Programmation orientée objet en Python

CPES2: Algorithmique et Applications

Pierre Senellart

24 septembre 2020

Plan

Paradigme de programmation

- Style d'approche de la programmation : comment exprimer la solution à un problème comme un programme informatique
- Différents paradigmes de programmation :

Impératif	le programme est une suite d'instructions exécutées les unes après les autres
Fonctionnel	le programme est une fonction mathématique
Objet	le programme est décrit par une collection d'objets, avec des propriétés et des méthodes
Logique	le programme est une combinaison de règles logiques de raisonnement
Évènementiel	le programme consiste en indiquer comment répondre à des événements
Synchrone	le programme décrit l'évolution, à chaque cycle d'horloge, de données de sortie en fonction des données d'entrée

...

Paradigme et langages de programmation

- Certains langages sont dédiés, ou particulièrement adaptés, à des paradigmes de programmation, par exemple :

Impératif	C, Pascal, FORTRAN 77
Fonctionnel	Haskell
Object	Smalltalk
Logique	Prolog
Évènementiel	Visual Basic
Synchrone	Lustre

- Mais la plupart des langages mélangent et encouragent plusieurs paradigmes de programmation
- Python : principalement impératif, avec des aspects majeurs de fonctionnel et d'objet – des extensions ou interfaces particulières utilisent de l'événementiel, du réactif, de la programmation logique...

Programmation orientée objet (POO)

- Un objet est une instance d'une classe bien définie, qui indiquent quelles sont ses propriétés et ses méthodes : tous les objets de la même classe ont les mêmes propriétés et méthodes
- Les classes sont organisées en une hiérarchie de super- et sous-classes, les sous-classes héritant des propriétés et des méthodes des super-classes
- Les classes permettent d'abstraire certains détails d'implémentation
- Une classe encapsule les données et les fonctions manipulant ces données, ce qui permet de cacher certaines données de l'utilisateur de la classe
- On utilise de manière uniforme des instances d'objets appartenant à différentes sous-classes de la même classe, qui se comportent de manière polymorphe

POO pure et impure

- Dans les langages de programmation objet purs (Smalltalk, Eiffel, Scala), tout ce qui est manipulé (en particulier toutes les valeurs du langage) est un objet, avec sa classe, etc.
- Dans les langages plus généralistes permettant la POO, dont Python (mais aussi C++, Java, JavaScript...), les objets s'utilisent de manière conjointe aux valeurs des types de base, aux fonctions, etc.

Utilité de la POO

La POO est particulièrement adaptée à la programmation sur des données complexes qui viennent avec leurs propres propriétés et méthodes :

- concepts issus du monde réel (véhicule, usine, personne, compte bancaire, etc.)
- éléments d'une interface graphique (bouton, menu, etc.)
- concepts mathématiques (ensemble, relation, graphe, etc.)
- structures de données algorithmiques (liste chaînée, tableaux, table de hachage, arbre de recherche, tas, pile, file de priorité, etc.)

Python : un bon langage pour apprendre la POO ?

- Python n'est pas un langage objet pur
- Même comparé à d'autres langages multi-paradigmes généralistes, la POO en Python n'est pas beaucoup mise en avant, et n'est pas très riche (peu de sucre syntaxique)
- Le fait que Python n'ait pas de typage fort rend difficile d'applications certains concepts de POO
- . . . donc ce n'est peut-être pas le meilleur cadre pour découvrir la POO
- Cela dit : la POO reste très utile (et utilisée) en Python, et la librairie standard de Python enrichit le langage de fonctions supplémentaires dédiées à la POO

Plan

Classes et objets : Concepts

- Une classe est la description informatique d'un type d'objet
- Un objet est défini comme une instance d'une classe
- Une classe indique en particulier :
  - les propriétés (ou champs, ou attributs) des objets de cette classe : des valeurs qui seront associées à chacun des objets de la classe
  - les méthodes des objets de cette classe : des fonctions qui peuvent être appliquées aux objets de la classe manipulent les propriétés
- Une classe peut aussi comporter des propriétés ou méthodes de classe (ou statiques) : des valeurs ou fonctions associées à l'ensemble de la classe, que partagent tous les objets de la classe et ne s'appliquent pas à un objet en particulier (moins utile et moins important)

Classes et objets : Exemple

Example

`Matrice` peut être une classe décrivant le concept mathématique de matrice. Un objet de type `Matrice`, aussi appelé une instance de la classe `Matrice` est la représentation informatique d'une matrice en particulier.

