

4 Introduction aux fonctions

Une *fonction* est une suite d'instructions préparée en avance pour réaliser une tâche donnée, qui pourra être appelée autant de fois que nécessaire dans la suite du programme. À chaque fois qu'une fonction est appelée, elle peut (suivant comment elle a été définie) :

- recevoir en entrée zéro, une ou plusieurs valeurs, appelées *arguments* ou *paramètres*,
- renvoyer une éventuelle *valeur de retour* récupérable et utilisable dans la suite du programme.

4.1 Définition et utilisation d'une fonction basique

Il est important de garder à l'esprit que l'utilisation d'une fonction se fait en deux temps.

1. D'abord, on *déclare* la fonction en donnant son nom et les instructions qu'elle est supposée effectuer. À ce moment-là, ces instructions ne sont **pas** exécutées, elles sont simplement gardées en mémoire afin de pouvoir être utilisées plus tard, lorsqu'on fera appel à la fonction.
2. Ensuite seulement, on peut *appeler* la fonction autant de fois qu'on le souhaite. Lors de chacun de ces appels, les instructions gardées en mémoire seront exécutées et un résultat final pourra éventuellement être renvoyé.

Déclaration d'une fonction

Un exemple valant mieux qu'un long discours, voici un programme définissant une fonction appelée `count_differences` qui compte le nombre de différences trouvées lorsqu'on compare caractère par caractère deux chaînes `msg1` et `msg2` de même taille :

```
1 def count_differences(msg1, msg2):
2     diff = 0
3     for i in range(len(msg1)):
4         if msg1[i] != msg2[i]:
5             diff = diff + 1
6
7     return diff
```

- Le mot-clé **def** indique à Python que l'on commence un bloc contenant les instructions qu'exécutera la fonction lorsqu'on l'appellera. Ce mot-clé est suivi du nom de la fonction, lui-même suivi d'une paire de parenthèses entre lesquelles on liste les éventuels *arguments* de la fonction (ici `msg1` et `msg2`). Notez que le nom de la fonction et le nom des arguments sont soumis aux mêmes contraintes que n'importe quel identifiant (cf. chapitre 1).

Remarque. Pour définir une fonction qui ne prend pas d'argument, on laisserait une paire de parenthèses vide après l'identifiant de la fonction. Par exemple :

```
def say_hello():
    print("Hello world!")
```

- Comme pour les boucles et structures conditionnelles, c'est ici l'*indentation* qui indique que les lignes 2 à 7 sont des instructions à l'intérieur du bloc « def ». On dit que ces instructions forment le *corps* de la fonction.
- L'instruction ligne 7 indique à la fonction que, lorsqu'elle sera appelée et que cette ligne sera atteinte, la fonction devra immédiatement renvoyer la valeur de l'expression placée après le mot-clé **return**. Attention : cela mettra aussitôt fin à l'exécution de la fonction (c'est-à-dire que si le corps de la fonction contenait d'autres instructions après la ligne 7, celles-ci seraient ignorées).

Remarque. Au sein d'une fonction, le mot-clé **return** est facultatif. Une fonction qui ne contient pas ce mot-clé ne renverra pas de résultat lorsqu'on l'appellera (voir paragraphe *Utilisation d'une fonction* ci-après).


Attention. Le mot-clé **return** n'a de sens que s'il est placé dans le corps d'une fonction.

- Enfin, on insiste encore une fois sur le fait que, lors de cette déclaration, le corps de la fonction n'est **pas** exécuté. On crée seulement un objet de type `function` qui garde en mémoire le corps de la fonction de sorte à pouvoir l'exécuter plus tard.

À tester. Exécutez le programme suivant.

```
def test(x):  
    return x / 0
```

1. Un message d'erreur s'affiche-t-il?
2. Qu'observe-t-on dans l'explorateur de variables?
3. Et si vous saisissez maintenant l'instruction `a = test(2)` ?

