Algorithmes et Pensée Computationnelle

Programmation orientée objet : Héritage et Polymorphisme - exercices avancés

Le but de cette séance est d'approfondir les notions de programmation orientée objet vues précédemment. Les exercices sont construits autour des concepts d'héritage, de surcharge d'opérateurs/méthodes et de polymorphisme. Au terme de cette séance, vous devez être en mesure de factoriser votre code afin de le rendre mieux structuré et plus lisible. Cette série d'exercices vous allez utiliser Python comme language de programmation.
Le code présenté dans les énoncés se trouve sur Moodle, dans le dossier **Ressources**.

1 Rappel : Surcharge des opérateurs - Python

Dans cette section, vous manipulerez des fractions sous forme d'objets. Vous ferez des opérations de base sur ce nouveau type d'objets.

Question 1: (🕓 *5 minutes*) **Création de classe**
Dans un projet que vous aurez au préalable préparé, créez un fichier appelé **surcharge.py**. À l'intérieur de ce fichier, créer une classe **Fraction** qui aura comme attributs privés un numérateur et un dénominateur.

Question 2: (🕓 *5 minutes*) **Constructeur par défaut**
Définir un constructeur à votre classe. Assignez des valeurs par défaut à vos attributs.
Si un seul argument est passé à votre constructeur, la fraction devra être égale à l'entier correspondant.
Empêchez l'utilisateur d'assigner la valeur **zéro** au dénominateur.

> **Conseil**
>
> Les valeurs par défaut seront assignées à votre objet au cas où il est instancié sans valeurs. Ainsi en faisant **f = Fraction()**, on obtiendra un objet **Fraction** ayant pour valeurs un numérateur à 0 et un dénominateur à 1 soit $\frac{0}{1}$.
> En Python, vous pouvez donner des valeurs par défaut aux arguments de vos méthodes lors de leur définition. Par exemple, vous pouvez faire : **def add(self, x=10, y=5):(...)**.

Question 3: (🕓 *5 minutes*) **Type casting**
Convertir les attributs en entier.

> **Conseil**
>
> Pensez à utiliser la fonction **int**.

Question 4: (🕓 *5 minutes*) **Redéfinition de méthodes**
Redéfinir la méthode **__str__()** pour produire une représentation textuelle de vos objets **Fraction**.

> **Conseil**
>
> Une fois la méthode **__str()__** redéfinie, lorsqu'on fera un **print()** sur une instance de votre classe **Fraction**, il affichera le message suivant : *Votre fraction a pour valeur* ***numérateur/dénominateur***. **numérateur** et **dénominateur** étant les valeurs que vous passerez à votre objet **Fraction**.

Question 5: (🕓 *5 minutes*) **Accesseurs et mutateurs**
Créer des **getters** et **setters** pour chacun des attributs de votre classe **Fraction**.

Question 6: (🕓 *10 minutes*) **Simplification de fractions**
Définir une méthode **simplification** qui réduit la **Fraction**. Cette méthode ne renverra rien, elle modifiera simplement l'instance. Pour la suite des exercices, assurez-vous de toujours manipuler des fractions simplifiées.

Pour ce faire, vous pouvez faire appel à votre méthode **simplification** après chaque opération sur un objet de type **Fraction**.

Conseil

Afin de simplifier une fraction, vous devez diviser le numérateur et le dénominateur par leur plus grand diviseur commun. Pensez à utiliser la méthode **math.gcd** pour trouver le plus grand diviseur commun entre deux nombres.

Question 7: (*15 minutes*) **Redéfinition de méthodes - __eq__**
Redéfinissez la méthode d'instance __eq__ qui prend en entrée un objet **Fraction** que vous nommerez *other* (en plus de **self**) et qui renvoie **True** si **self** et l'objet passé en argument ont la même valeur.

