Fonctions

I. Bibliothèques et modules

1) Introduction

Définition 1 : *Bibliothèques et modules*

Une bibliothèque ou un module est un fichier python qui contient des définitions de constantes et de fonctions.

Python possède quelques fonctions prédéfinies (on parle de fonctions natives) dont certaines que nous avons déjà utilisées :

- ⋆ Les fonctions d'affichage et d'interaction avec l'utilisateur : `print`, `input`.
- ⋆ Les fonctions de conversion de type : `int`, `float`, `str`, `bool`, `list`, `dict`, `tuple`.
- ⋆ Certaines fonctions mathématiques : `abs` (valeur absolue), `pow` (puissance), `round` (arrondi), `sum` (somme).

Il est possible d'avoir accès à de nombreuses autres fonctions grâce à l'importation de bibliothèques ou de modules :

- ⋆ La bibliothèque `math` définit la plupart des fonctions mathématiques utiles : `sqrt`, `sin`, `cos`, `tan`, `exp`, `log`, etc.
- ⋆ La bibliothèque `cmath` permet d'effectuer des calculs avec les nombres complexes.
- ⋆ La bibliothèque `random` permet de générer des nombres pseudo-aléatoires et de les utiliser pour programmer des simulations.
- ⋆ La bibliothèque `numpy` permet de travailler avec un nouveau type de données : les tableaux.
- ⋆ La bibliothèque `matplotlib` permet de tracer des courbes.

Installation d'une bibliothèque

Certaines bibliothèques sont préinstallées : c'est le cas de `math`, `cmath` et `random`. Les bibliothèques `numpy` et `matplotlib` doivent être installées avant de pouvoir être utilisées. Pour les installer, il faut avoir une connexion internet et taper l'instruction suivante dans le shell puis suivre les instructions :

```
>>> install numpy matplotlib
```

Bien entendu, l'installation n'est à faire qu'une seule fois.

2) Importer une bibliothèque

Pour utiliser une bibliothèque dans un fichier Python, nous avons trois méthodes possibles. Étant donné une bibliothèque appelée `module`, on peut :

Méthode 1 : Importer la bibliothèque avec l'instruction

```
import module
```

puis utiliser les fonctions et constantes de cette bibliothèque en les préfixant à l'aide du nom du module : `module.fonction(...)` et `module.constante`

Exemple :

```
>>> import math
>>> math.sqrt(4)
2.0
>>> math.cos(math.pi/4)
0.7071067811865476
```

Il est possible de raccourcir le nom de la bibliothèque pour faciliter l'utilisation des fonctions : on dit que l'on donne un alias à la bibliothèque. Pour cela, il utiliser l'instruction

```
import module as alias
```

Exemple :

```
>>> import math as m
>>> m.cos(m.pi/4)
0.7071067811865476
```

Méthode 2 : Importer uniquement les fonctions et les constantes que l'on veut utiliser avec l'instruction

```
from module import fonction1, fonction2, constante1, ...
```

puis utiliser les fonctions et constantes choisies sans avoir besoin de les préfixer.

Exemple :

```
>>> from math import sqrt, cos, pi
>>> sqrt(4)
2.0
>>> cos(pi/4)
0.7071067811865476
>>> sin(pi/4)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'sin' is not defined. Did you mean: 'bin'?
```

Méthode 3 : Importer toutes les fonctions et constantes à la fois à l'aide de l'instruction

```
from math import *
```

puis utiliser les fonctions et constantes de cette bibliothèque sans avoir besoin de les préfixer.

Exemple :

```
>>> from module import *
>>> cos(pi/4)
0.7071067811865476
>>> atan(cos(pi+ceil(2.4)))
0.7803692906480693
```

Remarque : La troisième méthode est à utiliser avec parcimonie : elle présente un risque car on importe toute la bibliothèque donc il peut y avoir des conflits de noms avec notre propre programme. Par exemple ce programme :

```
e = 12
from math import *
print(e)
```

affiche après exécution la valeur

```
2.718281828459045
```

Une bonne pratique consiste à mettre toutes les importations en début de fichier pour éviter ces problèmes.

