S54MA2M7 : Informatique 2
Programmation Orientée Objet (en Python)

Luigi Santocanale
LIS, Aix-Marseille Université

21 janvier 2019

Plan

Plan

Paradigmes/styles de programmation

- procédural : Fortran, Pascal, C,
- déclaratif :
  - fonctionnelle : LISP, Haskell, Ocaml, ML,
  - logique : Prolog,
  - algèbre relationnelle : SQL
- Programmation Orienté Objet

NB : la plupart des langages modernes (**Python** inclus) permettent de faire recours (et de mélanger) ces paradigmes de programmation.

Origine de la POO

- Premiers langages : Simula (1960), SmallTalk (1970), Objective C, C++, Eiffel (1981) . . .
- Naissance de la programmation des interfaces graphiques
- Dans ce cadre, les types de données élémentaires (entiers, chaînes de caractères, . . .) ne sont pas suffisant
- Necessités de types ad hoc pour représenter :
  - des objets géométriques (triangle, ronds, . . .)
  - des objets interfaces (fenêtres, utilisateurs,. . .)
  - les liens entres ces objets non uniformes

Succès la POO

Réponses apportées par la POO dans le cadre de larges projets :

- modélisation (représenter des concepts clairement)
- mantainibilité du code
  - gestion de l'espace de noms (pas de variables globales)
  - cacher les détails d'implémentation (encapsulation)

Autres :

- Réutilisation et adaptation du code existant

C'est quoi un objet

- Packet de :
  - variables
  - fonctions (et/ou procédures)
- Donnée structurée :
  - les variables sont les *propriétés* ou *attributs*
  - les fonctions sont appelés *méthodes*
- L'ensemble des valeurs des propriétés peut se considérer comme l'état de l'objet, chaque objet a son état.

Des objets déjà connus

```
>>> str = "{0}--{1}"
>>> str.format(20, 30)
'20--30'
>>> l = list(range(1,10))
>>> l.append(11)
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 11]
>>> import numpy as np
>>> a = np.arange(6)
>>> a.resize((2, 3))
```

- `format` est une méthode de l'objet `str`
- `append` est une methode de l'objet `l`
- `resize` est une méthode de l'objet `a`

C'est quoi une classe (I)

Ensemble d'objets $\sim$ type de données

```
>>> type('abc')
<class 'str'>          #  'abc' est un objet de la classe 'str'
>>> type(1.)
<class 'float'>        #  1.0 est un objet de la classe 'float'
>>> type(1)
<class 'int'>           #  1 est un objet de la classe 'int'
>>> type([1,2,3])
<class 'list'>          # [1,2,3] est un objet de la classe 'list'
>>> import numpy as np
>>> a = np.arange(6) ; type(a)
<class 'numpy.ndarray'>
```

C'est quoi une classe (II)

- Ensemble d'objets ayant des propriétés et méthodes
  - similaires et/ou,
  - en commun
- On peut penser une classe comme un type de données.
- La définition d'une classe permet (en **Python**) de définir de types de données non élémentaires.
- Objet ou classe ? (Oeuf ou la poule ?)
  Avec certain langages (anciennes versions de **Javascript**) on peut avoir des objets sans avoir des classes.

Plan

Définition d'une classe, un premier exemple

```
class Point():
    """Classe pour les points sur le plan"""

    def __init__(self,x=0.,y=0.):
        self.x = float(x)
        self.y = float(y)

    def __str__(self):
        return '({},{})'.format(self.x,self.y)

    def moveRight(self,step=1.):
        self.x += step
    def moveLeft(self,step=1.):
        self.x += step
    def moveUp(self,step=1.):
        self.y += step
    def moveDown(self,step=1.):
        self.y += step
```

Exemple

- Proprietés des objets de la classe Point : `x`, `y`
- Methodes des objets de la classe Point :
  - `moveRight`,
  - `moveLeft`,
  - `moveUp`,
  - `moveDown`
- En plus : méthodes *special* :
  - `__init__` (constructeur) et
  - `__str__` (utilisé par `print`).

