Autotest n°2 – Correction

Exercice 1 : *Cet exercice porte sur la programmation objet, les structures de données, les réseaux et l'architecture matérielle.*
On considère un réseau local constitué des trois machines de Alice, Bob et Charlie dont les adresses IP sont les suivantes :

- la machine d'Alice a pour adresse `192.168.2.1` ;
- la machine de Bob a pour adresse `192.168.2.2`.

On rappelle que l'adresse `192.168.2.255` est l'adresse de diffusion qui sert à communiquer avec toutes les machines du réseau local et le masque de ce réseau local est `255.255.255.0`. Cette adresse de diffusion est réservée et ne peut être attribuée à une machine.

Partie A

1) Donner une adresse IP possible pour la machine de Charlie afin qu'elle puisse communiquer avec celles d'Alice et Bob dans le réseau local. Justifier votre réponse en donnant toutes les conditions à respecter dans le choix de cette adresse IP.
Solution : On peut prendre n'importe quelle adresse entre `192.168.2.3` et `192.168.2.254`. Il faut que les 3 premiers octets soient les mêmes et le dernier doit être entre 1 et 254 et ne pas avoir été donné.

Ce réseau est utilisé pour effectuer des transactions financières en monnaie numérique `nsicoin` entre les trois utilisateurs. Pour cela, on crée la classe `Transaction` ci-dessous :

```
class Transaction:
    def __init__(self, expediteur, destinataire, montant):
        self.expediteur = expediteur
        self.destinataire = destinataire
        self.montant = montant
```

2) Toutes les dix minutes, les transactions réalisées pendant cet intervalle de temps sont regroupées par ordre d'apparition dans une liste Python. Dans un intervalle de dix minutes, Alice envoie cinq `nsicoin` à Charlie puis Bob envoie dix `nsicoin` à Alice. Écrire la liste Python correspondante à ces transactions.
Solution : `[Transaction('Alice', 'Charlie', 5), Transaction('Bob', 'Alice', 10)]`

Pour garder une trace de toutes les transactions effectuées, on utilise une liste chaînée de blocs (ou *blockchain*) dont le code Python est fourni ci-dessous. Toutes les dix minutes un nouveau bloc contenant les nouvelles transactions est créé et ajouté à la blockchain.

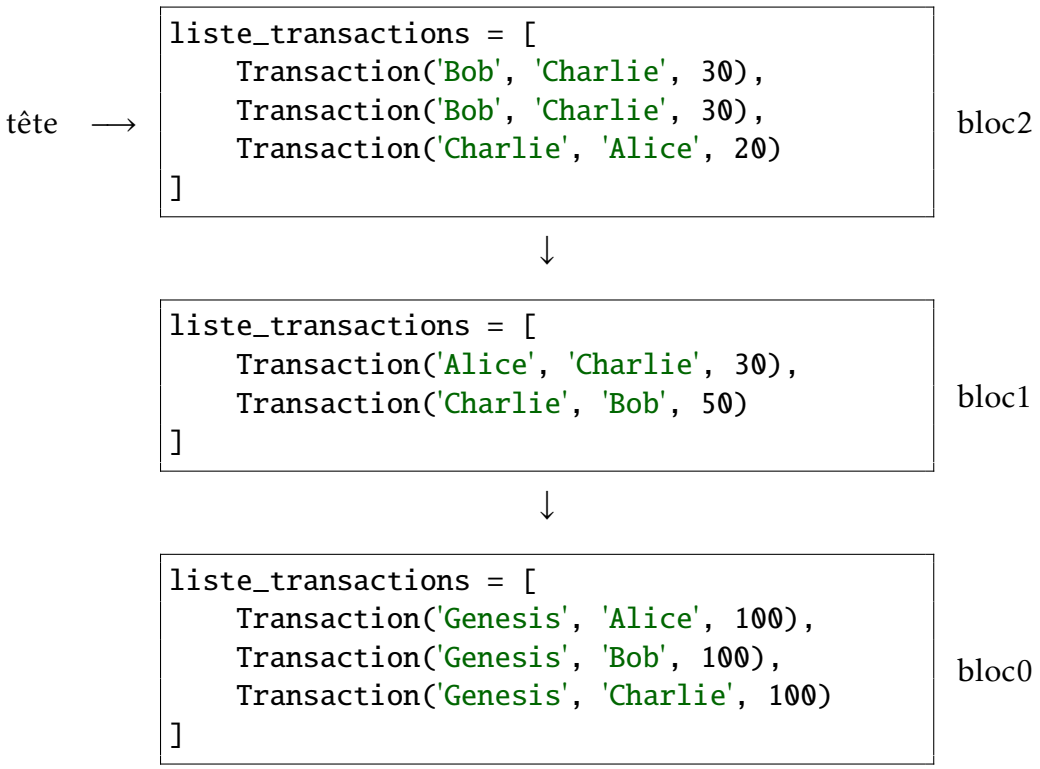
```
class Bloc:
    def __init__(self, liste_transactions, bloc_precedent):
        self.liste_transactions = liste_transactions
        self.bloc_precedent = bloc_precedent  # de type Bloc

class Blockchain:
    def __init__(self):
        self.tete = self.creer_bloc_0()

    def creer_bloc_0(self):
        """ Crée le premier bloc qui donne 100 nsicoin à chaque utilisateur
```

```
        (un pseudo-utilisateur Genesis est utilisé comme expéditeur) """
        liste_transactions = [Transaction("Genesis", "Alice", 100),
                              Transaction("Genesis", "Bob", 100),
                              Transaction("Genesis", "Charlie", 100)]
        return Bloc(liste_transactions, None)
```

