Héritage, Polymorphisme et Surcharge en Python

Anis SAIED

15 octobre 2023

Héritage de classes

- L'héritage est une caractéristique fondamentale de la programmation orientée objet (POO).
- Permet de créer de nouvelles classes basées sur des classes existantes.
- Favorise la réutilisation du code et la structuration des classes en hiérarchies.
- En Python, l'héritage s'effectue en spécifiant la classe parente entre parenthèses lors de la définition de la classe enfant.

```
1 class Parent:
2     # Attributs et méthodes de la classe parente
3
4 class Enfant(Parent):
5     # Attributs et méthodes spécifiques à la classe enfant
```

Exemple d'Héritage en Python I

Considérons une classe parente appelée `Véhicule` avec les attributs `marque` et `année`. La classe enfant appelée `Voiture` hérite la classe `Véhicule` et ajoute un attribut `modèle`.

```
1  class Véhicule: # classe de base
2      def __init__(self, marque, année):
3          self.marque = marque
4          self.année = année
5
6  class Voiture(Véhicule): # classe dérivée
7      def __init__(self, marque, année, modèle):
8          # Reutilisation de code
9          super().__init__(marque, année)
10         # Ajouter des attributs spécifiques
11         self.modèle = modèle
```

Exemple d'Héritage en Python II

```
12 # Créez une instance de Voiture et affichez ses attributs
13 voiture1 = Voiture("Toyota", 2022, "Camry")
14 print("Marque:", voiture1.marque)
15 print("Année:", voiture1.année)
16 print("Modèle:", voiture1.modèle)
```

super()

- est une fonction spéciale en Python qui est utilisée pour appeler une méthode de la classe parente (ou classe de base) à l'intérieur de la classe enfant (ou classe dérivée).
- Utilisée principalement dans le contexte de l'héritage lorsque vous souhaitez appeler le constructeur de la classe parente à partir de la classe enfant.
- permet d'initialiser les attributs hérités de la classe parente

Exercice 1

1. Définissez une classe `Forme` avec une méthode `aire` qui renvoie 0.
2. Définissez la classe `Point`, qui est une sous-classe de `Forme`. Ajoutez des attributs `x` et `y` pour représenter les coordonnées d'un point et une méthode `afficher` qui affiche les coordonnées.
3. Créez une classe `Cercle`, qui est une sous-classe de `Point`. Ajoutez un attribut `rayon` pour représenter le rayon du cercle et une méthode `aire` pour calculer l'aire du cercle.
4. Créez une classe `Rectangle`, qui est également une sous-classe de `Point`. Ajoutez des attributs `largeur` et `hauteur` pour représenter les dimensions du rectangle et une méthode `aire` pour calculer l'aire du rectangle.
5. Créez un objet de chaque classe (Point, Cercle et Rectangle) et utilisez les méthodes `aire` et `afficher` pour afficher leurs aires respectives et leurs coordonnées.

Solution I

```
1  class Forme:
2      def aire(self):
3          return 0 # valeur par défaut
4
5  class Point(Forme):
6      def __init__(self, x, y):
7          self.x = x
8          self.y = y
9
10     def afficher(self):
11         print("Point : ({}, {})".format(self.x, self.y))
12
13
14
```

Solution II

```
15 import math
16 class Cercle(Point):
17     def __init__(self, x, y, rayon):
18         super().__init__(x, y)
19         self.rayon = rayon
20     def aire(self):
21         return math.pi * self.rayon ** 2
22 class Rectangle(Point):
23     def __init__(self, x, y, largeur, hauteur):
24         super().__init__(x, y)
25         self.largeur = largeur
26         self.hauteur = hauteur
27     def aire(self):
28         return self.largeur * self.hauteur
```

Solution III

```
29 point = Point(2, 3)
30 cercle = Cercle(0, 0, 5)
31 rectangle = Rectangle(1, 1, 4, 3)
32
33 point.afficher()
34 print("Aire du point : {:.2f}".format(point.aire()))
35
36 cercle.afficher()
37 print("Aire du cercle : {:.2f}".format(cercle.aire()))
38
39 rectangle.afficher()
40 print("Aire du rectangle : {:.2f}".format(rectangle.aire()))
41
```

Relations "Est-un" (Héritage) et "A-un" (Composition)

En programmation orientée objet (POO), deux concepts clés pour modéliser les relations entre objets sont l'**héritage** et la **composition** :

- L'héritage permet de créer des classes dérivées (enfants) à partir de classes de base (parents), établissant ainsi une relation "est-un" où la classe dérivée est un type spécialisé de la classe de base.
  - Exemple : Une "Voiture" est un type de "Véhicule".

- La relation "a-un" est implémentée via la composition, qui consiste à inclure des objets d'une classe dans une autre, où la classe contenant l'objet est responsable de sa création et de sa gestion.
  - Exemple : Une "Voiture" a un "Moteur".

