

Python et la Programmation Orientée Objet

TANTINI FRÉDÉRIC

LABORATOIRE HUBERT CURIEN, Université Jean Monnet Saint-Étienne – France

Plan

- 1 POO - Généralités
 - Le concept d'objet
 - Les espaces de nom
- 2 L'objet en Python
 - Les classes génèrent des instances
 - L'héritage
 - Les classes sont aussi des exceptions
 - Quelques attributs et fonctions utiles
- 3 Sur les méthodes
 - Un peu de technique
 - Les méthodes réservées par Python
 - Décoration

Plan

- 1 **POO - Généralités**
 - Le concept d'objet
 - Les espaces de nom
- 2 **L'objet en Python**
 - Les classes génèrent des instances
 - L'héritage
 - Les classes sont aussi des exceptions
 - Quelques attributs et fonctions utiles
- 3 **Sur les méthodes**
 - Un peu de technique
 - Les méthodes réservées par Python
 - Décoration

Les objets du Réel

- Tout (ce que nous pouvons toucher) est objet
- Deux objets *identiques* sont issus d'une même fabrique (*classe*)
- Deux objets similaires ont les mêmes propriétés, les mêmes fonctions, mais peuvent avoir des attributs différents

Exemple

La voiture de X

- Couleur : noir
- Nb de portes : 5
- Avancer
- Tourner

La voiture de Y

- Couleur : bleu
- Nb de portes : 3
- Avancer
- Tourner

Les objets en programmation

- But : essayer de modéliser les objets du réel
- Une vraie chaise est composée d'atomes interagissant entre eux
- La définition de l'objet CHAISE dépendra de si l'on veut programmer un jeu, faire un magasin en ligne, . . .

- En tant qu'humain on regroupe les trucs qui ressemblent à des chaises sous l'étiquette *chaise*
- En programmation, on doit « définir » ce que doit être un OBJET CHAISE en premier

Utilité des objets

La ré-utilisabilité des *programmes*

Spécialisation/Généralisation (Héritage)

L'*objet* enfant *hérite* des propriétés de l'*objet* parent

Exemple

- La CLIO de X *est une* VOITURE : a 4 ROUES...
- La 306 de Y *est une* VOITURE
- Une VOITURE *est un* MOYEN DE TRANSPORT

Utilité des objets

La ré-utilisabilité des *programmes*

Composition

Utilisation d'objets pour définir d'autres objets

Exemple

L'objet VOITURE est composée de 4 objets ROUE, 3 objets ESSUIE-GLACE, 1 objet POT D'ÉCHAPPEMENT, ...

Utilité des objets

La ré-utilisabilité des *programmes*

Passage de *messages* à un objet (à partir d'un autre objet ou non)

Exemple

- Tourner la CLÉ fait démarrer le MOTEUR
- Appuyer sur la PÉDALE DE FREIN fait (généralement) ralentir la VOITURE

Utilité des objets

La ré-utilisabilité des *programmes*

L'encapsulation :

- les fonctionnalités internes de l'objet et ses variables ne sont accessibles qu'à travers des procédures bien définies
- évite l'utilisation de variables globales
- constructions des objets indépendantes

Définitions

Un objet-CLASSE est une fabrique d'objets-INSTANCE. Leurs attributs fournissent le comportement (données et fonctions) qui est hérité par toutes les instances générées par cette classe

Exemple

Fonction pour calculer le salaire d'un employé à partir de la paie et du nb d'heures travaillées

Un objet-INSTANCE représente ce qui est concrètement manipulé dans/par le programme. Leurs attributs enregistrent les données spécifiques à l'instance

Exemple

Numéro de sécurité social de l'employé

Ma première classe

Definition

```
class Prems:  
    x=33  
    def fonctionQuiFaitRien():  
        pass
```