Conseil

Pour vérifier l'égalité entre a/b et c/d, tester que $a * d$ est égal à $b * c$.
Utilisez la méthode **isinstance()** afin de vérifier que *other* est bien de type **Fraction**. Dans le cas contraire, affichez un message d'erreur.
La fonction **isinstance** prend en entrée une valeur et un type. Elle vérifie que cette valeur est du type défini. Par exemple : **isinstance(nombre, int)** renverra **True** si la variable **nombre** est de type **int** et **False** dans le cas contraire.

Question 8: (*15 minutes*) **Addition et multiplication**
Redéfinissez les méthodes __add__ et __mul__ afin d'effectuer des opérations d'addition et de multiplication sur vos objets de type **Fraction**. Attention, ces méthodes devront renvoyer des objets de type **Fraction**. Dans vos méthodes __add__ et __mul__, n'oubliez pas de simplifier ces fractions avant de les retourner.
Gérer le cas où l'élément passé en argument n'est ni une **Fraction**, ni un **int**.

Solution

```python
import math


# Question 1: Création de la classe Fraction
class Fraction:
    # Question 2: Déclaration du constructeur et initialisation des valeurs
    def __init__(self, numerateur=0, denominateur=1):
        # Question 3: type casting
        self.__num = int(numerateur)
        self.__den = int(denominateur)
        self.simplification()

    # Question 4: redéfinition de la méthode __str__()
    def __str__(self):
        return "Votre fraction a pour valeur {}/{}".format(self.__num, self.__den)

    # Question 5: Getters et Setters
    def get_num(self):
        return self.__num

    def get_den(self):
        return self.__den

    def set_num(self, n):
        self.__num = n

    def set_den(self, d):
        d = int(d)
        # Si on passe 0 au dénominateur, on lève une exception ce qui arrêtera le programme
        if d == 0:
            raise ZeroDivisionError
        self.__den = d

    # Question 6
    def simplification(self):
        if self.__num == 0:
            self.__den = 1
        if self.__den < 0:
            self.__num = -self.__num
            self.__den = -self.__den
        pgcd = math.gcd(self.__num, self.__den)
        self.__num = int(self.__num / pgcd)
        self.__den = int(self.__den / pgcd)

    # Question 7
    def __eq__(self, f):
        if isinstance(f, Fraction):
            # vu que les fractions sont toujours en représentation simplifiée, on pourrait se contenter de
            # self.__numerateur == f.__numerateur and self.__denominateur = f.denominateur
            return self.__num * f.__den == f.__num * self.__den
        # Au cas où on re coit un seul argument, on créé une fraction ayant pour numérateur l'argument et 1 comme
          dénominateur
        elif isinstance(f, int):
            return self.__eq__(Fraction(f))
        else:
            return False
```

Solution

```
    # Question 8
    def add(self, f):
        self.__num = self.__num * f.__den + f.__num * self.__den
        self.__den = self.__den * f.__den
        self.simplification()

    def plus(self, f):
        q = Fraction(self.__num, self.__den)
        q.add(f)
        return q

    def __add__(self, other):
        if isinstance(other, Fraction):
            return self.plus(other)
        elif isinstance(other, int):
            self.add__ = self.__add__(Fraction(other))
            return self.add__
        else:
            raise TypeError(
                "Unsupported operand types for +: '" + self.__class__.__name__ + "' and '" + other.__class__.__name__ +
            "'")

    def __mul__(self, other):
        if isinstance(other, Fraction):
            return Fraction(self.__num * other.__num, self.__den * other.__den)
        elif isinstance(other, int):
            return Fraction(self.__num * other, self.__den)
        # On affiche un message d'erreur lorsque other n'est pas une Fraction
        else:
            raise TypeError(
                "Unsupported operand types for *: '" + self.__class__.__name__ + "' and '" + other.__class__.__name__ +
            "'")

if __name__ == '__main__':
    f1 = Fraction()
    print(f1)
    f1 = Fraction(4)
    print(f1)
    f1 = Fraction(denominateur=5)
    print(f1)
    f1 = Fraction(4, -6)
    print(f1)
    f2 = Fraction(2, -8)
    print(f2)
    print(f1+f2)
    print(f1 == Fraction(22, -24))
    print(f1 == Fraction(1, 2))
```

2 Héritage en Python

Question 9: (*15 minutes*) **Classe Point (Suite)**

Dans la série dernière, vous avez rencontré un exemple de classe en Python. Celui-là représente un point de 2 dimensions, x et y, ainsi que des opérations basiques sur des points 2D. Pour cet exercice, vous allez implémenter une classe des points de 3 dimensions en utilisant de l'héritage sur la classe **Point** qu'on a implémentée !