Remarques syntaxiques

- `class:` est une instruction composée.
  Donc, elle se termine par les deux points :
  et ouvre un bloc indenté.
- Convention :
  les noms des classes débutent par une majuscule.
- La définition de chaque méthode contient un paramètre spécial (l'objet) en première position.
  Par convention, ce paramètre est nommé `self`.
- Les noms des attributs/methodes spéciaux debutent et terminent par __
- Les noms des attributs/méthodes privés débutent par __
- Les noms des attributs/méthodes protégés débutent par _
- Pour l'instant, oubliez le "privé" et le "protégé"
  (**python** n'encourage pas le recours à ces techniques)

Utilisation d'une classe

```
>>> from point import Point
>>> p = Point() ## Création de l'objet et initialisation
>>> type(p)
<class 'point.Point'>
>>> p.moveUp()
>>> print(p)
(0.0,1.0)
>>> q = Point(3,4) ## Création avec paramètres
>>> q.moveRight()
>>> print(q)
(4.0,4.0)
>>> r = Point()
>>> r.moveDown(7.5)
>>> print(r)
(0.0,7.5)
```

- `p`, `q`, `r` sont des instances de la classe `Point`

Plan

Gestion de l'espace des noms

Mauvais exemple :

```
times, parent = {}, {}


def dfsRecursive(adjLists):
    vertices = list(adjLists.keys())

    for x in vertices:
        times[x] = {'discovery': None, 'finish': None}

    for x in vertices:
        if times[x]['discovery'] is None:
            dfsRec(adjLists, x, 1)
    return parent


def dfsRec(adjLists, root, i):
    global times, parent
    times[root]['discovery'] = i
    for x in adjLists[root]:
        if times[x]['discovery'] is None:
            parent[x] = root
            i = dfsRec(adjLists, x, i+1)
    i += 1
    times[root]['finish'] = i
    return i
```

Problèmes

Quand ce code est utilisé

- dans le cadre d'un grand projet (autres programmeurs)
- en tant que bibliothèque
  - utilisable par d'autres programmeurs,
  - avec d'autres bibliothèques (écrites par d'autres programmeurs)

la probabilité qu'un autre programmeurs utilise (et donc "écrase") les mêmes noms `parent` et `times` variables globales n'est négligeable.

Attention : les noms de fonctions

- `dfsRecurisive`, et `dfsRec`

posent le même problème.

Solution, en utilisant les classes

```
class Dfs:
    times, parent = {}, {}

    def search(self, adjLists, root=None):
        vertices = list(adjLists.keys())

        for x in vertices:
            self.times[x] = {'discovery': None, 'finish': None}

        if root is None:
            for x in vertices:
                if self.times[x]['discovery'] is None:
                    self.__searchRec(adjLists, x, 1)
        else:
            self.__searchRec(adjLists, root, 1)
        return (self.parent, self.times)

    def __searchRec(self, adjLists, root, i):
        self.times[root]['discovery'] = i
        for x in adjLists[root]:
            if self.times[x]['discovery'] is None:
                self.parent[x] = root
                i = self.__searchRec(adjLists, x, i+1)
                i += 1
                self.times[root]['finish'] = i
        return i
```

Remarques

- `parent` et `times` sont *locales*, à la classe `Dfs`.
- On peut les acceder de cette facon :
  `self.parent` et `self.times`
- En le préfixant par `__`, nous avons caché l'existence de la méthode `searchRec`.
  `__searchRec` est *privé* à la classe `Dfs`
  `__searchRec` n'appartient pas à l'*interface* de la classe `Dfs`

```
>>> from dfsRecClass import *
>>> dfs = Dfs()
>>> dfs.__searchRec(petersen,4,1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Dfs' object has no attribute '__searchRec'
>>> dfs.search(petersen,4)
({5: 4, ... }, {1: {'discovery': 3, 'finish': 17}, ...})
```

- On aurait pu faire de même pour `parent` et `times`
- On a pas des vrais objets, mais seulement une classe/module (à instancier une seule fois)

Compréhensions des espaces des noms

Que s'affiche à l'écran ?

```
1  class Test():
2      x = 'b'
3
4      def __init__(self):
5          self.y = 'c'
6
7      def createX(self):
8          self.x = 'd'
9
10 if __name__ == '__main__':
11     x = 'a'
12     print(x, end=' ')
13     test = Test()
14     print(test.x, end=' ')
15     print(test.y, end=' ')
16     test.createX()
17     print(test.x)
```