Accès à un attribut : classe.attribut

Exemple

```
>>> Prems  
<class __main__.Prems at 0xb7dac11c>  
>>> type(Prems)  
<type 'classobj'>  
>>> dir(Prems)  
['__doc__', '__module__', 'fonctionQuiFaitRien', 'x']  
>>> Prems.x  
33
```

Mes premiers héritages

Definition

```
class Deuz(Prem):  
    pass  
  
class Troiz(Deuz):  
    x=44
```

Exemple

```
>>> dir(Deuz)  
['__doc__', '__module__', 'fonctionQuiFaitRien', 'x']  
>>> Deuz  
<class __main__.Deuz at 0xb7d9b14c>  
>>> dir(Troiz)  
['__doc__', '__module__', 'fonctionQuiFaitRien', 'x']  
>>> Troiz.x  
44
```

Plan

- 1 **POO - Généralités**
 - Le concept d'objet
 - **Les espaces de nom**
- 2 **L'objet en Python**
 - Les classes génèrent des instances
 - L'héritage
 - Les classes sont aussi des exceptions
 - Quelques attributs et fonctions utiles
- 3 **Sur les méthodes**
 - Un peu de technique
 - Les méthodes réservées par Python
 - Décoration

Les espaces de nom : rappel (?)

- Quand on utilise un nom dans un programme, Python crée, change, cherche dans un *espace de nom*
- L'endroit où est déclaré ce nom détermine la portée de la visibilité de ce nom
- Cherche *Localement*, puis dans les *Encapsulations*, au niveau *Global*, et enfin dans les fonctions *Built-in*

```
####thismod.py
var = 99

def local():
    var = 0                # changement local

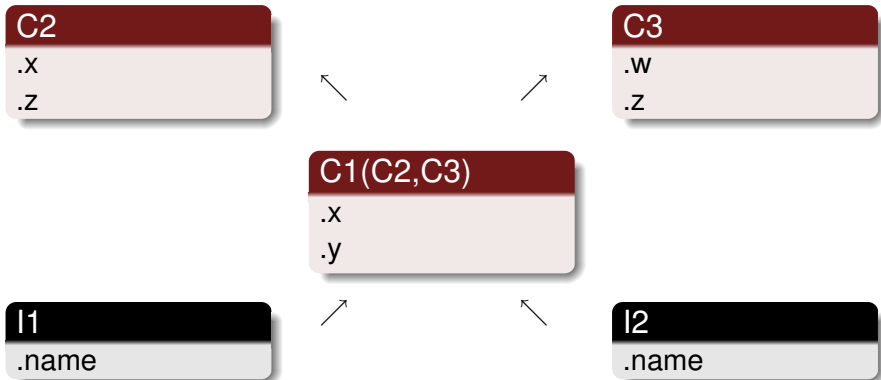
def glob1():
    global var             # déclaration globale
    var += 1              # changement global

def glob2():
    var = 0                # changement local
    import thismod         # s'importe lui-même
    thismod.var += 1      # changement global

def test():
    print var
    local(); glob1(); glob2();
    print var
```

```
>>> import thismod
>>> thismod.test()
99
101
>>> thismod.var
101
```


Espace de nom en objet



Espace de nom en objet

- C2 et C3 sont des « super-classes » de C1
- I1 et I2 sont des instances de C1

- Recherche dans l'objet, puis dans les classes, de bas en haut, de gauche à droite
- I2.w ? \rightarrow I2, C1, C2, C3
- I1.x = I2.x = C1.x
- I1.y = I2.y = C1.y
- I1.z = I2.z = C2.z

- Si w est une fonction, I2.w2() == C3.w(I2)

Exemple

```
class C2: pass
class C3: pass
class C1(C2,C3):
    def setname(self, qui): #methode
        self.nom = qui
I1=C1() #cree une instance de la classe C1
I2=C1() #cree une instance de la classe C1
```

```
>>> I1
<__main__.C1 instance at 0xb7d8578c>
>>> type(I1)
<type 'instance'>
>>> I1.setname('bob')
>>> I2.setname('garry')
>>> print I1.nom, I2.nom
bob garry
```