*Remarque : pour la seconde question, vérifiez au préalable que lorsque vous cliquez sur le menu  de l'explorateur de variables, l'option « Exclure les appelables et les modules » n'est **pas** cochée.*

Appel d'une fonction et valeur de retour

Un objet de type fonction est *appelable*. Concrètement cela signifie que, comme vous le faites déjà pour les fonctions rencontrées jusque là (`int`, `print`, `len`, `max` ...), vous pouvez lancer l'exécution de la fonction `count_differences` en saisissant son identifiant suivi d'une paire de parenthèses contenant les arguments que vous voulez faire passer à la fonction. Par exemple :

```
count_differences("HOTEL", "MOTOS")
```

Remarque. Si une fonction n'admet pas d'argument, on l'appelle en saisissant simplement son identifiant suivi d'une paire de parenthèses vide. Par exemple, la fonction `say_hello` définie plus haut s'appelle en saisissant l'instruction `say_hello()`.

Si, dans le corps de la fonction, le mot-clé **return** a été utilisé pour renvoyer une valeur, alors l'appel de la fonction joue le rôle d'une expression qu'on pourra utiliser dans une affectation de variable, insérer dans une expression plus complexe, passer en argument d'une autre fonction, ...

À tester. Observer dans la console et dans l'explorateur de variables l'effet du programme suivant.

```
def count_differences(msg1, msg2):
    diff = 0
    for i in range(len(msg1)):
        if msg1[i] != msg2[i]:
            diff = diff + 1

    return diff

a = count_differences("HOTEL", "MOTOS")
b = 5 * count_differences("CHAT", "CRAN") + 1
print(count_differences("AB" + "C", 3 * "A"))
```

Sinon, si lors de l'appel de la fonction l'instruction **return** n'est jamais rencontrée, alors la fonction renvoie un objet de type `NoneType` (c'est-à-dire, littéralement, un objet de type « Rien »). Par abus de langage, on dira qu'une telle fonction ne renvoie rien, ou n'a pas de valeur de retour, même si techniquement un objet de type `NoneType` est renvoyé.

À tester. Observer dans l'explorateur de variables le résultat du programme suivant.

```
def partial_return(x):
    if x > 0:
        return 3

a = partial_return(1)
b = partial_return(-2)
```

Remarque. Dans le corps d'une fonction, on peut aussi utiliser l'instruction **return** seule (c'est-à-dire sans la faire suivre d'une valeur) pour interrompre immédiatement l'exécution de la fonction sans renvoyer de valeur.

Avant de conclure cette section, signalons qu'en toute rigueur une fonction ne peut renvoyer qu'un seul objet. Toutefois ce n'est pas une contrainte en pratique car rien n'empêche de renvoyer un objet correspondant à une collection de plusieurs valeurs, comme par exemple une liste ou un tuple.

À tester. Exécutez le programme suivant et observez l'explorateur de variables.

```
1 def sum_and_diff(a, b):
2     return a+b, a-b
3
4 x, y = (3, 5)
5
6 T = sum_and_diff(10, 2)
7 a, b = T
8
9 u, v = sum_and_diff(6, 4)
```

1. Quel est le type de `T`?
2. Comment semble avoir été interprétée l'instruction ligne 4? Et l'instruction ligne 7?
3. Expliquer alors comment est interprétée l'instruction ligne 9.

Exercice 4.1. Écrire une fonction `sum_of_squares` qui prend en argument une liste `L` contenant des entiers, et qui renvoie la somme des carrés des valeurs contenues dans `L`.

Par exemple, `sum_of_squares([3, 1, 5, 2])` devra renvoyer 39 car $3^2 + 1^2 + 5^2 + 2^2 = 39$.

Exercice 4.2. Écrire une fonction `powers` qui prend en argument deux entiers `p` et `n` (avec $n \geq 0$), et qui renvoie une liste contenant les $n + 1$ premières puissances de `p` à partir de 1 (c'est-à-dire $p^0, p^1, p^2, \dots, p^n$).

Par exemple, `powers(2, 5)` devra renvoyer la liste `[1, 2, 4, 8, 16, 32]`.

Exercice 4.3. Écrire une fonction `triangle` qui prend en argument un entier `n` (supérieur ou égal à 1), et qui ne renvoie rien mais **affiche** `n` lignes formées respectivement de 1, 2, ..., `n` fois le caractère `"*"`.