3) Accès à la documentation d'une bibliothèque ou d'une fonction

Pour connaître la liste des fonctions d'une bibliothèque et pour savoir comment utiliser les fonction et à quoi elle servent, on peut utiliser la fonction `help` après avoir importé la bibliothèque.

- ⋆ `help(module)` affiche la liste de toutes les fonctions et constantes de la bibliothèque `module` ainsi que la documentation de chaque fonction.
- ⋆ `help(module.fonction)` affiche la documentation d'une fonction en particulier.

Exemple :

```
>>> import math
>>> help(math)
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

        The result is between 0 and pi.
    ...
DATA
    e = 2.718281828459045
    inf = inf
    nan = nan
    pi = 3.141592653589793
    tau = 6.283185307179586

FILE
    (built-in)
>>> help(math.cos)
Help on built-in function cos in module math:

cos(...)
    cos(x)

    Return the cosine of x (measured in radians).
```

L'aide permet surtout de savoir

- ⋆ combien de paramètres a besoin une fonction
- ⋆ à quoi correspond chaque paramètre et quel type de donnée est attendu
- ⋆ ce que retourne la fonction comme résultat

4) La bibliothèque `math`

Voici une liste non exhaustive des fonctions de la bibliothèque `math` :

Fonction	Équivalent mathématique	Fonction	Équivalent mathématique
`sqrt(x)`	$\sqrt{x}$	`exp(x)`	e^x
`log(x)`	$\ln x$	`log10(x)`	$\log x$
`cos(x)`	$\cos x$	`acos(x)`	$\arccos x$
`sin(x)`	$\sin x$	`asin(x)`	$\arcsin x$
`tan(x)`	$\tan x$	`atan(x)`	$\arctan x$
`floor(x)`	$\lfloor x \rfloor$	`factorial(n)`	$n!$
`comb(n, k)`	$\binom{n}{k}$		

Et voici une liste non exhaustive des constantes de la bibliothèque `math` :

Constante	Équivalent mathématique
`e`	e
`pi`	π
`inf`	$+\infty$

Les fonctions mathématiques suivantes ne font pas partie de la bibliothèque `math` :

Fonction	Équivalent mathématique
`abs(x)`	$\lvert x \rvert$
`round(x, n=0)`	Arrondi le nombre `x` avec `n` décimales

Remarque : Lorsque vous voyez une description de fonctions comme celle de la fonction `round`, les paramètres ayant une valeur associée comme `n=0` sont des paramètres optionnels : la valeur associé est la valeur par défaut.

```
>>> round(pi, 2)
3.14
>>> round(pi)
3
```

II. Définir ses propres fonctions

1) Notion de fonction informatique

Définition 2 : *Fonction*

Une fonction est un bloc de code identifié par un nom, qui peut prendre un certain nombre (éventuellement aucun) de paramètres (appelés aussi arguments) en entrée et qui peut éventuellement avoir un résultat en sortie (on dira que la fonction renvoie ou retourne ce résultat).

Schématiquement, on peut voir une fonction informatique comme une « boîte noire » qui, si on lui donne les paramètres qu'elle attend, va effectuer un travail et éventuellement nous renvoyer un résultat.

On parle de boîte noire car, comme nous allons le découvrir dans ce chapitre, en dehors des valeurs données en entrée et de l'éventuelle sortie de la fonction, l'utilisateur ne connaît rien du calcul réalisé et des variables utilisées à l'intérieur de la fonction.

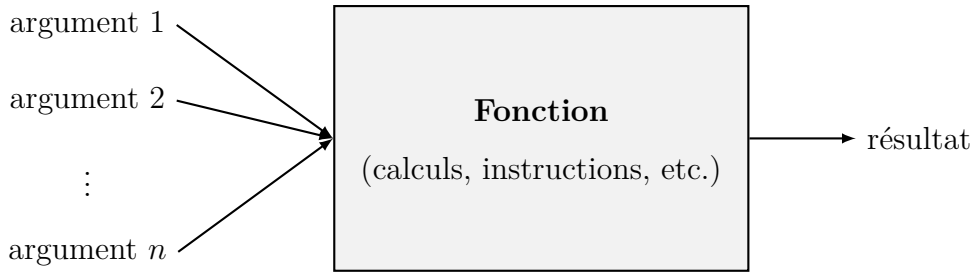

FIGURE 1 – Représentation schématique d'une fonction informatique

Remarque : Définir et utiliser une fonction permet :

