Méthodologie de la programmation

Chapitre 2

La programmation objet avec Python

Pablo Rauzy <pr@up8.edu>
pablo.rauzy.name/teaching/mp

La programmation objet avec Python

Cette séance

- Objectifs :
  - Donner un aperçu du paradigme de la programmation orientée objet.
  - Comprendre son intérêt, ses points forts.
- On va pour cela réutiliser le langage Python.

Programmation orientée objet

- Paradigme de programmation inventé par Ole-Johan Dahl et Kristen Nygaard au début des années 1960 et poursuivi par les travaux d'Alan Kay dans les années 1970.
- L'idée est de définir des *briques logicielles* appelées *objets*.
- Un objet représente un concept, une idée, ou toute entité du monde physique.
- Il possède une structure interne (*implémentation*) et un comportement (*interface*).
- Le but de cette méthode est double :
  - permettre une meilleure *modélisation* des problèmes, et donc une meilleure programmation,
  - obtenir des briques logiciel facilement réutilisables en dissociant l'implémentation de l'interface. On appelle cette pratique l'*encapsulation*.
- Explosion de la POO dans les années 80 et 90.

Différentes façons d'être orienté objet

- On peut programmer orienté objet dans n'importe quel langage.
- Certains langages offre un support natif de la POO :
  Smalltalk, Objective-C, Java, C++, Go, PHP, Python, Self, JavaScript, ...
- Mais pas tous de la même façon :
  - avec des classes qu'on instancie (C++, Java, Python) ;
  - avec des prototypes qu'on clone (Self, JavaScript) ;
  - d'autres choses moins clairement nommées (Go).
- Avec différentes façons de typer :
  - typage fort ou faible (Python vs. JavaScript) ;
  - typage statique ou dynamique (OCaml vs. Racket).

- Python supporte la programmation orientée objet à base de *classes*.
- Il est dynamiquement typé.

Classes

- Une classe est une structure qui peut aussi avoir des fonctions comme membre.
  On parle alors de *méthodes*.
- On appelle les variables membres des *attributs*.
- Comme pour les structures, on dit qu'on crée une *instance* d'une classe lors de la création d'un objet qui a cette classe pour type.

Méthodes

- Lorsqu'une méthode est appelée sur une instance, elle reçoit cette instance en argument.
  - Dans certains langages, le passage de cet argument est entièrement implicite, et son nom est alors le plus souvent le mot-clef `this`.
  - En Python, c'est le premier argument de la méthode et on le nomme généralement `self`, par convention.
- Il existe une méthode spéciale qu'on appelle le *constructeur* de la classe et qui est chargée de l'initialisation des attributs.
  - Parfois c'est la méthode qui a le même nom que la classe.
  - En Python, c'est la méthode qui s'appelle `__init__`.
  - C'est la méthode qui est appelée quand on *instancie* la classe.

Exemple

- Créons une classe qui représente une personne :

```
1  class Person:
2    def __init__ (self, age, name):
3      self.name = name
4      self.age = age
5
6    def presentation (self):
7      print("Bonjour je m'appelle " + self.name
8          + " et j'ai " + str(self.age) + " ans.")
9
10 jd = Person(22, "John Doe") # instanciation
11 jd.presentation()
12
13 jd.name = "Jane Doe"
14 print("Mon nouveau nom est " + jd.name + ".")
```

Encapsulation

- Le code qu'on vient de voir n'est pas bon !
- Une des forces de la programmation orientée objet est l'*encapsulation*.
- Il s'agit de cacher les détails de l'*implémentation* d'une classe à ses utilisateurs, et de leur offrir une *interface* stable.
- De cette façon,
  - l'implémentation peut changer sans que l'utilisateur ne s'en soucie,
  - et les objets sont seuls responsables du maintien de leur cohérence interne.

Exemple : changement d'implémentation, cassage d'interface

- Si on décide a un moment de changer l'implémentation, le code utilisateur de la classe peut casser :

```
1  class Person:
2    def __init__ (self, age, firstname, lastname = None):
3      if lastname is None:
4        firstname, lastname = firstname.split(" ")
5      self.firstname = firstname
6      self.lastname = lastname
7      self.age = age
8
9    def presentation (self):
10     print("Bonjour je m'appelle " + self.firstname + " "
11         + self.lastname + " et j'ai " + str(self.age) + " ans.")
12
13 jd = Person(22, "John Doe")
14 jd.presentation()
15
16 jd.name = "Jane Doe"
17 print("Mon nouveau nom est " + jd.name + ".")
```

- Pour mettre en œuvre l'encapsultation, la POO propose de restreindre l'accès aux membres d'une classe.
- Il existe trois niveaux de restriction :
  - publique : accès libre par tous,
  - privée : accès réservé aux méthodes de la classe propriétaire,
  - protégée : niveau intermédiaire que l'on verra plus tard en étudiant les notions d'*héritage* et de *hiérarchie* de classes.
- En Python ces notions ne sont pas directement supportées, mais il existe une convention :
  - quand son nom commence par __, un membre est privé,
  - quand son nom commence par _, un membre est protégé,
  - sinon, un membre est public.

- Pour cette raison l'interface d'une classe offre généralement des *accesseurs*.
- Un accesseur est une méthode qui permet de lire ou écrire dans un attribut.
- Cela permet à la classe de contrôler la bonne formation de ses objets, tout en offrant une interface stable aux utilisateurs.