Par exemple, `triangle(4)` devra afficher :

```
*
**
***
****
```

4.2 Portée des variables et muabilité

4.2.1 Variables locales et globales

Essayez d'exécuter le code suivant (dans un nouvel espace de travail remis à zéro).

```
1 def test(a):
2     x = 2 * a
3     return x
4
5 x = 3
6 y = test(x)
```

Si vous observez ensuite l'explorateur de variables, vous pourrez constater que la valeur de `y` est bien 6 comme attendu, mais que :

- l'explorateur de variables ne contient pas de variable nommée `a`,
- la valeur de `x` à la fin du programme est 3.

Ces deux derniers points s'expliquent par le fait que lorsqu'on appelle une fonction, celle-ci est exécutée dans un espace de noms *local* indépendant de l'espace de noms *global* dans lequel la fonction `test` et les variables `x` (ligne 5) et `y` ont été définies. Plus précisément, voici le comportement détaillé du programme ci-dessus (à lire en observant en parallèle la figure 4.1).

1. Les lignes 1 à 3 sont interprétées : un objet fonction est créé en mémoire, et est lié à l'identifiant `test` ajouté à l'espace de noms global. En revanche, il n'y a pas d'identifiants `a` et `x` créés en mémoire pour le moment car la fonction est seulement déclarée, pas exécutée.
2. La ligne 5 est interprétée : un objet de type entier représentant la valeur 3 est créé en mémoire, et est liée à l'identifiant `x` ajouté à l'espace de noms global.