- ⋆ d'éviter les répétitions de code (on parle aussi de redondance du code)
- ⋆ d'améliorer la lisibilité dun programme
- ⋆ de structurer un programme en sous-tâches plus simples
- ⋆ de faciliter la maintenance ou la modification du code

2) Syntaxes de définition et d'appel d'une fonction

Syntaxes d'une fonction

Pour **définir** une fonction on utilise la syntaxe suivante :

```
def nom_fonction(arg_1, arg_2,.., arg_n):
    bloc_d'instructions
```

où

- ⋆ `nom_fonction` est le nom de la fonction que l'on peut choisir en respectant les mêmes règles que pour les noms de variables
- ⋆ `arg_1`, `arg_2`, ..., `arg_n` est un ensemble de **noms de variables** qui représentent les paramètres nécessaires à l'exécution de la fonction (il peut y en avoir autant que l'on veut, y compris aucun)
- ⋆ `bloc_d'instructions` est un ensemble d'instructions, **indentées** par rapport au mot-clé `def`, qui seront exécutées à chaque appel de la fonction.

Une fois la fonction définie et cette définition exécutée, on peut **utiliser** la fonction (on dit qu'on appelle la fonction) avec l'une des syntaxes suivantes :

```
nom_fonction(val_1, val_2,.., val_n)
resultat = nom_fonction(val_1, val_2,.., val_n)    # si la fonction renvoie
                                                   # un résultat
```

où `val_1`, `val_2`, ..., `val_n` sont des valeurs qui seront affectées aux variables `arg_1`, `arg_2`, ..., `arg_n` lors de l'exécution de la fonction.

Exemple :

⋆ Soit $f : \mathbb{R} \to \mathbb{R}$ définie par $\forall x \in \mathbb{R}, \quad f(x) = (2x+1)^2 - x^3$. Définissons cette fonction en Python.

```
def f(x):
    y = (2 * x + 1)**2 - x**3
    return y    # Permet de déclarer que y est le résultat (la sortie)
```

Le nom de la fonction est `f` et elle possède un argument appelé `x`. Dans la fonction, on définit une variable `y` qui prend la valeur `(2 * x + 1)**2 - x**3`, puis on renvoie la valeur de `y`.

> Remarque : La variable `x` n'est définie ni avant, ni pendant, ni après la définition de la fonction `f` : c'est un argument de `f`. C'est lorsqu'on utilise la fonction que l'on va attribuer une valeur aux arguments.

Pour appeler la fonction, il suffit de taper son nom et de mettre entre parenthèses des valeurs pour les paramètres :

```
>>> f(3)
22
>>> a = 1.5
>>> x = f(a)      # On peut enregistrer la sortie dans une variable
>>> x
12.625
```

Remarque : On remarque que dans la définition de la fonction, le paramètre en entrée s'appelait `x` et le résultat en sortie s'appelait `y` mais lors de l'utilisation **on n'a absolument pas besoin de le savoir** !

On peut appeler la valeur retournée comme on veut et heureusement ! N'oubliez pas que certaines fonctions qu'on utilise n'ont pas été définies pas nous-même : s'il fallait savoir et se souvenir que la fonction `sqrt` a une sortie qui s'appelle `z` alors que la fonction `exp` a une sortie qui s'appelle `w` ce serait très vite ingérable...

