**Numérique et sciences informatiques**
**Classe de première**
**---------**
**I Programmation et algorithmes**
**A Techniques de programmation**
**4. Variables locales et variables globales**

## I Les notions de variables globales et locales

### I.1 La distinction entre variable globale et variable locale

• Considérons le programme suivant dans lequel on a défini une fonction `tabCarres(n)` qui prend en argument un entier positif `n` et qui renvoie le tableau des carrés d'entiers compris entre 1 et `n` inclus :

```
def tabCarres(n) :
    """..."""
    tab = []
    for i in range(1, n+1) :
        tab.append(i**2)
    return tab

x = 12
monTab = tabCarres(x)
print(monTab)
```

L'exécution de ce programme génère l'affichage suivant :

```
>>>
= RESTART: .../carres.py
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

• Dans ce programme, il faut distinguer deux ensembles de variables :

* celles qui sont créées dans le programme principal : `x` et `monTab` ;
* celles qui sont créées dans la fonction `tabCarres` : `n`, `tab` et `i`.

• On dit que les variables du premier ensemble sont des **variables globales** alors que celles du deuxième ensemble sont des **variables locales** à la fonction `tabCarres`.

• Il est notamment important de remarquer que *les arguments d'une fonction sont des variables locales* à cette fonction. Dans notre exemple, la variable `x` est une variable globale alors que `n` est une variable locale.

• L'importance de la distinction entre variables globales et locales tient à la différence de leur comportement. Une variable locale à une fonction n'existe que pendant le temps où la fonction est exécutée et elle est détruite dès que cette exécution est finie. Nous pouvons le constater avec notre exemple en ajoutant l'instruction suivante à la fin du programme principal :

```
...
print(monTab)
print(n)
```

et en exécutant à nouveau. On obtient alors l'affichage suivant :

```
>>>
= RESTART: .../carres.py
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

```
Traceback (most recent call last):
  File "/.../carres.py", line 9, in <module>
    print(n)
NameError: name 'n' is not defined
```

On constate que l'appel de fonction `monTab = tabCarres(x)` et l'impression du résultat `print(monTab)` ont bien été exécutés. En revanche l'instruction `print(n)` a généré un message d'erreur indiquant que la variable `n` n'existe plus après l'exécution de l'appel de la fonction `tabCarres`. Bien entendu, on obtiendrait un résultat similaire avec les instructions `print(tab)` ou `print(i)`.

### I.2 Passage par argument

• Lorsque, dans le corps d'une fonction, on change la valeur d'une variable locale qui est un argument de cette fonction, ceci ne doit avoir aucun effet sur la variable globale utilisée lors de l'appel de la fonction. Par exemple, si on exécute le programme suivant :

```
def decupler(x):
    x = x*10
    return x

a = 5
b = decupler(a)
print("a =", a, "b =", b)
```

on obtient l'affichage suivant :

```
>>>
= RESTART: .../Decupler.py
a = 5 b = 50
```

La variable `a` n'a pas été modifiée par l'appel de la fonction `decupler` et notamment par l'instruction `x = x*10`. En effet, la variable `x` a été créée au moment de l'appel de la fonction `b = decupler(a)` et elle a alors pris la valeur de la variable `a`, c'est à dire 5. Pendant l'exécution de la fonction `decupler`, l'instruction `x = x*10` a modifié la valeur de la variable locale `x` *sans pour autant modifier la variable* `a`. On dit qu'il y a eu un **passage d'argument par valeur**.

### I.3 Accéder à la valeur d'une variable globale dans le corps d'une fonction

• En Python, il est possible d'accéder à la valeur d'une variable globale dans le corps d'une fonction. Il suffit pour cela d'y faire référence.