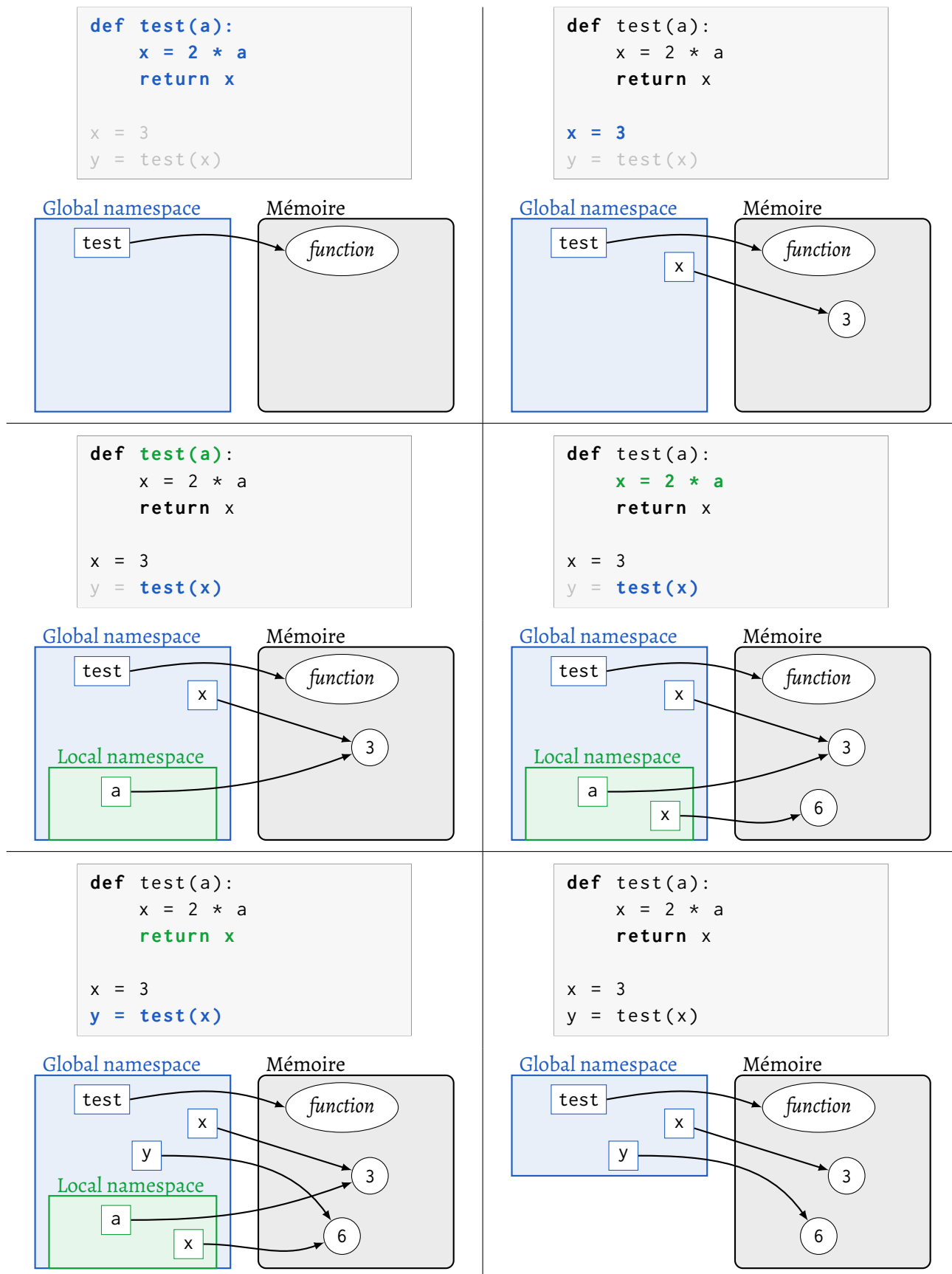


FIGURE 4.1 – Illustration de la mémoire lors de la déclaration puis l'appel d'une fonction.

3. Pour l'interprétation de la ligne 6, il faut d'abord évaluer l'expression à droite du signe `=`. La fonction `test` est alors appelée avec `x` pour argument. À ce moment-là, un espace de noms local est créé spécifiquement pour cet appel de fonction. Dans cet espace local, un identifiant `a` est créé, et on le fait pointer en mémoire vers la même valeur que l'identifiant `x` de l'espace de noms global (car c'est lui qu'on a choisi de passer en argument pour cet appel de la fonction `test`).
4. On exécute maintenant les instructions données lors de la déclaration de la fonction `test`, en commençant par `x = 2 * a`. L'expression à droite du signe `=` est évaluée et un objet représentant l'entier 6 est créé en mémoire. Cet objet est ensuite relié à l'identifiant `x` dans l'espace de noms **local**.
5. On exécute ensuite l'instruction `return x`. Ceci met fin à l'appel de fonction, et le résultat de l'expression `test(x)` est alors la valeur vers laquelle pointe l'identifiant `x` de l'espace de noms local (c'est-à-dire l'objet 6). L'affectation `y = test(x)` crée alors un identifiant `y` dans l'espace de noms global et le fait pointer vers cet objet 6 en mémoire.
6. Une fois la ligne 6 totalement interprétée, l'espace de noms local qui avait été temporairement créé peut être complètement oublié.

Par défaut, tous les identifiants utilisés dans le corps d'une fonction feront référence à l'espace de noms local. Toutefois, il existe deux exceptions à cette règle.

- Il est possible de préciser explicitement que certains identifiants doivent être récupérés depuis l'espace de noms global en utilisant le mot-clé **global** suivi des identifiants en questions (séparés par des virgules si on veut donner plusieurs identifiants) au début du corps de la fonction.

À tester. Comparer le résultat du code ci-dessous à celui donné en début de section, et essayer d'expliquer ce qui a changé par rapport au fonctionnement décrit dans la figure 4.1.

```
def test(a):
    global x
    x = 2 * a
    return x

x = 3
y = test(x)
```

- Si jamais, dans le corps de la fonction, un identifiant est utilisé dans une expression sans jamais être initialisé au préalable, alors l'interpréteur ira le chercher dans l'espace de noms global. Cela revient à dire que l'identifiant sera implicitement déclaré comme global par l'interpréteur sans que vous n'ayez besoin de le faire (c'est une pratique à éviter : il vaut mieux dans ce cas déclarer manuellement l'identifiant comme global afin qu'il n'y ait aucune ambiguïté lorsqu'on lit votre code).

À tester. Vérifiez que le code suivant ne donne pas d'erreur et agit comme si on avait donné l'instruction `global x` dans le corps de la fonction `test`.

```
def test():
    return 2 * x

x = 3
print(test())
```

4.2.2 Attention aux variables muables

En vertu du fonctionnement décrit ci-dessus, quand vous passez une variable en paramètre lors de l'appel d'une fonction :

- si cette variable contient une valeur de type **immuable** (int, str, tuple, ...), alors vous avez la garantie que cette variable **ne pourra pas être modifiée** par votre fonction (sauf si vous avez utilisé explicitement le mot-clé `global`),
- en revanche si cette variable contient une valeur de type **muable** (par exemple `list`), alors la fonction **peut modifier** la valeur de votre variable (même si vous n'avez pas utilisé `global`).