3) La figure 1 représente les trois premiers blocs d'une Blockchain. Expliquer pourquoi la valeur de l'attribut `bloc_precedent` du bloc0 est **None**.



Solution : Puisque le bloc `bloc_precedent` est le premier créé, il n'y a pas de bloc précédent.

4) Donner la valeur de l'attribut `bloc_precedent` du bloc1 afin que celui-ci soit lié au bloc0.
Solution : Il faut que ce soit `Blockchain()`, ou `bloc0` s'il a été créé.

5) À l'aide des classes `Bloc` et `Blockchain`, écrire le code Python permettant de créer un objet `ma_blockchain` de type `Blockchain` représenté par la figure 1.

```
ma_blockchain = Blockchain()
transactions = [Transaction('Alice', 'Charlie', 30),
                Transaction('Charlie', 'Bob', 50)]
bloc1 = Bloc(transactions, ma_blockchain.tete)
ma_blockchain.tete = bloc1
transactions = [Transaction('Bob', 'Charlie', 30),
                Transaction('Bob', 'Charlie', 30),
                Transaction('Charlie', 'Alice', 20)]
bloc2 = Bloc(transactions, ma_blockchain.tete)
ma_blockchain.tete = bloc2
```

6) Donner le solde en `nsicoin` de Bob à l'issue du bloc2.
Solution : Le solde de Bob est de :

$$-30 - 30 + 50 + 100 = 90 \,\texttt{nsicoin}$$

7) On souhaite doter la classe `Blockchain` d'une méthode `ajouter_bloc` qui prend en paramètre la liste des transactions des dix dernières minutes et l'ajoute dans un nouveau bloc. Écrire le code Python de cette méthode ci-dessous.

```
def ajouter_bloc(self, liste_transactions):
    self.tete = Bloc(liste_transaction, self.tete)
```

8) Lorsqu'un utilisateur ajoute un nouveau bloc à la Blockchain, il l'envoie aux autres membres. Ainsi chaque utilisateur dispose sur sa propre machine d'une copie identique de la Blockchain. Donner le nom et la valeur de l'adresse IP à utiliser pour effectuer cet envoi.
Solution : Il faut l'envoyer à `192.168.2.255`.

