Informatique

9

Fonctions

Cours

Fonctions **4**

1.I. Utilité **4**

1.II. Les fonctions par l'exemple **4**

1.III. Structure **5**

1.III.1 Définition d'une fonction 5

1.III.1.a Syntaxe 5

1.III.1.b Noms des fonctions - Dangers 5

1.III.1.c Cas particulier des fonctions mathématiques 5

1.III.2 Arguments 6

1.III.2.a Aucun argument 6

1.III.2.b Arguments obligatoires 6

1.III.2.c Arguments optionnels prédéfinis 6

1.III.2.d Arguments optionnels en quantité quelconque 7

1.III.2.e Listes optionnelles – Danger 7

1.III.3 Contenu 8

1.III.3.a Généralités 8

1.III.3.b Champs « aide » 8

1.III.3.c Renvoie d'un objet 9

1.III.3.c.i Renvoie 9

1.III.3.c.ii Récupération 9

1.III.3.d Aucun renvoie 10

1.IV. Remarques **11**

1.IV.1 Récupérer TOUS les objets renvoyés 11

1.IV.2 Return : une option de Pyhton 12

1.IV.3 Oublie des parenthèses 12

1.IV.4 Fonctions de plusieurs variables ramenées à 1 variable 13

1.IV.5 Renvoie d'un booléen 13

1.IV.6 Import d'une fonction d'un autre fichier 13

1.V. Notions de variables locales et globales **14**

1.V.1 Structure générale d'un code avec fonctions 14

1.V.2 Variables locales 15

1.V.2.a Définition 15

1.V.2.b Dangers 15

1.V.3 Variables globales 16

1.V.3.a Définition 16

1.V.3.b Utilisation dans une fonction 16

1.V.3.c Modification de la variable localement 16

1.V.3.d Modification de la variable globalement 18

1.V.4 Problèmes rencontrés 19

1.V.4.a Noms des variables 19

1.V.4.b Gestion des listes 20

1.V.4.c Fonctions dans des fonctions 21

1.VI. Débogage des erreurs **22**

1.VII. Mise en mémoire des fonctions **22**

1.VIII. Pile d'exécution (stack) **23**

Fonctions

1.I. Utilité

Une fonction est une portion de code qui peut être « appelée », exécutée, à n'importe quel endroit d'un code informatique, à partir du moment où elle a été définie, c'est-à-dire lue une fois et mise en mémoire par l'ordinateur.

Elle peut exécuter une ou plusieurs opérations en prenant ou non des arguments en entrée, et en renvoyant ou non des variables en sortie.

Elle prend en entrée des variables bien définies, et renvoie le résultat voulu. On peut donc aisément copier/coller une fonction faite par quelqu'un d'autre à partir du moment où l'on maîtrise ses entrées et sorties, et ce sans modifier les variables déjà existantes (notion de variables locales vue par la suite).

1.II. Les fonctions par l'exemple

Supposons que nous souhaitons déterminer les diviseurs de 3 nombres. Mettons en œuvre ce programme avec et sans fonctions.

Sans fonctions

```
a = 10
b = 5
c = 200

Diviseurs_a = []
Diviseurs_b = []
Diviseurs_c = []

for i in range(1,a+1):
    if a%i == 0:
        Diviseurs_a.append(i)

for i in range(1,b+1):
    if b%i == 0:
        Diviseurs_b.append(i)

for i in range(1,c+1):
    if c%i == 0:
        Diviseurs_c.append(i)
```

Avec fonctions

```
def f_Diviseurs(Nombre):
    Diviseurs = []
    for i in range(1,Nombre+1):
        if Nombre%i == 0:
            Diviseurs.append(i)
    return Diviseurs

a = 10
b = 5
c = 200

Diviseurs_a = f_Diviseurs(a)
Diviseurs_b = f_Diviseurs(b)
Diviseurs_c = f_Diviseurs(c)
```

Sans fonctions, on réécrit le code qui détermine les diviseurs autant de fois qu'il faut afin d'avoir toutes les listes nécessaires.

Lorsque l'on utilise une fonction, celle-ci a pour rôle de déterminer tous les diviseurs d'un nombre en entrée et de renvoyer cette liste. Ainsi, on l'exécute en début de code puis à chaque appel, elle donne les diviseurs du nombre qu'elle a en argument. On voit clairement l'intérêt de simplification par l'utilisation des fonctions.

Et il est parfaitement possible d'appeler une fonction dans une fonction, voire de créer une sous-fonction locale dans une fonction.

cc BY NC SA

1.III. Structure

1.III.1 Définition d'une fonction

1.III.1.a Syntaxe

Pour définir une fonction, on écrit la première ligne suivante :

```
def Nom_Fonction(Argument_1, Argument_2,…,Argument_n):
```

def permet de définir une fonction dont le nom est `Nom_Fonction` (sans espaces et éviter les accents). Personnellement, j'aime bien appeler les fonctions par « f_Nom_Fonction » afin de bien faire la différence entre d'éventuelles variables et les fonctions associées.

