

PYTHON : LA PROGRAMMATION ORIENTEE OBJET

BEN DAKHLIA Sonia

Objectif de la programmation objet

Qualités requises d'un logiciel

- Simplicité : faciliter la maintenance.
- Extensibilité : faciliter l'adaptation aux changements du domaine d'application.
- Décentralisation : faciliter le développement collaboratif.
- Réutilisabilité : augmenter la fiabilité et réduire les coûts et les délais de développement.

Objectif de la programmation objet

Approches

- La modularité : Structurer un logiciel en ensemble de modules.
- Un module = ensemble de fonctions spécifiques pour traiter un problème donné
- L'approche Objet : Un objet = Un module autonome qui encapsule des données et des fonctions de manipulation de ces données.

LES CLASSES SOUS PYTHON

- Une classe est un type permettant de regrouper dans la même structure : les informations (champs, propriétés, attributs) relatives à une entité ; les procédures et fonctions permettant de les manipuler (méthodes). Champs et méthodes constituent les membres de la classe.
- Remarques :
 - La classe est un type structuré qui va plus loin que l'enregistrement (ce dernier n'intègre que les champs)
 - Les champs d'une classe peuvent être de type quelconque
 - Ils peuvent faire référence à d'instances d'autres classes

LES CLASSES SOUS PYTHON

Termes techniques :

- « Classe » est la structure ;
- « Objet » est une instance de la classe (variable obtenue après instantiation) ;
- « Instantiation » correspond à la création d'un objet
- L'objet est une référence

Définition et implémentation d'une classe

- Une classe est définie par le mot-clé **class**.

```
class NomDeLaClasse :  
    # corps de la classe # ...
```

- NomDeLaClasse est le nom identifiant d'une classe (qu'on appelle aussi son identifiant). Par convention, il débute par une majuscule.
- Une classe porte en elle-même ses données et ses méthodes : c'est l'encapsulation.

Exemple :

```
class Rectangle():  
    """cette classe définit les rectangles """
```

```
rect1 = Rectangle()  
print rect1  
print rect1.__doc__
```

← Instanciation

← __doc__ est une méthode

Définition et implémentation d'une classe

Instanciación

Une instance d'une classe C désigne une variable de type C. Le terme instance ne s'applique qu'aux variables dont le type est une classe.

```
rect1=Rectangle()
```


Définition et implémentation d'une classe

Exemple

```
class Rectangle:
```

Définition de la classe Rectangle

```
    pass
```

La classe ne contient rien, on met le mot-clé **pass** qui ne fait rien

```
print(type(Rectangle))
```

```
rect=Rectangle()
```

Python nous indique qu'il s'agit d'un type

```
print(rect)
```

On crée une instance de la classe

```
test=isinstance(rect , Rectangle)
```

Affiche qu'il s'agit d'un objet de type Rectangle ainsi que son adresse en mémoire

Retourne vrai si rect est une instance de Rectangle

Définition et implémentation d'une classe : attribut d'instance et attribut de classe

Attributs d'instance : Une variable ou attribut d'instance est une variable accrochée à une instance et qui est spécifique à cette instance. Cet attribut n'existe donc pas forcément pour toutes les instances d'une classe donnée, et d'une instance à l'autre il ne prendra pas forcément la même valeur. On peut retrouver tous les attributs d'instance d'une instance donnée avec une syntaxe `instance.__dict__`

Exemple

```
class Rectangle:
```

```
    """cette classe instancie des rectangles"""
```

```
    pass
```

```
rect1 = Rectangle()
```

```
rect1.x=5
```

```
rect1.y=12
```

```
print ("longueur= ",rect1.x,"largeur=",rect1.y)
```

```
print(rect1.__dict__)
```

A EVITER

```
rect1 = Rectangle()
longueur= 5 largeur= 12
cette classe instancie des rectangles
{'x': 5, 'y': 12}
```

In [47]:

Définition et implémentation d'une classe : attribut d'instance et attribut de classe

- Attributs de classe :


Les attributs sont des variables qui sont associées de manière explicite à une classe. Les attributs de la classe se comportent comme des variables globales pour toutes les méthodes de cette classe.

- Un attribut de classe est un attribut qui sera identique pour chaque instance.
- Il n'y a pas d'attributs privés en Python, tout est toujours accessible.