• Par exemple, l'exécution du programme :

```
PI = 3.14

def perimetreCercle(r) :
    return 2*r*PI

rayon = 7
print(perimetreCercle(rayon))
```

donne l'affichage suivant :

```
>>>
= RESTART: .../Perimetre.py
43.96
```

Dans ce programme la variable globale `PI` est créée *avant* l'appel de fonction `perimetreCercle(rayon)` et elle est utilisée dans le corps de la fonction `perimetreCercle`. On voit que ceci ne génère aucun message d'erreur.

Bien entendu, une variable globale utilisée dans le corps d'une fonction doit avoir été définie *avant l'appel de cette fonction*. Dans le cas contraire l'exécution du programme générera un message d'erreur indiquant que la variable n'a pas été définie.

• Par exemple, l'exécution du programme

```
def perimetreCercle(r) :
    return 2*r*PI

rayon = 7
print(perimetreCercle(rayon))
PI = 3.14
```

génère l'affichage suivant :

```
>>>
= RESTART: .../Perimetre.py
Traceback (most recent call last):
  File " .../Perimetre.py ", line 16, in <module>
    print(perimetreCercle(rayon))
  File " .../Perimetre.py ", line 13, in perimetreCercle
    return 2*r*PI
NameError: name 'PI' is not defined
```

• L'utilisation d'une variable globale dans une fonction est toutefois considérée comme une mauvaise pratique. En effet, le comportement de cette fonction va alors dépendre de la valeur de cette variable globale *au moment de l'appel de la fonction*. Cette technique est cependant utile et légitime si la variable globale est utilisée *comme une constante* qui ne sera donc jamais modifiée pendant l'exécution du programme. C'est bien ainsi que nous utilisons la variable `PI` dans notre exemple.

### I.4 Variable locale qui masque une variable globale de même nom

• Si on définit dans le programme principal une variable globale qui porte *le même nom* qu'une variable locale définie dans une fonction, alors cette variable globale n'est plus accessible dans le corps de la fonction. En effet, toute utilisation de ce nom dans le corps de la fonction sera considérée comme une référence *à la variable locale* qui porte le même nom. On dit alors que **la variable locale masque la variable globale de même nom**.

• Par exemple, si on exécute le programme suivant :

```
def maFct1():
    a = 14
    print("Dans la fonction, a =", a)

a = 25
maFct1()
print(("Dans le programme principal, a =", a)
```

on obtient l'affichage suivant :

```
>>>
= RESTART: .../varMasquee.py
Dans la fonction, a = 14
Dans le programme principal, a = 25
```

* Dans l'instruction `print(..., a)` de la fonction `maFct1`, le nom `a` est interprété comme désignant la variable locale `a` dont la valeur est 14

* L'instruction `print(..., a)` du programme principal conduit à l'affichage `a = 25`. On voit donc que la variable globale `a` n'a pas été modifiée par l'exécution de la fonction `maFct1()`.

• Il faut aussi noter que si on exécute le programme suivant :

```
def maFct2():
    print(a)
    a = 14

a = 25
maFct2()
```

on obtient le message d'erreur suivant :

```
>>>
= RESTART: .../varMasquee.py
Traceback (most recent call last):
  File ".../varMasquee.py", line 6, in <module>
    maFct()
  File ".../varMasquee.py", line 2, in maFct
    print(a)
UnboundLocalError: local variable 'a' referenced before assignment
```

Ici encore, l'instruction `print(a)` est interprétée comme faisant référence à la variable locale `a` définie *plus loin* dans le corps de la fonction `maFct()` et non pas à la variable globale `a`. Ceci génère donc un message d'erreur puisque on utilise cette variable locale avant de l'avoir définie.

• En revanche, si on exécute le programme suivant :

```
def maFct3():
    print(a)

a = 25
maFct3()
```

on obtient l'affichage suivant :

```
>>>
= RESTART: .../varMasquee.py
25
```

L'instruction `print(a)` est interprétée ici comme faisant référence à la variable globale `a` puisqu'aucune variable locale de même nom n'a été définie dans la fonction `maFct3()`.