Constructeur

```
class C1(C2,C3):  
    def __init__(self, qui):  
        self.nom = qui  
I1=C1('bob')  
I2=C1('garry')
```

```
>>> dir(I1)  
['__doc__', '__init__', '__module__', 'nom']  
>>> I1.nom  
'bob'
```

Plan

- 1 POO - Généralités
 - Le concept d'objet
 - Les espaces de nom
- 2 L'objet en Python
 - Les classes génèrent des instances
 - L'héritage
 - Les classes sont aussi des exceptions
 - Quelques attributs et fonctions utiles
- 3 Sur les méthodes
 - Un peu de technique
 - Les méthodes réservées par Python
 - Décoration

Vu jusqu'à maintenant

Objet CLASSE vs. objet INSTANCE

- `class X` : pass → crée un objet CLASSE et lui donne un nom
- les affectations à l'intérieur de *class* définissent les attributs de la classe (les fonctions sont appelées *méthodes*)
- appeler une classe comme une fonction crée une nouvelle instance
- chaque instance hérite des attributs de la classe et possède son *propre* espace de nom
- le premier argument des méthodes est *self* (par convention) : utilisable sur l'instance, pas la classe

Définition d'une classe

Definition

```
class <NomAvecMajAuDebut> (Superclasse, ...):  
    donnee = valeur  
    def methode(self, ...):  
        self.membre=valeur2  
    ...
```

L'espace de nom

Classe différent de instance

```
class Espace:
    aa=33
    def affiche(self):
        print aa, Espace.aa, self.aa
    def enregistre(self, val):
        self.aa=val
```

```
>>> aa=12
>>> test=Espace()
>>> test.affiche()
12 33 33
>>> test.aa=44
>>> test.affiche()
12 33 44
>>> test.enregistre(55)
>>> test.affiche()
12 33 55
```


L'espace de nom

Instance1 différent des autres instances

```
class Espace:
    aa=33
    def affiche(self):
        print aa, Espace.aa, self.aa
    def __init__(self,val):
        self.aa=val
```

```
>>> aa=12
>>> test1=Espace(44)
>>> test2=Espace(55)
>>> test1.affiche()
12 33 44
>>> test2.affiche()
12 33 55
```

Plan

- 1 POO - Généralités
 - Le concept d'objet
 - Les espaces de nom
- 2 L'objet en Python
 - Les classes génèrent des instances
 - L'héritage
 - Les classes sont aussi des exceptions
 - Quelques attributs et fonctions utiles
- 3 Sur les méthodes
 - Un peu de technique
 - Les méthodes réservées par Python
 - Décoration

Sur la réutilisabilité

Definition

```
class <NomAvecMajAuDebut> (Superclasse, ...):  
    donnee = valeur  
    def methode(self, ...):  
        self.membre=valeur2  
    ...
```

- une classe hérite de tous les attributs de ses super-classes
- ses instances aussi
- une classe peut *remplacer* ou *étendre* les méthodes de ses *parents*
- elle peut en avoir des nouvelles

Sur la réutilisabilité

```
class Super:
    def methode(self):
        print 'Dans super'
    def delegate(self):
        self.action()
```

```
class Herite(Super):
    pass
```

```
>>> Herite().methode()
Dans super
```

Sur la réutilisabilité

```
class Super:
    def methode(self):
        print 'Dans super'
    def delegate(self):
        self.action()
```

```
class Remplace(Super): #polymorphisme
    def methode(self):
        print 'Dans la methode de Remplace'
```

```
>>> Remplace().methode()
Dans la methode de Remplace
```

