Exercice 1. Corriger un bogue (6 points)

1. (a) Si l'on exécute l'instruction `mystere(3)`, aucune erreur n'est levée car `3` est bien du type `int` et la valeur renvoyée est $2 + 3 = 5$.

   (b) Si l'on exécute l'instruction `mystere(3.0)`, une erreur d'assertion est levée car `3.0` est du type `float`. Ainsi, aucune valeur n'est renvoyée.

   (c) On donne ci-dessous deux solutions équivalentes.

   — Proposition 1.

   ```
   def mystere(x):
       assert isinstance(x, (int, float))
       return 2 + x
   ```

   — Proposition 2.

   ```
   def mystere(x):
       assert isinstance(x, int) or isinstance(x, float)
       return 2 + x
   ```

2. (a) Cette fonction tente de calculer la somme des éléments du tableau `t` passé en paramètre à l'aide de la variable d'accumulation `var`.

   (b) Cette fonction est incorrecte en raison d'une erreur d'accès. En effet, prenons comme exemple le tableau `[10, 11, 12, 13]`. Pour ce tableau, `len(t)` vaut 4 donc la première ligne de la boucle `for` est `for i in range(1, 5)`. Les accès suivants vont donc être tentés `t[1]`, `t[2]`, `t[3]` et finalement `t[4]`. Ce dernier accès va déclencher une erreur d'accès car les rangs du tableau s'échelonnent entre `0` et `3`. De plus le rang `0` a été oublié. Il faut donc décaler les rangs de 1.

   (c) On donne ci-dessous deux solutions équivalentes.

   — Propositions 1.

   ```
   def mystere(t):
       var = 0
       for i in range(len(t)):
           var += t[i]
       return var
   ```

   — Propositions 1.

   ```
   def mystere(t):
       var = 0
       for i in range(1, len(t) + 1):
           var += t[i - 1]
       return var
   ```

3. (a) Ce programme est incorrect car la boucle `while` ne se termine pas. En effet, en raison de la représentation inexacte des nombres décimaux par les nombres flottants en Python, les ajouts successifs de `0.1` induisent des erreurs croissantes sur le résultats. Au bout de neuf tours, la variable `x` n'est pas exactement égale à `1.0`. La condition `x != 1.0` est donc toujours vérifiée et ainsi la boucle `while` ne se termine pas.

(b) On respecte le principe de programmation défensive en uttilisant l'opérateur de comparaison `<`. Ainsi, dès que le seuil de `1.0` est dépassé la boucle `while` se termine.

```
x = 0.1
while x < 1.0:
    print(x)
    x += 0.1
```

Exercice 2. Spécification et tests (4 points)

1.
```python
import pytest

def test_precondition():
    with pytest.raises(AssertionError):                          # test n°1
        partage(1)
    with pytest.raises(AssertionError):                          # test n°2
        partage('mot')

def jeu_de_test():
    assert partage([7, 5, 2, 1, 8, 3]) == ([7, 2, 8], [5, 1, 3]) # test n°3
    assert partage([7, 5, 2, 1, 8]) == ([7, 2, 8], [5, 1])       # test n°4
    assert partage([]) == ([], [])                               # test n°5
```

2. (a)
   — Tests n°1 et n°2.
   La fonction `partage` ne présente aucune assertion permettant de vérifier la validité de l'argument : est-ce une liste Python ? Les tests n°1 et n°2 vont donc échouer systématiquement.
   — Test n°4.
   La boucle `while` est conçue pour avancer dans les rangs de deux en deux. Cela n'est possible que si le tableau passé en argument a une longueur paire. Le test n°4 va donc échouer déclenchant une erreur d'accès : au premier tour de boucle on accède aux éléments `7` et `5`, au second à `2` et `1`. Le problème intervient au troisième tour de boucle lorsque le programme accède à `8` puis tente d'accéder à un élément suivant qui n'existe pas.
   — Par contre le test n°5 réussit car la longueur du tableau étant égale à `0` la boucle `while` n'est pas exécutée et les tableaux `pairs` et `impairs` restent vident comme attendus.

   (b) On propose ci-dessous deux solutions possibles.
   — Proposition 1. Avec une boucle `while`

```python
def partage(t):
    assert isinstance(t, list), 't doit être du type list !'
    pairs = []
    impairs = []
    i = 0
    while i < len(t):
        pairs.append(t[i])
        i += 1 # on avance seulement de 1 dans les rangs
        if i < len(t): # si le bout du tableau pas encore atteint
            impairs.append(t[i]) # + 1 enlevé en raison de i += 1
            i += 1 # on avance à nouveau de 1 dans les rangs
    return pairs, impairs
```

   — Proposition 2. Avec une boucle `for`

```python
def partage(t):
    assert isinstance(t, list), 't doit être du type list !'
    pairs = []
    impairs = []
    for i in range(len(t)):
        if i % 2 == 0: # si i est pair
            pairs.append(t[i])
        else: # si i est impair
            impairs.append(t[i])
    return pairs, impairs
```