Après le nom de la fonction viennent des parenthèses dans lesquelles on définit si nécessaire les noms des variables qui seront les paramètres d'entrée de la fonction. Il est tout à fait possible de laisser ces parenthèses vides si aucun argument ne doit être défini.

Enfin, il ne faut pas oublier les « : » comme pour les boucles.

1.III.1.b Noms des fonctions - Dangers

Quand on ne fait pas attention aux noms, on peut rencontrer des erreurs. Voici quelques exemples :

<table>
<tr>
<td>

```
def Maximum(L):
    return max(L)

L = [0,1,2]
Maximum = Maximum(L)

LL = [4,5,6]
Maximum = Maximum(L)
```

</td>
<td>

```
Maximum = Maximum(L)
TypeError: 'int' object is not callable
```

On a appelé la fonction Maximum et le maximul de L avec le même nom. Deux problèmes :

- On ne sait plus si Maximum est la fonction ou un nombre
- On remplace la fonction par un nombre qui cause le bug lors d'un nouvel appel de la fonction qui nous dit qu'un int n'est pas une fonction…

Préférer f_Maximum pour la fonction (par exemple)

</td>
</tr>
<tr>
<td>

```
L = [1,2,3]
M = max(L)
print(M)

def max(L):
    return 10

M = max(L)
print(M)

max = 50
M = max(L)
print(M)
```

</td>
<td>

```
M = max(L)
TypeError: 'int' object is not callable
```

Au premier appel de max, la fonction python est utilisée, le résultat est bon.
Au second appel de max, la fonction python a été remplacée par notre fonction du même nom
Au 3° appel de max, la variable max a pris la place des fonctions, on ne peut plus appeler la fonction.

</td>
</tr>
</table>

1.III.1.c Cas particulier des fonctions mathématiques

On peut définir très simplement une fonction mathématique (les deux codes sont équivalents) :

<table>
<tr>
<td>

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

</td>
<td>

```
g = lambda x: x**2
```

</td>
</tr>
</table>

1.III.2 Arguments

1.III.2.a Aucun argument

Une fonction peut tout à fait ne pas avoir d'arguments :

```
def Message():
print('Veuillez patienter')
```

```
>>> Message()
Veuillez patienter
```

1.III.2.b Arguments obligatoires

Supposons que l'on ait défini la fonction suivante :

```
from math import sqrt
def f(x,y,z):
    Norme = sqrt(x**2+y**2+z**2)
    return Norme
```

Les arguments non optionnels sont forcément attendus lors de l'appel de la fonction :

```
>>> f(1)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: f() missing 2 required positional arguments: 'y' and 'z'
```

Lors de l'appelle de la fonction, il suffit d'écrire par exemple :

```
f(1,2,3)
```

```
>>> f(1,2,3)
3.7416573867739413
```

Attention à l'ordre des arguments de la fonction, en effet si l'on écrit :

```
x = 1
y = 2
z = 3
f(z,y,x)
```

```
>>> f(z,y,x)
3.7416573867739413
```

La fonction va associer la valeur z à x, y à y et x à z !!!

1.III.2.c Arguments optionnels prédéfinis

Il est simple de définir des arguments optionnels. Ils sont nécessairement introduits à la fin de la parenthèse, après les arguments obligatoires. Leur valeur est définie par défaut :

```
def f(x,a=0,b=0):
    return a*x+b
```

```
>>> f(2)
0

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

Ainsi, si les arguments ne sont pas définis lors de l'appel de la fonction, ils seront égaux à la valeur de base, ici 0. On veillera à mettre le plus à droite les arguments toujours optionnels. Dans l'exemple ci-dessus, si l'on veut préciser la valeur de b, il faudra aussi préciser a…

cc BY NC SA

1.III.2.d Arguments optionnels en quantité quelconque

```
def f(*args):
    Somme = 0
    for arg in args:
        Somme += arg
    return Somme
print(f(2,1))
```

`*args` permet de dire que les arguments peuvent avoir n'importe quelle taille. L'appel `(f(2))` renvoie 2, `(f(2,1))` renvoie 3…

Cela revient au même que de demander une liste en argument, et d'appeler la fonction pour une liste dans laquelle on choisit les arguments :

```
def f(L):
    Somme = 0
    for Terme in L:
        Somme += Terme
    return Somme
```

1.III.2.e Listes optionnelles – Danger

Soit l'exemple suivant :

```
def f(a=[]):
    a.append(1)
    print(a)
def g(a=[]):
    a.append(2)
    print(a)
    a = [1,2,3]
def h(L,a=[]):
    for t in L:
        a.append(t)
    print(a)
```