Syntaxe : Relations "Est-un" et "A-un"

Héritage (Relation "Est-un") :

```
class ClasseParent:
    # Attributs et méthodes de la classe parente
class ClasseEnfant(ClasseParent):
    # Attributs et méthodes spécifiques à la classe enfant
```

Composition (Relation "A-un") :

```
class ObjetContenu:
    def __init__(self, nom):
        self.nom = nom
    # Autres attributs et méthodes de la classe ObjetContenu

class ClasseContenant:
    def __init__(self):
        self.objet = ObjetContenu("nom_de_l_objet")
    # Autres attributs et méthodes de la classe ClasseContenant
```

Exemple : Relations "Est-un" et "A-un"

```python
class Vehicule:
    def __init__(self, nom):
        self.nom = nom
class Moteur:
    def __init__(self, type):
        self.type = type
class Voiture(Vehicule): # Héritage (est-un)
    def __init__(self, nom, moteur):
        super().__init__(nom)
        self.moteur = moteur # Composition (a-un)

    def afficher_info(self):
        print(self.nom+"a un moteur de type"+self.moteur.type)

moteur_voiture = Moteur("Essence")
ma_voiture = Voiture("Sedan", moteur_voiture)
ma_voiture.afficher_info()
```

Exercice 2

1. Créez une nouvelle classe "MoteurElectrique" avec un attribut "batterie" pour représenter le type de batterie (par exemple, "lithium-ion").
2. Ensuite, modifiez la classe "Voiture" pour inclure un attribut "moteur_electrique" qui sera une instance de la classe "MoteurElectrique."
3. Créez une instance de "Voiture" avec un moteur électrique et accédez à l'attribut "batterie" de la classe "MoteurElectrique."

Solution I

```python
class MoteurElectrique:
    def __init__(self, batterie):
        self.batterie = batterie
class Voiture:
    def __init__(self, marque, annee, moteur_electrique):
        self.marque = marque
        self.annee = annee
        self.moteur_electrique = moteur_electrique
# Création d'instances
moteur_elect = MoteurElectrique("Lithium-ion")
voiture_electrique = Voiture("Tesla", 2023, moteur_elect)
# Accès à l'attribut "batterie" de la classe MoteurElectrique
# via la classe Voiture
# Affiche "Lithium-ion"
print(voiture_electrique.moteur_electrique.batterie)
```

Exercice 3 :

Créez une hiérarchie de classes pour gérer une bibliothèque et définir les méthodes appropriées pour accéder et manipuler les attributs des classes.

1. Créez une classe de base appelée "Livre" avec les attributs suivants : titre, auteur, année de publication.
2. Créez deux sous-classes : "LivrePapier" et "EBook".
3. La classe "LivrePapier" devrait avoir un attribut supplémentaire pour le nombre de pages.
4. La classe "EBook" devrait avoir un attribut supplémentaire pour la taille du fichier en mégaoctets.
5. Créez une classe "Bibliotheque" pour stocker une liste de livres.
6. Ajoutez des méthodes à la classe "Bibliotheque" pour ajouter un livre, supprimer un livre et afficher la liste des livres disponibles.
7. Créez quelques instances de livres (à la fois "LivrePapier" et "EBook") et ajoutez-les à la bibliothèque.
8. Affichez la liste des livres disponibles dans la bibliothèque.

Solution I

```
1  class Livre:
2      def __init__(self, titre, auteur, annee):
3          self.titre = titre
4          self.auteur = auteur
5          self.annee = annee
6
7  class LivrePapier(Livre):
8      def __init__(self, titre, auteur, annee, nombre_pages):
9          super().__init__(titre, auteur, annee)
10         self.nombre_pages = nombre_pages
11
12     def description(self):
13         return "{} ({}), par {} - {} pages"
14                 .format(self.titre, self.annee, self.auteur,
15                         self.nombre_pages)
16
```

Solution II

```
17
18 class EBook(Livre):
19     def __init__(self, titre, auteur, annee, taille_fichier):
20         super().__init__(titre, auteur, annee)
21         self.taille_fichier = taille_fichier
22
23     def description(self):
24         return "{} ({}), par {} - {} Mo"
25                 .format(self.titre, self.annee, self.auteur,
26                         self.taille_fichier)
27
28 class Bibliotheque:
29     def __init__(self):
30         self.collection = []
31     def ajouter_livre(self, livre):
32         self.collection.append(livre)
```

Solution III

```
33
34      def supprimer_livre(self, titre):
35          self.collection = [livre for livre in self.collection
36                             if livre.titre != titre]
37
38      def afficher_livres(self):
39          for livre in self.collection:
40              print(livre.description())
41
42  # Exemple d'utilisation
43  biblio = Bibliotheque()
44  livre1 = LivrePapier("Harry Potter", "J.K. Rowling", 1997, 350)
45  livre2 = EBook("Python for Beginners", "John Smith", 2021, 5)
46  biblio.ajouter_livre(livre1)
47  biblio.ajouter_livre(livre2)
48  biblio.afficher_livres()
```

Polymorphisme

- Le polymorphisme permet de traiter différents objets de manière uniforme, simplifiant ainsi le code.
- Il simplifie le code en permettant d'utiliser des méthodes ou des opérateurs sur des objets de différentes classes.
- Cela repose sur le fait que des objets de classes différentes peuvent répondre de manière cohérente aux mêmes méthodes ou opérations.
- Le polymorphisme en Python s'exprime généralement à travers les méthodes.

