JOUR 1_1

Programmation Orientée Objet (POO)

Introduction

La programmation orientée objet (POO) est un paradigme de programmation qui permet de modéliser le monde réel en utilisant des objets.

Les objets sont des entités qui possèdent des propriétés et des méthodes.

La POO avancé est une extension de la POO de base. Elle permet de créer des programmes plus complexes et plus flexibles.

Le cours couvrira les sujets suivants :

- **Revoir les concepts de base de la POO**
- **L'héritage** : l'héritage permet de créer des classes qui héritent des propriétés et des méthodes d'autres classes.
- **La surcharge de méthodes** : la surcharge de méthodes permet de définir plusieurs méthodes avec le même nom, mais avec des signatures différentes.
- **La polymorphisme** : le polymorphisme permet d'appeler une méthode sur un objet, quelle que soit la classe à laquelle appartient l'objet.
- **Les classes abstraites** : les classes abstraites sont des classes qui ne peuvent pas être instanciées. Elles sont utilisées pour définir des interfaces.
- **Les interfaces** : les interfaces sont des contrats qui spécifient les méthodes qu'une classe doit implémenter.

Les concepts de base de la POO

Les concepts de base de la POO sont les suivants :

- **Les classes** : les classes sont des modèles qui définissent les propriétés et les méthodes des objets.
- **Les objets** : les objets sont des instances de classes. Ils possèdent les propriétés et les méthodes définies dans la classe.
- **Les attributs** : les attributs sont des variables qui sont associées à une classe ou à un objet.
- **Les méthodes** : les méthodes sont des fonctions qui sont associées à une classe ou à un objet.

Les classes

Une classe est un modèle qui définit les propriétés et les méthodes des objets. Une classe est définie à l'aide du mot-clé `class`.

Voici un exemple de définition d'une classe :

```
class Voiture:
    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele

    def demarrer(self):
        print("La voiture démarre")

    def accelerer(self):
        print("La voiture accélère")
```

Les objets

Un objet est une instance d'une classe. Un objet est créé à l'aide du mot-clé `new`.

Voici un exemple de création d'un objet :

```
voiture = Voiture("Peugeot", "308")
```

Cette ligne de code crée un objet `voiture` de la classe `Voiture`. L'objet `voiture` possède les valeurs `"Peugeot"` pour l'attribut `marque` et `"308"` pour l'attribut `modele`.

Les attributs

Un attribut est une variable qui est associée à une classe ou à un objet. Un attribut est accessible à l'aide du symbole de point `.`.

Voici un exemple d'accès à un attribut :

```
voiture.marque
```

Cette ligne de code renvoie la valeur de l'attribut `marque` de l'objet `voiture`.

Les méthodes

Une méthode est une fonction qui est associée à une classe ou à un objet. Une méthode est appelée à l'aide du symbole de point `.`.

Voici un exemple d'appel d'une méthode :

```
voiture.demarrer()
```

Cette ligne de code appelle la méthode `demarrer()` de l'objet `voiture`.

Exercices

- Créez une classe `Personne` qui possède les attributs `nom` et `prenom`.
- Créez une méthode `se_presenter()` dans la classe `Personne` qui affiche les informations de la personne.
- Créez un objet `personne` de la classe `Personne` et appelez la méthode `se_presenter()`.

L'Héritage

L'héritage est un concept important de la programmation orientée objet qui permet de créer des classes qui héritent des propriétés et des méthodes d'autres classes.

Principe de l'héritage

L'héritage permet de créer une relation de parenté entre deux classes. La classe parente est appelée la classe de base, et la classe enfant est appelée la classe dérivée.

La classe dérivée hérite de toutes les propriétés et méthodes de la classe de base. Elle peut également ajouter ses propres propriétés et méthodes.

Exemple

Voici un exemple de deux classes qui utilisent l'héritage :

```
class Animal:
    def __init__(self, nom):
        self.nom = nom

    def manger(self):
        print("L'animal mange")

class Chat(Animal):
    def miauler(self):
        print("Le chat miaule")
```

La classe `Animal` est la classe de base. Elle définit une propriété `nom` et une méthode `manger()`.

La classe `Chat` est la classe dérivée. Elle hérite de la propriété `nom` et de la méthode `manger()` de la classe de base. Elle définit également une méthode `miauler()`.