Une matrice peut par exemple avoir les propriétés suivantes :

- n1 : première dimension (un entier naturel)
- n2 : seconde dimension (un entier naturel)
- lignes : les lignes de la matrice (chacune une liste de nombres)

Une matrice peut par exemple avoir les méthodes suivantes :

- copie : pour produire une copie de la matrice
- addition : pour calculer le résultat de l'addition de la matrice avec une autre matrice de même dimension

Classes et objets : En Python

- Une classe est définie de la manière suivante :

  ```
  class MaClasse:
      ...
  ```

- À l'intérieur de la classe, on met :
  - des définitions de fonctions : les méthodes et méthodes statiques de la classe
  - des définitions de variables : les propriétés statiques de la classe
- Les méthodes prennent un premier argument supplémentaire, conventionnellement appelé `self`, qui fait référence à l'objet à laquelle on applique la méthode
- Les propriétés des objets de la classe ne sont pas explicitement déclarés, mais sont automatiquement créés dans les méthodes en affectant une valeur à `self.propriete`

Classes et objets : Exemple Python

```
class Matrice:
    # dimensions est une méthode
    # _n1 et _n2 sont deux propriétés
    def dimensions(self):
        return [self._n1, self._n2]
    # copy est une méthode
    def copy(self):
        from copy import deepcopy
        return deepcopy(self)

    # somme est une méthode statique
    def somme(a, b): return a+b
    # description est une propriété statique
    description = "Matrice"

print(Matrice.somme(35, 7))
print(Matrice.description)
```

Constructeur et destructeur : Concepts

- Quand on construit un objet d'une classe (on dit qu'on instancie la classe), on peut utiliser un constructeur pour spécifier quelles sont les valeurs initiales des propriétés de l'objet
- Quand un objet est détruit (parce qu'on ne l'utilise plus), on peut utiliser un destructeur pour spécifier des opérations à réaliser à la fin de la vie d'un objet

Constructeur et destructeur : En Python

- Le constructeur est une méthode spéciale dont le nom est `__init__`. On peut donner un nombre arbitraire d'arguments, qui seront utilisés pour construire l'objet.
- Pour construire un objet, on utilise la syntaxe `objet = Classe()`. Les paramètres éventuels du constructeur se placent entre les parenthèses.
- Le destructeur est une méthode spéciale dont le nom est `__del__` (on préfère parfois le nom *finaliseur*). Mais c'est peu utile en Python, car on n'a pas de garantie sur quand le constructeur va être appelé (dépend de la gestion de la mémoire). Dans d'autres langages (C++ par exemple), le destructeur est beaucoup plus important pour la gestion de la mémoire.

Constructeur et destructeur : Exemple Python

```python
class Matrice:
    # Constructeur
    # On initialise les propriétés _n1, _n2 et _lignes
    # avec les arguments du constructeur
    def __init__(self, n1, n2, valeur = 0):
        self._n1 = n1
        self._n2 = n2
        self._lignes = [[valeur]*n2 for i in range(n1)]

    # Destructeur (ou finaliseur)
    def __del__(self):
        print(f"Matrice {self._n1}x{self._n2} détruite")

# On crée une matrice 5,5 nulle
m = Matrice(5, 5)
# Le destructeur sera peut-être appelé à la fin du programme
```

Spécialisation, redéfinition : Concepts

- Dans une classe on peut redéfinir ou spécialiser des méthodes pré-existantes ou définies dans une super-classe (voir plus loin) pour avoir un comportement approprié pour les objets de la classe
- On peut également redéfinir le comportement des opérateurs du langage de programmation (`+`, `*`, `**`, `<`, `==`, etc.) pour pouvoir les utiliser avec des objets de la classe

Spécialisation, redéfinition : En Python

- Pour spécialiser une méthode, il suffit de la redéfinir dans la classe comme toute autre méthode
- Il existe en Python des méthodes magiques appelées automatiquement par Python dans certains contextes ; leur nom est toujours de la forme `__methode__`. On a déjà vu `__init__`, `__del__`, mais il y a par exemple aussi `__str__` qui est une méthode appelée quand on doit convertir un objet en chaîne de caractères (par exemple pour l'afficher)
- Le comportement des opérateurs peut aussi être redéfini avec des méthode magiques : `__add__` pour l'addition (à gauche), `__rmul__` pour la multiplication (à droite), `__getitem__` pour l'indexation par des indices (`[]`), etc.