9) On souhaite doter la classe `Bloc` d'une nouvelle méthode `calculer_solde` permettant de renvoyer le solde à l'issue de ce bloc. Recopier et compléter sur votre copie le code Python de cette méthode :

```
def calculer_solde(self, utilisateur):
    if self.bloc_precedent is None: # cas de base
        solde = 0
    else:
        # appel récursif : calcul du solde au bloc précédent
        solde = self.bloc_precedent.calculer_solde(utilisateur
    for transaction in self.liste_transactions:
        if utilisateur == transaction.expediteur:
            solde = solde - transaction.montant
        elif utilisateur == transaction.destinataire:
            solde = solde + transaction.montant
    return solde
```

10) En utilisant `calculer_solde`, écrire l'expression permettant de calculer le solde actuel de Alice dans `ma_blockchain`.
Solution : `ma_blockchain.tete.calculer_solde('Alice')`

Partie B

Dans cette partie, on va améliorer la sécurité de la blockchain. Pour cela on enrichit la classe `Bloc` comme indiqué ci-dessous :

```
class Bloc:
    def __init__(self, liste_transactions, bloc_precedent):
        self.liste_transactions = liste_transactions
        self.bloc_precedent = bloc_precedent
        # Définition de trois nouveaux attributs
        self.hash_bloc_precedent = self.donner_hash_precedent()
        self.nonce = 0 # Fixé arbitrairement et temporairement à 0
                       # avant le minage du bloc
        self.hash = self.calculer_hash()

    # Définition de trois nouvelles méthodes
    def donner_hash_precedent(self):
        if self.bloc_precedent is not None:
            return self.bloc_precedent.hash
        else :
            return "0"

    def calculer_hash(self):
        """calcule et renvoie le hash du bloc courant"""
        # Le code python n'est pas étudié dans cet exercice
```

```
    def minage_bloc(self):
        """modifie le nonce d'un bloc pour que son hash commence par '00' """
        # A compléter
```

La fonction `calculer_hash` produit une chaîne de caractères appelée `hash` qui possède les propriétés suivantes :

- Le hash d'un bloc dépend des valeurs de tous ses attributs, dont `nonce`, et uniquement de ces valeurs ;
- Le calcul du hash d'un bloc est rapide et facile à calculer par une machine ;
- La moindre modification dans le bloc produit un hash complètement différent ;
- Il est impossible de déduire le bloc à partir de son hash.
- Si deux blocs ont le même hash, c'est qu'ils sont parfaitement identiques.

11) L'attribut `nonce` est de type entier. Miner un bloc signifie trouver une valeur de nonce de telle façon que l'attribut `hash` du bloc commence par les deux caractères `"00"`. Compte tenu des propriétés précédentes, la seule façon de trouver cette valeur est de procéder à une recherche exhaustive. Expliquer en quoi consiste le fait de trouver une valeur par recherche exhaustive.
Solution : Cela revient à essayer toutes les valeurs une à une.

Dans la suite de l'exercice, on considère que tous les utilisateurs cherchent à miner le nouveau bloc. Le premier qui réussit, l'ajoute à la blockchain et gagne une récompense en `nsicoin`.

12) En justifiant votre réponse, donner la valeur de l'attribut `hash_bloc_precedent` du bloc0.

Solution : Ce sera `"0"` car le précédent est `None`.

13) Sachant que le `hash` est écrit sur 256 bits, donner le calcul permettant d'obtenir le nombre de hash possibles.
Il y a 2^{256} hash possibles.

14) Recopier et compléter sur votre copie le code python de la méthode `minage_bloc`.

```
def minage_bloc(self):
    """modifie le nonce d'un bloc pour que son hash commence par
    '00' en énumérant tous les entiers naturels en partant de 0."""
    self.nonce = 0
    self.hash = self.calculer_hash()
    while self.hash[0] != "0" or self.hash[1] != "0":
        self.nonce = self.nonce + 1
        self.hash = self.calculer_hash()
```

Exercice 2 : *Cet exercice porte sur la programmation Python, la programmation orientée objet et l'algorithmique.*
Une entreprise doit placer des antennes relais le long d'une rue rectiligne. Une antenne relais de portée (ou rayon) p couvre toutes les maisons qui sont à une distance inférieure ou égale à p de l'antenne.
Connaissant les positions des maisons dans la rue, l'objectif est de placer les antennes le long de la rue, pour que toutes les maisons soient couvertes, tout en en minimisant le nombre d'antennes utilisées.

