Programmation Orientée Objet

C3P

Vincent Aranega
vincent.aranega@univ-lille.fr

Université de Lille

Table of contents

Table of Contents

C'est quoi la programmation orientée objet ?

Des idées ?

C'est quoi la programmation orientée objet ?

- Création d'artefacts "autonomes"
- Encapsulation
- Un objet représente un concept, une idée ou une entitée
- On met ensemble:
  - les données
  - les comportements associés
- L'état de l'application passe par les différents états des objets
- Paradigme de programmation
  $\hookrightarrow$ il s'agit d'une façon de penser la construction d'un logiciel basée sur certains principes fondamentaux et définis

Quelles conséquences ?

En pratique

- On créer des classes d'objets
- Un objet possède une structure interne et un comportement
- Un objet sait comment intéragir avec ses paires

En résumé

La programmation orientée objet met en avant la définition et l'**intéraction** de briques logicielles appelées **objets**.

Histoire de Python

- Langage développé à par Guido van Rossum à partir de 1989 aux Pays-bas
- Recherche sur les systèmes d'exploitation :
  - développé initialement pour simplifier la production de programmes pour l'OS Amoeba
  - typage dynamique
  - langage hybride

1989 Guido débute le développement de Python

1991 Python 0.9 est rendu publique

1994 Python 1.0 est distribué avec des éléments fonctionnels

2000 Python 2.0 est distribué

2008 Python 3.0 est distribué

Points clé de la programmation objet ?

- Système de classe (et/ou méta-classes)
- Héritage
- Polymorphisme
- Encapsulation

Et pour Python ?

- Héritage multiple
- Modèle objet à base d'attributs
- Attributs (presque) toujours publique (rarement d'accesseurs)
- Duck typing

Table of Contents

Rappels généraux

- Pas de séparateurs d'instructions, on revient à la ligne.
- Les commentaires commencent par #.
- L'instruction print permet d'afficher n'importe quel objet (fonction polymorphe).
- `https://docs.python.org/3/tutorial/introduction.html`

```
a = 1
b = a + 4
a, b = b, a
if b == 5:
  print('Value is', b)
else:
  print("I don't care")
```

Attention

Gestion des blocs par indentation

Rappels – list/tuple/dict

- Les listes sont mutables et peuvent contenir n'importe quel type d'éléments.
- Les tuples sont immutables.
- Les dictionnaires permettent d'associer n'importe quel valeur à une autre.

```
a = [1]
a.append(2)  # a == [1, 2], a[0] vaut 1 et a[1] vaut 2

b = (1, 2, 3)  # a[0] vaut 1, a[1] vaut 2 ...etc

c = {'a': 2, 'b': 3}
c['c'] = 4  # c['a'] vaut 2 ...etc
```

Rappels – boucles for

- Les boucles for itère sur des iterable .
- Beaucoup de type sont itérables (ex: les dicts, les tuples, string...etc)

```
c = {'a': 2, 'b': 3, 'c': 4}
for x in c:
  print(x)  # Iteration sur les clefs seulement

for k, v in c.items():
  print(k, v)  # Iteration sur les clefs valeurs
```

Création de classes

- La création de classe se fait avec le mot clef : class
- La méthode permettant d'init. les instances est __init__
- On accède à un attribut ou une fonction par la notation pointé.

```
class MaClass(object):
  pass

class Point(object):
  def __init__(self, x, y=0):
    self.x = x
    self.y = y

c1 = Point(x=3)
c2 = Point(x=3, y=4)
c3 = Point(4, 5)

print(c1.y)  # affiche 0
print(c3.x)  # affiche 4
```

Accès aux attributs

- Pas d'accesseurs ! En Python, on est entre adultes consentant
- Si on veut un attribut privé, la conventions veut qu'on le préfixe avec un _
- Si on veut vraiment masquer un attribut, il faut préfixer avec __ (il est rare de devoir avoir recours à ça en Python)

```
c1 = Point(x=3)
c2 = Point(y=4, x=3)
c3 = Point(4, 5)

print(c1.y)  # affiche 0
print(c3.x)  # affiche 4
```

Attributs dérivés

- possibilité d'avoir des attributs en lecture seule
- possibilité d'impacter la façon dont on lit ou écrit une donnée

```
class Person(object):
  def __init__(self, birth_year):
    self.year = birth_year

  @property
  def age(self):
    import datetime; now = datetime.datetime.now()
    return now.year - self.year

  @age.setter
  def age(self, new_age):
    import datetime; now = datetime.datetime.now()
    self.year = now.year - new_age

p = Person(2000)
print(p.age)  # Declanche la methode 'age'
p.age = 30  # Declanche la methode 'age' comme setter
p.year  # 1989 si l'annee courante est 2019
```