Sur la réutilisabilité

```
class Super:
    def methode(self):
        print 'Dans super'
    def delegate(self):
        self.action()
```

```
class Etend(Super):
    def methode(self):
        print 'debut de methode dans etend'
        Super.methode(self)
        print 'fin de methode dans etend'
```

```
>>> Etend().methode()
debut de methode dans etend
Dans super
fin de methode dans etend
```

Sur la réutilisabilité

```
class Super:
    def methode(self):
        print 'Dans super'
    def delegue(self):
        self.action()
```

```
class Nouvelle(Super):
    def action(self):
        print 'en action'
```

```
>>> Nouvelle().delegue()
en action
```

Pas inutile d'ajouter à Super la méthode `action(self) : assert 0, 'doit définir action'`

Sur la ré-utilisabilité

Marche aussi en *cascade* :

```
class Mammifere:
    caract1="Allaite"

class Carnivore(Mammifere):
    c2="Mange de la viande"

class Chien(Carnivore):
    c3="Aboit"
```

```
>>> sparky=Chien()
>>> sparky.caract1,sparky.c2,sparky.c3
('Allaite', 'Mange de la viande', 'Aboit')
```


Sur la ré-utilisabilité

Une classe est un attribut d'un module

```
#mod1.py
class ClasseModule:
    x=12
    def truc(self):
        print "truc"
```

```
from mod1 import ClasseModule
class MaClasse(ClasseModule): pass
```

```
>>> dir(MaClasse)
['__doc__', '__module__', 'truc', 'x']
>>> import mod1
>>> dir(mod1)
['ClasseModule', '__builtins__', '__doc__', ...]
```

Plan

- 1 POO - Généralités
 - Le concept d'objet
 - Les espaces de nom
- 2 L'objet en Python
 - Les classes génèrent des instances
 - L'héritage
 - **Les classes sont aussi des exceptions**
 - Quelques attributs et fonctions utiles
- 3 Sur les méthodes
 - Un peu de technique
 - Les méthodes réservées par Python
 - Décoration

Comment lever une exception-classe

Definition

```
raise Class, inst
raise inst
```

Avec inst une instance de Class (ou d'une classe héritant de Class)

```
raise inst == raise inst.__class__, inst
```

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass
```

```
>>> for c in [B, C, D]:
...     try:
...         raise c()
...     except D:
...         print "err D"
...     except C:
...         print "err C"
...     except B:
...         print "err B"
...
err B
err C
err D
```

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass
```

```
>>> for c in [B, C, D]:
...     try:
...         raise c()
...     except B:
...         print "err B"
...     except C:
...         print "err C"
...     except D:
...         print "err D"
...
err B
err B
err B
```

Plan

- 1 POO - Généralités
 - Le concept d'objet
 - Les espaces de nom
- 2 L'objet en Python
 - Les classes génèrent des instances
 - L'héritage
 - Les classes sont aussi des exceptions
 - **Quelques attributs et fonctions utiles**
- 3 Sur les méthodes
 - Un peu de technique
 - Les méthodes réservées par Python
 - Décoration

Definition

Pour les classes :

- `__dict__` : un dictionnaire de l'espace de noms (lecture / écriture)
- `__name__` : le nom de la classe (lecture seule)
- `__bases__` : tuples de classes (les classes parentes) (lecture seule)
- `__doc__` : documentation de la classe (string OU None) (lecture / écriture)
- `__module__` : le nom du module dans laquelle la classe a été définie (lecture / écriture)

```
#mod1.py
class ClasseModule:
    x=12
    def truc(self):
        print "truc"
```

```
from mod1 import ClasseModule
class MaClasse(ClasseModule): pass
```

```
>>> ClasseModule
<class mod1.ClasseModule at 0xb7d0550c>
>>> MaClasse
<class __main__.MaClasse at 0xb7d0553c>
>>> MaClasse.__bases__
(<class mod1.ClasseModule at 0xb7d0550c>,)

```


Definition

Pour les instances :