Définition et implémentation d'une classe : attribut d'instance et attribut de classe

Exemple

```
class Rectangle():  
    """attribut de classe"""  
    longueur=0  
    largeur=0
```



Attributs de classe

```
rect1=Rectangle()
```

```
print("longueur rect1=",rect1.longueur)  
print("largeur rect1=",rect1.largeur)  
rect1.largeur=20  
print("nouvelle largeur  
rect1=",rect1.largeur)  
rect2=Rectangle()
```

```
print("longueur rect1=",rect2.longueur)  
print("largeur rect1=",rect2.largeur)
```

```
In [57]: runfile('C:/annee2019-2020/semestre2/me  
wdir='C:/annee2019-2020/semestre2/mes tps/TP4_ob  
longueur rect1= 0  
largeur rect1= 0  
nouvelle largeur rect1= 20
```


Définition et implémentation d'une classe :



attribut d'instance et attribut de classe

```
class Rectangle():  
    """attribut de classe"""  
    longueur=0  
    largeur=0
```

```
rect1=Rectangle()  
print("longueur rect1=",rect1.longueur)  
print("largeur rect1=",rect1.largeur)  
rect1.largeur=20  
print("nouvelle largeur  
rect1=",rect1.largeur)  
rect2=Rectangle()
```

```
print("longueur rect1=",rect2.longueur)  
print("largeur rect1=",rect2.largeur)  
rect2.couleur='jaune'  
print(rect1.couleur)
```

Attribut d'objet



```
File "C:/annee2019-2020/semestre2/mes tps/TP4  
in <module>  
    print(rect1.couleur)
```

Définition et implémentation d'une classe

Les méthodes :

- Les méthodes sont des fonctions qui sont associées de manière explicite à une classe.
- Les méthodes sont donc des fonctions définies dans une classe.
- Elles ont comme particularité un accès privilégié aux données de la classe elle-même.
- avec le concept d'encapsulation, un objet est utilisé via ses méthodes, peu importe pour l'utilisateur de connaître le contenu (le code) des dites méthodes.

syntaxe : objetInstancié.méthode()

Définition et implémentation d'une classe

Les méthodes : définition

Exemple :

```
class Point:
    x=0
    y=0
    def deplace(self, dx, dy):# méthode deplace(dx,dy)
        self.x = self.x + dx
        self.y = self.y + dy

    def affiche(self): # méthode affiche()
        print("x =", self.x, "y =", self.y)
```

```
a = Point()
a.x = 1
a.y = 2
a.affiche()
a.deplace(3, 5)
a.affiche()
```

```
In [61]: runfile('C:/annee2019-2020/semestre2/mes tp
x = 1 y = 2
```


Définition et implémentation d'une classe

Constructeur :

- Si lors de la création d'un objet nous voulons qu'un certain nombre d'actions soit réalisées (par exemple une initialisation), nous pouvons utiliser un **constructeur**.
- C'est une méthode sans valeur de retour qui permet d'initialiser et de définir les attributs d'une classe.
- Le constructeur d'une classe porte un nom imposé par le langage Python :
`__init__()`.
- Ce nom est constitué de **init** entouré avant et après par **__** (**deux fois** le symbole **underscore** qui est le tiret sur la touche 8).
- Cette méthode sera appelée lors de la création de l'objet.
- Le constructeur peut disposer d'un nombre quelconque de paramètres, éventuellement aucun.

Définition et implémentation d'une classe

Constructeur :

Syntaxe

```
def __init__(self, arg1=v1, arg2=v2, .. )  
    self.argument1 = arg1  
    self.argument2 = arg2  
    .....
```

- `__init__` est le constructeur de la classe;
- `self` est le premier argument : `self` rappel l'objet instancié
- `v1` et `v2` sont des valeurs par défaut optionnelles

Définition et implémentation d'une classe

Constructeur : Exemple

```
class classe1:  
    def __init__(self):  
        # pas de paramètre supplémentaire  
        print("constructeur de la classe classe1")  
        self.n = 1 # ajout de l'attribut n  
  
x = classe1() # affiche constructeur de la classe classe1  
print(x.n)    # affiche 1
```


Définition et implémentation d'une classe

Constructeur : Exemple

```
class classe2:
```

```
    def __init__(self, a, b):
```

```
        # deux paramètres supplémentaires
```

```
        print("constructeur de la classe classe2")
```

```
        self.n = (a + b) / 2 # ajout de l'attribut n
```

```
x = classe2(5, 9) # affiche constructeur de la classe classe2
```

```
print(x.n)        # affiche 7
```

Définition et implémentation d'une classe

Constructeur : Exemple

```
class Rectangle():
```

```
    """cette classe instancie des rectangles"""
```

```
    def __init__(self, dim1, dim2):
```

```
        self.longueur = dim1
```

```
        self.largeur = dim2
```

```
rect=Rectangle(12,15)
```

```
print(rect.largeur)
```


Définition et implémentation d'une classe

Les propriétés

Quelque soit le langage, pour la **programmation orientée objet**, il est préférable de passer par des propriétés pour changer les valeurs des attributs.