Variables de classes

- Les variables de classes sont partagées par toutes les instances
- On accède aux variables depuis une instance ou directement depuis la classe

```
class Point(object):
  default_x = 5
  default_y = 5

  def __init__(self):
      self.x = self.default_x
      self.y = self.default_y

  def addition(self, other):
    return Point(self.x + other.x, self.y + other.y)

p1 = Point()
p2 = Point()
p2.x = 10
p3 = p1.addition(p2)
assert p1.default_x == Point.default_x
```

Méthodes statiques et méthodes des classes

Attention

I y a une différence entre méthode statique et méthode de classes

```
class Spam(object):
  @classmethod
  def egg(cls, i):
    print("I know", cls, i)

  @staticmethod
  def bacon(i):
    print("I only know", i)

  def usage(self):
    self.egg(4)
    self.bacon(3)

Spam().usage()
Spam.egg(4)
Spam.bacon(5)
```

Classes et héritage multiple

- Python supporte l'héritage multiple

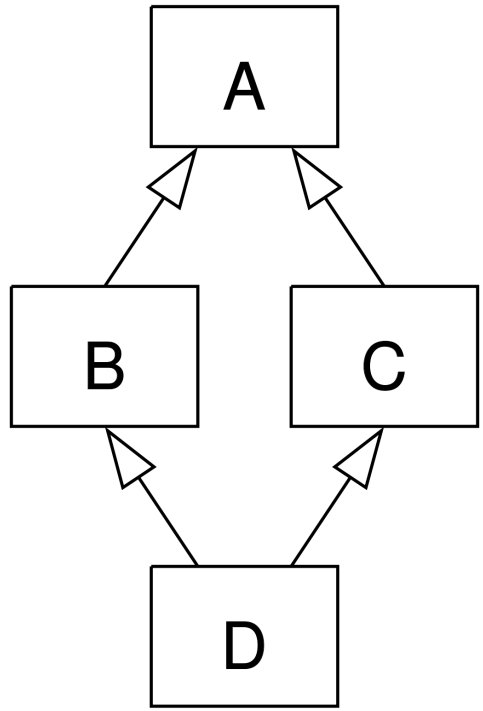


Figure: Problème du diamant

- Résolution par l'algorithme de linéarisation C3

Classes et héritage multiple

```
class A(object):
  def display(self):
    self.disp()

class B(A):
  def disp(self):
    print('In B')

class C(A):
  def disp(self):
    print('In C')

class D(B, C):
  pass

D.mro()  # returns (D, B, C, A)
d = D()
d.display()  # print what?
```

Surcharge de méthodes

- La surcharge de méthode se fait uniquement par rapport au nom de la méthode.

```
class A(object):
  def spam(self):
    cls_name = self.__class__.__name__
    print(f"I'm in spam from A, called from {cls_name}")

class B(A):
  def spam(self):
    print("I'm in spam from B")

class C(A):
  def spam(self):
    super().spam()
    print("Now I'm spam from C")

B().spam()
C().spam()
```

Typage dynamique et Duck Typing

- "If it walks like a duck and it quacks like a duck, then it must be a duck"
- On considère des protocols (équivalent d'interfaces) mais qui ne sont pas explicitement définis

```
class Duck(object):
  def fly(self):
    print("Duck flying")

class Airplane(object):
  def fly(self):
    print("Airplane flying")

class Whale(object):
  def swim(self):
    print("Whale swimming")

for animal in Duck(), Airplane(), Whale():
  animal.fly()
```

Table of Contents

Dunder methods (ou magic methods)

- Il est possible de surcharger/réutiliser les opérateurs et structures de base de Python
- Cela permet de donner des APIs plus fluide et simple à manipuler
- ATTENTION : Il ne faut pas en abuser si ce n'est pas nécessaire

```
class Point(object):
  def __init__(self, x=5, y=5):
    self.x = x
    self.y = y

  def __add__(self, other):
    return Point(self.x + other.x, self.y + other.y)

p1 = Point()
p2 = Point(y=6)
p3 = p1 + p2
```

Dunder methods – liste vraiment non exhaustive

```
__eq__(self, other)     self == other
__pos__(self)           +self
__neg__(self)           -self
__invert__(self)        ~self
__bool__(self)          bool(self)

__getattr__(self, name)       self.name
__setattr__(self, name, val)  self.name = val
__getattribute__(self, name)  self.name

__getitem__(self, key)        self[key]
__setitem__(self, key, val)   self[key] = val

__iter__(self)                for x in self
__contains__(self, value)     value in self
__call__(self [,...])         self(args)
```

