Algorithmique et Programmation

Arnaud Durand

L2 : MIASHS et Mathématiques, Paris-Diderot

2014

1

Programmation objet

Programmation objet

Si on prend le monde réel, nous sommes entourés d'objets : une chaise, une table, une route, une voiture, une personne etc. Ces objets forment un tout.

- Ils possèdent des propriétés (la chaise possède 4 pieds, elle est de couleur bleue, etc.).
- Ces objets peuvent faire des actions (la voiture peut rouler, klaxonner, etc.).
- Ils peuvent également interagir entre eux (l'objet conducteur démarre la voiture, l'objet voiture fait tourner l'objet roue, etc.).

Objets et instances

La définition de l'objet permet d'indiquer ce qui compose un objet $\Rightarrow$ ses propriétés, ses actions etc.
Ex : une chaise a des pieds et on peut s'assoir dessus.

Par contre, l'objet chaise est concret. On peut donc avoir plusieurs objets chaises : on parle également **d'instances**.

Chaque instance d'un objet a sa propre vie et est différent d'une autre. Nous pouvons avoir une chaise bleue, une autre rouge, une autre avec des roulettes, une cassée...

Encapsulation

Le fait de concevoir une application comme un système d'objets intéragissant entre eux apporte une certaine souplesse et une forte abstraction.
Ex : la machine à café : tout ce qui nous importe c'est que le fait de mettre des sous dans la machine nous permet d'obtenir un café. Peu importe comment cela fonctionne.

- Propriétés : Allumée/éteinte, présence de café, de gobelets, de sucre, ...
- Actions : AccepterMonnaire, DonnerCafé, ...

L'encapsulation protège donc les données contenues dans l'objet (pas d'accès au café ou à la monnaie) et son fonctionnement interne.

Exemples : listes, tuples, chaînes de caractères, dictionnaires, ...

En python : exemple

```
class Compte:
    # définition de la méthode spéciale __init__ (constructeur)
    def __init__(self,soldeInitial):
        # assignation de l'attribut d'instance solde
        self.solde = float(soldeInitial)
    #Méthodes
    def Credit(self,somme):
        self.solde += somme
        return self.solde

# l'instanciation de l'objet appelle la methode init
monCompte=Compte(25000)
monCompte.Credit(2000)
print monCompte.solde
>>>> 27000.0
```

En python : exemple

- `__init__` est appelé immédiatement après qu'une instance de la classe est créée. Ce n'est pas le constructeur (au sens Java) mais cela s'en rapproche.
- Le premier argument de chaque méthode de classe, est toujours une référence à l'instance actuelle de la classe. Par convention, cet argument est toujours nommé self.
- Dans la méthode `__init__`, `self` fait référence à l'objet nouvellement créé, dans les autres méthodes de classe, il fait référence à l'instance dont la méthode a été appelée. Non nécessaire au moment de l'appel.
- Une méthode `__init__` peut prendre n'importe quel nombre d'arguments et tout comme pour les fonctions, les arguments peuvent être définis avec des valeurs par défaut, ce qui les rend optionnels lors de l'appel.
- L'instanciation de classes en Python est simple et directe. Les classes s'appellent comme des fonctions avec les arguments de la méthode `__init__`. La valeur de retour sera l'objet nouvellement créé.

Plusieurs “constructeurs”

- Il ne peut y avoir qu'une méthode __init__ par classe.
- Pour avoir plusieurs initialisation possible (par copie, ...) il faut utiliser *args et **kwargs.

```
#!/usr/bin/env python

class Compte(object):
    def __init__(self,*args,**kwargs):
        if args!=():
            if len(args)==1:
                if type(args[0]) is int or type(args[0]) is float:
                    self.solde=float(args[0])
                if isinstance(args[0],Compte):
                    self.solde=args[0].solde
        if kwargs!={}:
            if "soldeinitial" in kwargs.keys():
                self.solde=float(kwargs["soldeinitial"])
            if "compte" in kwargs.keys():
                self.solde=float(kwargs["compte"].solde)
    def Credit(self,somme):
        self.solde+=float(somme)
        return self.solde
```

Plusieurs “constructeurs”

On a maintenant plusieurs possibilités d'instanciation de la classe Compte.

```
monCompte = Compte(25000)
monCompte1 = Compte(monCompte)
monCompte2 = Compte(soldeinitial=25000)
monCompte3 = Compte(compte=monCompte)
print monCompte.Credit(2000)
print monCompte1.Credit(2000)
print monCompte2.Credit(2000)
print monCompte3.Credit(2000)
```