```
>>> f()
[1]
>>> f()
[1, 1]
>>> f()
[1, 1, 1]
>>> g()
[2]
>>> g()
[2, 2]
>>> h([1,2])
[1, 2]
>>> h([3,4])
[1, 2, 3, 4]
```

```
>>> f()
[1, 1]

>>> f([2])
[2, 1]

>>> f()
[1, 1, 1]
```

Oui, vous avez bien vu. On pourrait croire que la liste a sera remise à 0 si on ne précise rien, mais non. Pire encore, même si on précise un argument, elle n'est pas remise à 0. La liste a est même mémorisée, et ce pour chaque fonction (3 listes a différentes mémorisées). La remise en mémoire de la fonction remet à 0 les listes optionnelles en mémoire ce celles-ci.

Par ailleurs, on en parlera un peu plus tard, mais la ligne « `a = [1,2,3]` » crée une nouvelle liste locale a qui ne remplace pas la liste a sur laquelle on travaille…

Je ferai en sorte d'éviter de travailler avec des listes optionnelles dans mes TD/TP pour éviter ce piège !

On peut aussi réinitialiser la liste en mettant une liste vide en argument :

```
def f(a=[]):
    a.append(1)
    print(a)
```

```
>>> f()
[1]
>>> f()
[1, 1]
>>> f()
[1, 1, 1]
>>> f([])
[1]
```

Attention enfin : écrire `def f(a,i=0,j=len(a)):` conduira au même type de problèmes avec len(a) qui (1) nécessite que a existe à la mise en mémoire de la fonction et (2) prendra ensuite la liste a en mémoire pour créer j qui ne vaudra donc pas len(a) de la liste mise en argument.

cc BY NC SA

1.III.3 Contenu

1.III.3.a Généralités

Ensuite, il faut indenter le contenu de la fonction et mettre le code que l'on souhaite. Nous verrons dans le prochain paragraphe comment manipuler les variables dites locales ou globales liées aux fonctions.

Attention : un contenu de fonction vide empêche l'exécution d'un programme. Inscrire **pass** ou quelque chose d'inutile comme `a = 0` afin que le code fonctionne.

Il est parfaitement possible d'avoir un contenu de fonction possédant des erreurs (autres que de syntaxe) et que sa mise en mémoire fonctionne. C'est lors de son appel qu'il y aura une erreur :

```
def f(x):
    return a
```

L'exécution du code ci-dessus permet de mettre en mémoire la fonction sans problèmes. C'est au moment de son appel qu'il y a un problème si a n'existe pas :

```
>>> f(1)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<tmp 1>", line 2, in f
    return a
NameError: name 'a' is not defined
```

Conclusion : pensez à mettre en mémoire (F5 par exemple) puis à utiliser vos fonctions pour vérifier qu'elles fonctionnent, dès qu'elles sont écrites.

1.III.3.b Champs « aide »

On peut définir un champ d'aide dans une fonction, c'est-à-dire un texte qui sera retourné si on tape dans la console « `help(f_Fonction)` »

Exemple :

```
def f_Fonction(x):
    ''' Cette fonction prend en argument un reel et renvoie son carre'''
    return x**2
```

Rq : Ne pas mettre d'accents dans le champs d'aide

Lors de l'exécution de la fonction help(), on obtient :

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

f_Fonction(x)

    Cette fonction prend en argument un reel et renvoie son carre
```

Le fonction « help » fonctionne sur toutes les fonctions de python.

cc BY NC SA

1.III.3.c Renvoie d'un objet

1.III.3.c.i Renvoie

Si l'on souhaite que la fonction renvoie un résultat, il suffit d'inscrire à la fin : `return Objet`

Attention, ce que vous mettrez après ce return ne sera pas exécuté, return marque la fin de la fonction.

Pour retourner plusieurs objets :

Ne fonctionne pas	Fonctionne
`return Objet_1` `return Objet_2`	`return [Objet_1,Objet_2]`

Attention : `return Objet_1,Objet_2` sans crochets ou avec des parenthèses renvoie un tuple non modifiable. En effet, regarder l'exemple suivant :

```python
def f(x):
    return x,x,x,x
A = f(1)
print(type(A))
A[0] = 1
```

```python
def f(x):
    return (x,x,x,x)
A = f(1)
print(type(A))
A[0] = 1
```

```
<class 'tuple'>
Traceback (most recent call last):
  File "<tmp 1>", line 5, in <module>
    A[0] = 1
TypeError: 'tuple' object does not support item assignment
```

Toutefois, si la fonction renvoie des listes par exemple, il sera possible de modifier ce qu'il y a dans ces listes. Mais je recommande donc de bien retenir de mettre les crochets !