### I.5 Modifier une variable globale dans une fonction : le mot-clé `global`

• Exécutons le programme suivant :

```
def fct1():
    var = 5

var = 10
fct1()
print(var)
```

on obtient l'affichage suivant :

```
>>>
= RESTART: .../global.py
10
```

Ce comportement illustre ce qui a été dit dans la partie précédente : l'instruction `var = 5` dans le corps de la fonction `fct1()` est interprété *comme la création d'une variable locale* `var` et *non comme une modification de la variable globale* de même nom.

• Ceci conduit à se demander s'il est possible en Python de modifier une variable globale dans le corps d'une fonction. La réponse est que l'on peut le faire en indiquant explicitement que l'on veut agir sur une variable globale à l'aide du mot **`global`**.

• Pour agir sur une variable globale dans une fonction, on doit utiliser la syntaxe suivante :

```
def maFct(...) :
    ...
    global maVar
    ...
    maVar = expression
```

• Cette syntaxe indique que, dans le corps de cette fonction, la variable *`maVar`* doit être considérée comme une variable globale *et non comme une variable locale*. L'instruction *`maVar = expression`* sera alors interprétée comme la modification d'une variable globale et non comme la création d'une variable locale de même nom.

Il est impératif que, dans le corps de la fonction, l'instruction **`global`** *`maVar`* *soit placée avant* l'instruction *`maVar = expression`*.

• Par exemple, l'exécution du programme suivant :

```
def fct2():
    global var
    var = 5

var = 10
fct2()
print(var)
```

générera l'affichage suivant :

```
>>>
= RESTART: .../global.py
5
```

On constate que la variable globale `var` a bien été modifiée par l'appel de fonction `fct2()`.

• En fait, si la fonction *`maFct(...)`* contient une instruction de la forme `global maVar` et une instruction de la forme `maVar = expression` alors un appel de fonction de la forme *`maFct(...)`* peut avoir *deux effets différents selon le contexte* de cet appel :

* si la variable `maVar` existe déjà au moment de l'appel, cette variable globale `maVar` sera *modifiée* ;
* si elle n'existe pas déjà, cette variable globale `maVar` *sera créée* et sa valeur initiale sera le résultat de `expression`. Dans ce cas, cette variable deviendra utilisable dans le programme principal ainsi que dans toute autre fonction.

## II Objets mutables et effets de bord

### II.1 La particularité des objets de types mutables

• En Python, il est important de distinguer deux catégories de types d'objets : les types immuables et les types mutables :

* un objet de **type immuable** est un objet dont la valeur ne peut pas être modifiée. C'est le cas d'un grand nombre des types que nous avons rencontrés, notamment les objets de types **`int`**, **`float`**, **`str`**, **`bool`** et **`tuple`**. Une variable dont la valeur est d'un de ces types peut changer de valeur mais l'objet python qui constitue cette valeur ne peut lui-même être modifié.

* un objet de **type mutable** peut être modifié. Nous avons jusqu'à présent rencontré deux types python mutables : les objets de type **`list`** (tableaux) et les objets de type **`dict`** (dictionnaires).

• Pour comprendre la particularité du comportement d'un objet mutable, prenons un exemple. Exécutons dans la console IDLE la succession de commandes suivantes :

```
>>> T1 = [1, 2, 3, 4]
>>> T2 = T1
>>> T2[0] = 50
>>> T2
[50, 2, 3, 4]
>>> T1
[50, 2, 3, 4]
```

Nous constatons qu'en modifiant le tableau T2 avec l'instruction T2[0] = 50, nous avons modifié dans le même temps le tableau T1.