Exemple

- Première version de la classe **`Person`** avec accesseurs :

```
1  class Person:
2    def __init__ (self, age, name):
3      self._name = name
4      self._age = age
5
6    def set_name (self, name):
7      self._name = name
8
9    def get_name (self):
10     return self._name
11
12   # …
13
14 jd = Person(22, "John Doe")
15 jd.presentation()
16
17 jd.set_name("Jane Doe")
18 print("Mon nouveau nom est " + jd.get_name() + ".")
```

Exemple

- Seconde version de la classe `Person` avec accesseurs :

```
1  class Person:
2
3    def __init__ (self, age, firstname, lastname = None):
4      if lastname is None:
5        firstname, lastname = firstname.split(" ")
6      self._firstname = firstname
7      self._lastname = lastname
8      self._age = age
9
10   def set_name (self, firstname, lastname = None):
11     if lastname is None:
12       firstname, lastname = firstname.split(" ")
13     self._firstname = firstname
14     self._lastname = lastname
15
16   def get_name (self):
17     return self._firstname + " " + self._lastname
18
19   # …
20
21 jd = Person(22, "John Doe")
22 jd.presentation()
23
24 jd.set_name("Jane Doe")
25 print("Mon nouveau nom est " + jd.get_name() + ".")
```

Héritage

- Une classe qui hérite d'une autre récupère ses membres et peut en définir de nouveaux ou redéfinir ceux qui sont hérités.
- En Python, une classe peut hériter de plusieurs classes.

- La classe héritante est appelée *classe dérivée* (ou *sous classe*).
- Les classes héritées sont appelées *classes de base directes* (ou *super classe*).
- Une *classe de base* d'une classe dérivée est soit une classe de base directe, soit une classe de base directe d'une classe de base.
- On parle d'*héritage simple* quand il n'y a qu'une seule classe de base directe.
- Sinon, on parle d'*héritage multiple*.

- Un·e étudiant·e est une personne, il partage donc les même attributs :

```
1  class Student:
2    def __init__ (self, age, firstname, lastname, year, department):
3      self._firstname = firstname
4      self._lastname = lastname
5      self._age = age
6      self._year = year
7      self._dept = department
8    def set_name (self, name): # …
9    def get_name (self): # …
10   def set_year (self, year):
11     self._year = year
12   def get_year (self):
13     return self._year
14   def presentation (self):
15     print("Bonjour je m'appelle " + self.get_name()
16         + " et j'ai " + str(self._age) + " ans.")
17     print("Je suis en " + self._year + " au département "
18         + self._dept + ".")
19
20 jd = Student(22, "Jane", "Doe", "L3", "PIF")
21 jd.set_name("John Doe")
```

- C'est donc une bonne idée d'utiliser l'héritage :

```
1  class Student (Person):
2
3    def __init__ (self, age, firstname, lastname, year, department):
4      Person.__init__(self, age, firstname, lastname)
5      self._year = year
6      self._dept = departement
7
8    def set_year (self, year):
9      self._year = year
10
11   def get_year (self):
12     return self._year
13
14   def presentation (self):
15     Person.presentation(self)
16     print("Je suis en " + self._year + " au département "
17         + self._dept + ".")
18
19 jd = Student(22, "Jane", "Doe", "L3", "PIF")
20 jd.set_name("John Doe")
```

Héritage et accessibilité des membres

- Il existe trois niveaux de restriction :
  - publique : accès libre par tous,
  - privée : accès réservé aux méthodes de la classe propriétaire,
  - protégée : comme privée mais accès possible aussi aux méthodes de classes dérivées.

- Si une classe D dérive de B alors l'interface de B est accessible sur les instances de D.
- On peut donc utiliser un D là où un B est prévu :

```
1  class Group:
2    def __init__ (self):
3      self._people = []
4
5    def add (self, pers):
6      self._people.append(pers)
7      print("Added: " + pers.get_name())
8
9    def remove (self, pers):
10     if pers in self._people:
11       self._people.remove(pers)
12       print("Removed: " + pers.get_name())
13
14 g = Group()
15 jd = Person(21, "John", "Doe")
16 js = Student(22, "Jane", "Smith", "L3", "PIF")
17 g.add(jd)
18 g.add(js)
```

- Parfois, on veut définir un *comportement* (i.e., une interface), mais sans spécifier son implémentation, en laissant le soin de le faire aux classes dérivées.
- Dans ce cas on peut définir ce qu'on appelle des *interfaces* ou *classes abstraites*
- Cette notion n'est pas directement supportée en Python non plus, mais il existe une convention : on fait remonter une erreur aux méthodes qui doivent être redéfinies :

```
1 class Animal:
2   def __init__ (self, name):
3     self._name = name
4
5   def speak (self):
6     raise NotImplementedError()
```

```
1 class Cat (Animal):
2   def speak (self):
3     print(self._name + ": Miaou !")
```

```
1 class Dog (Animal):
2   def speak (self):
3     print(self._name + ": Ouaf !")
```

- Une classe qui implémente toutes les méthodes abstraites de ses classes parentes est dite *concrète*.

1. Écrire une classe `StudentGroup` qui a comme attribut le niveau d'étude du groupe et qui vérifie à l'ajout que l'étudiant·e ajouté·e est bien du bon niveau, sinon affiche une erreur.
2. Écrire du code qui teste que la classe `StudentGroup` fonctionne comme attendu.
3. Écrire une classe abstraite `Shape` pour représenter des figures géométriques du plan, avec une position et la possibilité de se déplacer par translation, ainsi que deux méthodes donnant sa circonférence et son aire.
4. Écrire des classes concrètes `Circle` et `Square`.