1.III.3.c.ii Récupération

Lors de l'appel d'une fonction $f_Fonction$ qui renvoie deux objets en sortie, on écrira lors de l'appel de la fonction :

`a,b = f_Fonction(...)`	

`v = f_Fonction(...)`
`a = v[0]`
`b = v[1]`
	`v = f_Fonction(...)`
	`a,b = v`

ATTENTION : si vous appelez uniquement la fonction `f_Fonction(...)`, les variables a et b ne seront pas créées !!!

```python
def f(x):
    a = 2*x
    b = 3+x
    return a,b
```

```
>>> f(2)
(2, 3)

>>> (executing lines 1 to 4 of "<tmp 1>")

>>> f(2)
(4, 5)

>>> a
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'a' is not defined

>>> b
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'b' is not defined
```


CC BY NC SA

1.III.3.d Aucun renvoie

Une fonction peut réaliser un travail sans rien renvoyer, par exemple une modification de liste :

```
def Ajout(x,L):
    L.append(x)
```

```
>>> L=[]
>>> Ajout(1,L)
>>> L
[1]
>>> Ajout(1,L)==None
True
>>> G = [Ajout(1,L)]
>>> G
[None]
```

Si la fonction ne retourne rien, elle retourne en réalité un résultat qui s'appelle « None ».

On peut aussi écrire « `return` » tout seul pour arrêter une fonction, ce qui retourne aussi « None ».

On peut arrêter l'exécution d'une fonction simplement en inscrivant « `return` » et rien derrière. La fonction ne renvoie alors rien, ou plutôt, renvoie l'objet None.

```
def Message():
    print('Veuillez patienter')
    return
    print('Impossible')
```

```
>>> Message()
Veuillez patienter
```


CC BY NC SA

1.IV. Remarques

1.IV.1 Récupérer TOUS les objets renvoyés

Soit l'exemple suivant, où à partir d'une liste de 4 listes, renvoie les 4 maximums de celles-ci.

```
def Max(L):
    M = L[0]
    for t in L:
        if t > M:
            M = t
    return M

L = [1,2,3,4,3]
print(Max(L))

from random import randint as rand
n = 1000
L1 = [rand(0,1000) for i in range(n)]
L2 = [rand(0,1000) for i in range(n)]
L3 = [rand(0,1000) for i in range(n)]
L4 = [rand(0,1000) for i in range(n)]
Listes = [L1,L2,L3,L4]

def Max_Listes(Listes):
    Liste_M = []
    for Liste in Listes:
        Max_L = Max(Liste)
        Liste_M.append(Max_L)
    return Liste_M
```

Voici maintenant deux manières qui semblent similaires pour créer les 4 maximums :

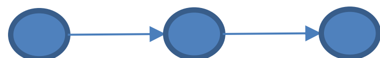
Gauche	Droite

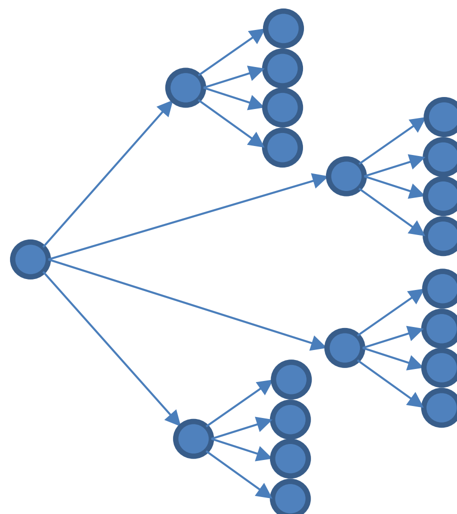
`M1,M2,M3,M4 = Max_Listes(Listes)`	`M1 = Max_Listes(Listes)[0]`
	`M2 = Max_Listes(Listes)[1]`
	`M3 = Max_Listes(Listes)[2]`
	`M4 = Max_Listes(Listes)[3]`

Rendez-vous compte de l'inutilité de la version de droite… La fonction est de complexité $O(4n)$ à droite au lieu de $O(n)$ à gauche. On détermine 4 fois chaque maximum, pour n'en stocker qu'un à chaque fois.

On veillera donc à créer les différents éléments renvoyés par une fonction, quitte à ne pas en utiliser…

Et cela aura beaucoup d'influence dans les codes récursifs de seconde année, où la complexité pourra passer de $O(n)$ à $O(4^n)$!