Les deux situations sont illustrées par les programmes ci-dessous, dans lesquels une même fonction ne se comportera pas de la même manière si on lui donne une variable de type entier (immuable) ou de type liste (muable).

Programme 1.

```
1 def exemple(x):
2     x *= 2
3     return x
4
5 A = 3
6 B = exemple(A)
```

Programme 2.

```
1 def exemple(x):
2     x *= 2
3     return x
4
5 A = [8, 5]
6 B = exemple(A)
```

En observant l'explorateur de variables, vous pourrez constater que dans le programme 1, l'appel à la fonction `exemple` n'a pas modifié la valeur de `A` alors que dans le programme 2, l'appel à la fonction `exemple` a modifié la valeur de `A`. En effet, dans les deux cas l'interprétation du programme se déroule d'abord de la même manière :

- un objet de type fonction est créé en mémoire, relié à l'identifiant `exemple` dans l'espace de noms global,
- un objet soit de type entier (programme 1), soit de type liste (programme 2) est créé en mémoire, relié à l'identifiant `A` dans l'espace de noms global,
- la fonction `exemple` est appelée avec pour argument `x` : un espace de noms local est alors créé, et dans cet espace local un identifiant `x` est ajouté et pointe vers la même valeur que l'identifiant `A` de l'espace de noms global.

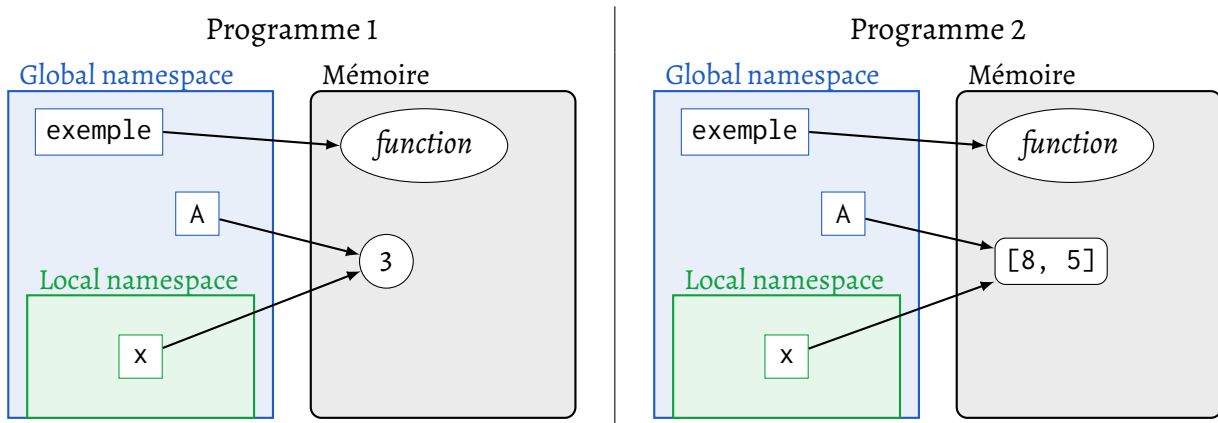


FIGURE 4.2 – Interprétation des lignes 1 à 5 identique pour les programmes 1 et 2.

En revanche, c'est au moment de l'instruction `x *= 2` que les comportements des deux programmes divergent.

1. Dans le premier programme, l'opérateur `*` est appliqué sur un objet de type entier. Or le type entier est immuable, c'est-à-dire que l'objet lui-même ne peut pas être modifié en mémoire. C'est donc un nouvel objet qui est créé en mémoire pour représenter le résultat de l'opération (ici 6), et l'identifiant `x` pointe maintenant vers ce nouvel objet. L'identifiant `A` de l'espace de noms global, qui n'a rien à voir avec tout ça, n'est pas impacté et continue à pointer vers l'objet « 3 ».
2. Dans le second programme, l'opérateur `*` est appliqué sur un objet de type liste. Le type liste étant muable, cet opérateur modifie directement l'objet en mémoire de sorte qu'il ne représente plus la liste `[8, 5]` mais la liste `[8, 5, 8, 5]`. Cette modification impacte automatiquement tous les identifiants qui pointaient vers cet objet : l'identifiant local `x` mais aussi l'identifiant global `A`.