Méthodes spéciales

Les méthodes spéciales sont appelées par python dans des cas prédéfinis.
__getitem__ : utiles lorsque les attributs de classe sont stockés sous forme de dictionnaire :

```
class Personne:
    """Classe permettant la modelisation d'une personne"""
    def __init__(self,prenom,nom,age):
        self.data={'prenom':prenom, 'nom':nom, 'age':age}
    def __getitem__(self,key):
        return self.data[key]

p = Personne("Emmanuel","Temam",40)
print p.__getitem__("age")
>>>> 40
print p["age"]
>>>> 40
```

Appel direct (à ne pas faire). Utiliser __getitem__ pour que Python fasse l'appel.

Méthodes spéciales

__setitem__

: même idée que __getitem__ mais en écriture.

```
class Personne:
    """Classe permettant la modelisation d'une personne"""
    def __init__(self,prenom,nom,age):
        self.data={'prenom':prenom, 'nom':nom, 'age':age}
    ...
    def __setitem__(self,key,value):
        self.data[key]=value

p = Personne("Emmanuel","Temam",40)
p["age"]=35
print p["age"]
>>>> 35
```

Cela nous permet de définir des classes qui se comportent en partie comme des dictionnaires, mais qui ont leur propre comportement dépassant le cadre d'un simple dictionnaire.

Méthodes spéciales

__str__ : permet de modifier l'affichage de la classe

```
class Personne:
    """Classe permettant la modelisation d'une personne"""
    def __init__(self,prenom,nom,age):
        self.data={'fn':prenom, 'ln':nom, 'age':age}
    ...
    def __str__(self):
        return "Personne %(prenom)s %(nom)s d'age %(age)d"
                    % self.data

p = Personne("Emmanuel","Temam",40)
print p
>>>> Personne Emmanuel Temam d'age 40
```

Cela nous permet de créer un affichage personnalisé de chaque classe créée.
Sans cette méthode on aurait :

```
print p
>>> <__main__.Personne instance at 0x801427d88>
```

Méthodes spéciales

__len__ : permet de définir la "longueur" d'une personne et d'utiliser la fonction len avec notre classe. Ici nous pouvons par exemple décider que la longueur d'une personne est son âge.

```
class Personne:
    """Classe permettant la modelisation d'une personne"""
    def __init__(self,prenom,nom,age):
        self.data={'fn':prenom, 'ln':nom, 'age':age}
    ...
    def __len__(self):
        return self.data["age"]
p = Personne("Emmanuel","Temam",40)
print len(p)
>>>> 40
```

Surcharges d'opérateur

D'autres méthodes spéciales peuvent être intéressantes, en particulier celles qui permettent de surcharger un opérateur :

- **__add__(self,other)** : surcharge de l'addition.
- **__sub__(self,other)** : surcharge de la soustraction.
- **__mul__(self,other)** : surcharge de la multiplication.
- **__div__(self,other)** : surcharge de la division.
- **__hash__(self)** : attribut un identifiant unique à une instance (méthode id, influe sur __eq,cmp__)
- **__eq,ne,lt,le,gt,ge(self,other)__** : surcharge de l'égalité, $!=$, $<>$, $<$, $<=$, $>$, $>=$.
- **__cmp__(self,other)** : surcharge de la comparaison si les méthodes ci-dessus ne sont pas codés.

Exemples opérations arithmétiques

```
class Complexe:
    def __init__(self,x,y):
        self.x=x
        self.y=y
    def __add__(self,other):
        return Complexe(self.x+other.x,self.y+other.y)
    def __sub__(self,other):
        return Complexe(self.x-other.x,self.y-other.y)
    def __mul__(self,other):
        return Complexe(self.x*other.x-self.y*other.y,self.x*other.
    y+self.y*other.x)
    def __div__(self,other):
     module = other.x**2+other.y**2
        real = (self.x*other.x+self.y*other.y)/module
        img  = (self.x*other.y-self.y*other.x)/module
        return Complexe(real,img)
    def __str__(self):
        if self.y < 0:
            return "%s - %s*i" % (str(self.x),str(abs(self.y)))
        else:
            return "%s + %s*i" % (str(self.x),str(self.y))
```

Exemples opérations arithmétiques

Avec cette classe le code suivant devient correct :

```
z1 = Complexe(2,3)
z2 = Complexe(1,4)
print z1+z2
>>>> 3 + 7*i
print z1-z2
>>>> 1 - 1*i
print z1*z2
>>>> -10 + 11*i
print z1/z2
>>>> 0.823529411765 + 0.294117647059*i
```