⋆ Une fonction peut ne pas prendre de paramètre en entrée et peut également ne pas avoir de résultat en sortie :

```
def dire_bonjour():
    print("Bonjour !")  # Ce n'est pas une sortie car il n'y a pas
                        # de return
```

puis lors de l'utilisation :

```
>>> dire_bonjour()      # Même s'il n'y a pas de paramètre,
Bonjour !               # les parenthèses sont obligatoires
```

⋆ Une fonction peut prendre plusieurs paramètres en entrée (mais il ne peut y avoir qu'une seule sortie). Par exemple, écrivons une fonction prenant en argument deux nombres réels a et b, avec $a \neq 0$, et renvoyant la solution de l'équation $ax + b = 0$.

```
def solution(a, b):
    return -b / a
```

puis lors de l'utilisation :

```
>>> solution(2, 7)
-3.5
>>> solution(3, 1)
-0.3333333333333333
```

3) Instruction `return`

Instruction `return`

Dans la définition d'une fonction, une instruction spéciale permet de spécifier une valeur de sortie (un résultat) :

```
return resultat
```

Cette instruction **interrompt immédiatement** l'exécution de la fonction même s'il y a d'autres instructions après et même si on est dans une boucle. Le résultat associé à cette instruction est la seule valeur qui peut être récupérée lors de l'appel de la fonction :

```
resultat = nom_fonction(val_1, val_2,.., val_n)
```

S'il n'y a pas d'instruction de ce type dans la définition d'une fonction, alors celle-ci n'a pas de valeur de sortie. Il est donc dangereux d'écrire

```
resultat = nom_fonction(val_1, val_2,.., val_n)
```

car alors `resultat` sera une variable vide. C'est le cas de la fonction `print` : elle n'a aucune valeur de sortie.
En réalité, une fonction a toujours une valeur de sortie : lorsqu'il n'y a pas de `return` la valeur de sortie est la valeur `None` par défaut.

a) Différences entre `print` et `return`

Regardons les trois fonctions suivantes :

```
def discriminant1(a, b, c):
    b**2 - 4 * a * c

def discriminant2(a, b, c):
    print(b**2 - 4 * a * c)

def discriminant3(a, b, c):
    return b**2 - 4 * a * c
```

Quelle est la différence entre ces trois fonctions ?

- ⋆ La première fonction ne sert absolument à rien : le discriminant est calculé mais il n'est ni affiché à l'écran, ni renvoyé, ni enregistré dans la mémoire.
- ⋆ La deuxième fonction **affiche** le discriminant à l'écran : elle n'a pas de sortie.
- ⋆ La troisième fonction **renvoie** le discriminant : elle a une sortie.

Ainsi

```
>>> discriminant1(1, 2, 3)

>>> discriminant2(1, 2, 3)
-8
>>> discriminant3(1, 2, 3)
-8

>>> delta = discriminant2(1, 2, 3)
-8
>>> delta
>>> print(delta)    # Variable vide
None
>>> type(delta)
<class 'NoneType'>

>>> delta = discriminant3(1, 2, 3)
>>> delta
-8
>>> print(delta)
-8
>>> type(delta)
<class 'int'>
```

On voit que la différence entre `discriminant2` et `discriminant3` se fait lorsqu'on essaye d'enregistrer la sortie dans une variable. C'est possible pour `discriminant3` mais pas pour `discriminant2`. Une autre différence apparaît lorsqu'on essaye de faire des calculs avec le résultat des fonctions :

```
>>> discriminant2(1, 2, 3) + discriminant2(4, 5, 6)
-8
-71
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
>>> discriminant3(1, 2, 3) + discriminant3(4, 5, 6)
-79
```

À retenir

Lorsqu'on veut que notre fonction ait un résultat, **il faut toujours** mettre un `return`. La seule exception est lorsque l'énoncé nous demande explicitement de coder une fonction qui **affiche** quelque chose : dans ce cas on mettra un `print` (mais ces cas-là sont très rares).
En général, on utilise des `print` pour debogger un programme afin de contrôler les valeurs des variables et voir si tout se passe bien. Une fois les problèmes corrigés, on enlève les `print`.

b) Rôle d'interrupteur de fonction

Étant donné qu'il ne peut y avoir qu'une seule sortie à une fonction, si Python exécute une instruction `return`, l'exécution de la fonction se termine instantanément, peut importe où on se trouve dans le programme.

