Exercices

Exercice ITC1.1 : Appel de fonctions et valeur renvoyée [*]

On définit les fonctions suivantes :

```
def somme(x:int,y:int) -> int:
    """ renvoie la somme de x et y """
    s=x+y
    return s

def somme_entiers(n:int) -> int:
    """ renvoie la somme des entiers de 1 à n inclus """
    somme=0
    for i in range(n+1):
        somme=somme+i
    return somme

def compare_avec_a(x:int) -> int:
    """ compare x avec a; renvoie +1 si x>a, -1 si x<a et 0 sinon """
    if x>a:
        return 1
    elif x<a:
        return -1
    else:
        return 0
```

Pour chaque extrait suivant, indiquez si le programme fonctionne, et si c'est le cas, la valeur affichée.

1.
```
print(somme(3,2))
```

2.
```
somme(1,2)
print(s)
```

3.
```
somme=0
somme_entiers(5)
print(somme)
```

4.
```
a=input("Entrez un nombre: ")
print(somme(a,2))
```

5.
```
print(compare_avec_a(12))
```

6.
```
a=10
x=8
print(compare_avec_a(12))
```

7.
```
a=10
s=somme_entiers(6)
print(compare_avec_a(somme))
```

Exercice ITC1.2 : État de la mémoire lors de l'exécution d'un programme [*]

On considère le programme (stupide) suivant :

```
1  a=3
2  def somme(x,y):
3      """ renvoie la somme de x et y """
4      z=x+y
5      return z
6
7  def incremente_a(valeur):
8      """ ajoute valeur à a """
9      global a
10     a=a+valeur
11
12 def maximum(a,b):
13     """ renvoie le maximum entre a et b """
14     if a>=b:
15         return a
16     else:
17         return b
18 x=2
19 y=1
20 y=somme(y,x)
21 incremente_a(y)
22 x=somme(y,-a)
23 c=maximum(x+y,a)
24 print(c)
```

1. Indiquez l'ordre d'exécution des lignes de ce programme (on pourra admettre que lorsque la ligne 14 est atteinte, a est plus petit que b).
2. Indiquez l'état de la mémoire globale et locale à la fin de l'exécution des lignes : 4 (premier appel), 20, 10, 22, 14 (exception pour celui-ci : donnez l'état **avant** l'exécution de cette ligne), et 23.

Exercice ITC1.3 : Appels de fonctions avec des listes [* à **]

On définit les fonctions suivantes qui prennent toutes comme argument deux listes de même longueur et ajoutent terme à terme les listes, mais avec des implémentations différentes :

```
1  def ajoute1(L1:list,L2:list) -> list:
2      for i in range(len(L1)):
3          L1[i]=L1[i]+L2[i]
4      return L1
5
6  def ajoute2(L1:list,L2:list) -> list:
7      L=L1.copy()
8      for i in range(len(L)):
9          L[i]=L[i]+L2[i]
10     return L
11
12 def ajoute3(L1:list,L2:list) -> list:
13     L=L2.copy()
14     for i in range(len(L1)):
15         L1[i]=L1[i]+L[i]
16     return L1
17
18 def ajoute4(L1:list,L2:list) -> list:
19     L=[]
20     for i in range(len(L1)):
21         L.append(L1[i]+L2[i])
22     return L
```

Pour chaque extrait suivant, indiquez la valeur affichée pour les 4 programmes différents (`X` étant à remplacer

par 1, 2, 3 ou 4).

1.

```
A=[1,2,3,4]
B=[3,2,-1,0]
C=ajouteX(A,B)
print(A,B,C)
```

2. Plus dur :

```
A=[1,2,3,4]
B=[3,2,-1,0]
C=ajouteX(A,B)
D=ajouteX(A,C)
print(C,D)
```

Exercice ITC1.4 : Extraction des nombres positifs d'une liste [**]

On propose deux fonctions qui prennent comme argument une liste de réels, et qui en extrait uniquement les réels positifs ou nuls, et en renvoie la liste. Par exemple avec [1,3,-1,0,-2,5,3,9] la fonction renverra [1,3,0,5,3,9].

```
def extrait1(L:list)->list:
    L2=[]
    for i in range(len(L2)):
        if L[i]>=0:
            L2.append(L[i])
    return L2

def extrait2(L:list)->list:
    i=0
    while i<len(L):
        if L[i]<0:
            L.pop(i) # supprime le ième terme de L
        else:
            i=i+1
    return L
```