Remarque : si l'on ne veut pas récupérer un résultat, on pourra par exemple écrire :

- `M1,RAS,M3,M4 = Max_Listes(Listes)`
- `M1,_,M3,M4 = Max_Listes(Listes)`

1.IV.2 Return : une option de Pyhton

Comme nous l'avons déjà abordé, certains logiciels de programmation n'utilisent pas l'indentation pour organiser les boucles, conditions, fonctions. Par exemple, voici la comparaison de deux codes identiques sous Python et Matlab :

Python

```
def f(x):
    if x == 0:
        s = 0
    else:
        s = 1/x
    return s

print(f(2))
```

Matlab

```
function [ s ] = f( x )
if x == 0
s = 0;
else
s = 1/x;
end
end
disp(f(1));
```

L'absence d'indentation est contrée par la présence d'un « end » pour marquer la fin de la condition if. Sous Python, vous risquez de prendre la « mauvaise » habitude de ne pas forcément écrire else en présence de return :

Python

```
def f(x):
    if x == 0:
        return 0
    return 1/x
print(f(2))
```

Ce code fonctionne sous Python, mais ce n'est pas possible sous Matlab en l'état. En effet, Matlab ne permet pas de s'arrêter à un « return », qui d'ailleurs n'existe pas. Matlab lit tout le code de la fonction et renvoie la variable de sortie. Il faudra donc forcément utiliser un « else », ou donner une valeur initiale à s, modifiée par la suite si une condition est validée…

Comme l'objectif est de vous apprendre à programmer, quel que soit le logiciel, pensez à toujours mettre « else ».

1.IV.3 Oublie des parenthèses

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

```
>>> f
<function f at 0x0000028964CD28C8>
```

Sans parenthèses, f est la fonction, et non son résultat. Ainsi, on peut par exemple créer une fonction identique à f en écrivant g=f… Vous ferez peut-être un jour l'erreur d'oublier des parenthèses, alors vous aurez un message d'erreur du genre (multiplication de la fonction f par un entier): « unsupported operand type(s) for *: 'int' and 'function' »…

CC BY NC SA

1.IV.4 Fonctions de plusieurs variables ramenées à 1 variable

Il arrive que l'on souhaite utiliser une fonction de plusieurs variables pour une seule variable évolutive, les autres étant fixées. Il est très simple de se ramener à une fonction d'une variable, en procédant ainsi :

```
def f(x,y,z):
    return 2*x+3*y+4*z

def F(x):
    y = 2
    z = 3
    return f(x,y,z)
```

```
def f(x,y=2,z=3):
    return 2*x+3*y+4*z
```

Cela est souvent utilisé pour résoudre des équations par Dichotomie et Newton lorsqu'elles possèdent plusieurs variables.

1.IV.5 Renvoie d'un booléen

Il arrivera fréquemment que vous souhaitiez renvoyer un booléen valant True ou False avec une condition.

Les deux codes ci-dessous sont équivalents :

```
def f(x):
    if x >=0 :
        return True
    else:
        return False
```

```
def f(x):
    return x>=0
```

Préférez donc renvoyer le test !

1.IV.6 Import d'une fonction d'un autre fichier

Supposons que vous disposez dans le même dossier de deux fichiers « Essai.py » et « Fichier.py » et que vous avez bien coché l'option de pyzo dans run « Change directory when executing file ».

Fichier.py

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

Essai.py

```
from Fichier import Fct
```

Il est possible d'importer la fonction Fct du fichier « Fichier.py » dans le fichier « Essai.py ».

1.V. Notions de variables locales et globales

1.V.1 Structure générale d'un code avec fonctions

En général, lorsque l'on crée un code avec des fonctions, on réserve tout un premier espace au début du fichier pour définir toutes les fonctions.

Ensuite, on créer le code à proprement parler, qui résout le problème traité, et qui fait appel aux fonctions précédemment créées.

```
## Définition des fonctions

def f_Fonction_1(n):
    ''' Aide '''
    Var = 1
    ...
    return Var

def f_Fonction_2(...):
    ...
    ...
    ...

def f_Fonction_3(...):
    ...
    ...
    ...

## Programme

Sol = f_Fonction_1(...)+f_Fonction_2(...)/f_Fonction_3(...)
```


CC BY NC SA

1.V.2 Variables locales

1.V.2.a Définition

Une variable est locale si :

- Elle est présente dans la parenthèse des arguments
- Elle est créée dans une fonction.

Une variable locale n'existe que dans la fonction dans laquelle elle est créée.

On fait ce que l'on veut des variables locales tant que l'on reste dans le cadre de la fonction considérée. Elles n'existent plus à la sortie des fonctions. Tout se passe comme si, lorsque l'on appelle une fonction, on ouvrait **un second Pyzo** dans lequel on exécute le code de la fonction et l'on définit de nouvelles variables. A la différence près que ce second Pyzo a le droit d'aller chercher des variables existantes dans le premier (abordé plus tard), l'inverse étant impossible. A la fin de l'exécution de la fonction, c'est comme si l'on fermait le second Pyzo et toutes les variables créées dans celui-ci disparaissent.

D'une manière générale, je mettrai dans la suite « `loc_` » pour indiquer que la variable est locale.