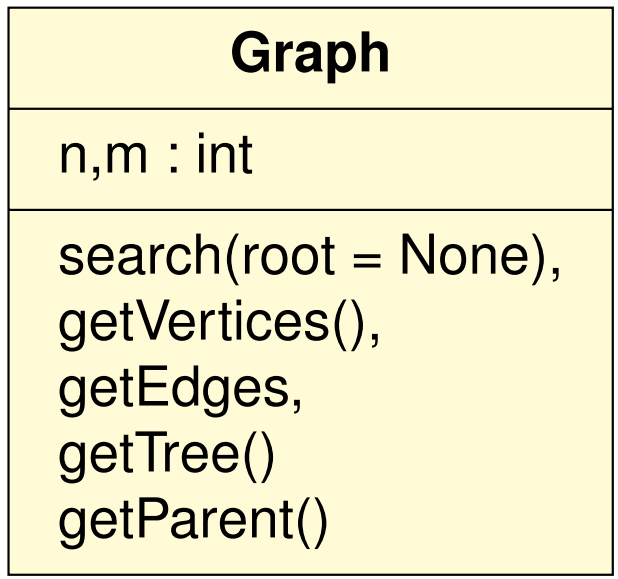
- `x` (ligne 11) est une variable globale
- `x` (ligne 14) est un attribut de la classe
- `y` (ligne 15) et `x` (ligne 17) sont des attributs des objets (attributs d'instance)

Encapsulation et masquage

- Encapsulation : désigne le principe de regrouper des données brutes avec un ensemble de routines permettant de les lire ou de les manipuler.
- Masquage des données brutes : s'assurer que l'utilisateur ne contourne pas l'interface qui lui est destinée.
- L'ensemble se considère alors comme une boîte noire ayant un comportement et des propriétés spécifiés.

Troisième version de `dfsRec` ($\rightsquigarrow$ `Graph`)

- On applique ces principes, encapsulation et masquage
- On masque la structure interne des données :
  - tous les attributs deviennent privés
  - on accédera aux attributs seulement par des méthodes faisant partie de l'interface (*publiques*)
  - on ne souhaite pas connaître que la structure de données utilisée sont des listes d'adjacence
- On specifie ainsi l'interface :



- Les attributs `parent` et `times` sont maintenant des attributs d'instance (et non de classe).

Troisième version de `dfsRec`

```
1  class Graph:
2      def __init__(self, adjLists):
3          self.__times, self.__parent = {}, {}
4          self.__adjLists = adjLists
5          self.__vertices = list(adjLists.keys())
6          self.n = len(self.__vertices)
7          self.m = sum([len(neighbours)for node,neighbours in
       ↪   self.__adjLists.items()])//2
8
9      def search(self, root=None):
10         self.__times, self.__parent = {}, {}
11         if root is None:
12             for x in self.__vertices:
13                 if x not in self.__times.keys():
14                     self.__searchRec(x, 1)
15         else:
16             self.__searchRec(root, 1)
17         return self
18
19     def __searchRec(self, root, i):
20         self.__times[root] = {}
21         self.__times[root]['discovery'] = i
22         for x in self.__adjLists[root]:
23             if x not in self.__times.keys():
24                 self.__parent[x] = root
25                 i = self.__searchRec(x, i+1)
26                 i += 1
27                 self.__times[root]['finish'] = i
28         return i
```

Troisième version de `dfsRec`

```
30      def getTree(self):
31          return self.__parent
32
33      def getTimes(self):
34          return self.__times
35
36      def getVertices(self):
37          return self.__vertices
38
39      def getEdges(self):
40          return [(x, y) for x in self.__vertices for y in
    ↪   self.__adjLists[x]]


60  if __name__ == '__main__':
61      petersenGraph = Graph(petersen)
62      print(petersenGraph.m)
63      petersenGraph.search()
64      print(petersenGraph.getTree())
65      petersenGraph.search(5)
66      print(petersenGraph.getTree())
67      print(petersenGraph.getEdges())
68      c3tree = Graph(c3).search(2).getTree()
69      print(c3tree)
```

Héritage (dérivation)

Mécanisme/principe fondamentale/fondante de la POO.

Permet de :

- définir une nouvelle classe à partir d'une (ou plus) classe(s) existante(s)
- réutiliser du code existant

et aussi

- construire un modéle hiérarchique du programme.

