Langage Python

Programmation orientée objet (POO) - partie 2

Formation POEC Cybersécurité
Théo Hubert

Syntaxe

- *Surcharge*

La surcharge d'opérateurs nous permet d'**utiliser des opérateurs standards** tels que +, -, * etc sur nos **classes personnalisées**. Il est également **possible de surcharger certaines fonctions** tel que print().

Constructeur →

Surcharge de l'opérateur + →

Surcharge de la fonction print() →

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, p2):
        return Point2D(self.x + p2.x, self.y + p2.y)

    def __str__(self):
        return str((self.x, self.y))

p1 = Point2D(2,3);
p2 = Point2D(4,5);

result = p1 + p2

print(result)
```

Syntaxe

- *Surcharge*

Exemple de surcharge n°2 :

Par exemple pour définir ce que signifie l'**égalité de deux objets,** il faut redéfinir l'opérateur == en définissant une méthode **__eq__**.

Constructeur →

Surcharge de l'opérateur == →

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

p1 = Point2D(2,2);
p2 = Point2D(2,2);

if p1 == p2 :
  print("Les 2 vecteurs sont égaux")
```

```
Les 2 vecteurs sont égaux
```

Syntaxe

- *Surcharge d'opérateurs*

Opérateur	Notation	Méthode à définir
Signe positif	+	__pos__
Signe négatif	-	__neg__
Addition	+	__add__
Soustraction	-	__sub__
Multiplication	*	__mul__
Division	/	__truediv__
Égal	==	__eq__
Différent	!=	__ne__

Pour voir la totalité : http://lptms.u-psud.fr/wiki-cours/index.php/Python:_Surcharge

Concept

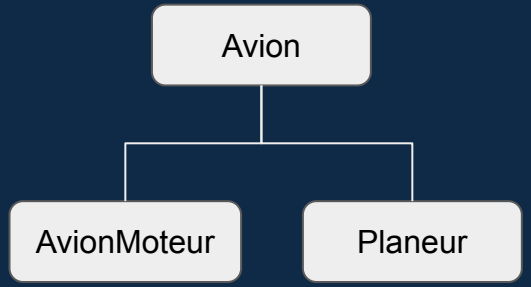
- *Héritage*

Le concept d'héritage constitue l'un des **fondements** de la programmation orientée objet. Il est notamment à l'origine des possibilités de **réutilisation des composants logiciels** que sont les classes.

En effet, il permet de définir une nouvelle classe, dite ***classe dérivée***, à partir d'une classe existante dite ***classe de base***.

Cette nouvelle classe hérite d'emblée **des fonctionnalités de la classe de base** (champs et méthodes) qu'elle pourra modifier ou compléter à volonté sans qu'il soit nécessaire de remettre en question la classe de base.

Cette technique permet de développer de nouveaux outils en se fondant sur un certain acquis, ce qui justifie le terme d'**héritage**. Comme on peut s'y attendre, il sera possible de développer à partir d'une classe de base, autant de classe dérivées qu'on le désire. De même, une classe dérivée pourra à son tour servir de classe de base pour une nouvelle classe dérivée.

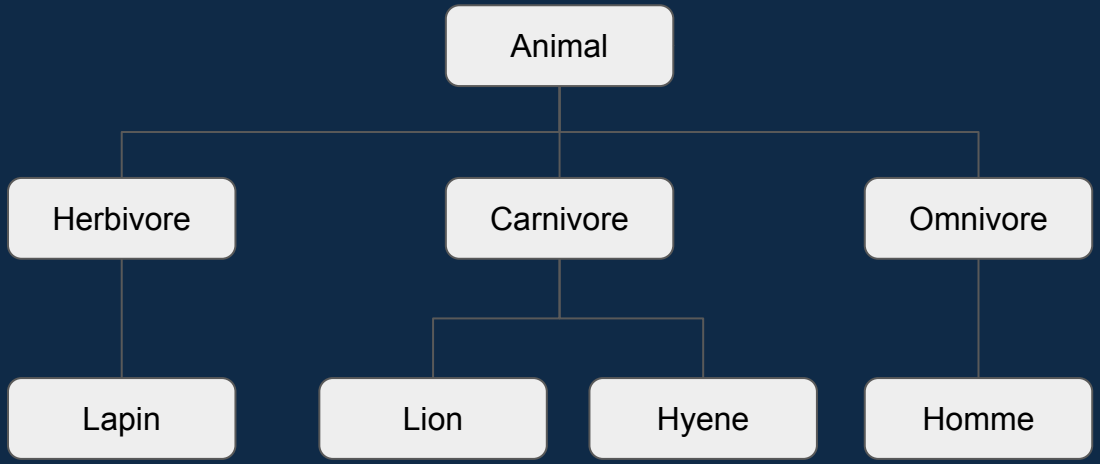


Concept

- *Héritage*

En bref : “L’héritage est un mécanisme qui permet à une classe de disposer des champs et des méthodes d’une autre classe”

Exemple :



Syntaxe

- *Héritage*

Pour utiliser l'héritage avec Python il faut spécifier le **nom de la classe mère en argument de la classe fille** puis il faut faire appel à **super()** dans le constructeur de la classe fille.