Bien que cela ne soit pas obligatoire, il existe une convention de passer par des **getter** (ou **accesseur** en français) et des **setter** (**mutateurs**) pour changer la valeur d'un attribut. Cela permet de garder une cohérence pour le programmeur.

Si on change un attribut, on peut également impacter d'autres attributs et les mutateurs permettent de faire cette modification une fois pour toute.

Définition et implémentation d'une classe

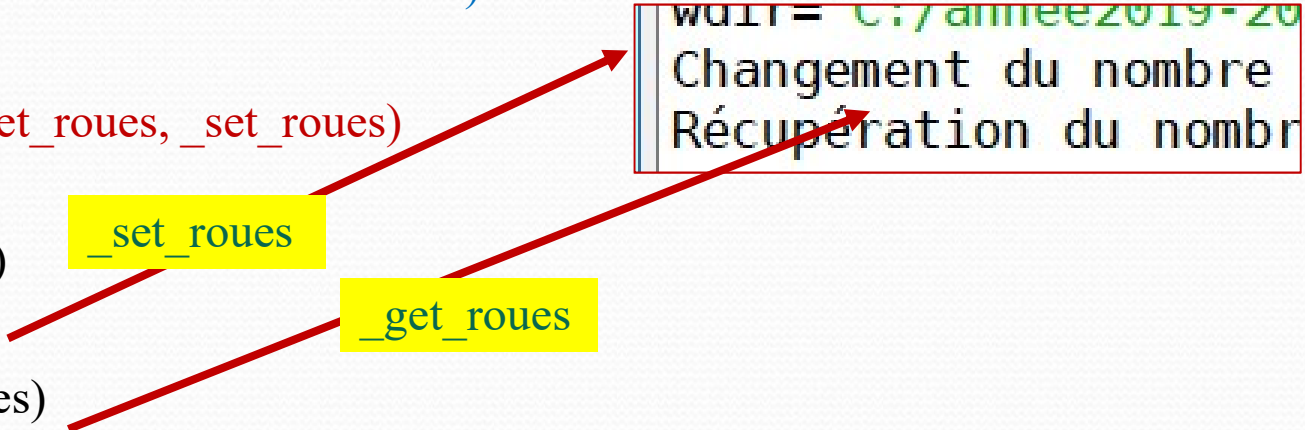
Exemple d'utilisation de propriétés:

```
class Voiture(object):  
    def __init__(self):  
        self._roues=4  
    def _get_roues(self):  
        print("Récupération du nombre de roues")  
        return self._roues  
    def _set_roues(self, v):  
        print("Changement du nombre de roues")  
        self._roues = v  
roues=property(_get_roues, _set_roues)
```

```
ma_voiture=Voiture()  
ma_voiture.roues=5  
print(ma_voiture.roues)
```

`_set_roues`

`_get_roues`



```
python C:/annee2019-20  
Changement du nombre  
Récupération du nombr
```

Méthodes spéciales

- On a déjà rencontré une méthode spéciale :

__init__

- Ces méthodes portent des noms prédéfinis, précédés et suivis de deux caractères de soulignement.
- Elles servent à:
 - initialiser l'objet instancié ;
 - modifier son affichage ;A
 - surcharger ses opérateurs ;
 -

Méthodes spéciales : `__repr__`

La fonction `repr()` donne une représentation textuelle de n'importe quel objet, sous forme de chaîne.

```
In [73]: repr(rect)
Out[73]: '<__main__.Rectangle object at 0x000001D68F1AA708>'
```

Pour contrôler cette conversion il faut définir une méthode spéciale `__repr__`

```
class Rectangle():
    """cette classe instancie des rectangles"""
    def __init__(self, dim1, dim2):
        self.longueur = dim1
        self.largeur = dim2
    def __repr__(self):
        return 'Rectangle({}, {})'.format(self.longueur, self.largeur)
rect=Rectangle(12,15)
print(repr(rect)) #Rectangle(12, 15)
```

Méthodes spéciales et opérateurs

On considère l'exemple :

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

```
p1 = Point(2, 4)  
p2 = Point(5, 1)  
p3 = p1+p2
```

Génère une erreur

p3 = p1+p2



```
p3 = p1+p2  
TypeError: unsupported operand type(s) for +
```

TypeError a été généré car Python ne savait pas comment ajouter deux objets Point ensemble.