1.V.2.b Dangers

Attention aux noms choisis pour les variables locales. Voici un exemple d'erreur à ne pas faire :

```
def f(min):
    L = [1,2,3]
    return min(L)

f(10)
```

```
>>> (executing file "<tmp 1>")
Traceback (most recent call last):
  File "<tmp 1>", line 5, in <module>
    f(10)
  File "<tmp 1>", line 3, in f
    return min(L)
TypeError: 'int' object is not callable
```

Lorsque l'on définit l'argument min dans la fonction, on déclare que min sera une variable locale. La fonction native de python min pour minimum se retrouve donc remplacée par la variable min qui est dans le cas de l'appel de f(10) un entier. Lorsque l'on appelle min(L), on essaie d'appeler un entier…

CC BY NC SA

1.V.3 Variables globales

1.V.3.a Définition

Toutes les variables créées dans le code principal, autrement dit toute variable qui n'est pas créée dans une fonction, est une variable globale que l'on retrouvera dans le « Workspace ».

1.V.3.b Utilisation dans une fonction

Il est possible d'appeler une variable globale dans une fonction sans la mettre en argument :

```
## Définition des fonctions

def f_Fonction():
    loc_Somme = 0
    for i in range(n):
        loc_Somme += i
    return loc_Somme

## Programme

n = 10
Sol = f_Fonction()
print(Sol)
```

```
45
```

1.V.3.c Modification de la variable localement

Si maintenant on veut changer la valeur de n localement dans la fonction :

```
## Définition des fonctions

def f_Fonction():
    n = n + 1
    loc_Somme = 0
    for i in range(n):
        loc_Somme += i
    return loc_Somme

## Programme

n = 10
Sol = f_Fonction()
print(Sol)
```

```
Traceback (most recent call last):
  File "<tmp 1>", line 13, in <module>
    Sol = f_Fonction()
  File "<tmp 1>", line 4, in f_Fonction
    n = n + 1
UnboundLocalError: local variable 'n' referenced before assignment
```

Cette erreur se produit même si n est créé avant la fonction. En effet, écrire « n = » consiste à créer une variable locale n. Il veut alors mettre dans cette variable locale n la variable n (nous savons qu'elle existe de manière globale… mais la fonction est en train de créer celle du même nom, de manière locale…). Il nous indique que l'on demande de mettre dans la variable locale n la même variable locale n qui n'existe pas, « referenced before assignment - référencée avant de l'affecter ». Ce n'est pas possible.

CC BY NC SA

Solution : si l'on veut modifier une variable globale dans une fonction **sans** la modifier en tant que variable globale, il faut la faire devenir locale en la mettant en argument de la fonction comme proposé ci-dessous :

```
## Définition des fonctions

def f_Fonction(loc_n):
    loc_n = loc_n + 1
    loc_Somme = 0
    for i in range(loc_n):
        loc_Somme += i
    return loc_Somme

## Programme

n = 10
Sol = f_Fonction(n)
print(Sol,n)
```

```
55 10
```

Mais attention : à la sortie de la fonction, cette variable n'est pas modifiée ! n vaudra toujours 10.

Pour compléter, regardez les exemples très simples ci-dessous :

```
## Définition des fonctions

def test(x):
    x = 10

## Programme

x = 1
test(x)
print('x: ',x)
```

```
x:  1
```

```
## Définition des fonctions

def test(x):
    x += 10

## Programme

x = 1
test(x)
print('x: ',x)
```

```
x:  1
```

Que l'on cherche à modifier x en le remplaçant ou en lui ajoutant quelque chose, la variable globale x n'est pas modifiée. C'est très important de s'en souvenir !!!

1.V.3.d Modification de la variable globalement

Si l'on souhaite modifier une variable globale dans une fonction et que cette modification soit maintenue après exécution de la fonction, il faut :

- Soit qu'elle soit locale dans la fonction en la mettant en argument, puis faire en sorte que la fonction retourne cette valeur de variable globale et que lors de l'appelle de la fonction, on affecte la nouvelle valeur de la variable à l'ancienne (`n = f_Fonction(n)`)

```python
## Définition des fonctions

def f_Fonction(loc_n):
    loc_n = loc_n + 1
    loc_Somme = 0
    for i in range(loc_n):
        loc_Somme += i
    return loc_Somme,loc_n

## Programme

n = 10
Sol, n = f_Fonction(n)
print(Sol,n)
```

```
55 11
```

- Soit la déclarer comme variable globale à l'intérieur de la fonction en écrivant `global n` et alors la modifier dans la fonction

```python
## Définition des fonctions
def f_Fonction():
    global n
    n = n + 1
    loc_Somme = 0
    for i in range(n):
        loc_Somme += i
    return loc_Somme

## Programme
n = 10
Sol = f_Fonction()
print(Sol,n)
```

```
55 11
```

Dans ce code, on a déclaré la variable n comme une variable globale en écrivant `global n`. Ainsi, la variable n utilisée dans la fonction est la variable globale et à la fin de l'exécution du code, n vaut réellement n+1 !