La rue est représentée par un axe, et les maisons sont représentées des points sur cet axe :

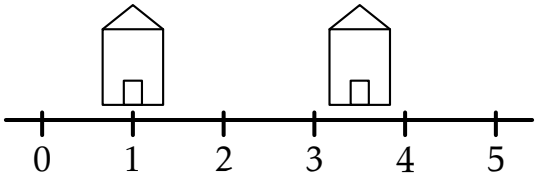


Figure 1. Deux maisons sur une rue, repérée par leur abscisse : 1 et 3,5

Les entités manipulées sont modélisées en utilisant la programmation orientée objet.

```
class Maison:
    def __init__(self, position):
        self._position = position

    def get_pos_maison(self):
        return self._position

class Antenne:
    def __init__(self, position, rayon):
        self._position = position
        self._rayon = rayon

    def get_pos_antenne(self):
        return self._position

    def get_rayon(self):
        return self._rayon
```

1) Donner le code qui crée et initialise deux variables `m1` et `m2` avec des instances de la classe Maison situées aux abscisses 1 et 3,5 (Figure 1).

   Solution :

   ```
   m1 = Maison(1)
   m2 = Maison(3.5)
   ```

On ajoute à présent une antenne ayant un rayon d'action de 1 à la position 2,5 :

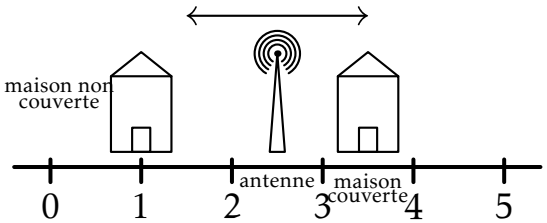


Figure 2. L'antenne placée en 2,5 et de rayon d'action 1 couvre la maison en 3,5 mais pas celle en 1

2) Donner le code qui crée la variable a correspondant à l'antenne à la position 2,5 avec le rayon d'action 1.

   Solution : `Antenne(2.5, 1)`

On souhaite modéliser une rue par une liste d'objets de type `Maison`. Cette liste sera construite à partir d'une autre liste contenant des nombres correspondant aux positions des maisons.

La fonction `creation_rue` réalise ce travail. Elle prend en paramètre une liste de positions et renvoie une liste d'objets de type `Maison`.

3) Recopier le schéma ci-dessous et le compléter pour donner une représentation graphique de la situation créée par :
`creation_rue([0, 2, 4, 5, 7, 9, 10.5, 11.5])`

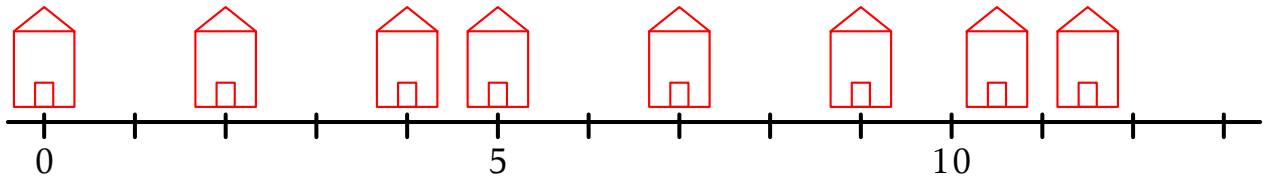


Figure 3. Axe pour représenter le problème avec 8 maisons

4) Compléter le code donné ci-dessous de la fonction `creation_rue`.

```
def creation_rue(pos):
    pos.sort()
    maisons = []
    for p in pos:
        m = Maison(p)
        maisons.append(m)
    return maisons
```

Pour rappel : la commande `tab.sort()` trie la liste `tab`. La méthode `couvre` de la classe `Antenne` prend en paramètre un objet de type `Maison` et indique par un booléen si l'antenne couvre la maison en question ou non. La méthode peut être utilisée ainsi (en supposant que les objets précédents `m1`, `m2` et `a` existent).

```
>>> a.couvre(m1)
False
>>> a.couvre(m2)
True
```

5) Compléter la fonction `couvre`, ci-dessous, en veillant à ne pas accéder directement aux attributs d'une maison depuis la classe `Antenne`. On pourra utiliser les méthodes `get_pos_maison`, `get_pos_antenne` et `get_rayon`.
Pour rappel : la fonction valeur absolue se nomme **`abs`** en Python.

```
# Méthode à ajouter dans la classe Antenne
def couvre(self, maison):
    pa = self.get_pos_antenne()
    pm = self.get_pos_maison()
    return abs(pa-pm) <= self.get_rayon()
```