Exemple :

⋆ **Définition :**

```
def f(x):
    y = x**2 + x
    return y
    return y + 1
```

Utilisation :

```
>>> f(3)
12
```

⋆ **Définition :**

```
def f(x):
    y = x**2 + x
    return y
    y = 3 / 0   # Ligne ignorée
```

Utilisation :

```
>>> f(3)
12
```

⋆ **Définition :**

```
def f(x):
    if x > 0:
        return x + 1
    else:
        return x - 1
```

Utilisation :

```
>>> f(3)
4
>>> f(-3)
-4
```

⋆ **Définition :**

```
def f(x):
    if x > 0:
        return x + 1
    return x - 1
```

Utilisation :

```
>>> f(3)
4
>>> f(-3)
-4
```

Le deuxième `return` n'est pas exécuté si `x > 0` car le premier `return` aura été exécuté avant.

⋆ **Définition :**

```
def f(x):
    for i in range(x):
        return i
    return x
```

Utilisation :

```
>>> f(10)
0
```

La boucle n'est pas vraiment une boucle : à la première itération, l'instruction `return` interrompt tout !

⋆ **Définition :**

```
def f(x):
    S = 0
    for i in range(x):
        S += i
        if S > 15:
            return S
```

Utilisation :

```
>>> f(10)
21
```

La boucle est exécutée mais pas en totalité : ici la boucle `for` se comporte comme une boucle `while`.

4) Documenter ses fonctions

Nous avons vu que la fonction `help` permettait d'accéder à la documentation des fonctions natives ou importées de bibliothèques. Qu'en est-il de nos propres fonctions ?

Documenter ses fonctions

On peut écrire la documentation de nos fonctions au moment de la définition de celles-ci : en début de fonction (juste après la ligne du `def`), on peut créer une chaîne de caractères multiligne (entre triple guillemets `"""`) dans laquelle on peut expliquer

- ⋆ combien la fonction possède de paramètres
- ⋆ quel est le rôle de chaque paramètre (et quel type de donnée est attendu)
- ⋆ si la fonction a une valeur de sortie et quel est le rôle de cette valeur

Ce texte sera alors celui affiché lors de l'appel de la fonction `help` sur le nom de notre fonction.

Exemple :

```
def solutions(a, b, c):
    """
    Entrées :
        a : réel non nul
        b : réel
        c : réel
    Sortie : solutions de l'équation ax^2 + bx + c = 0 sous forme de liste
    """
    Instructions
```

puis si on utilise la fonction `help` :

```
>>> help(solutions)
Help on function solutions in module __main__:

solutions(a, b, c)
    Entrées :
        a : réel non nul
        b : réel
        c : réel
    Sortie : solutions de l'équation ax^2 + bx + c = 0 sous forme de liste
```

5) Complément : mot-clé `assert`

Souvent, lorsqu'on écrit une fonction, les paramètres de notre fonction doivent répondre à des conditions pour que la fonction ait le bon comportement. Par exemple, pour la fonction `discriminant(a, b, c)`, les paramètres `a`, `b` et `c` doivent être des nombres réels et `a` ne doit pas être nul (on ne calcule pas de discriminant si `a = 0`).

Si l'utilisateur de notre fonction fait n'importe quoi, il risque d'y avoir des erreurs ou des résultats incohérents :

⋆ **Définition :**

```
def discriminant(a, b, c):
    return b**2 - 4 * a * c
```

⋆ **Utilisation :**

```
>>> discriminant(0, 1, 2)
1
>>> discriminant("a", "b", "c")
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "d:\downloads\tp2.py", line 12, in discriminant
    return b**2 - 4 * a * c
           ~^^~
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

On peut bien sûr ajouter une documentation pour expliquer à l'utilisateur ce qu'il faut faire et ne pas faire mais ce qui serait encore mieux c'est d'ajouter nous-même les conditions d'utilisation de la fonction et les messages d'erreur à afficher en cas de non respect de ces conditions. C'est ce qu'on peut faire grâce au mot-clé `assert` :

⋆ **Définition :**