Spécialisation, redéfinition : Exemple Python (1/2)

```
class Matrice:
    def __getitem__(self, i):
        return self._lignes[i]

    def __str__(self):
        result = ""
        for i in range(self._n1):
            for j in range(self._n2):
                result += str(self[i][j]) + " "
            if(i != self._n1 - 1):
                result += "\n"
        return result

# On modifie la case 2,3 de la matrice m
m[2][3] = 42
# On affiche la matrice m
print(m)
```

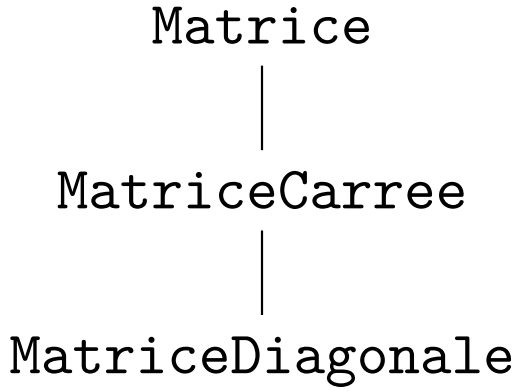
Spécialisation, redéfinition : Exemple Python (2/2)

```
class Matrice:
    def __rmul__(self, scalaire):
        copie = self.copy()
        for i in range(self._n1):
            for j in range(self._n2):
                copie[i][j] *= scalaire
        return copie

# On met dans n la multiplication de la matrice m par 3
n = 3*m
# On affiche la matrice n
print(n)
```

Héritage : Concepts

- Une sous-classe (ou classe dérivée) d'une classe est une classe décrivant un type d'objet plus spécifique que le type d'objet décrit par la super-classe (ou classe de base)
- Une sous-classe hérite des propriétés et méthodes de sa super-classe
- Une sous-classe peut spécialiser les méthodes de la super-classe
- Plusieurs sous-classes peuvent hériter de la même super-classe
- Une sous-classe peut hériter de plusieurs super-classes (mais ça rend l'héritage plus complexe)



Héritage : En Python

- On indique la ou les classes de base au moment de la définition de la sous-classe :

  ```
  class MaClasse(SuperClasse1, SuperClasse2):
  ```

- La spécialisation de méthodes fonctionne de la même manière que pour la redéfinition des méthodes par défaut ou des opérateurs
- Quand on redéfinit une méthode `ma_methode`, on peut utiliser `super().ma_methode` pour appeler la méthode du même nom dans la super-classe ; ça fonctionne également pour les méthodes magiques. En cas d'héritage multiple, fonctionne, mais plus complexe pour déterminer quelle méthode est appelée.
- On peut utiliser `isinstance(mon_objet, MaClasse)` pour tester si `mon_objet` a comme classe `MaClasse` ou une classe en-dessous de MaClasse dans la hiérarchie

Héritage : Exemple Python (1/2)

```
class MatriceCarree(Matrice):
    def __init__(self, n, valeur = 0):
        self._n = n
        super().__init__(n, n, valeur)

a = MatriceCarree(5, 2)
print(3*a)
```

Héritage : Exemple Python (2/2)

```
class MatriceDiagonale(MatriceCarree):
    def __init__(self, n, valeur = 0):
        self._n1 = n
        self._n2 = n
        self._n = n
        self._diagonale = [valeur] * n

    def __getitem__(self, i):
        return [0] * i + [self._diagonale[i]] \
                       + [0] * (self._n - i - 1)

    def __rmul__(self, scalaire):
        copy = self.copy()
        for i in range(self._n):
            copy._diagonale[i]*=scalaire
        return copy
```

Encapsulation : Concepts

- Les classes peuvent servir à encapsuler propriétés d'un objet et manières dont accéder et manipuler ces propriétés
- Il peut être utile de cacher certaines propriétés ou méthodes à l'utilisateur de la classe, pour éviter de l'exposer à des détails d'implémentation, ou pour éviter qu'il utilise l'objet de manière incorrecte
- Par exemple, on ne veut pas qu'un utilisateur de la classe `Matrice` puisse modifier dimension d'une matrice sans modifier le vrai nombre de lignes / colonnes de données
- Trois degrés de visibilité d'une propriété ou méthode :

  public accessible sans contrainte de l'extérieur de la classe

  protégé accessible uniquement à l'intérieur de la classe ou de ses sous-classes

  privé accessible uniquement à l'intérieur de la classe

Encapsulation : En Python

- Pas de réelle implémentation de l'encapsulation en Python, comparé à d'autres langages (C++, Java)
- Mais une convention à respecter dans le nommage des méthodes et propriétés (sauf méthodes magiques !) :

  public : nom commençant par une lettre
  protégé : nom commencent par un tiret bas (_)
  privé : nom commençant par deux tirets bas (__)

- Idem pour les méthodes et propriétés statiques
- Les noms privés sont vraiment cachés (en fait, renommés en `_Classe__methode`) en dehors de la classe ; mais pas de protection pour les noms protégés, juste une convention