1. La première fonction contient une erreur ; laquelle ? Corrigez-la.
2. Laquelle de ces deux fonctions a un effet de bord ?
3. Indiquez ce qu'affiche le programme suivant :

```
L1=[1,3,-1,3,-1]
L2=[-1,-2,-3,-4]
print(extrait2(L1))
L3=L1
L1.append(-3)
L4=L2.copy()
L2.append(4)
L5=extrait2(L2)
print(L1,L2,L3,L4,L5)
```

Exercice ITC1.5 : Fonctions et listes [**]

On considère le programme suivant :

```
s=8
def somme(L:[float]) -> float:
    s=0
    for i in range(len(L)):
        s=s+L[i]
    return s
```

```
7
8  def enleve_minimum(L:[float]):
9      imin=0
10     for i in range(len(L)):
11         if L[i]<L[imin]:
12             imin=i
13     L.pop(imin) # enlève le ième terme de la liste
14
15 liste=[-1,6,-2,6,-3]
16 for k in range(5):
17     total=somme(liste)
18     enleve_minimum(liste)
19     liste.append(s-total)
20     print(imin)
21     print(liste)
22 print(s)
```

1. Ce programme renvoie une erreur ; à quelle ligne ? Dans la suite, on supposera que cette ligne est remplacée par une ligne vide.
2. Que font les deux fonctions ? Ont-elles un effet de bord sur la liste passée en paramètre ?
3. Indiquez l'état de la mémoire globale et de la mémoire locale à la ligne 6 lors du premier appel à la fonction `somme`.
4. Indiquez ce que ce programme va afficher ; on indiquera clairement les 5 itérations de la boucle principale.

Exercice ITC1.6 : Tests de fonctions [*]

Proposez un jeu de tests pour les fonctions suivantes :

1. `est_pair(n:int)->bool` : renvoie True si l'entier est pair, False sinon
2. `tous_positifs(L:[float])->bool` : renvoie True si tous les termes de la listes sont positifs ou nuls, False sinon
3. `partie_entiere(x:float)->int` : renvoie la partie entière de x
4. `position_lettre(message:str, lettre:str)->int` : renvoie la première position où on trouve la lettre dans le message, et -1 si on ne la trouve pas
5. `trie(a:float, b:float, c:float)->[float]` : renvoie les trois réels classés dans l'ordre croissant

Corrections des exercices

Corrigé de l'exercice ITC1.1 : Appel de fonctions et valeur renvoyée [*]

1. Affiche 5.
2. Affiche une erreur : `s` est locale à la fonction.
3. Affiche 0 car la fonction a créé et modifié une variable locale. Si on avait mis `global somme` au début de la fonction, ç'aurait été différent.
4. Affiche une erreur car `a` n'a pas été converti en entier ; on ajoute donc un entier et un nombre.
5. Affiche une erreur car `a` n'est pas défini.
6. `a` n'existant pas en local, la fonction utilise la variable globale qui vaut 10 et la compare avec le `x` local qui vaut 12, le résultat affiché est donc 1.
7. Affiche une erreur car `somme` est une variable locale à la fonction `somme_entier`.

Corrigé de l'exercice ITC1.2 : État de la mémoire lors de l'exécution d'un programme [*]

1. Rappel : les fonctions sont chargées en mémoire mais non exécutées tant qu'on ne les appelle pas.
   L'ordre d'exécution est donc : 1, 18, 19, 20→4, 5→20, 21→9, 10→21, 22→4, 5→22, 23→14, et là il faut faire un petit calcul pour voir que à ce moment-là on a en mémoire locale a qui vaut 0 et b qui vaut 6, donc on va ligne 16, 17→ 23, enfin 24.
2.

Ligne	Mémoire globale	Mémoire locale
4	a:3 ; x:2 ; y:1	x:1 ; y:2 ; z:3
20	a:3 ; x:2 ; y:3	
10	a:6 ; x:2 ; y:3	valeur:3
22	a:6 ; x:2 ; y:3	
14	a:6 ; x:-3 ; y:3	a:0 ; b:6
23	a:6 ; x:-3 ; y:3 ; c:6	

Corrigé de l'exercice ITC1.3 : Appels de fonctions avec des listes [* à **]