```
def discriminant(a, b, c):
    assert type(a) == int or type(a) == float, "a doit être un nombre"
    assert type(b) == int or type(b) == float, "b doit être un nombre"
    assert type(c) == int or type(c) == float, "c doit être un nombre"
    assert a != 0, "a doit être non nul"

    return b**2 - 4 * a * c
```

⋆ **Utilisation :**

```
>>> discriminant(0, 1, 2)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "d:\downloads\tp2.py", line 15, in discriminant
    assert a != 0, "a doit être non nul"
           ^^^^^^
AssertionError: a doit être non nul
>>> discriminant("a", "b", "c")
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "d:\downloads\tp2.py", line 12, in discriminant
    assert type(a) == int or type(a) == float, "a doit être un nombre"
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: a doit être un nombre
```

Enfin, le mot-clé `assert` peut aussi servir à se créer un « jeu de tests » qui permet de s'assurer que la fonction qu'on écrit a le bon comportement et renvoie le bon résultat sur différents exemples dont on connait la valeur de sortie attendue. Par exemple :

```
def discriminant(a, b, c):
    return b**2 - 4 * a * c

# Jeu de tests
assert discriminant(1, 2, 3) == -8
assert discriminant(1, 2, 1) == 0
assert discriminant(1, 3, 2) == 1
```

Si un message d'erreur apparaît lors de l'exécution, on saura que notre fonction ne fait pas exactement ce qu'elle devrait faire : il faut donc chercher l'erreur.

III. Portées des variables

Dans le chapitre 1, nous avons brièvement parlé de la notion de portée d'une variable. Il y a deux types de variables :

- ⋆ les variables globales : ce sont les variables définies en dehors d'une fonction. Elle sont accessibles partout dans le programme (après leur définition bien sûr) y compris dans les définitions de fonctions.
- ⋆ les variables locales : ce sont les variables définies dans le corps d'une fonction ou les paramètres d'une fonction. Elle sont accessibles uniquement dans cette fonction (d'où le terme de boîte noire : ce qui est à l'intérieur d'une fonction ne se voit pas de l'extérieur).

Il est possible d'avoir à la fois une variable globale et une variable locale qui portent le même nom. Si, à l'intérieur d'une fonction, une expression fait référence à une variable, Python commence par chercher si une variable locale porte ce nom. Si ce n'est pas le cas, il cherchera alors parmi les variables globales. S'il n'y en a toujours pas, il affichera un message d'erreur du type `NameError`.

Ainsi, on peut comprendre pourquoi Python renvoie certains messages d'erreur dans les exemples qui suivent :

Exemple :

⋆ **Éditeur :**

```
x = 12

def f(a):
    y = a + x
    return y
```

Shell :

```
>>> f(5)
17
>>> a
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'a' is not defined
>>> y
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'y' is not defined
>>> x
12
```

Ici `x` est une variable globale, alors que `a` et `y` sont locales. La fonction `f` peut utiliser `x` dans sa définition mais même une fois la fonction `f` appelée sur le paramètre `5`, les variables `a` et `y` n'existent pas en dehors de la fonction.

⋆ **Éditeur :**

```
x = 12

def f(a):
    x = 1
    y = a + x
    return y
```

Shell :

```
>>> f(5)
6
>>> x
12
```

Ici c'est plus subtil : il y a bien une variable `x` globale mais il y a une autre variable `x` définie dans la fonction et celle-ci est locale. **Dans une fonction**, Python cherche d'abord parmi les variables locale, donc c'est bien la variable `x` qui vaut `1` qui est utilisée pour l'instruction `y = a + x`. Une fois en dehors de la fonction, les variables locales n'existent plus donc la seule variable `x` qui existe est celle qui vaut `12`.

⋆ **Éditeur :**

```
x = 12

def f(x):
    y = 3 + x
    return y
```

Shell :

```
>>> f(5)
8
>>> x
12
```

C'est exactement la même chose que pour le dernier exemple : deux variables portent le même nom, une locale et l'autre globale. Celle utilisée dans la fonction est la variable locale et celle accessible depuis l'extérieur est la variable globale.

⋆ **Éditeur :**

```
x = 12

def f(a):
    x = x + 1
    y = a + x
    return y
```

Shell :

```
>>> f(5)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "d:\downloads\tp2.py", line 4, in f
    x = x + 1
        ^
UnboundLocalError: cannot access local variable 'x' where it is not
   associated with a value
```

On remarque ici que Python nous interdit de modifier une variable globale dans une fonction. Nous reparlerons de cela dans le chapitre suivant sur les listes.