FORMATION PYTHON

POO AVANCÉ - Partie 2

Héritage simple

- Python supporte l'héritage simple et l'héritage multiple. *Nous parlerons ici de l'héritage simple.*
- Prenons l'exemple de la création d'une classe *fille* `Etudiant` à partir de notre classe `Personne` (un peu modifiée) :

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

```
class Etudiant(Personne):
    pass
```

La classe `Etudiant` hérite de la classe `Personne`. Mais là, elle ne fait pas grand chose de plus.

Créons une instance de chacune des classes :

```
p = Personne(nom="DUPONT", prenom="Jean")
print(p)

e = Etudiant(nom="LAURENT", prenom="Xavier")
print(e)
print(e.nom, e.prenom)
```

- Ajoutons du code d'initialisation spécifique.

```
class Etudiant(Personne):
    def __init__(self, nom, prenom="", ecole=""):
        self.nom = nom
        self.prenom = prenom
        self.ecole = ecole
```

Testons :

```
e = Etudiant(nom="LAURENT", prenom="Xavier", ecole="HEC")
print(e.nom, e.prenom, e.ecole)
```

Utilisation de la fonction super()

Plutôt que de réécrire tout pour l'initialisation, avec le risque d'erreur, on va utiliser la fonction `super()` pour **déléguer à la classe parent l'initialisation des attributs hérités**.

Dans notre cas :

```
class Etudiant(Personne):
    def __init__(self, nom, prenom="", ecole=""):
        super(Etudiant, self).__init__(nom, prenom)
        self.ecole = ecole
```

Note : `super(Etudiant, self).__init__()` est équivalent à `super().__init__(nom, prenom)`.

Héritage et Polymorphisme

Le **polymorphisme** nous permet de modifier le comportement d'une classe fille par rapport à sa classe mère. On utilise l'héritage comme un mécanisme d'extension en adaptant le comportement des objets.

Soit la classe Personne définit comme suit :

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

    def generer_identifiant(self):
        print("Identifiant générique")
        print(self.nom.lower() + self.prenom.lower())
```

Dans la classe fille `Etudiant`, nous pouvons redéfinir le comportement de la méthode `generer_identifiant()` (**override**).

```
class Etudiant(Personne):
    def __init__(self, nom, prenom="", ecole=""):
        super().__init__(nom, prenom)
        self.ecole = ecole

    def generer_identifiant(self):
        ident = self.ecole + "-" + self.nom + self.prenom
        ident = ident.lower()
        print(ident)
```

À tester : Créez une classe fille `Professeur` sur le même principe. Elle pourrait ressembler à ceci :

```
class Professeur(Personne):
    def __init__(self, nom, prenom="", matiere=""):
        super().__init__(nom, prenom)
        self.matiere = matiere

    def generer_identifiant(self):
        ident = self.matiere + "-" + self.nom + self.prenom
        ident = "prof-" + ident
        ident = ident.lower()
        print(ident)
```

Un étudiant est bien une personne. Un professeur est une personne. Les objets de ces types disposent bien de la méthode `generer_identifiant()`, mais son comportement est **polymorphe**. Il dépend du type réel de l'objet.

Héritage multiple

Une classe peut hériter de plusieurs classes "parent".

```
class ClasseParent1:
    pass

class ClasseParent2:
    pass

...

class ClasseEnfant(ClasseParent1, ClasseParent2, ...):
    pass
```

Exemple

```
class Etudiant:
    ...

class Employe:
    ...

class EtudiantQuiTravaille(Etudiant, Employe):
    ...
```

Method Resolution Order (MRO)

Dans l'héritage multiple, Python cherche chaque attribut spécifié sur l'objet, en suivant un certain ordre :

1. dans la classe de l'objet (classe courante),
2. si pas trouvé, la recherche passe aux classes parent et dans le sens de la gauche vers la droite,
3. si pas trouvé, l'interpréteur Python termine sa recherche avec la classe dont héritent tous les objets, la classe `object`.

On peut visualiser le "MRO" d'une classe en utilisant :

- soit avec l'attribut spécial `__mro__`,
- soit avec la méthode correspondante `mro()`.

```
>>> ClasseEnfant.__mro__
(<class '__main__.ClasseEnfant'>, <class '__main__.ClasseParent1'>, <c
lass '__main__.ClasseParent2'>, <class 'object'>)
```

À tester : Expérimentez avec le code suivant, qui correspond à un cas plus complexe.

```
class Classe1:
    def m(self):
        print("Dans Classe1")

class Classe2(Classe1):
    def m(self):
        print("Dans Classe2")
        super().m()

class Classe3(Classe1):
    def m(self):
        print("Dans Classe3")
        super().m()

class Classe4(Classe2, Classe3):
    def m(self):
        print("Dans Classe4")
        super().m()

obj = Classe4()
obj.m()
```

```
>>> Classe4.__mro__
(<class '__main__.Classe4'>, <class '__main__.Classe2'>, <class '__mai
n__.Classe3'>, <class '__main__.Classe1'>, <class 'object'>)
```

La technique du "Mixin"

- Un Mixin est un ensemble d'attributs et de méthodes pouvant être utilisées dans différentes classes, qui ne proviennent pas d'une classe de base.
- En général, nous héritons d'une classe "mixin" pour donner aux objets de classes différentes la même fonctionnalité.

Exemple

```
class ExperienceProfessionnelleMixin:
    ...

class Stagiaire(Personne, ExperienceProfessionnelleMixin):
    ...

class Employe(Personne, ExperienceProfessionnelleMixin):
    ...
```

Cas pratique : Expérimenter avec une classe mixin `ExperienceProfessionnelleMixin` qui apporterait les fonctionnalités complémentaires utiles pour construire les classes `Stagiaire` et `Employe`. ***Voir exercices.***