Polymorphisme : Syntaxe

```
1  class ClasseParente:
2      def methode(self):
3          pass # Traitement générique
4
5  class ClasseEnfant1(ClasseParente):
6      def methode(self):
7          # Implémentation spécifique à la ClasseEnfant1
8
9  class ClasseEnfant2(ClasseParente):
10     def methode(self):
11         # Implémentation spécifique à la ClasseEnfant2
```

Polymorphisme : Exemple

```
1  class Animal:
2      def faire_son_cri(self):
3          pass # Traitement générique
4
5  class Chien(Animal):
6      def faire_son_cri(self):
7          return "Le chien aboie"
8
9  class Chat(Animal):
10     def faire_son_cri(self):
11         return "Le chat miaule"
12
13 animaux = [Chien(), Chat()]
14
15 for animal in animaux:
16     print(animal.faire_son_cri())
```

Exercice : Polymorphisme

- Créez une classe "Véhicule" avec une méthode "description" qui renvoie une description générique.
- Créez ensuite des classes "Voiture" et "Moto" qui héritent de "Véhicule" et redéfinissez la méthode "description" pour renvoyer des descriptions spécifiques.

Solution

```
1  class Véhicule:
2      def description(self):
3          return "Ceci est un véhicule."
4
5  class Voiture(Véhicule):
6      def description(self):
7          return "Ceci est une voiture."
8
9  class Moto(Véhicule):
10     def description(self):
11         return "Ceci est une moto."
12
13 # Utilisation du polymorphisme
14 véhicules = [Véhicule(), Voiture(), Moto()]
15
16 for véhicule in véhicules:
17     print(véhicule.description())
```

Surcharge de méthodes

- La surcharge de méthodes permet à une classe enfant de redéfinir une méthode héritée de sa classe parente.
- La méthode redéfinie dans la classe enfant doit avoir la même signature (c'est-à-dire le même nombre et le même type de paramètres) que la méthode de la classe parente.

Exemple :

- La méthode *description* de la classe Véhicule est redéfinie dans la classe Voiture.
- Lorsque vous appelez la méthode description sur un objet de la classe Voiture, c'est la version de la méthode définie dans la classe Voiture qui est exécutée, ce qui signifie que "Ceci est une voiture." sera renvoyé.

Surcharge de méthodes : Syntaxe

```
1 class Parent:
2     def afficher_info(self):
3         print("Méthode de la classe parente")
4
5 class Enfant(Parent):
6     def afficher_info(self):
7         print("Méthode redéfinie dans la classe enfant")
```

Dans cet exemple, la classe enfant redéfinit la méthode `afficher_info()` héritée de la classe parente.

Exercice : Surcharge de méthodes

1. Créez une classe Personne avec une méthode presentation qui renvoie "Je suis une personne."
2. Définissez une sous-classe Etudiant qui hérite de Personne. Surchargez la méthode presentation dans la classe Etudiant pour qu'elle renvoie "Je suis un étudiant."
3. Créez un objet de la classe Personne et appelez la méthode presentation.
4. Créez un objet de la classe Etudiant et appelez la méthode presentation.

Solution

```python
class Personne:
    def presentation(self):
        return "Je suis une personne."

class Etudiant(Personne):
    def presentation(self):
        return "Je suis un étudiant."

personne = Personne()
etudiant = Etudiant()

# Résultat attendu : "Je suis une personne."
print(personne.presentation())
# Résultat attendu : "Je suis un étudiant."
print(etudiant.presentation())
```

À retenir

- **Héritage** : L'héritage est un mécanisme en POO qui permet de créer de nouvelles classes (enfants) basées sur des classes existantes (parents). Les classes enfants héritent des attributs et des méthodes des classes parentes. Cela favorise la réutilisation du code et la structuration des classes en hiérarchies.
- **Polymorphisme** : Le polymorphisme est la capacité à traiter des objets de différentes classes de manière uniforme. En Python, le polymorphisme est souvent implémenté via la surcharge des méthodes. Il permet de réutiliser des méthodes ou des opérateurs sur des objets de différentes classes, simplifiant ainsi le code.
- **Surcharge des Méthodes** : La surcharge des méthodes consiste à redéfinir une méthode héritée dans une classe enfant. Les méthodes redéfinies dans la classe enfant doivent avoir la même signature que celles de la classe parente. Cela permet d'adapter le comportement des méthodes pour chaque classe.