- ceux des classes (héritage)
- `__dict__` : le dictionnaire de l'espace de noms de l'instance (lecture / écriture)
- `__class__` : nom de la classe de l'instance (lecture / écriture)

```
class C1:
    x=33
class C2:
    y=44
```

```
>>> c=C1()
>>> c.z=55
>>> dir(c)
['__doc__', '__module__', 'x', 'z']
>>> c.__class__=C2
>>> dir(c)
['__doc__', '__module__', 'y', 'z']
>>> c.__class__
<class __main__.C2 at 0xb7d5014c>
>>> C2
<class __main__.C2 at 0xb7d5014c>
```

Definition

- `isinstance`
- `issubclass`

```
class C2: pass
class C1(C2): pass
```

```
>>> c=C1()
>>> isinstance(c,C1),isinstance(c,C2)
(True, True)
>>> isinstance(C1,C2)
False
>>> issubclass(C1,C2),issubclass(C2,C1)
(True, False)
```

Plan

- 1 POO - Généralités
 - Le concept d'objet
 - Les espaces de nom
- 2 L'objet en Python
 - Les classes génèrent des instances
 - L'héritage
 - Les classes sont aussi des exceptions
 - Quelques attributs et fonctions utiles
- 3 **Sur les méthodes**
 - **Un peu de technique**
 - Les méthodes réservées par Python
 - Décoration

Ce qui se passe vraiment

```
class C:
    def f(self): print "Hello"
class D(C):
    def f(self): print "bye"
c=C()
d=D()
```

```
>>> D.f
<unbound method D.f>
>>> d.f
<bound method D.f of <__main__.D instance at 0xb7a9fdec>>
>>> d.f()
bye
>>> D.f()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unbound method f() must be called with D
instance as first argument (got nothing instead)
```

Ce qui se passe vraiment

```
class C:
    def f(self): print "Hello"
class D(C):
    def f(self): print "bye"
c=C()
d=D()
```

```
>>> D.__dict__['f'].__get__(c,D) ()
bye
>>> C.__dict__['f'].__get__(d,D) ()
Hello
>>> D.__dict__['f'].__get__(c,D) # D.f.__get__(c,D)
<bound method D.f of <__main__.C instance at 0xb7a9fd2c>>
>>> C.__dict__['f'].__get__(d,D) # C.f.__get__(d,D)
<bound method D.f of <__main__.D instance at 0xb7a9fdec>>
```

Ce qui se passe vraiment

```
class C:
    def f(self): print "Hello"
class D(C):
    def f(self): print "bye"
c=C()
d=D()
```

```
>>> if 1>2 : x=d.f
... else: x=c.f
...
>>> x()
Hello
>>> x
<bound method C.f of <__main__.C instance at 0xb7a9fd2c>>
```

Plan

- 1 POO - Généralités
 - Le concept d'objet
 - Les espaces de nom
- 2 L'objet en Python
 - Les classes génèrent des instances
 - L'héritage
 - Les classes sont aussi des exceptions
 - Quelques attributs et fonctions utiles
- 3 **Sur les méthodes**
 - Un peu de technique
 - **Les méthodes réservées par Python**
 - Décoration

__fctn__

Definition

- `__new__` : utilisée pour créer l'instance. Valeur de retour l'objet.
- `__init__` : appelée quand l'instance est créée
- `__del__` : appelée lorsque l'instance va être détruite

cf obj1.py

```
>>> a #ou >>> print a  
<__main__.Personne instance at 0xb7d3878c>
```

__fctn__

Definition

- `__repr__` : retourne une représentation valide d'un objet
- `__str__` : retourne une représentation d'un objet

```
class Personne:
    def __init__(self, nom, age=100):
        self.nom = nom
        self.age=age
    def __str__(self):
        return "<Personne s'appelant %s
        ayant %d ans>" % (self.nom,self.age)
```

```
>>> a=Personne('Albert')
>>> print a
<Personne s'appelant Albert
    ayant 100 ans>
>>> repr(a) # ou juste >>> a
'<__main__.Personne instance at 0xb7d4ea2c>'
```