Reflexion dans les langages

- Abilité d'un programme à s'examiner et à controler sa propre implémentation
- La reflexion est composé de deux "techniques":
  1. l'instropection : capacité d'un programme à examiner son propre état
  2. l'intercession : capacité d'un programme à modifier son propre état d'exécution ou d'altérer sa propre interprétation ou signification
- Pourquoi est-ce utile ?
  - Étendre un langage ou son design
  - Construire des environnements de programmation
  - Produire d'outils de développement avancés
  - Produire des applications s'auto-modifiant ou s'auto-optimisant

Accès reflexifs aux éléments d'un programme

- la fonction globals() retourne un dictionnaire des modules/classes/fonctions/variables du scope global.
- la fonction locals() retourne un dictionnaire des modules/classes/fonctions/variables du scope courrant.
- les éléments récupérés peuvent être utilisés directement

```
class A(object):
  pass

def foo(o):
  print(o)

locs = locals()
locs['foo'](3)

myclass = locs['A']
instance = myclass()
locs['A'] = type('B', [A], {'x': 0})
instance2 = A()
print(instance2.x)
```

Accès reflexifs aux éléments d'un objet

- Le module *inspect* permet d'aider à l'inspection d'éléments `https://docs.python.org/3/library/inspect.html`.
- il est possible d'utiliser la fonction vars (...) pour accéder aux éléments d'un objet.
- la fonction dir (...) donne l'intégralité des méthodes appelable depuis un élément.
- la dictionnaire `__dict__` permet d'accéder à tout les attributs explicite d'un élément.
- les méthodes sont des attributs que l'on peut appeler comme des fonctions (ils sont callable).
- code.inspect permet d'accéder à tout les attributs d'un élément.
- l'attribut `__class__` permet d'accéder à la classe d'un objet.

Accès reflexifs aux éléments d'un objet

```
class A(object):
  var1_int = None
  var2_float = None

  def printme(self):
    print("I'm an object", self, "in", self.__class__)
    print("var1", self.var1, "var2", self.var2)

def update_instance(inst, default):
  from inspect import isfunction
  for key, val in inst.__class__.__dict__.items():
    if not isfunction(val) and not key.startswith('__'):
      varname, vartype = key.split('_')
      new_val = default[vartype]
      setattr(inst, varname, new_val)

a, b = A(), A()
update_instance(a, {'int': 0, 'float': 1.0})
update_instance(b, {'int': 1234, 'float': 45.3})
a.printme()
b.printme()
```

Compilation at run time

- Il est possible de compiler du code lors de l'exécution du code.
- Une fois compilé, le code peut-être directement utilisé.

```
def gen_getter_for_classvar(cls, attributes):
  for name in attributes:
    fun_name = "get_{}".format(name)
    fun = """def {}(self):
  return self.{}
""".format(fun_name, name)
    loc, glob = {}, {}
    exec(compile(fun, '<string>', 'exec'), glob, loc)
    setattr(cls, fun_name, loc[fun_name])

class A(object):
  x = 0
  y = 0

gen_getter_for_classvar(A, ["x", "y"])
# Comment modifier pour l'appliquer a des instances ?
```

Typed Python

- Il est possible de typer explicitement les fonctions Python
- Les types sont conservé à l'exécution et ne sont pas analysé par Python (on peut y accéder par introspection)
- Utilisation par des projets annexes pour founir de la validation (ex: mypy)

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

  def addition(self, other: Point) -> Point:
    return Point(self.x + other.x, self.y + other.y)
```

Table of Contents

Tests avec `pytest`

- Il existe plusieurs framework de test en Python.
- PyTest est framework intéressant et très utilisés dans la communauté même par de très gros projets
- Les tests reposent sur l'utilisation du mot clef `assert`.

```
def test__point_initialisation_without_args():
  p = Point()
  assert p.x == 5
  assert p.y == 5

def test__point_initialisation_with_args():
  p = Point(x=6)
  assert p.x == 6
  assert p.y == 5