Utilisation de l'héritage

L'héritage peut être utilisé pour :

- Réutiliser du code existant : l'héritage permet de réutiliser le code des classes de base dans les classes dérivées.
- Créer des classes plus complexes : l'héritage permet de créer des classes plus complexes en combinant les propriétés et les méthodes de plusieurs classes.
- Favoriser la modularité : l'héritage permet de favoriser la modularité des programmes en encapsulant les propriétés et les méthodes communes dans des classes de base.

Exemple de réutilisation du code

Voici un exemple de réutilisation du code avec l'héritage :

```
class Animal:
    def __init__(self, nom):
        self.nom = nom

    def manger(self):
        print("L'animal mange")

class Chien(Animal):
    def aboyer(self):
        print("Le chien aboie")
```

```
class Chat(Animal):
    def miauler(self):
        print("Le chat miaule")
```

Dans cet exemple, les classes `Chien` et `Chat` réutilisent la méthode `manger()` de la classe de base `Animal`.

Exemple de création de classes plus complexes

Voici un exemple de création de classes plus complexes avec l'héritage :

```
class Vehicule:
    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele

    def demarrer(self):
        print("Le véhicule démarre")

    def accelerer(self):
        print("Le véhicule accélère")

class Voiture(Vehicule):
    def klaxonner(self):
        print("La voiture klaxonne")

class Moto(Vehicule):
    def faire_wheeling(self):
        print("La moto fait un wheeling")
```

Dans cet exemple, la classe `Voiture` hérite de la classe de base `Vehicule`. Elle ajoute la méthode `klaxonner()`.

La classe `Moto` hérite également de la classe de base `Vehicule`. Elle ajoute la méthode `faire_wheeling()`.

Exemple de modularité

Voici un exemple de modularité avec l'héritage :

```
class Personnage:
    def __init__(self, nom, prenom):
        self.nom = nom
        self.prenom = prenom

    def se_presenter(self):
        print("Je m'appelle", self.nom, ".")

class Guerrier(Personnage):
    def attaquer(self):
        print("Le guerrier attaque")

class Magicien(Personnage):
    def lancer_sort(self):
        print("Le magicien lance un sort")
```

Dans cet exemple, les classes `Guerrier` et `Magicien` héritent de la classe de base `Personnage`. Elles ajoutent des méthodes spécifiques à leur rôle.

Conclusion

L'héritage est un concept puissant de la programmation orientée objet qui permet de créer des programmes plus complexes et plus flexibles.

La surcharge de méthodes

La surcharge de méthodes est un concept de la programmation orientée objet qui permet de définir plusieurs méthodes avec le même nom, mais avec des signatures différentes.

Principe de la surcharge de méthodes

La surcharge de méthodes est basée sur le principe de l'overloading, qui est un mécanisme qui permet à une fonction d'être appelée avec différents types d'arguments.

En Python, la surcharge de méthodes est réalisée en définissant plusieurs méthodes avec le même nom, mais avec des listes de paramètres différentes.

Exemple

Voici un exemple de surcharge de méthodes :

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

    def distance(self, point):
        return abs(self.x - point.x) + abs(self.y - point.y)

    def distance(self, x, y):
        return abs(self.x - x) + abs(self.y - y)
```

La classe `Point` définit deux méthodes `distance()`. La première méthode prend un objet de type `Point` en tant qu'argument. La deuxième méthode prend deux nombres en tant qu'arguments.

Utilisation de la surcharge de méthodes

La surcharge de méthodes peut être utilisée pour :

- Offrir une flexibilité supplémentaire aux utilisateurs : la surcharge de méthodes permet aux utilisateurs de choisir la méthode la plus appropriée pour leurs besoins.

- Favoriser la modularité : la surcharge de méthodes permet de séparer les différentes fonctionnalités d'une classe.

Exemple d'offre de flexibilité supplémentaire

Voici un exemple d'offre de flexibilité supplémentaire avec la surcharge de méthodes :

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

    def distance(self, point):
        return abs(self.x - point.x) + abs(self.y - point.y)

    def distance(self, x, y):
        return abs(self.x - x) + abs(self.y - y)

    def distance(self, point, unit="pixel"):
        if unit == "pixel":
            return abs(self.x - point.x) + abs(self.y -
point.y)
        elif unit == "cm":
            return (abs(self.x - point.x) + abs(self.y -
point.y)) * 0.0393701
        else:
            raise ValueError("Unité invalide")
```