Pour pouvoir le faire on doit surcharger les opérateurs.

Méthodes spéciales : surcharge d'opérateurs

La surcharge d'opérateurs permet de redéfinir la signification d'opérateur en fonction de la classe.

Cette fonctionnalité en Python, qui permet à un même opérateur d'avoir une signification différente en fonction du contexte, est appelée **surcharge d'opérateur**.

Méthodes spéciales : surcharge d'opérateurs

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):
        a = self.x + p.x
        b = self.y + p.y
        return Point(a, b)
```

```
p1 = Point(2, 4)
p2 = Point(5, 1)
```

```
p3 = p1+p2
print(p3)
```



(7.5)

Méthodes spéciales : surcharge

Le tableau suivant répertorie les opérateurs et leur méthode spéciale correspondante.

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>
Décalage binaire gauche	$p1 << p2$	<code>p1.__lshift__(p2)</code>
Décalage binaire droite	$p1 >> p2$	<code>p1.__rshift__(p2)</code>
ET binaire	$p1 \& p2$	<code>p1.__and__(p2)</code>
OU binaire	$p1 p2$	<code>p1.__or__(p2)</code>
XOR	$p1 \wedge p2$	<code>p1.__xor__(p2)</code>
NON binaire	$\sim p1$	<code>p1.__invert__()</code>

Méthodes spéciales : surcharge

Opérateurs de comparaison sont résumés ci-dessous.

Opérateur	Expression	Interprétation Python
Inférieur à	$p1 < p2$	<code>p1.__lt__(p2)</code>
Inférieur ou égal	$p1 \leq p2$	<code>p1.__le__(p2)</code>
Egal	$p1 == p2$	<code>p1.__eq__(p2)</code>
différent	$p1 \neq p2$	<code>p1.__ne__(p2)</code>
Supérieur à	$p1 > p2$	<code>p1.__gt__(p2)</code>
Supérieur ou égal	$p1 \geq p2$	<code>p1.__ge__(p2)</code>

Méthodes spéciales : surcharge

Exemple

Voir exercice 6 du TP4

Héritage

- L'*héritage* est le mécanisme qui permet de se servir d'une classe préexistante pour en créer une nouvelle qui possédera des fonctionnalités différentes ou supplémentaires.
- *Hériter* : Récupérer les attributs et méthodes d'une classe (la mère) dans une autre classe (la fille)
- *Surcharge* : redéfinition d'une méthode qui a été héritée.

Héritage

- En Python, lorsque l'on veut créer une classe héritant d'une autre classe, on ajoutera après le nom de la classe fille le nom de la ou des classe(s) mère(s) entre parenthèses :

Exemple :

```
class Mere1:
```

```
    # contenu de la classe mère 1
```

```
class Mere2:
```

```
    # contenu de la classe mère 2
```

```
class Fille1(Mere1):
```

```
    # contenu de la classe fille 1
```

```
class Fille2(Mere1 , Mere2):
```

```
    # contenu de la classe fille 2
```

la classe Fille1 hérite de la classe Mere1

**et la classe Fille2 hérite des deux classes
Mere1 et Mere2**

Héritage : exemple

```
class Rectangle:
```

```
    def __init__(self, long=0.0, larg=0.0, coul="blanc"):
```

```
        self.longueur = long
```

```
        self.largeur = larg
```

```
        self.couleur = coul
```

```
    def calcule_surface(self):
```

```
        return self.longueur * self.largeur
```

```
class Carre(Rectangle):
```

```
    """cette classe instancie des carrés"""
```

```
    def __init__(self, dim1):
```

```
        Rectangle.__init__(self,dim1, dim1)
```

```
c=Carre(10)
```

```
s=c.calcule_surface()
```

```
print(s)
```


Polymorphisme

- Si une **classe** hérite d'une autre classe, **elle hérite les méthodes de son parent.**

Exemple:

```
c=Carre(10)
```

```
s=c.calcule_surface()
```

```
print(s)
```

Méthode de la classe Rectangle



Polymorphisme

Il est cependant possible d' **écraser** la méthode de la classe parente en la redéfinissant. On parle alors de *surcharger une méthode* ou de **Polymorphisme**

```
class Carre(Rectangle):  
    """cette classe instancie des carrés"""  
    def __init__(self, dim1):  
        Rectangle.__init__(self, dim1, dim1)  
  
    def calcule_surface(self):  
        """Méthode qui calcule la surface."""  
        return self.largeur * self.largeur
```