La fonction `strategie_1` est donnée ci-dessous. L'objectif est de placer des antennes dans une rue. Elle est fournie à la société qui place les antennes. La fonction prend en paramètre une liste d'objets de type `Maison` (qu'on supposera triée par abscisse croissante) et le rayon d'action des antennes (**`float`**). Cette fonction renvoie une liste d'objets de type `Antenne` ayant ce rayon d'action et couvrant toutes les maisons de la rue.

```
def strategie_1(maisons, rayon):
    ''' Prend en paramètre une liste de maisons et le rayon
    d'action des antennes et renvoie une liste d'antennes'''
    antennes = [Antenne(maisons[0].get_pos_maison(), rayon)]
```

```
    for m in maisons[1:]:
        if not antennes[-1].couvre(m):
            antennes.append(Antenne(m.get_pos_maison(), rayon))
    return antennes
```

Pour rappel :

- `tab[1:]` correspond aux éléments de `tab` à partir de l'indice 1 jusqu'à la fin de la liste ;
- `tab[-1]` correspond au dernier élément de la liste `tab`.

6) Indiquer ce que renvoie cette suite d'instructions après exécution.

```
>>> maisons = creation_rue([0, 2, 4, 5, 7, 9, 10.5, 11.5])
>>> antennes = strategie_1(maisons, 2)
>>> print([a.get_pos_antenne() for a in antennes])
[0, 4, 7, 10.5]
```

Une amélioration est possible et la société qui pose les antennes souhaite implémenter l'algorithme suivant :

- considérer les maisons dans l'ordre des abscisses croissantes ;
- dès qu'une maison n'est pas couverte, placer une antenne à la plus grande abscisse telle qu'elle couvre cette maison. Par exemple, si la maison d'abscisse 5 est la première maison non couverte, alors, on placera l'antenne en $5 + r$ si r est le rayon d'action de l'antenne.

7) On considère la rue composée des maisons situées aux abscisses :
`[0, 2, 4, 5, 7, 9, 10.5, 11.5]`.
Recopier et compléter le schéma ci-dessous en indiquant l'emplacement des antennes selon cette nouvelle stratégie. On suppose que le rayon d'action est toujours 2.

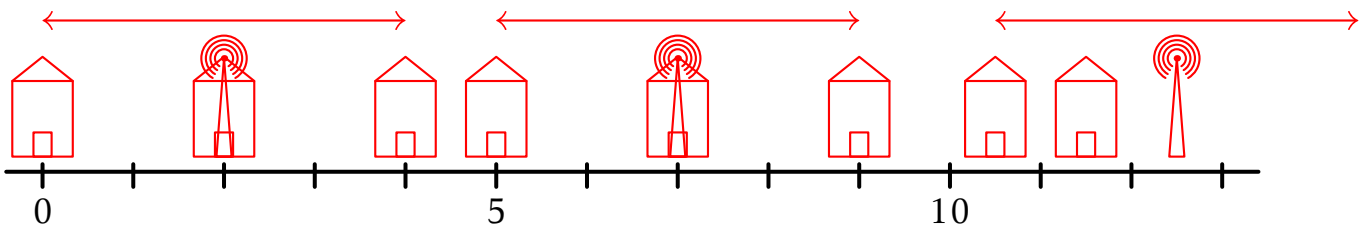


Figure 4. Axe pour représenter le résultat de la nouvelle stratégie

8) Cet algorithme étant a priori plus économe en antennes, proposer une fonction `strategie_2`, sur le modèle de `strategie_1` qui implémente cette nouvelle stratégie.

```
def strategie_2(maisons, rayon):
    antennes = [Antenne(maisons[0].get_pos_maison()+rayon, rayon)]
    for m in maisons[1:]:
        if not antennes[-1].couvre(m):
            antennes.append(Antenne(m.get_pos_maison()+rayon, rayon))
    return antennes
```

9) Comparer le coût en nombre d'opérations des deux stratégies en fonction du nombre `n` de maisons dans la rue. On admet que le coût de la fonction `append` est constant.
Solution : Les deux fonctions n'utilisent qu'une boucle simple parcourant toutes les maisons. Le coût est donc linéaire par rapport au nombre de maison.