• On peut se représenter le phénomène observé en disant que l'exécution de la commande `T2 = T1` a pour effet que les variables `T1` et `T2` « pointent » toutes les deux vers un même objet de type **`list`** dont la valeur initiale était `[1, 2, 3, 4]`. En exécutant l'instruction `T2[0] = 50`, on ne se contente pas de modifier la valeur de la variable `T2` mais on modifie le tableau vers lequel pointent simultanément les variables `T1` et `T2`. Ainsi cette instruction modifie également la valeur de la variable `T1`. Ce phénomène s'appelle un **effet de bord**.

• Si, en revanche, on donne à la variable `T2` la valeur d'un autre tableau, même identique par ses valeurs, alors une action sur l'un des tableaux n'aura aucun effet sur l'autre tableau. Par exemple, observons la suite de commandes suivante :

```
>>> T1 = [1, 2, 3, 4]
>>> T2 = [1, 2, 3, 4]
>>> T2[0] = 50
>>> T2
[50, 2, 3, 4]
>>> T1
[1, 2, 3, 4]
```

• On peut de la même façon observer un effet de bord avec les dictionnaires qui sont aussi des objets mutables :

```
>>> dico1 = {"prénom": "james", "nom": "bond"}
>>> dico2 = dico1
>>> dico1["prénom"] = "hubert"
>>> dico1
{'prénom': 'hubert', 'nom': 'bond'}
```

```
>>> dico2
{'prénom': 'hubert', 'nom': 'bond'}
```

• Comment récupérer la valeur de type mutable d'une variable dans une autre variable sans risque de générer ces effets de bord ? Il faut procéder à une copie de l'objet avec la syntaxe suivante :

* pour une copie de tableau :

```
tab2 = list(tab1)
```

ou bien :

```
tab2 = tab1.copy()
```

* pour une copie de dictionnaire :

```
dico2 = dict(dico1)
```

• Il faut toutefois remarquer que cette méthode permet une copie superficielle (en anglais *shallow copy*) qui ne fonctionnera parfaitement que si les éléments du tableau ou les valeurs du dictionnaires ne sont pas eux-mêmes des objets de type mutable. Par exemple, cette méthode n'évitera pas les effets de bord sur des *tableaux de tableaux*.

### II.2 Le mot clé `is`

• Pour savoir si deux variables `var1` et `var2` « pointent » sur le même objet, on peut utiliser le mot-clé **`is`** de la façon suivante :

```
var1 is var2
```

* Si ce test renvoie `True` alors cela signifie que les variables ont pour valeur le même objet python, situé à un emplacement donné de la mémoire de l'ordinateur. Dans ce cas le test `var1 == var2` renvoie `True` aussi.

* Si le test renvoie False alors les deux variables ont pour valeur deux objets python différents situés à deux emplacements différents de la mémoire. Toutefois, il reste possible que ces deux objets aient la même valeur et que le test `var1 == var2` renvoie aussi la valeur `True`.

• Si `tab1` est un tableau (de type **`list`**) et que l'on exécute l'instruction `tab2 = tab1` alors les variables `tab1` et `tab2` pointent sur le même objet python. Le test `tab1 is tab2` renvoie donc `True`. Toute modification du tableau, que ce soit par l'intermédiaire de la variable `tab1` ou de la variable `tab2` modifiera donc la valeur de l'autre variable.

• Si, en revanche, on effectue une opération de copie telle que `tab2 = tab1.copy()`, alors les variables `tab1` et `tab2` pointent sur deux tableaux différents qui ont la même valeur. En conséquence, le test `tab1 is tab2` renvoie `False` mais le test `tab1 == tab2` renvoie `True`. Une modification du tableau pointé par une des deux variables n'aura donc aucun effet sur l'autre variable.

### II.3 Appel d'une fonction prenant pour argument un objet de type mutable

• Les effets de bord peuvent aussi se produire lorsque, dans une fonction, on modifie un tableau qui était un argument de la fonction.

• Par exemple, exécutons le programme suivant :

```
def ajouter_elem(tab, x):
    tab.append(x)

T = [1, 2, 3]
ajouter_elem(T, 15)
print(T)
```