1. Avec `ajoute1`, la première liste passée en argument est modifiée. Par conséquent on aura à la fin [4,4,2,4],[3,2,-1,0],[4,4,2,4].
   Avec `ajoute2`, la première liste passée en argument n'est pas modifiée car la copie la préserve. Par conséquent on aura à la fin [1,2,3,4],[3,2,-1,0],[4,4,2,4].
   Avec `ajoute3`, la première liste passée en argument est modifiée. Par conséquent on aura à la fin [4,4,2,4],[3,2,-1,0],[4,4,2,4].
   Avec `ajoute4`, la première liste passée en argument n'est pas modifiée car on crée une nouvelle liste. Par conséquent on aura à la fin [1,2,3,4],[3,2,-1,0],[4,4,2,4].
2. Avec `ajoute2` et `ajoute4`, après la ligne `C=...`, on a A, B et C qui sont indépendants, donc D se calculera normalement et vaudra [5,6,5,8].
   Par contre, avec `ajoute1` et `ajoute3`, après la ligne `C=...`, on a A et C qui correspondent à la même liste, ce qui fait que l'appel de la fonction va traiter la même liste deux fois et modifier A donc aussi C ; à la fin A, C et D valent tous la même liste : [8,8,4,8].

Corrigé de l'exercice ITC1.4 : Extraction des nombres positifs d'une liste [**]

1. `for i in range(len(L)):` car la liste L2 est vide.
2. La seconde modifie directement la liste L donc elle a un effet de bord.
3. La ligne 3 modifie L1 en [1,3,3] ; la ligne 4 fait que L3 et L1 sont la même liste, donc la ligne 5 ajoute -3 à la fin de L1 et L3 ; L4 par contre est une copie de L2, donc la ligne 7 ne modifie que L2 ; La ligne 8 extrait [4] de L2 en la modifiant ; donc à la fin le programme affiche [1,3,3,-3],[4],[1,3,3,-3],[-1,-2,-3,-4,4],[4]

Corrigé de l'exercice ITC1.5 : Fonctions et listes [**]

1. La ligne 18 émet une erreur car `imin` était une variable locale de la fonction `enleve_minimum` donc n'est plus définie.
2. La fonction `somme` renvoie la somme des termes de la liste passée en argument ; elle n'a pas d'effet de bord.
   La fonction `enleve_minimum` enlève à la liste son plus petit terme (s'il y en a plusieurs, le premier d'entre eux). Elle a un effet de bord.
3. Mémoire globale : s:8 ; liste:[1,4,-2,6,3]
   Mémoire locale : L:[1,4,-2,6,-3] ; i:5 ; s:6
4. Itération $k = 0$: `total` vaut 6, donc on enlève le -3, puis on ajoute 2 à la fin de la liste ; la ligne 19 affiche [-1,6,-2,6,2]
   Itération $k = 1$: `total` vaut 11, donc on enlève le -2, puis on ajoute 8-11=-5 à la fin de la liste ; la ligne 19 affiche [-1,6,6,2,-5]
   Itération $k = 2$: `total` vaut 8, donc on enlève le -5, puis on ajoute 8-8=0 à la fin de la liste ; la ligne 19 affiche [-1,6,6,2,0]
   Itération $k = 3$: `total` vaut 13, donc on enlève le -1, puis on ajoute 8-13=-5 à la fin de la liste ; la ligne 19 affiche [6,6,2,0,-5]
   Itération $k = 4$: `total` vaut 9, donc on enlève le -5, puis on ajoute 8-9=-1 à la fin de la liste ; la ligne 19 affiche [6,6,2,0,-1]
   Enfin la ligne 20 affiche 8 car la variable globale s n'a jamais varié.

Corrigé de l'exercice ITC1.6 : Tests de fonctions [*]

1.
```
def teste_est_pair() -> bool:
    if est_pair(2)!=True:
        return False
    if est_pair(-1)!=False:
        return False
    return True
```

2.
```
def teste_tous_positifs() -> bool:
    if tous_positifs([1,3,2,0])!=True:
        return False
    if tous_positifs([1,-2,3,-4,5])!=False:
        return False
    return True
```

3.
```
def teste_partie_entiere() -> bool:
    if partie_entiere(2.4)!=2:
        return False
    if partie_entiere(-2.4)!=-3:
        return False
    return True
```

4.
```
def teste_position_lettre() -> bool:
    if position_lettre("Bonjour","o")!=1:
        return False
    if position_lettre("Bonjour","r")!=6:
        return False
    if position_lettre("Bonjour","z")!=-1:
        return False
    return True
```

5.
```
def teste_trie() -> bool:
    if trie(1,2,3)!=[1,2,3]:
```

```
        return False
    if trie(1,3,2)!=[1,2,3]:
        return False
    if trie(3,2,1)!=[1,2,3]:
        return False
    if trie(3,1,2)!=[1,2,3]:
        return False
    if trie(2,3,1)!=[1,2,3]:
        return False
    if trie(2,1,3)!=[1,2,3]:
        return False
    return True
```