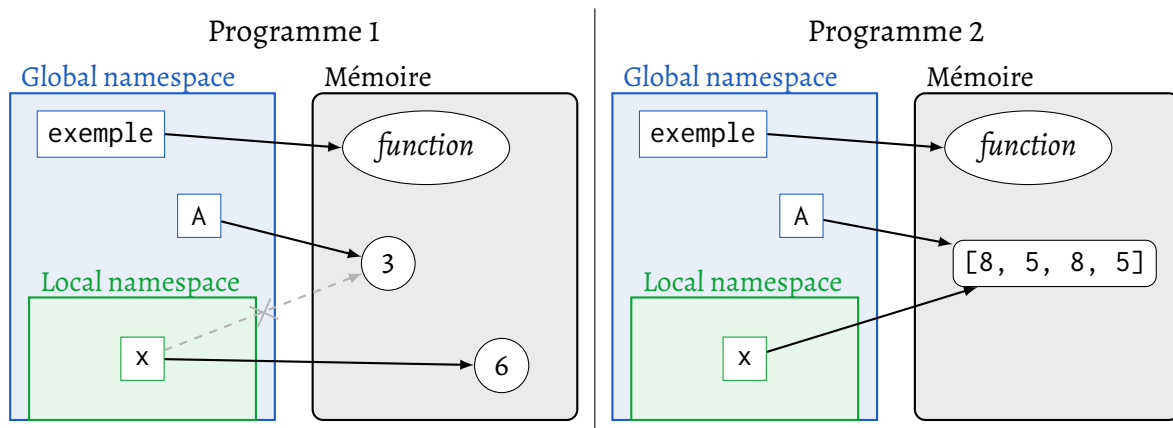


FIGURE 4.3 – Effets différents de l'instruction `x *= 2` les programmes 1 et 2.

Exercice 4.4.

1. Écrire une fonction `translate_in_place(L, x)` qui prend comme argument une liste d'entiers `L` et un entier `x`, et qui **modifie** la liste `L` de sorte à ajouter `x` à chacune de ses valeurs.
2. Écrire une fonction `translate_copy(L, x)` qui prend comme argument une liste d'entiers `L` et un entier `x`, et qui **renvoie une nouvelle liste** formée des valeurs de `L` auxquelles on a ajouté `x`, sans modifier la liste `L` de départ.

Par exemple si `L` est la liste `[5, 0, 8, 3, 6]`, alors :

- `translate_in_place(L, 2)` devra transformer `L` en `[7, 2, 10, 5, 8]`,
- `translate_copy(L, 2)` devra renvoyer la liste `[7, 2, 10, 5, 8]` sans modifier `L`.

4.3 Arguments nommés, arguments optionnels

4.3.1 Arguments positionnels et arguments nommés

Considérons la déclaration de fonction suivante :

```
def f(a, b, c):
    print("a = " + str(a) + ", b = " + str(b) + ", c = " + str(c))
```


Pour appeler cette fonction, vous avez jusqu'ici appris à utiliser une notation *positionnelle*, c'est-à-dire dans laquelle l'ordre des arguments lors de l'appel indique de manière naturelle à quels arguments ils correspondent dans la définition de `f`. Par exemple, en écrivant `f(5, 1, 3)`, on comprend implicitement que vous voulez faire passer à l'argument `a` la valeur 5, à l'argument `b` la valeur 1 et à l'argument `c` la valeur 3.

Il est en réalité possible de préciser explicitement à quel argument vous voulez faire passer quelle valeur en utilisant une syntaxe telle que `f(a=5, b=1, c=3)`. Dans ce cas, l'ordre n'a plus d'importance et vous pouvez vérifier que les appels `f(b=1, a=5, c=3)` et `f(c=3, b=1, a=5)` donneraient exactement le même résultat. On parle alors d'arguments *nommés* (ou *keyword arguments* en anglais).

