

CHAPITRE 1

Coder une solution orientée objet

1. Création d'un package
2. Codage d'une classe
3. **Intégration des concepts POO**



Getter et Setter en Python

- Les Getters et Setters en Python sont souvent utilisés pour éviter l'accès direct à un champ de classe, c'est-à-dire que les variables privées ne peuvent pas être accessibles directement ou modifiées par un utilisateur externe.

Utilisation de la fonction normale pour obtenir le comportement des Getters et des Setters

- Pour obtenir la propriété Getters et Setters, si nous définissons les méthodes `get()` et `set()`, cela ne reflètera aucune implémentation spéciale.
- Exemple :**

```
class EtudiantGeek:
    def __init__(self, age = 0):
        self._age = age

    def get_age(self): #déclarartion d'un getter
        return self._age

    def set_age(self, x): #declaration d'un setter
        self._age = x
```

```
if __name__=='__main__':
    raj = EtudiantGeek()
    raj.set_age(21)
    print(raj.get_age())
    print(raj._age)
#Affiche:
#21
#21
```

01 - Coder une solution orientée objet

Intégration des concepts POO



Getter et Setter avec property()

- En Python `property()` est une fonction intégrée qui crée et renvoie un objet de propriété.
- Un objet de propriété a trois méthodes, `getter()`, `setter()` et `delete()`.
- La fonction `property()` en Python a trois arguments **`property(fget, fset, fdel, doc)`**
 - **`fget`** est une fonction pour récupérer une valeur d'attribut
 - **`fset`** est une fonction pour définir une valeur d'attribut
 - **`fdel`** est une fonction pour supprimer une valeur d'attribut
 - **`doc`** est une chaîne contenant la documentation (docstring à voir ultérieurement) de l'attribut

01 - Coder une solution orientée objet

Intégration des concepts POO



- Exemple

```
class EtudiantGeek :  
    def __init__(self):  
        self._age = 0  
  
    def get_age(self):  
        print("getter est appelé")  
        return self._age  
  
    def set_age(self, a):  
        print("setter est appelé")  
        self._age = a  
  
    def del_age(self):  
        del self._age  
    age = property(get_age, set_age, del_age)
```

appel

```
if __name__ == '__main__':  
    mark = EtudiantGeek()  
    mark.age = 10  
    print(mark.age)  
  
    #affiche:  
    #setter est appelé  
    #getter est appelé  
    #10
```

01 - Coder une solution orientée objet

Intégration des concepts POO



Utilisation de @property

- **@property** est un décorateur qui évite d'utiliser la fonction getter explicite
- **@attribut.setter** est un décorateur qui évite d'utiliser la fonction setter explicite

Exemple :

```
class EtudiantGeek:
    def __init__(self):
        self._age = 0

    @property
    def age(self):
        print("getter method called")
        return self._age

    @age.setter
    def age(self, a):
        print("setter method called")
        self._age = a
```

appel

```
if __name__ == '__main__':
    mark = EtudiantGeek()
    mark.age = 19 #appel de setter
    print(mark.age) #appel de getter
    #affiche:
    #setter method called
    #getter method called
    #19
```

Héritage en Python

- Rappel :** l'héritage est défini comme la capacité d'une classe à dériver ou à hériter des propriétés d'une autre classe et à l'utiliser chaque fois que nécessaire.

Syntaxe en Python

```
Class Nom_classe(Nom_SuperClass)
```

Exemple :

```
class Enfant:  
    def __init__(self, nom):  
        self.nom = nom  
  
    def getNom(self):  
        return self.nom  
  
    def isEtudiant(self):  
        return False
```

```
class Etudiant(Enfant):  
    def isStudent(self):  
        return True
```

```
if __name__ == '__main__':  
    std = Enfant("Ram")  
    print(std.getNom(), std.isEtudiant())  
  
    std = Etudiant("Shivam")  
    print(std.getNom(), std.isEtudiant())
```

#Affiche:

#Ram False

#Shivam <bound method Enfant.isEtudiant of <__main__.Etudiant object at 0x0000024192C89250>

Classe Etudiant hérite de Enfant

Héritage unique

- **Rappel** : L'héritage unique permet à une classe dérivée d'hériter des propriétés d'une seule classe parente, permettant ainsi la réutilisation du code et l'ajout de nouvelles fonctionnalités au code existant.

Exemple :

```
class Parent:
    def func1(self):
        print("This function is in parent class.")

class Enfant(Parent): #hérite de la classe Parent
    def func2(self):
        print("This function is in child class.")
```

```
if __name__=='__main__':
    object = Enfant()
    object.func1()
    object.func2()

#affiche:
#This function is in parent class.
#This function is in child class.
```

Héritage multiple

- Rappel :** Lorsqu'une classe peut être dérivée de plusieurs classes de base, ce type d'héritage est appelé héritage multiple. Dans l'héritage multiple, toutes les fonctionnalités des classes de base sont héritées dans la classe dérivée.