Remarque : Si la variable n n'existe pas, écrire `global n` permet de créer la variable globale n dans la fonction afin de l'utiliser ensuite à l'extérieur ☺

```python
def f(x):
    global n
    n = 2
    return n*x
print(f(2))
print(n)
```

```
>>> (executing file "<tmp 1>")
4
2
```


cc BY NC SA

1.V.4 Problèmes rencontrés

1.V.4.a Noms des variables

Supposons que l'on appelle une fonction `f_Fonction(a,b,c)` dans le programme principal (pas depuis une fonction) et que l'on définit la fonction avec `def f_Fonction(a,b,c):`. Il faut faire attention car les variables a, b et c ne sont pas vues de la même manière dans les deux cas :

`def f_Fonction(a,b,c):`	`Sol = f_Fonction(a,b,c)`
a, b et c sont des variables locales	a, b et c sont des variables globales
La modification de ces variables dans la fonction ne va pas changer les variables globales !	
Préférer leur donner des noms différents afin d'en être conscient : `loc_a,loc_b,loc_c`	

Prenons un exemple. Soit le code suivant, proprement rédigé avec distinction des noms des variables locales et globales :

```
## Définition des fonctions
def f_Fonction(loc_n,loc_Somme): # On ajoute
les loc_n premiers entiers à loc_Somme
    for i in range(loc_n):
        loc_Somme += i
    return loc_Somme

## Programme
n = 10
Somme = 0
Sol = f_Fonction(n,Somme)
print(Sol,Somme)
```

```
45 0
```

Prenons maintenant le code suivant où variables locales et globales ont le même nom :

```
## Définition des fonctions
def f_Fonction(n,Somme):
    for i in range(n):
        Somme += i
    return Somme

## Programme
n = 10
Somme = 0
Sol = f_Fonction(n,Somme)
print(Sol,Somme)
```

```
45 0
```

Dans le premier code, tout va bien, on différencie variables locales et globales. Somme vaut 0 à la fin.

Dans le second code, on a fait exprès de ne pas faire attention aux noms des variables locales dans la définition de la fonction. Ainsi, en particulier, `loc_Somme` est devenu `Somme`. Lors de l'appel de la fonction `f_Fonction`, on pense que la variable globale `Somme` qui porte le même nom que la variable locale `Somme` de la fonction changent toutes les deux. Mais à la fin, le `print(Somme)` renvoie une valeur nulle ! C'est normal.

Soyez donc conscients de la différence entre les variables locales et globales, lorsque vous leur donnez le même nom !

cc BY NC SA

1.V.4.b Gestion des listes

Le cas des listes est quelque peu problématique vis-à-vis des variables locales/globales.

Soit le premier exemple suivant :

```
def f_Fonction(loc_L):
    for i in range(n):
        loc_L[i] = 0

n = 10
L = [i for i in range(n)]
f_Fonction(L)
print('L: ',L)
```

```
L:  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

On croirait que la fonction a modifié la liste locale `loc_L` qui serait une copie de `L` alors qu'en réalité, elle a modifié les blocs mémoire vers lesquels pointe la liste `loc_L` qui sont les mêmes que ceux de la liste `L`.

Soit le second exemple ou on ajoute une égalisation de listes :

```
def f_Fonction(loc_L):
    loc_LL = loc_L
    for i in range(n):
        loc_LL[i] = 0

n = 10
L = [i for i in range(n)]
f_Fonction(L)
print('L: ',L)
```

```
L:  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

En créant la variable locale `loc_LL`, on espère créer une nouvelle liste que l'on va pouvoir modifier. Sauf que : La liste en variable globale L a été modifiée aussi…

On retiendra que pour les listes, tout est global. Le nom de la liste pointe vers des cases mémoire qui sont modifiées par la fonction.

Et ce dernier exemple, où l'on veut modifier la liste L dans la fonction :

```
def test(L):
    L = [1,2,3,4,5,6]

L = [1,2,3]
test(L)
print('L: ',L)
```

```
L:  [1, 2, 3]
```

Dans le code proposé ci-dessus, écrire `L = [1,2,3,4,5,6]` crée une nouvelle liste L locale, indépendante de la liste L en argument.

A retenir donc : Pour modifier une liste dans une fonction, il faut obligatoirement modifier ses valeurs avec par exemple L.pop(), L.append(), L[i] = 1, L+= ... Remarquer que le return est inutile

```
def test(L):
    L += [1,2,3,4,5,6]

L = [1,2,3]
test(L)
print('L: ',L)
```

```
L:  [1, 2, 3, 1, 2, 3, 4, 5, 6]
```


CC BY NC SA

1.V.4.c Fonctions dans des fonctions

Soit l'exemple suivant :

```
k = 2

def f(x):
    return k*x

def g(x):
    k = 10000
    Val = 2 * f(x)
    return Val

Sol = g(3)
print(Sol)
```

```
12
```

La fonction f fait appel à une variable globale k.

