TD2 - Encapsulation & Héritage

Imene Kerboua

2022/2023

1 Rappel du cours

1.1 Encapsulation

L'encapsulation permet de définir la visibilités des attributs et méthodes d'un objet.

```python
class Vehicule: # definition de la classe
    def __init__(self, marque: str): # constructeur
        self.marque = marque # attribut public
        self.__vitesse = 0 # attribut prive

    @property
    def vitesse(self):
        return self.__vitesse

    @vitesse.setter
    def vitesse(self, value: float):
        if value < 0:
            raise ValueError("La valeur de la vitesse doit tre > 0.")
        else:
            self.__vitesse = value

    def accelerer(self, delta_vitesse: float):
        self.__vitesse += delta_vitesse

    def decelerer(self, delta_vitesse: float):
        self.__vitesse -= delta_vitesse


if __name__ == "__main__":
    v = Vehicule(marque="BMW")
    print(f"Marque {v.marque} et vitesse {v.vitesse}")
    # out => Marque BMW et vitesse 0

    v.accelerer(80)
    print(v.vitesse)
    # out => 80

    v.decelerer(20)
    print(v.vitesse)
    # out => 60
```

1.2 Héritage

La classe `Voiture` hérite des propriétés et comportements de la classe mère `Vehicule`.

```python
class Vehicule: # definition de la classe
    def __init__(self, marque: str): # constructeur
        self.marque = marque  # attribut public
        self._vitesse = 0 # attribut portege

    @property
    def vitesse(self) -> float:
        return self._vitesse

    @vitesse.setter
    def vitesse(self, value: float):
        if value < 0:
            raise ValueError("La valeur de la vitesse doit etre > 0.")
        else:
            self._vitesse = value

    def accelerer(self, delta_vitesse: float):
        self._vitesse += delta_vitesse

    def decelerer(self, delta_vitesse: float):
        self._vitesse -= delta_vitesse

class Voiture(Vehicule):
    def __init__(self, marque: str, couleur: str):
        super().__init__(marque)
        self.__couleur = couleur # attribut prive

    @property
    def couleur(self) -> str:
        return self.__couleur

    @vitesse.couleur
    def couleur(self, value: str):
        self.__couleur = value


if __name__ == "__main__":
    v = Voiture(marque="BMW", couleur="rouge")
    print(f"Marque {v.marque}, couleur {v.couleur} et vitesse {v.vitesse}")
    # out => Marque BMW, couleur rouge et vitesse 0

    v.accelerer(80)
    print(v.vitesse)
    # out => 80

    v.decelerer(20)
    print(v.vitesse)
    # out => 60
```

1.3 Fonctions et attributs utiles

- `__class__`: retourne la classe de l'object instancié.
- `__class__.__name__`: retourne le nom de la classe de l'object instancié
- `isinstance(object, class_name)`: retourne `True` si l'objet est une instance de `class_name`.

- `issubclass(class_name_1, class_name_2)`: retourne `True` si la classe `class_name_1` hérite de la classe `class_name_2`.

2 Application

Questions

Q1 - Exécuter le code suivant et expliquer les résultats obtenus :

```
class A:
    def __init__(self, attribute_1: str, attribute_2: str) -> None:
        self.__attribute_1 = attribute_1
        self.__attribute_2 = attribute_2

    @property
    def attribute_1(self) -> str:
        return self.__attribute_1

    @attribute_1.setter
    def attribute_1(self, value: str):
        self.__attribute_1 = value


if __name__ == "__main__":
    obj = A("a", "b")

    print(obj.attribute_1)
    obj.attribute_1 = "c"
    print(obj.attribute_1)
    print(obj._A__attribute_2)
    print(obj.__attribute_2)
```

Q2 - Exécuter le code suivant et expliquer les résultats affichés :

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

d = B()
print(isinstance(d, A))
print(isinstance(d, B))
print(isinstance(d, C))
print(isinstance(d, (B, C)))
print(issubclass(A, B))
print(issubclass(B, A))
```

Q3 - Executer le code suivant, que remarquez-vous ?

```
class Shape:
    def __init__(self, name: str) -> None:
        self.name = name

    def __str__(self) -> str:
        return self.name

class Circle(Shape):
    def __init__(self, name: str, radius: float) -> None:
        super().__init__(name)
        self.radius = radius

    def __str__(self) -> str:
        return super().__str__() + f"{self.radius}"

    def get_area(self):
        return 3.14 * self.radius ** 2

class Square(Shape):
    def __init__(self, name: str, side: float) -> None:
        super().__init__(name)
        self.side = side

    def __str__(self) -> str:
        return super().__str__() + f"{self.side}"

    def get_area(self):
        return self.side ** 2

if __name__ == "__main__":
    li = [Square(name="square 1", side=5), Circle(name="circle 1", radius=2), Square(name="square
        2", side=2)]
    for i in li : print(i)