Avant de commencer, nous voudrions attirer votre attention sur les points suivants de la classe mère **Point** :

- Nous avons changé le nom de la méthode **distance()** en **distance_euclidean()** pour la distinguer des autres types de distance.
- Traiter la classe **Point** comme une classe mère et la faire hériter de la classe **Point3D** n'est pas la meilleure structure d'un programme Python, mais on la garde pour le moment afin de vous montrer comment les méthodes de classe mère peuvent être manipulées dans une classe fille.

Voici la classe **Point** qui a été légèrement modifiée (Vous trouverez le fichier sur Moodle, dans le dossier **Ressources**) :

```
import math

class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def get_x(self):
        return self._x

    def get_y(self):
        return self._y

    def set_x(self, x):
        self._x = x

    def set_y(self, y):
        self._y = y

    def distance_euclidean(self, p2):
        return math.sqrt((self._x - p2.get_x()) ** 2 + (self._y - p2.get_y()) ** 2)

    def milieu(self, p2):
        x_M = (self._x + p2.get_x()) / 2
        y_M = (self._y + p2.get_y()) / 2
        M = Point(x_M, y_M)
        return M

    def __str__(self):
        return "Les coordonnées du point sont: x=" + str(self.get_x()) + ", y=" + str(self.get_y())
```

Écrivez une classe qui hérite de **Point**. Nommez-la **Point3D**. Après avoir rajouté la 3ème dimension comme attribut, implémentez les opérations ci-dessous :

- Rajoutez une méthode qui renvoie une représentation vectorielle du point. Vous pouvez utiliser la list en Python.
- Recalculez la distance euclidienne et le milieu pour le point 3D.
- Pour aller plus loin Si vous voulez vous familiariser encore plus avec les méthodes de classe en Python, implémentez deux autres calculs de distance : Manhattan et Minkowski.

```
class Point3D(Point):
    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.__z = z

    def get_z(self):
        ...

    def set_z(self, z):
        ...

    def vector_representation(self): # représentée sous forme de liste
        ...
```

```
15      def distance_euclidean(self, p2): # i.e norme
16          ...
17
18      def distance_manhattan(self, p2):
19          ...
20
21      def distance_minkowski(self, p2, order=3):
22          ...
23
24      def milieu(self, p2):
25          ...
```

Conseil

Que fait `super().__init__()` ?
Dans un espace à 3 dimensions, la formule pour calculer la distance entre un $p_1 = (x_1, y_1, z_1)$ et $p_2 = (x_2, y_2, z_2)$ est $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (y_1 - y_2)^2}$.

Solution

```
class Point3D(Point):
    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.__z = z

    def get_z(self):
        return self.__z

    def set_z(self, z):
        self.__z = z

    def vector_representation(self): # représentée sous forme de liste
        return [self.__x, self.__y, self.__z]

    def distance_euclidean(self, p2): # i.e norme
        other_x = p2.get_x()
        other_y = p2.get_y()
        other_z = p2.get_z()
        return math.sqrt((self.__x - other_x)**2 + (self.__y - other_y)**2 + (self.__z - other_z)**2)

    def distance_manhattan(self, p2):
        other_x = p2.get_x()
        other_y = p2.get_y()
        other_z = p2.get_z()
        return sum((abs(self.__x - other_x), abs()), abs(self.__y - other_y), abs(self.__z - other_z))

    def distance_minkowski(self, p2, order=3):
        other_x = p2.get_x()
        other_y = p2.get_y()
        other_z = p2.get_z()
        return sum((abs(self.__x - other_x)**order), abs(self.__y - other_y)**order,\
                abs(self.__z - other_z)**order)**(1/order)

    def milieu(self, p2):
        other_x = p2.get_x()
        other_y = p2.get_y()
        other_z = p2.get_z()

        x_M = (self.__x + other_x)/2
        y_M = (self.__y + other_y)/2
        z_M = (self.__z + other_z)/2
        return Point3D(x_M, y_M, z_M) # renvoie un point!


point1 = Point3D(1, 2, 3)
point2 = Point3D(3, 4, 5)

# exemple
point1.vector_representation()
```