Nom de la classe mère en argument de la classe fille →

Fonction super() pour faire appel au constructeur de la classe mère →

```python
class Personne():

    # Constructeur
    def __init__(self, nom, prenom):
        self.nom = nom
        self.prenom = prenom

    def ToPrint(self):
        print("Nom : ", self.nom," & Prénom : ", self.prenom)


class Employe(Personne):

    def __init__(self, nom, prenom, salaire):
        super().__init__(nom, prenom)
        self.salaire = salaire

employe1 = Employe("Jean", "David", 1600)
employe1.ToPrint()
```

```
Nom :  Jean  & Prénom :  David
```

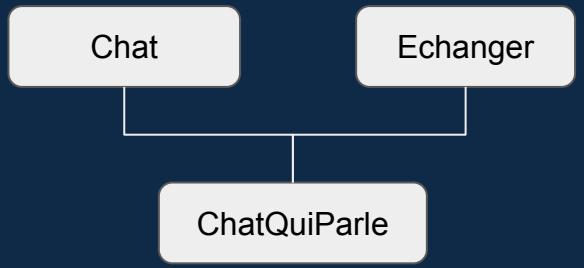
Concept

- *Héritage multiple*

L'**héritage multiple** suppose qu'une classe ait de **multiples classes parents** à l'opposée de l'héritage ou il y a une hiérarchie, c'est à dire plusieurs niveaux d'héritage (une classe **à un parent qui a un parent**).

L'héritage multiple a **mauvaise réputation** en programmation orientée objet car les systèmes qui utilisent l'héritage multiple peuvent être difficiles à comprendre.

Le plus souvent, l'héritage multiple **n'est pas une bonne solution** au problème à résoudre, mais il n'en reste pas moins qu'il existe *quelques* situations où il représente la meilleure solution possible.



Syntaxe

- *L'héritage multiple*

Python permet de l'héritage multiple, cela signifie qu'il est possible de créer une classe étendue (extends) depuis **deux ou plusieurs classes**.

La classe hérite de 2 classes mères Chat et Echanger

```
class Chat:
    def meow(self):
        """Miaule."""
        print("Meow!")


class Echanger:

    def parler(self, parole):
        print(parole)


class ChatQuiParle(Chat, Echanger):
    pass
```

Syntaxe

- *Polymorphisme*

Le polymorphisme est un concept fondamental de la programmation orientée objet, qui vient compléter le concept d'héritage.
On peut caractériser le polymorphisme en disant que c'est l'aptitude d'un objet à **pouvoir prendre plusieurs formes**.

```
class Oiseau:

  def chanter(self):
    print("Coui Coui Coui Coui")

class Pigeon(Oiseau):

  def chanter(self):
    print("Piou Piou Piou Piou")

oiseau1 = Oiseau()
pigeon1 = Pigeon()
oiseau1.chanter()
pigeon1.chanter()
```

```
Coui Coui Coui Coui
Piou Piou Piou Piou
```

Contrairement à d'autre langage de programmation tel que Java, il n'existe pas de notation override dans Python.

Syntaxe

- *Classe abstraite*

Le concept de méthode abstraite (abstract method) ou une classe abstraite (abstract class) sont définis dans des langages comme Java, C#.

Pour rappel, une **classe abstraite** est une classe que l'on ne peut pas instancier. Elle **sert uniquement de classe de base** pour la dérivation et elle peut obliger une classe dérivée à implémenter certaines méthodes qui sont dites abstraites.

Dans Python il n'existe pas de mots clés abstract mais on peut utiliser le décorateur suivant @abstractmethod

Syntaxe

- *Classe abstraite*

Dans Python il n'existe pas de mots clés abstract mais on peut utiliser le décorateur suivant **@abstractmethod** et la classe **ABC**. Pour ce faire il faut faire appel à la librairie **abc**.

```
from abc import ABC, abstractmethod

class Animal(ABC): # hériter de ABC(Abstract base class)

    @abstractmethod  # un décorateur pour définir une méthode abstraite
    def nourrir(self):
        pass

class Panda(Animal):
    def nourrir(self):
        print("Nourrir le panda avec du bamboo!")

panda1 = Panda()
panda1.nourrir()
```

```
Nourrir le panda avec du bamboo!
```

Syntaxe

Travaux Pratique

Références

Histoire de la POO :
https://bpesquet.developpez.com/tutoriels/csharp/programmation-orientee-objet-csharp/?page=initiation-a-la-programmation-orientee-objet
https://www.jedha.co/blog/quest-ce-que-la-programmation-orientee-objet

POO/Procédurale :
https://waytolearnx.com/2018/09/difference-entre-programmation-procedurale-et-orientee-objet.html
https://practicalprogramming.fr/paradigme-programmation-orientee-objet-et-programmation-fonctionnelle

Intérêt de la POO :
https://www.powerpress.fr/ads/avantages-de-la-programmation-orientee-objet/

Classes abstraites :
https://pythonforge.com/classes-abstraites-en-python/#:~:text=Les%20classes%20abstraites%20sont%20des,'a%20pas%20d'impl%C3%A9mentation.

Fin

La suite :
Programmation Orienté Objet