    li_areas = [obj.get_area() for obj in li]
    print(li_areas)
```

Exercice 1

Créer la classe `Cylinder` qui hérite de la classe `Circle` de l'exercice 1 du TD1. Elle aura comme attribut additionnel la hauteur (`height`) et une méthode pour calculer son volume (`get_volume()`). Créer la méthode adaptée pour afficher l'objet `Cylinder`.

- Créer la classe `Cylinder`, ses attributs et ses méthodes, avec les *getters* et *setters* adéquats. Penser à appliquer l'encapsulation à la classe `Circle` aussi.
- Créer un objet `Cylinder` avec un rayon = 3 et une hauteur = 5.
- Afficher l'objet créé.
- Calculer et afficher son volume.

Exercice 2 : Ville

- Ecrire une classe Ville où une ville est définie par son nom et son nombre d'habitants.
- Implémenter les *getters*, *setters* et `__str__()` pour permettre l'affichage des objets `Ville`.

- A partir de cette classe, dérivez une classe `Capitale` où l'on mémorise en plus le nom des monuments qu'elle abrite.
- Implémenter les *getters*, *setters* et `__str__()` pour permettre l'affichage des objets `Capitale`.

Exercice 3 : Géométrie

Implémenter la classe `Parallelepipede` :

- Prévoir un constructeur avec des valeurs par défaut pour les longueur, largeur et hauteur.
- Affectez le nom de cette figure (parallélépipède) comme attribut d'instance ; définissez la méthode volume() qui retourne le volume d'un parallélépipède rectangle.
- Munir la classe d'une méthode de représentation permettant que l'affichage d'une instance de `Parallelepipede` produise : Le parallélépipède de côtés 12, 8 et 10 a un volume de 960.

Testez votre classe, puis ajoutez une classe `Cube` qui hérite de la classe `Parallelepipede`. Son constructeur aura une valeur d'arête par défaut et, dans sa définition :

- Appellera le constructeur de la classe `Parallelepipede`.
- Surchargera son propre nom (attribut d'instance) : 'cube'.
- L'affichage d'un objet Cube de côté 10 doit produire le message : Le cube de côtés 10, 10 et 10 a un volume de 1000.

Exercice 4 : Pokémon - Devoir :)

Les Pokémon sont certes de très mignonnes créatures, mais ils sont également un bon exemple pour illustrer l'héritage. Je vous propose donc de commencer par créer une classe `Pokemon` qui contient (entre autres) :

- Un attribut `nom` qui représente le nom du Pokémon.
- Un attribut `hp` (pour Health Points) qui représente les points de vie du Pokémon.
- Un attribut qui s'appelle `atk` qui représente la force de base de l'attaque du Pokémon.
- Un constructeur pour instancier des Pokémon adéquatement.
- Des getters (accesseurs) qui permettent de consulter les attributs du Pokémon.
- Une méthode `is_dead()` qui retourne un `bool` pour indiquer si un Pokémon est mort (`hp == 0`) ou non.
- Une méthode `attaquer(p: Pokemon)` qui permet au Pokémon appelant d'attaquer le Pokémon passé en paramètre. L'attaque déduit `atk` points de la vie `hp` du Pokémon attaqué `p`.
- Une redéfinition de la méthode `__str__()` qui affiche les informations du Pokémon.

En plus des Pokémon normaux (décrits à travers la classe `Pokemon`) on ressence trois types de Pokémon. Les Pokémon de type Feu, les Pokémon de type Eau et les Pokémon de type Plante (en réalité il existe 17 types en tout mais on ne va pas s'amuser à tous les coder) :

- Les Pokémon de type Feu sont super efficaces contre les Pokémon de type Plante et leur infligent deux fois plus de dégâts (`2*atk`). Par contre, ils sont très peu efficaces contre les Pokémon de type Eau ou de type Feu et ne leur infligent que la moitié des dégâts (`0.5*atk`). Ils infligent des dégâts normaux aux Pokémon de type Normal.

- Les Pokémon de type Eau sont super efficaces contre les Pokémon de type Feu et leur infligent deux fois plus de dégâts (`2*atk`). Par contre, ils sont très peu efficaces contre les Pokémon de type Eau ou de type Plante et ne leur infligent que la moitié des dégâts (`0.5*atk`). Ils infligent des dégâts normaux aux Pokémon de type Normal.
- Enfin, les Pokémon de type Plante sont super efficaces contre les Pokémon de type Eau et leur infligent deux fois plus de dégâts (`2*atk`). Par contre, ils sont très peu efficaces contre les Pokémon de type Plante ou de type Feu et ne leur infligent que la moitié des dégâts (`0.5*atk`). Ils infligent des dégâts normaux aux Pokémon de type Normal.

Créez trois classes `PokemonFeu`, `PokemonEau` et `PokemonPlante` qui héritent de la classe `Pokemon` et qui représentent les trois types de Pokémon susmentionnés. Ensuite, amusez-vous à faire des combats de Pokémon.

- Quel principe de la POO permet d'utiliser la méthode `attaquer()` adéquate à chaque objet qui lui fait appel ?