Exemples comparaisons

On peut également comparer deux nombres complexes de la manière suivante :

```
class Complexe:
    ....
    def __cmp__(self,other):
        if self.x==other.x and self.y==other.y:
            return 0
        else:
            if self.module() > other.module():
                return 1
            else:
                return -1
z1 = Complexe(2,3)
z2 = Complexe(1,4)
print z1 < z2
>>>> True
print z1 >= z2
>>>> False
print z1==Complexe(2,3)
>>>> True
```

Exemples iterateurs

Quelques définitions sur les itérateurs et les intégrables :

- Itérable - Un objet est dit itérable si la méthode __iter__ est définie.
- Iterateur - Un itérateur est un objet qui supporte le protocole iterator , ce qui signifie que les deux méthodes suivantes doivent être définies :
  - Il a une méthode __iter__ qui retourne lui-même.
  - Il a une méthode next (__next__ pour Python 3.x) qui retourne la valeur suivante chaque fois qu'elle est appelée.

Exemple : les listes :

```
>>> a=[1,2,3,4]
>>> a.__iter__
<method-wrapper '__iter__' of list object at 0x80140a0e0>
>>> a.next
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'next'
>>> ia=iter(a)
>>> ia.__iter__
<method-wrapper '__iter__' of listiterator object at 0x801431350>
>>> ia.next
<method-wrapper 'next' of listiterator object at 0x801431350>
```

Exemples iterateurs

```
class carte(object):
    def __init__(self, rang, couleur):
        FACE_CARD = {11: 'V', 12: 'D', 13: 'R'}
        self.couleur = couleur
        self.rang = rang if rang <=10 else FACE_CARD[rang]
    def __str__(self):
        return "%s%s" % (self.rang, self.couleur)

class jeu(object):
    def __init__(self):
        self.cartes = []
        for col in ['T', 'P', 'Ca', 'Co']:
            for r in range(1, 14):
                self.cartes.append(carte(r, col))
    def __iter__(self):
        return iter(self.cartes)

j = jeu()
for c in j.cartes: print c
for c in j: print c
```

Exemples iterateurs

```
class Jeu(object):
    .....
    def __iter__(self):
        return iterCartes(self)

class iterCartes:
    def __init__(self,j):
        self.couleur='Co'
        self.rang=0
    def next(self):
        if self.couleur=='Co':
            if self.rang==13:
                raise StopIteration
            else:
                self.couleur='T'
                self.rang+=1
        else:
            couleurs=['T','P','Ca','Co']
            self.couleur=couleurs[couleurs.index(self.couleur)+1]
        return Carte(self.rang,self.couleur)

j = Jeu()
for c in j: print c
```

Attributs et méthodes de classe

Un attribut ou une méthode de classe est une variable ou une fonction appartenant à la classe elle-même et non seulement aux instances. On définit les variables de classe avant la méthode __init__ .
Un attribut de classe peut être modifié en l'appellant précédé de la notation __class__ .

```
class Personne:
    count = 0
    def __init__(self,prenom,nom,age):
        self.__class__.count+=1
        ....
    ....
```

La variable count étant présente dans toutes les instances, elle permet de compter le nombre de personnes instanciées.

Attributs et méthodes privés

Comme la plupart des langages, Python possède le concept d'éléments privées :

- des fonctions privées, qui ne peuvent être appelées de l'extérieur de leur module ;
- des méthodes de classe privées, qui ne peuvent être appelées de l'extérieur de leur classe ;
- des attributs privés, qui ne peuvent être accédé de l'extérieur de leur classe.

Contrairement à la plupart des langages, le caractère privé ou public d'une fonction, d'une méthode ou d'un attribut est déterminé en Python entièrement par son nom.

Son nom doit commencer par _.

Un exemple global : les fractions - construction

```
class Fraction(object):
    def __init__(self,numerateur,denominateur):
        if type(denominateur) is not int:
            raise ValueError("Denominateur '%s' doit etre de type
    int" % str(denominateur))
        if type(numerateur) is not int:
            raise ValueError("Numerateur '%s' doit etre de type int
    " % str(numerateur))
        if denominateur==0:
            raise ValueError("Le denominateur doit etre different
    de 0")
        self.__datas=[numerateur,denominateur]
        self.reduce()
    def reduce(self):
        a,b=self.__datas
        if a<0 and b<0:
            self.__datas=[-a/pgcd(-a,-b),-b/pgcd(-a,-b)]
        if a<0 and b>0:
            self.__datas=[a/pgcd(-a,b),b/pgcd(-a,b)]
        if a>=0 and b<0:
            self.__datas=[-a/pgcd(a,-b),-b/pgcd(a,-b)]
        if a>=0 and b>0:
            self.__datas=[a/pgcd(a,b),b/pgcd(a,b)]
```