__fctn__

Definition

- `__getattr__` : utilisé pour rechercher un attribut s'il n'est pas dans `__dict__`
- `__setattr__` : appelé lors de l'affectation d'un attribut

```
def __setattr__(self, name, value):  
    self.__dict__[name] = value
```

```
class C:  
    a=0  
    def __getattr__(self, name):  
        return "%s: Default" % name  
  
i = C()  
i.b = 1
```

```
>>> i.a,i.b,i.c  
(0, 1, 'c: DEFAULT')
```

__fctn__

Definition

Moults autres : `__cmp__`, `__len__`, `'__add__'`, `'__contains__'`, `'__eq__'`, `'__ge__'`, `'__iter__'`, `'__radd__'`, `__getitem__(self, key)` (Retourne `self[key]`), `__getslice__(self,i,j)` (Retourne `self[i :j]`), ...

cf `recap.py`

Variable Cachée

```
class C1:
    def meth1(self): self.X=88
    def meth2(self): print self.X

class C2:
    def methA(self): self.X=99
    def methB(self): print self.X

class C3(C1,C2):pass
```

```
>>> c=C3()
>>> c.meth1()
>>> c.methA()
>>> c.meth2()
99
>>> c.methB()
99
```

__variablePseudoCachée

Definition

```
__X = _NomClasse__X
```

```
class C1:
    def meth1(self): self.__X=88
    def meth2(self): print self.__X
class C2:
    def methA(self): self.__X=99
    def methB(self): print self.__X
class C3(C1,C2):pass
```

```
>>> c=C3()
>>> c.meth1()
>>> c.methA()
>>> c.meth2()
88
>>> c.methB()
99
```

__variablePseudoCachée

```
class C1:
    def meth1(self): self.__X=88
    def meth2(self): print self.__X
class C2:
    def methA(self): self.__X=99
    def methB(self): print self.__X
class C3(C1,C2):pass
```

```
>>> c.X
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: C3 instance has no attribute 'X'
>>> c.__X
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: C3 instance has no attribute '__X'
>>> c.__dict__
{'_C2__X': 99, '_C1__X': 88}
```

Caché

Rien n'est vraiment caché en Python. Ce n'est pas le cas de tous les autres langages orientés objets. De façon générale, une bonne pratique est de passer par fonction `get_maVar()` pour accéder aux attributs de l'instance (ou de la classe).

Exemple

```
class C:
    def get_X(self):
        print self.X
    def set_X(self, var):
        self.X=var
```

`__getattr__` et `__setattr__` peuvent également être utilisés pour simuler des variables privées.

Caché

Definition

```
class C(object):  
    __slots__=['age', 'nom']  
    ...
```

provoquera une erreur si on utilise autre chose que 'age' et 'nom' comme attribut (doit hériter de OBJECT ; slot *remplace* dict)

```
>>> c=C()  
>>> c.age=3  
>>> c.age  
3  
>>> c.truc=4  
Traceback (most recent call last):  
  File "<stdin>", line 1, in ?  
AttributeError: 'C' object has no attribute 'truc'
```

Plan

- 1 POO - Généralités
 - Le concept d'objet
 - Les espaces de nom
- 2 L'objet en Python
 - Les classes génèrent des instances
 - L'héritage
 - Les classes sont aussi des exceptions
 - Quelques attributs et fonctions utiles
- 3 **Sur les méthodes**
 - Un peu de technique
 - Les méthodes réservées par Python
 - **Décoration**

Déco

Definition

```
@dec2
@dec1
def func(arg1, arg2, ...):
    pass
```

est équivalent à

```
def func(arg1, arg2, ...):
    pass
func = dec2(dec1(func))
```

Utile pour débbugger, pour les méta-classes, ...

cf `recap2.py`