Héritage, par l'exemple

```
from point import Point

class Circle(Point):

    def __init__(self,x=0.,y=0.,radius=1.):
        Point.__init__(self,x,y)
        self.radius = float(radius)

    def __str__(self):
        return '({},{},{})'.format(self.x,self.y,self.radius)

    def shrink(self,scale):
        self.radius *= scale
```

Héritage, par l'exemple

- `Circle` hérite de `Point`
- Aussi, on dit que `Point` est la classe parent de `Circle`
- Cela veut dire :
  - tous les attributs de `Point` sont aussi des attributs de `Circle`
  - tous les méthodes de `Point` sont aussi des méthodes de `Circle`
  - on peut définir des nouveaux attributs et méthodes
  - on peut redéfinir des attributs/méthodes de la classe parent

Héritage, par l'exemple

- Proprietés des objets de la classe `Circle` : `x`, `y`, `radius`,
- Methodes des objets de la classe `Circle` :
  - `moveRight`, `moveLeft`, `moveUp`, `moveDown`,
  - `shrink`.
- Les méthodes `__init__` et `__str__` ont été redéfinis.

```
>>> from circle import Circle
>>> c = Circle(radius = 3)
>>> print(c)
(0.0,0.0,3.0)
>>> c.moveUp(1)
>>> c.shrink(2)
>>> print(c)
(0.0,1.0,6.0)
```

Remarques +- syntaxiques

- La specification d'heritage apparit ici :

```
class Circle(Point):
```

- On a pas à définir les attributs/méthodes hérités depuis la classe parent.
- Jusqu'à **python** 2.7 la definition d'une classe non héritée se faisait ainsi (example ave `Point`) :

```
class Point(object):
```

- La classe `object` est l'ancêtre de toutes les classes.

Polymorphisme

- Du grec : plusieurs formes.
- Une fonction est polymorphe si elle peut avoir des types de données (en input ou en output) différents.
- Exemple, la fonction identité : `lambda x : x`

Polymorphisme comme surcharge (overloading)

En POO : *surcharge*. Une fonction de même nom peut s'appliquer à des types de données différents.

Exemple :

```
>>> 1 + 1
2
>>> 'abc'+'efg'
'abcefg'
>>> import numpy as np
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> a + a
array([0, 2, 4, 6])
>>> len(a)
4
>>> len({1,2,3,4,5,5})
5
```

Polymorphisme comme redéfinition (overriding)

Exemple :

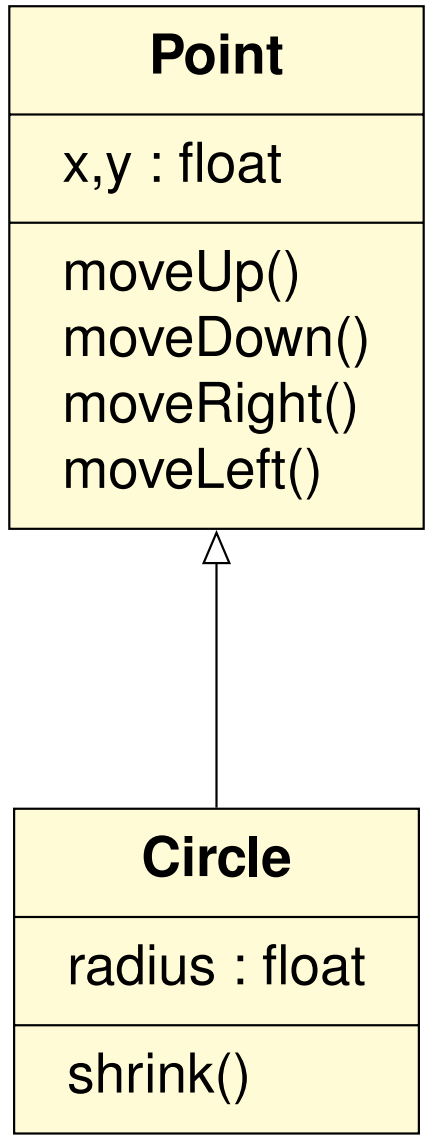
- Jeu d'échec.
- Classe de base : `Piece`, avec la méthode `mouvement()`.
- Classes héritant depuis `Piece` : `Roi`, `Reine`, `Fou`, `Cavalier`, `Tour`, `Pion`.
- Chaque classe dérivée redéfinit la méthode `mouvement()` : ainsi, cette méthode pourra effectuer le mouvement approprié en fonction de la classe de l'objet référencé au moment de l'appel.
- Le programme pourra donc utiliser `piece.mouvement()` sans avoir à se préoccuper de la classe de la pièce.

Plan

Diagrammes de classe UML

- Les bonnes normes de programmation suggèrent avoir une classe par fichier
- La structure de notre code devient complexe par la multiplication des fichiers.
- Il se rend nécessaire avoir un langage graphique pour “naviguer” le projet.
- UML (Unified Modeling Language) donne ces outils.
- Sujet d’un cours Génie Logiciel en Informatique.

Exemple


Point
x,y : float
moveUp()
moveDown()
moveRight()
moveLeft()
Circle
radius : float
shrink()

Un exemple moins simple