  p = Point(y=8)
  assert p.x == 5
  assert p.y == 8
```

Table of Contents

Pile à partir d'une liste

Les questions à se poser ?

- Quelles sont les entitées du système ?
- Quelles sont les méthodes que les entitées doivent gérer ?
- De manière générale, en OO, un diagramme de classe est souvent pratique à utiliser

Pile à partir d'une liste

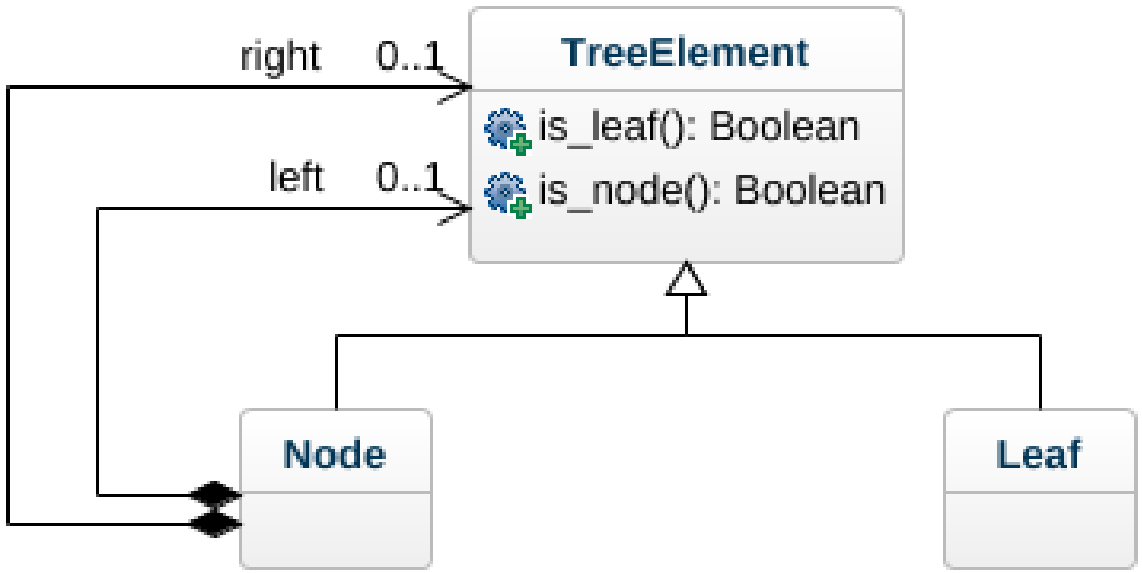
```
def test_stack_empty():
  s = Stack()
  assert s.is_empty() is True

def test_stack_push_peek():
  s = Stack(); s.push('a'); s.push(3)
  assert s.peek() == 3
  assert s.peek() == 3

def test_stack_push_pop():
  s = Stack(); s.push('a'); s.push(3)
  assert s.pop() == 3
  assert s.peek() == 'a'; assert s.pop() == 'a'
  assert s.is_empty() is True

def test_exception():
  s = Stack()
  with pytest.raises(EmptyStack):
    s.peek()
  with pytest.raises(EmptyStack):
    s.pop()
```

Pile à partir d'une liste

```
class EmptyStack(Exception): pass
class Stack(object):
  def __init__(self, l=None):
    self.inner_list = list(l) if l else []

  def is_empty(self):
    return self.inner_list == []

  def peek(self):
    try:
      return self.inner_list[-1]
    except IndexError:
      raise EmptyStack()

  def pop(self):
    i = self.peek()
    self.inner_list = self.inner_list[:-1]
    return i

  def push(self, item):
    self.inner_list += [item]
```

Arbre binaire

- Utilisation du pattern composite



Arbre binaire

- Utilisation du pattern composite

```
class TreeElement(object):
    def is_leaf(self): pass
    def is_node(self): pass

class Node(TreeElement):
  def __init__(self, value, left=None, right=None):
    self.left = left
    self.right = right
    self.value = value

    def is_leaf(self): ??
    def is_node(self): ??

  class Leaf(TreeElement):
    def __init__(self, value=None):
      self.value = value

    def is_node(self): ??
    def is_leaf(self): ??
```

Arbre binaire

Exercices

- Fonction d'affichage infix , prefix et postfix
- Fonction profondeur
- Fonction maximum
- Considérons arbre binaire de recherche, faire un fonction insertion

Table of Contents

Exemples – Ruby

```ruby
class Bibliotheque
   attr_reader :livres

   def initialize
     @livres = []
   end

   def ajouter livre
     raise "Already exist #{livre}" if @livres.include?
         livre
     @livres << livre
   end

   def taille
     @livres.length
   end

   def auteurs
     @livres.map { |livre| livre.auteurs }.flatten.uniq
   end
 end
```

Exemples – C#

```
public class Person {
    private string m_name;
    public string Name {
        get { return m_name; }
        set { m_name = value; }
    }

    public Person() { }

    public Person(string name, ushort age) {
        this.m_age = age;
        this.m_name = name;
    }

    ~Person() {
        Console.WriteLine("Destruction");
    }

    public void SayHi() {
        Console.WriteLine("Hello " + this.m_name);
    }
}
```

Exemples – Smalltalk

Table of Contents

Ressources

Liens intéressants

- Le blog de Sam&Max une référence sur Python en France (plutôt NSFW)
  `http://sametmax.com/`