Dans cet exemple, la méthode `distance()` prend un troisième argument en tant que nom d'unité. La méthode renvoie la distance entre deux points, en fonction de l'unité spécifiée.

```
class Point:
    def __init__(self, x, y):
```

```
        self.x = x
        self.y = y

    def distance(self, arg1, arg2=None, unit="pixel"):
        if arg2 is None:
            # Si un seul argument est fourni, il est traité
comme un objet Point
            point = arg1
            if unit == "pixel":
                return abs(self.x - point.x) + abs(self.y -
point.y)
            elif unit == "cm":
                return (abs(self.x - point.x) + abs(self.y -
point.y)) * 0.0393701
            else:
                raise ValueError("Unité invalide")
        else:
            # Si deux arguments sont fournis, ils sont traités
comme les coordonnées x et y
            x, y = arg1, arg2
            if unit == "pixel":
                return abs(self.x - x) + abs(self.y - y)
            elif unit == "cm":
                return (abs(self.x - x) + abs(self.y - y)) *
0.0393701
            else:
                raise ValueError("Unité invalide")

# Exemple d'instanciation
point1 = Point(1, 2)
point2 = Point(3, 4)

# Utilisation de la méthode distance
distance_pixels = point1.distance(point2)
distance_cm = point1.distance(3, 4, unit="cm")
```

```
print(f"Distance en pixels : {distance_pixels}")
print(f"Distance en cm : {distance_cm}")
```

Exemple de modularité

Voici un exemple de modularité avec la surcharge de méthodes :

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

    def distance(self, point):
        return abs(self.x - point.x) + abs(self.y - point.y)

    def distance_carre(self, point):
        return (self.x - point.x)**2 + (self.y - point.y)**2

    def distance_hypotenuse(self, point):
        return math.sqrt((self.x - point.x)**2 + (self.y -
point.y)**2)
```

Dans cet exemple, la classe `Point` définit trois méthodes `distance()`. Ces méthodes calculent la distance entre deux points, en utilisant différentes formules.

Conclusion

La surcharge de méthodes est un concept puissant de la programmation orientée objet qui permet d'offrir une flexibilité supplémentaire aux utilisateurs et de favoriser la modularité.

Le polymorphisme

Le polymorphisme est un concept de la programmation orientée objet qui permet d'appeler une méthode sur un objet, quelle que soit la classe à laquelle appartient l'objet.

Principe du polymorphisme

Le polymorphisme est basé sur le principe de l'héritage. En effet, lorsqu'une classe dérivée hérite d'une classe de base, elle hérite également de toutes les méthodes de la classe de base.

En Python, le polymorphisme est réalisé en utilisant la méthode `__call__()`. Cette méthode est appelée lorsqu'un objet est appelé comme une fonction.

Exemple

Voici un exemple de polymorphisme :

```
class Animal:
    def __init__(self, nom):
        self.nom = nom

    def manger(self):
        print("L'animal mange")

class Chien(Animal):
    def aboyer(self):
        print("Le chien aboie")

class Chat(Animal):
    def miauler(self):
        print("Le chat miaule")

chien = Chien("Rex")
chat = Chat("Miaou")
```

```
chien.manger()
chat.manger()
```

Dans cet exemple, les classes `Chien` et `Chat` héritent de la classe de base `Animal`. La classe `Chien` définit la méthode `aboyer()`, et la classe `Chat` définit la méthode `miauler()`.

Dans l'appel de la méthode `manger()`, l'objet `chien` est passé en tant que paramètre. La méthode `manger()` est définie dans la classe `Animal`. Cependant, l'objet `chien` est un objet de la classe `Chien`. La méthode `manger()` de la classe `Chien` est donc appelée.

De même, dans l'appel de la méthode `manger()`, l'objet `chat` est passé en tant que paramètre. La méthode `manger()` est définie dans la classe `Animal`. Cependant, l'objet `chat` est un objet de la classe `Chat`. La méthode `manger()` de la classe `Chat` est donc appelée.

Utilisation du polymorphisme

Le polymorphisme peut être utilisé pour :

- Offrir une flexibilité supplémentaire aux utilisateurs : le polymorphisme permet aux utilisateurs de travailler avec des objets de différents types sans avoir à se soucier de la classe de l'objet.
- Favoriser la modularité : le polymorphisme permet de séparer les différentes fonctionnalités d'une classe.

Exemple d'offre de flexibilité supplémentaire

Voici un exemple d'offre de flexibilité supplémentaire avec le polymorphisme :