Un exemple global : les fractions - affichage

```
    def __str__(self):
        if self.__datas[0] < 0:
            chaine = "("
        else:
            chaine = ""
        if self.__datas[1]==1:
            chaine+= "%d" % self.__datas[0]
        else:
            chaine+= "%d / %d" % (self.__datas[0],self.__datas[1])
        if self.__datas[0] < 0:
            chaine += ")"
        return chaine
```

Un exemple global : les fractions - opérations (à droite)

```
    def convert(self,a):
        if type(a) is int:
            return Fraction(a,1)
        elif isinstance(a,Fraction):
            return a
        else:
            raise ValueError("'%s' ne peut etre converti en
    fraction" % a)
    def __pow__(self,b):
        if type(b) is not int:
            raise ValueError("L'exposant '%s' doit etre entier" % b
    )
        return Fraction(self.__datas[0]**b,self.__datas[1]**b)
    def __add__(self,other):
        f=self.convert(other)
        return Fraction(self.__datas[0]*f.__datas[1]+self.__datas
    [1]*f.__datas[0], self.__datas[1]*f.__datas[1])
    def __neg__(self):
        return Fraction(-self.__datas[0],self.__datas[1])
    def __sub__(self,other):
        f=self.convert(-other)
        return self.__add__(f)
```

Un exemple global : les fractions - opérations (à droite)

```
    def __mul__(self,other):
        f=self.convert(other)
        return Fraction(self.__datas[0]*f.__datas[0],self.__datas
    [1]*f.__datas[1])
    def __div__(self,other):
        f=self.convert(other)
        return Fraction(self.__datas[0]*f.__datas[1],self.__datas
    [1]*f.__datas[0])
```

Un exemple global : les fractions - opérations (à gauche)

```
    def __rmul__(self,other):
        return self.__mul__(other)
    def __rdiv__(self,other):
        return self.__div__(other)
    def __rsub__(self,other):
        return self.__sub__(other)
    def __radd__(self,other):
        return self.__add__(other)
    def __getitem__(self,index):
        if index ==0 or index == 1:
            return self.__datas[index]
        else:
            raise ValueError("index '%s' doit etre 0 ou 1 (0>
numerateur, 1>denominateur)" % index)
```

Un exemple global : les fractions - comparaisons et indexeur

```
    def __setitem__(self,index,value):
        if index ==0 or index == 1:
            self.__datas[index]=value
            self.reduce()
        else:
            raise ValueError("index '%s' doit etre 0 ou 1 (0>
    numerateur, 1>denominateur)" % index)
    def realvalue(self):
        return self.__datas[0]/float(self.__datas[1])
    def __cmp__(self,other):
        f = self.convert(other)
        if self.realvalue()==f.realvalue():
            return 0
        elif self.realvalue()<f.realvalue():
            return -1
        else:
            return 1
```

Un exemple global : les fractions - résultats

```
f1=Fraction(60,-18)
f2=Fraction(2,5)
## Operation entre fractions
print "%s + %s = %s" % (f1,f2,f1+f2)
print "%s - %s = %s" % (f1,f2,f1-f2)
print "%s * %s = %s" % (f1,f2,f1*f2)
print "%s / %s = %s" % (f1,f2,f1/f2)
## Operations avec des entiers
print "2 + %s = %s = %s + 2 = %s" % (f1,2+f1,f1,f1+2)
print "2 - %s = %s = %s - 2 = %s" % (f1,2-f1,f1,f1-2)
print "2 * %s = %s = %s * 2 = %s" % (f1,2*f1,f1,f1*2)
print "2 / %s = %s = %s / 2 = %s" % (f1,2/f1,f1,f1/2)
>>> (-10 / 3) + 2 / 5 = (-44 / 15)
>>> (-10 / 3) - 2 / 5 = (-56 / 15)
>>> (-10 / 3) * 2 / 5 = (-4 / 3)
>>> (-10 / 3) / 2 / 5 = (-25 / 3)
>>> 2 + (-10 / 3) = (-4 / 3) = (-10 / 3) + 2 = (-4 / 3)
>>> 2 - (-10 / 3) = (-16 / 3) = (-10 / 3) - 2 = (-16 / 3)
>>> 2 * (-10 / 3) = (-20 / 3) = (-10 / 3) * 2 = (-20 / 3)
>>> 2 / (-10 / 3) = (-5 / 3) = (-10 / 3) / 2 = (-5 / 3)
```