Il est même possible de mélanger les deux syntaxes, à la condition de donner d'abord les premiers arguments de manière positionnelle, puis de donner par leurs noms les arguments restants dans l'ordre de votre choix. Il n'est en aucun cas possible de donner des arguments positionnels *après* avoir utilisé des arguments nommés.

À tester. Après avoir déclaré la fonction `f` donnée ci-dessus, testez les commandes suivantes :

- | | | |
|----------------------------------|--------------------------------|------------------------------|
| 1. <code>f(5, 1, 3)</code> | 3. <code>f(5, c=3, b=1)</code> | 5. <code>f(a=5, 1, 3)</code> |
| 2. <code>f(c=3, a=5, b=1)</code> | 4. <code>f(5, 1, c=3)</code> | 6. <code>f(a=5, c=3)</code> |

4.3.2 Arguments optionnels et valeurs par défaut

Lors de la déclaration de la fonction, il est possible de faire en sorte que les derniers arguments soient facultatifs (*optional* en anglais) en leur attribuant dès la définition une valeur par défaut. Par exemple, dans le code ci-dessous, les arguments `b` et `c` de la fonction `f` sont maintenant optionnels avec pour valeurs par défaut respectives 4 et 8.

```
def f(a, b = 4, c = 8):
    print("a = " + str(a) + ", b = " + str(b) + ", c = " + str(c))
```

Cela signifie que lorsqu'on fera appel à la fonction `f` on pourra choisir de donner ou non des valeurs à `b` et `c` (de manière nommée ou de manière positionnelle). Si l'on donne une valeur à un argument, c'est celle-ci qui sera retenue, sinon c'est la valeur par défaut donnée dans la déclaration qui sera utilisée.

À tester. Avec la nouvelle définition de `f` ci-dessus, testez les commandes suivantes :

- | | | |
|----------------------------|--------------------------------|------------------------------|
| 1. <code>f(5, 1, 3)</code> | 3. <code>f(5)</code> | 5. <code>f(5, c=3)</code> |
| 2. <code>f(5, 1)</code> | 4. <code>f(5, c=3, b=1)</code> | 6. <code>f(5, 1, c=3)</code> |

Remarque. Une fonction peut avoir entre zéro et tous ses arguments de facultatifs. En revanche, il faut que dans la définition de la fonction les (éventuels) arguments obligatoires se situent toujours **avant** les (éventuels) arguments facultatifs. La définition ci-dessous ne serait par exemple pas valide.

```
def f(a, b = 4, c):
    print("a = " + str(a) + ", b = " + str(b) + ", c = " + str(c))
```

Exercice 4.5. Écrire une fonction `count` permettant de compter le nombre d'occurrences d'un caractère dans une chaîne, ou dans une portion d'une chaîne donnée. La fonction admettra les arguments suivants :

- `msg` (obligatoire) : la chaîne de caractère dans laquelle effectuer la recherche,
- `c` (obligatoire) : le caractère à rechercher,
- `start` (optionnel, par défaut 0) : l'indice à partir duquel commencer la recherche (inclus).

La fonction renverra un entier contenant le nombre d'occurrences trouvées.

Par exemple,

- `count("Le Soleil est une étoile", "e")` devra renvoyer 5,
- `count("Le Soleil est une étoile", "e", 10)` devra renvoyer 3.

Exercice 4.6. Complétez la fonction `triangle` de l'exercice 4.3 de sorte à ce qu'elle prenne en compte les arguments suivants :

- `n` (obligatoire) : le nombre de lignes à afficher,
- `symbol` (facultatif, par défaut '*') : le caractère à utiliser pour afficher le triangle,
- `align` (facultatif, par défaut 'left') : une chaîne de caractère précisant l'alignement horizontal du triangle ('left' ou 'right').

Par exemple,

- | | |
|---|--|
| <ul style="list-style-type: none"> — <code>triangle(4, symbol='+')</code> devra afficher : <li style="margin-left: 20px;">+ <li style="margin-left: 20px;">++ <li style="margin-left: 20px;">+++ <li style="margin-left: 20px;">++++ | <ul style="list-style-type: none"> — <code>triangle(5, align='right')</code> devra afficher : <li style="margin-left: 20px;">* <li style="margin-left: 20px;">** <li style="margin-left: 20px;">*** <li style="margin-left: 20px;">**** <li style="margin-left: 20px;">***** |
|---|--|

Exercices supplémentaires

Exercice 4.7. Écrire une fonction `convert_seconds` qui prend en argument un entier `t` représentant un nombre de secondes, et qui renvoie un tuple `(h, m, s)` représentant la même durée en heures/minutes/seconde.

