie en plaçant tous les éléments pairs de `L` au début de la liste et tous les éléments impairs de `L` à la fin de la liste.