La fonction g fait appel à la fonction f. Dans la fonction g, on définit k=10000. Dans l'ordinateur, une nouvelle variable k différente de la variable globale est créée. Elle est locale, uniquement pour g. Lors de l'appel de g(2), on fait appel à f qui recherche la variable globale k qui vaut 2. Le k=10000 est donc totalement sans effets ☹

Pour corriger cela, deux solutions :

Méthode 1 : il suffit de modifier f pour qu'elle prenne comme variable d'appel la valeur de k :

```
def f(x,k):
    return k*x

def g(x):
    k = 10000
    Val = 2 * f(x,k)
    return Val

Sol = g(3)
print(Sol)
```

```
60000
```

La définition de k en dehors des fonctions devient alors inutile et ce code prendra bien en compte la valeur k=10000.

Méthode 2 : déclarer k global et lui affecter la valeur adéquate pour qu'elle soit utilisée dans f :

```
def f(x):
    return k*x

def g(x):
    global k
    k = 10000
    Val = 2 * f(x)
    return Val

Sol = g(3)
print(Sol)
```

```
60000
```


cc BY NC SA

1.VI. Débogage des erreurs

La manipulation de variables locales dans les fonctions rend assez délicat le débogage. En effet, en cas d'arrêt du code, toutes les variables locales sont effacées et on ne peut pas simplement déterminer où est l'erreur.

Il y a donc plusieurs solutions, les trois meilleures étant :

- Créer le code de la fonction directement dans le programme afin de manipuler des variables globales accessibles après arrêt du code puis mettre le code dans une fonction lorsque tout fonctionne
- Mettre des print(…) un peu partout dans les fonctions afin d'afficher les variables lors de l'exécution de la fonction pour trouver le problème
- Exécuter les lignes de la fonction à l'aide de l'option « Execute selection » en les surlignant sans modifier le code (def et indentation). Les variables censées être locales dans la fonction seront créées de manière globale, ce qui permettra de voir combien elles valent

1.VII. Mise en mémoire des fonctions

A partir du moment ou une fonction a été mise en mémoire, on peut directement l'utiliser « à la main » dans la console « Shells ».

Pour mettre une fonction en mémoire à la main, il suffit de la copier dans le code et de la coller dans la console puis d'appuyer 2 fois sur « Entrée ». On peut ensuite l'utiliser.

1.VIII. Pile d’exécution (stack)

A chaque fois qu’une fonction est appelée, cet appel est stocké dans ce que l’on appelle la pile d’exécution.

Lorsqu’une fonction Fonction_1 qui appelle une seconde fonction Fonction_2 qui appelle une 3° fonction Fonction_3 est exécutée, voici schématiquement ce qu’il se passe dans la pile d’exécution :

Appel fonction 1	Appel fonction 2	Appel fonction 3
Fonction_1 - Var_Loc_1	Fonction_2 - Var_Loc_2 Fonction_1 - Val_Loc_1	Fonction_3 - Var_Loc_3 Fonction_2 - Var_Loc_2 Fonction_1 - Val_Loc_1

A chaque étape, la pile stocke les fonctions en cours dans l’ordre d’exécution ainsi que les variables locales associées. A partir du moment où la dernière fonction Fonction_3 est appelée, elle exécute son script puis la fonction Fonction_2 peut terminer son travail, puis la fonction Fonction_1 et c’est terminé.

On peut simplement voir cette pile d’exécution en intégrant une erreur dans la fonction 3. La console nous renvoie un « Traceback »

Exemple :

```
def Fonction_1(n):
    return Fonction_2(n)

def Fonction_2(n):
    return Fonction_3(n)

def Fonction_3(n):
    return (1/n)
```

La console nous renvoie :

```
>>> Fonction_1(0)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "C:\Users\DDP\Dropbox\Privé\Professionnel\Cours\CPGE\Chapitres\Info IPT\IPT 2\TP\TP2 - Récursivité\TP2 - Récursivité.py", line 39, in Fonction_1
    return Fonction_2(n)
  File "C:\Users\DDP\Dropbox\Privé\Professionnel\Cours\CPGE\Chapitres\Info IPT\IPT 2\TP\TP2 - Récursivité\TP2 - Récursivité.py", line 42, in Fonction_2
    return Fonction_3(n)
  File "C:\Users\DDP\Dropbox\Privé\Professionnel\Cours\CPGE\Chapitres\Info IPT\IPT 2\TP\TP2 - Récursivité\TP2 - Récursivité.py", line 45, in Fonction_3
    return (1/n)
ZeroDivisionError: division by zero
```

On voit clairement qu’après appelle de Fonction_1 a été appelée Fonction_2, qui a appelé Fonction_3, lieu de l’erreur.

CC BY NC SA