Exemple :

```
class Mere:
    nomMere = ""
    def mere(self):
        print(self. nomMere)

class Pere:
    nomPere = ""
    def pere(self):
        print(self. nomPere)

class Enfant(Mere, Pere): #hérite de la classe Mere et Pere
    def parents(self):
        print("Father :", self. nomPere) # nomPere est attribut hérité
        print("Mother :", self. nomMere) # nomMere est attribut hérité
```

```
if __name__=='__main__':
    s1 = Enfant()
    s1. nomPere = "RAM"
    s1. nomMere = "SITA"
    s1.parents()

#Affiche:
#Father : RAM
#Mother : SITA
```


Héritage en cascade

- Dans l'héritage en cascade, les fonctionnalités de la classe de base et de la classe dérivée sont ensuite héritées dans la nouvelle classe dérivée

```
class GrandPere:
    def __init__(self, nomGrandPere):
        self.nomGrandPere = nomGrandPere
class Pere(GrandPere):
    def __init__(self, nomPere, nomGrandPere):
        self.nomPere = nomPere
        GrandPere.__init__(self, nomGrandPere)
class Enfant(Pere):
    def __init__(self, nomEnfant, nomPere, nomGrandPere):
        self.nomEnfant = nomEnfant
        Pere.__init__(self, nomPere, nomGrandPere)
    def affiche_name(self):
        print("nom grand_père :", self.nomGrandPere)
        print("nom_père:", self.nomPere)
        print("nom enfant:", self.nomEnfant)
```

```
if __name__=='__main__':
    s1 = Enfant('Prince', 'Rampal', 'Lal mani')
    print(s1.nomGrandPere)
    s1.affiche_name()
#affiche:
#Lal mani
# nom grand_père : Lal mani
# nom_père: Rampal
# nom enfant: Prince
```

Chainage de constructeurs

- Le chaînage de constructeurs est une technique **d'appel d'un constructeur de la classe mère à partir d'un constructeur de la classe fille**

Exemple :

```
class GrandPere:
    def __init__(self, nomGrandPere):
        self.nomGrandPere = nomGrandPere
class Pere(GrandPere):
    def __init__(self, nomPere, nomGrandPere):
        self.nomPere = nomPere
        GrandPere.__init__(self, nomGrandPere)
```

Appel du constructeur de la classe mère pour initialiser l'attribut hérité
nomGrandPere

Chainage de constructeurs

```
class Enfant(Pere):  
    def __init__(self,nomEnfant, nomPere, nomGrandPere):  
        self.nomEnfant = nomEnfant  
        Pere.__init__(self, nomPere, nomGrandPere)  
  
    def affiche_name(self):  
        print('nom grand_père :', self.nomGrandPere)  
        print("nom_père:", self.nomPere)  
        print("nom enfant:", self.nomEnfant)
```

Appel du constructeur de la classe mère pour initialiser les attributs hérités **nomPere** et **nomGrandPere**

```
if __name__=='__main__':  
    s1 = Enfant('Prince', 'Rampal', 'Lal mani')  
    print(s1.nomGrandPere)  
    s1.affiche_name()  
  
#affiche:  
#Lal mani  
# nom grand_père : Lal mani  
# nom_père: Rampal  
# nom enfant: Prince
```

Surcharge des opérateurs en Python

- La surcharge d'opérateurs vous permet de redéfinir la signification d'opérateur en fonction de votre classe.
- Cette fonctionnalité en Python, qui permet à un même opérateur d'avoir une signification différente en fonction du contexte.

Opérateurs arithmétiques

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

if __name__ == '__main__':
    p1 = Point(2, 4)
    p2 = Point(5, 1)
    p3 = p1+p2 #erreur
```

Affiche:

Traceback (most recent call last):

File "prog.py", line 10, in < module>

p3 = p1+p2

TypeError: unsupported operand type(s) for +: 'Point' and 'Point'

Erreur!!!

Python ne savait pas comment ajouter deux objets Point ensemble.

Surcharge des opérateurs en Python

- Pour surcharger l'opérateur +, nous devons implémenter la fonction `__add__()` dans la classe

Exemple :

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

    def __str__(self):
        return "{0},{1}".format(self.x, self.y)

    def __add__(self, p): #surcharge de l'opérateur +
        a = self.x + p.x
        b = self.y + p.y
        return Point(a, b)

if __name__ == '__main__':
    p1 = Point(2, 4)
    p2 = Point(5, 1)

    p3 = p1+p2
    print(p3) #affiche (7,5)
```

Fonctions spéciales de surcharge de l'opérateur en Python

Opérateur	Expression	Interprétation Python
Addition	$p1=p2$	<code>p1.__add__(p2)</code>
Soustraction	$p1-p2$	<code>p1.__sub__(p2)</code>
Multiplication	$p1*p2$	<code>p1.__mul__(p2)</code>
Puissance	$p1**p2$	<code>p1.__pow__(p2)</code>
Division	$p1/p2$	<code>p1.__truediv__(p2)</code>
Division entière	$p1//p2$	<code>p1.__floordiv__(p2)</code>
Le reste(modulo)	$p1\%p2$	<code>p1.__mod__(p2)</code>
ET binaire	$p1\&p2$	<code>p1.__and__(p2)</code>
OU binaire	$p1 p2$	<code>p1.__or__(p2)</code>
XOR	$p1^p2$	<code>p1.__xor__(p2)</code>
NON binaire	$\sim p1$	<code>p1.__invert__(p2)</code>

Surcharge des opérateurs en Python

Opérateurs de comparaison

- En Python, il est possible de surcharger les opérateurs de comparaison.