Question 10: (*15 minutes*) **Un exemple appliqué**

Dans les établissements universitaires, on rencontre souvent des problèmes lors du calcul de salaires du personnel. Sans penser aux recherches effectuées par certains professeurs, on va essayer de calculer les salaires de ceux qui sont reconnus comme 'Professeur' (ordinaire, titulaire, associé ou assistant) à l'université et ceux qui y donnent des cours à temps partiel (on va les considérer comme 'Collaborateurs' dans cet exercice).

La classe mère dans ce cas est nommée **Enseignant**, qui possède une propriété - le salaire annuel moyen. On voudrait que la méthode qui calcule cette quantité renvoie 60 000 (dollars américains) si l'enseignant a moins de 10 ans d'expérience, et 100 000 sinon. Si l'enseignant travaille à temps partiel, la méthode devrait renvoyer une chaîne qui dit 'Le salaire annuel ne s'applique pas aux collaborateurs'.

Ensuite, on veut calculer la paye mensuelle pour chaque type d'employé. Pour les **Professeurs**, la paye devrait être calculée sur la base de deux sources de revenu : un salaire mensuel et une commission pour chaque comité où ils participent.

D'autre part, pour les **Collaborateurs**, la paye est calculée sur une base horaire i.e *taux horaire* $\times$ *nombres*

d'heures de travail (par mois).
Complétez le code ci-dessous :

```
class Enseignant:
    def __init__(self, name, years_experience, full_time):
        ...
    def salaire_annuel_moyen(self):
        ...

class Professeur(Enseignant):
    def __init__(self, name, years_experience, monthly_salary, commission, num_committees):
        ...
    def paye_mensuelle(self):
        ...

class Collaborateur(Lecturer):
    def __init__(self, name, years_experience, hours_per_month, rate):
        ...
    def paye_mensuelle(self):
        ...

prof1 = Professeur("Alexandra", 8, 3000, 200, 4)
prof2 = Collaborateur("David", 10, 40, 30)
# exemples
print(prof1.salaire_annuel_moyen())
print(prof2.paye_mensuelle())
```

Conseil

Pensez à redéfinir les attributs de la classe mère en utilisant `super().__init__()`.

Solution

```
class Enseignant:
    def __init__(self, name, years_experience, full_time):
        self.name = name
        self.years_experience = years_experience
        self.full_time = full_time

    def salaire_annuel_moyen(self):
        if self.full_time:
            if self.years_experience < 10:
                return 60000

            else:
                return 100000

        else:
            return "Le salaire annuel ne s'applique pas aux collaborateurs"


class Professeur(Enseignant):
    def __init__(self, name, years_experience, monthly_salary, commission, num_committees):
        super().__init__(name, years_experience, True)
        self.monthly_salary = monthly_salary
        self.commission = commission
        self.num_committees = num_committees

    def paye_mensuelle(self):
        return self.monthly_salary + self.commission*self.num_committees

class Collaborateur(Enseignant):
    def __init__(self, name, years_experience, hours_per_month, rate):
        super().__init__(name, years_experience, False)
        self.hours_per_month = hours_per_month
        self.rate = rate

    def paye_mensuelle(self):
        return self.hours_per_month*self.rate

prof1 = Professeur("Alexandra", 8, 3000, 200, 4)
prof2 = Collaborateur("David", 10, 40, 30)

# exemples
print(prof1.salaire_annuel_moyen())
print(prof2.paye_mensuelle())
```