```
class Animal:
    def __init__(self, nom):
        self.nom = nom
```

```
    def manger(self):
        print("L'animal mange")

class Chien(Animal):
    def aboyer(self):
        print("Le chien aboie")

class Chat(Animal):
    def miauler(self):
        print("Le chat miaule")

def nourrir_animal(animal):
    animal.manger()

chien = Chien("Rex")
chat = Chat("Miaou")

nourrir_animal(chien)
nourrir_animal(chat)
```

Dans cet exemple, la fonction `nourrir_animal()` prend un objet de type `Animal` en tant que paramètre. La fonction appelle la méthode `manger()` de l'objet.

Dans l'appel de la fonction `nourrir_animal()`, l'objet `chien` est passé en tant que paramètre. La méthode `manger()` de la classe `Chien` est donc appelée.

De même, dans l'appel de la fonction `nourrir_animal()`, l'objet `chat` est passé en tant que paramètre. La méthode `manger()` de la classe `Chat` est donc appelée.

Conclusion

Le polymorphisme est un concept puissant de la programmation orientée objet qui permet d'offrir une flexibilité supplémentaire aux utilisateurs et de favoriser la modularité.

Les classes abstraites

Les classes abstraites sont un concept de la programmation orientée objet qui permet de définir des classes qui ne peuvent pas être instanciées.

Principe des classes abstraites

Les classes abstraites sont utilisées pour définir des interfaces. Une interface est un contrat qui spécifie les méthodes qu'une classe doit implémenter.

En Python, les classes abstraites sont définies en utilisant le mot-clé `abc.ABCMeta`.

Exemple

Voici un exemple de classe abstraite :

```python
from abc import ABCMeta, abstractmethod

class Animal(metaclass=ABCMeta):
    @abstractmethod
    def manger(self):
        pass

    @abstractmethod
    def dormir(self):
        pass

class Chien(Animal):
    def manger(self):
        print("Le chien mange")
```

```
    def dormir(self):
        print("Le chien dort")

class Chat(Animal):
    def manger(self):
        print("Le chat mange")

    def dormir(self):
        print("Le chat dort")
```

La classe `Animal` est une classe abstraite. Elle définit deux méthodes abstraites : `manger()` et `dormir()`.

Les classes `Chien` et `Chat` héritent de la classe `Animal`. Elles doivent donc implémenter les méthodes `manger()` et `dormir()`.

Utilisation des classes abstraites

Les classes abstraites peuvent être utilisées pour :

- **Définir des interfaces** : les classes abstraites peuvent être utilisées pour définir des interfaces qui spécifient les méthodes qu'une classe doit implémenter.
- **Favoriser la modularité** : les classes abstraites peuvent être utilisées pour favoriser la modularité des programmes en encapsulant les interfaces dans des classes abstraites.

Exemple de définition d'une interface

Voici un exemple de définition d'une interface :

```
from abc import ABCMeta, abstractmethod

class Animal(metaclass=ABCMeta):
    @abstractmethod
```

```
    def manger(self):
        pass

    @abstractmethod
    def dormir(self):
        pass
```

La classe `Animal` définit deux méthodes abstraites : `manger()` et `dormir()`. Ces méthodes spécifient les comportements que doivent avoir tous les animaux.

Exemple de favoriser la modularité

Voici un exemple de favoriser la modularité avec les classes abstraites :

```
from abc import ABCMeta, abstractmethod

class Animal(metaclass=ABCMeta):
    @abstractmethod
    def manger(self):
        pass

    @abstractmethod
    def dormir(self):
        pass

class Zoo:
    def __init__(self):
        self.animaux = []

    def ajouter_animal(self, animal):
        self.animaux.append(animal)

    def nourrir_animaux(self):
        for animal in self.animaux:
            animal.manger()
```

```
    def faire_dormir_animaux(self):
        for animal in self.animaux:
            animal.dormir()

class Chien(Animal):
    def manger(self):
        print("Le chien mange")

    def dormir(self):
        print("Le chien dort")

class Chat(Animal):
    def manger(self):
        print("Le chat mange")

    def dormir(self):
        print("Le chat dort")

zoo = Zoo()
zoo.ajouter_animal(Chien())
zoo.ajouter_animal(Chat())

zoo.nourrir_animaux()
zoo.faire_dormir_animaux()
```

Dans cet exemple, la classe `Zoo` gère les animaux du zoo. La classe `Zoo` utilise la classe abstraite `Animal` pour définir les méthodes `manger()` et `dormir()`.

Les classes `Chien` et `Chat` héritent de la classe `Animal` et implémentent les méthodes `manger()` et `dormir()`.

La classe `Zoo` peut ajouter des animaux de n'importe quel type, tant que ces animaux implémentent les méthodes `manger()` et `dormir()`.