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

    def __lt__(self, p):
        m_self = math.sqrt((self.x ** 2) + (self.y ** 2))
        m2_p = math.sqrt((p.x ** 2) + (p.y ** 2))
        return m_self < m2_p
```

```
if __name__ == '__main__':
    p1 = Point(2, 4)
    p2 = Point(5, 1)
    if p1 < p2:
        print("p2 est plus loin que p1")
```

#affiche:

#p2 est plus loin que p1

Surcharge de l'opérateur inférieur

Fonctions spéciales de surcharge de l'opérateur en Python

Opérateur	Expression	Interprétation Python
Inférieur à	<code>p1<p2</code>	<code>p1.__lt__(p2)</code>
Inférieur ou égal	<code>p1<=p2</code>	<code>p1.__le__(p2)</code>
Egal	<code>p1==p2</code>	<code>p1.__eq__(p2)</code>
Différent	<code>p1!=p2</code>	<code>p1.__ne__(p2)</code>
Supérieur à	<code>p1>p2</code>	<code>p1.__gt__(p2)</code>
Supérieur ou égal	<code>p1>=p2</code>	<code>p1.__ge__(p2)</code>

Polymorphisme et héritage

- En Python, le polymorphisme permet de définir des méthodes dans la classe enfant qui ont le même nom que les méthodes de la classe parent.
- En héritage, la classe enfant hérite des méthodes de la classe parent.
- Il est possible de modifier une méthode dans une classe enfant dont elle a hérité de la classe parent (redéfinition de la méthode).

Exemple :

```
class Oiseau:
    def intro(self):
        print("Il y a différents types d'oiseaux")
    def vol(self):
        print("il y a des oiseaux qui volent d'autres non")
class Moineau(Oiseau):
    def vol(self):
        print("moineau peut voler.")
class Autruche (Oiseau):
    def vol(self):
        print("autruche ne volent pas.")
```

```
if __name__ == '__main__':
    obj_bird = Oiseau()
    obj_spr = Moineau()
    obj_ost = Autruche()
    obj_bird.intro() #affiche: Il y a différents types d'oiseaux
    obj_bird.vol() #affiche: il y a des oiseaux qui volent d'autres non.
    obj_spr.intro() #affiche: Il y a différents types d'oiseaux
    obj_spr.vol() #affiche: moineau peut voler.
    obj_ost.intro() #affiche : Il y a différents types d'oiseaux.
    obj_ost.vol() #affiche: autruche ne volent pas.
```

01 - Coder une solution orientée objet

Intégration des concepts POO



Classe abstraite

- Rappel: Les classes abstraites sont des classes qui ne peuvent pas être instanciées, elles contiennent une ou plusieurs méthodes abstraites (méthodes sans code)
- Une classe abstraite nécessite des sous-classes qui fournissent des implémentations pour les méthodes abstraites sinon ces sous-classes sont déclarées abstraites
- Une classe abstraite hérite de la **classe abstraite de base – ABC**
- L'objectif principal de la classe de base abstraite est de fournir un moyen standardisé de tester si un objet adhère à une spécification donnée
- Pour définir une méthode abstraite dans la classe abstraite, on utilise un décorateur **@abstractmethod**

01 - Coder une solution orientée objet

Intégration des concepts POO



Exemple

```
from abc import ABC, abstractmethod # abc est un module python intégré, nous importons ABC et
abstractmethod

class Animal(ABC): # hériter de ABC(Abstract base class)
    @abstractmethod # un décorateur pour définir une méthode abstraite
    def DonnerAManger(self):
        pass

class Cheval(Animal): #classe qui hérite de Animal
    def autre_nom(self):
        print("« Bonjour Cheval!»")

if __name__=='__main__':
    po = Cheval() #instanciation de Cheval impossible car la méthode DonneAManger est abstraite elle doit être
    implémentée dans Cheval
```

Affiche:
Traceback (most recent call last):
File
"C:\Users\DELL\AppData\Local\Programs\Python\Python39\ex0.py", line 14, in <module>
 po = Cheval() #instanciation de Cheval
impossible car la méthode nourrir est abstraite
elle doit être implémentée dans panda
TypeError: Can't instantiate abstract class Cheval
with abstract method DonnerAManger