Par exemple, `convert_seconds(11253)` doit renvoyer `(3, 7, 33)` car $11\,253\text{ s} = 3\text{h}07\text{m}33\text{s}$.

Exercice 4.8. Écrire une fonction `last_index(seq, val)` qui prend en arguments :

- `seq` (obligatoire) : une liste ou un tuple,
- `val` (obligatoire) : une valeur à rechercher dans la liste,

et qui renvoie l'indice (positif) de la **dernière** apparition de `val` dans la liste ou le tuple `seq`. Par convention, si la valeur `val` n'apparaît jamais dans `seq`, on renverra `-1`.

Par exemple, si `L = [8, 30, 12, 12, 26, 12, 45, 12, 30, 8]`, alors :

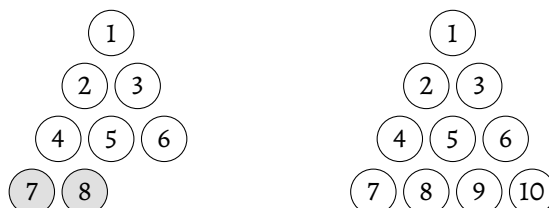
- `last_index(L, 12)` doit renvoyer 7 car le dernier 12 dans la liste est `L[7]`,
- `last_index(L, 26)` doit renvoyer 4 car l'unique (et donc dernier) 26 dans la liste est `L[4]`,
- `last_index(L, 23)` doit renvoyer `-1` car la liste `L` ne contient pas 23.

Exercice 4.9. Écrire une fonction `bowling` qui prend en argument un nombre de quilles N , et qui renvoie un tuple de la forme `(rows, left)` avec :

- `rows` le nombre de rangées complètes que l'on peut créer avec ces N quilles sachant qu'on dispose 1 quille sur la première rangée, 2 quilles sur la deuxième rangée, 3 quilles sur la troisième rangée, etc.,
- `left` le nombre de quilles éventuellement restantes.

Par exemple,

- `bowling(8)` doit renvoyer `(3, 2)` car on peut faire 3 rangées complètes et il restera 2 quilles,
- `bowling(10)` doit renvoyer `(4, 0)` car on peut faire 4 rangées complètes et il restera 0 quille.



Exercice 4.10. Écrire une fonction `alternate` qui prend les arguments suivants :

- `L1` (obligatoire) : une liste,
- `L2` (obligatoire) : une liste,
- `truncate` (facultatif, par défaut `False`) : un booléen,

et qui renvoie une nouvelle liste `L` créée en insérant alternativement des valeurs de `L1` et de `L2`. Le booléen `truncate` indique le comportement à adopter dans le cas où les deux listes ne sont pas de même longueur :

- s'il vaut `True`, le programme doit s'arrêter dès que la dernière valeur de la liste la plus courte a été insérée,
- s'il vaut `False`, une fois que toutes les valeurs de la liste la plus courte ont été insérées, on insère les valeurs restantes de la liste la plus longue.

Par exemple, si `A = [1, 2, 3, 4, 5, 6]` et `B = ['a', 'b', 'c']`, alors :

- `truncate(A, B)` doit renvoyer la liste `[1, 'a', 2, 'b', 3, 'c', 4, 5, 6]`,
- `truncate(A, B, True)` doit renvoyer la liste `[1, 'a', 2, 'b', 3, 'c']`.

À retenir

- Connaître le vocabulaire *fonction*, *argument/paramètre*, *valeur de retour*, *corps*.
- Savoir définir une fonction avec ou sans arguments facultatifs.
- Savoir appeler une fonction avec ou sans arguments facultatifs.
- Savoir faire la différence entre *afficher* une valeur et *renvoyer* une valeur.
- Comprendre la notion de variable globale, et savoir définir une variable comme globale.
- Comprendre qu'un paramètre de type muable peut être modifié par une fonction.
- Mots-clés à connaître : `def`, `global`, `return`.