Un exemple global : les fractions- résultats

```
## indexeurs
f2[1]=8
print "f2[0]=%d, f2[1]=%d, f2=%s" % (f2[0],f2[1],f2)
## Comparaisons avec des entiers
f3=Fraction(4,2)
print "f1=%s, f3=%s, f1>0=%r. f1<=0=%r, f3==2=%r, f1!=1=%r" % (f1,
    f3,f1>0,f1<=0,f3==2,f1!=1)
## Comparaisons
print "f1=%s, f2=%s, f1>f2=%r. f1<=f2=%r, f1==f2=%r, f1!=f2=%r" % (
    f1,f2,f1>f2,f1<=f2,f1==f1,f2!=f1)
## Operations complexes.
print "2 * ( %s + %s ) = %s" % (f1,f2,2*(f1+f2))
>>> f2[0]=1, f2[1]=4, f2=1 / 4
>>> f1=(-10 / 3), f3=2, f1>0=False. f1<=0=True, f3==2=True, f1!=1=
    True
>>> f1=(-10 / 3), f2=1 / 4, f1>f2=False. f1<=f2=True, f1==f2=True,
    f1!=f2=True
>>> 2 * ( (-10 / 3) + 1 / 4 ) = (-37 / 6)
```

Héritage

l'héritage est une fonctionnalité objet qui permet de déclarer que telle classe (**la classe fille**) sera elle-même modelée sur une autre classe, qu'on appelle la **classe parente, ou la classe mère**.
Si une classe B hérite de la classe A, les objets créés sur le modèle de la classe B auront accès aux **méthodes et attributs** de la classe A.

```
class A(object):
    def __init__(self):
        self.attr_a="test"
class B(A):
    def __init__(self,attr_b):
        self.attr_b=attr_b

a=A()
b=B(2)
print b.attr_b
>>>> 2
print b.attr_a
>>>>Traceback (most recent call last):
>>>>  File "personne.py", line 45, in <module>
>>>>    print b.attr_a
>>>>AttributeError: 'B' object has no attribute 'attr_a'
```

Héritage

Lorsque Python essaye de créer l'objet B, il va appeler le constructeur de la classe B. Si celui-ci ne fait pas référence au constructeur de la classe A, la méthode A.__init__ n'est pas appelée.

```
class Personne(object):
    def __init__(self,prenom,nom,age):
        self.data={'prenom':prenom, 'nom':nom, 'age':age}
    def __str__(self):
        return self.__class__.__name__+" %(prenom)s %(nom)s d'age
    %(age)d" % self.data
    ....
class Etudiant(Personne):
    def __init__(self,prenom,nom,age,etuid):
        super(Etudiant,self).__init__(prenom,nom,age)
        self.data.update({'etuid':etuid})
    def __str__(self):
        return super(Etudiant,self).__str__()+" d'id %(etuid)d" %
    self.data
e = Etudiant("Toto","titi",22,214000233)
print e
>>>> Etudiant Toto titi d'age 22, d'id 214000233
```

Héritage

Plusieurs méthodes du même nom définies dans différentes classes ⇒ Laquelle appeler ?

1. Celle définie directement dans la classe dont est issu l'objet
2. Parcourt de la hiérarchie de l'héritage jusqu'à tomber sur la méthode.

Mais on peut aussi se servir de la notation MaClasse.ma_methode(mon_objet) pour appeler une méthode précise d'une classe précise.
Ex : si on enlève __str__ à la classe étudiant on obtient :

```
e = Etudiant("Toto","titi",22,214000233)
print e
>>> Personne Toto titi d'age 22
```

Deux fonctions

Python définit deux fonctions qui peuvent se révéler utiles dans bien des cas : issubclass et isinstance .

- issubclass : comme son nom l'indique, elle vérifie si une classe est une sous-classe d'une autre classe. Elle renvoie True si c'est le cas, False sinon.
- isinstance : isinstance permet de savoir si un objet est issu d'une classe ou de ses classes filles.

Polymorphisme

Soit la fonction :

```
def affiche(p):
    if isinstance(p,Personne):
        print p

p = Personne("Emmanuel","Temam",40)
e = Etudiant("Toto","titi",22,214000233)
```

Python va automatiquement sélectionner la bonne méthode __str__ :

```
Personne Emmanuel Temam d'age 40
Etudiant Toto titi d'age 22 d'id 214000233
```