Encapsulation : Remarque supplémentaire

- Une manière d'assurer l'encapsulation est d'interdire l'accès à toutes les propriétés et de forcer à passer par des méthodes spéciales (accesseurs) pour lire ou modifier le contenu des propriétés – courant dans des langages comme Java
- Utile si on doit vérifier des contraintes chaque fois avant d'autoriser les modifications de l'objet
- En Python, on peut aussi le faire (en déclarant les propriétés comme privées ou protégées et définissant des méthodes publiques) – mais pas toujours approprié
- On peut aussi le faire quasi-automatiquement avec le décorateur `@property`

Encapsulation : Exemple Python

```
class Matrice:
    @property
    def couleur(self):
        return self._couleur

    @couleur.setter
    def couleur(self, nouvelle_couleur):
        if nouvelle_couleur == 'rose':
            nouvelle_couleur = 'rouge'
        self._couleur = nouvelle_couleur

m = Matrice()
m.couleur = 'rose'
print(m.couleur)
```

Abstraction : Concepts

- Il est parfois utile de définir des classes abstraites : des classes qu'on ne veut pas qu'un objet instancie, mais qui peuvent avoir des sous-classes concrètes
- Les classes abstraites peuvent avoir des méthodes abstraites : des méthodes sans contenu (ou avec un contenu par défaut), qui ne peuvent pas être directement appelées, et qui doivent être spécialisées dans des sous-classes concrètes
- Utile pour spécifier des interfaces : des fonctionnalités qu'on veut que des classes dérivées implémentent, mais sans spécifier comment

Abstraction : En Python

- Pas de support de base dans le langage des classes abstraites, mais un support via le module `abc` (*abstract base class*)
- Après avoir fait

  ```
  from abc import ABC, abstractmethod
  ```

  il suffit de définir comme classe de base de notre classe abstraite la classe `ABC`, et d'indiquer le décorateur `@abstractmethod` devant les définitions de méthodes abstraites

Abstraction : Exemple Python

```
from abc import ABC, abstractmethod

class ObjetMathematique(ABC):
    @abstractmethod
    def taille(self):
        pass

class Matrice(ObjetMathematique):
    def taille(self):
        return self._n1*self._n2

class Ensemble(ObjetMathematique):
    def taille(self):
        return self._nb_elements

    ...
```

Polymorphisme : Concepts

- On peut manipuler des objets instanciant une classe dérivée d'une classe `MaClasse`, même abstraite, comme s'ils étaient des instances de la classe `MaClasse` (par exemple, en appelant des méthodes de `MaClasse` redéfinies dans les classes dérivées)
- En fonction de la classe réelle de l'objet, le comportement pourra être différent : on dit qu'on a un comportement polymorphe
- Par exemple, toute matrice carrée a un déterminant qu'on peut calculer ; si la matrice est en particulier diagonale (ou triangulaire), son déterminant est beaucoup plus facile à manipuler

Polymorphisme : En Python

- Dans les langages avec typage statique (comme Java), le polymorphisme veut dire qu'on peut manipuler un objet de type `ClasseDerivee` comme s'il avait pour type `ClasseDeBase`, mais que son comportement sera celui de `ClasseDerivee`
- Python étant un langage sans système de type statique, un objet sera toujours manipulé comme étant de son type – mais on peut par exemple l'envoyer à une fonction qui n'a pas besoin de connaître son type précis, tant qu'il peut être utilisé comme un objet d'un autre type (va même au-delà des hiérarchies d'héritage, *typage canard*)

Polymorphisme : Exemple Python (1/2)

```
class MatriceCarree(Matrice):
    def determinant(self):
        if self._n == 2 and len(self[0]) == 2:
            val = self[0][0] * self[1][1] - self[1][0] * self[0][1]
            return val
        total = 0
        for c in range(self._n):
            copy = self.copy()
            copy._n -= 1
            copy._n1 -= 1
            copy._n2 -= 1
            copy._lignes = copy._lignes[1:]
            for i in range(self._n - 1):
                copy._lignes[i] = copy[i][0:c] + copy[i][c+1:]
            signe = (-1) ** (c % 2)
            sub_determinant = copy.determinant()
            total += signe * self[0][c] * sub_determinant
        return total
```

Polymorphisme : Exemple Python (2/2)

```
class MatriceDiagonale(MatriceCarree):
    def determinant(self):
        from functools import reduce
        from operator import mul
        return reduce(mul, self._diagonale, 1)

a = MatriceCarree(5,1)
b = MatriceDiagonale(5, 2)
print(a.determinant())
print(b.determinant())
```

Plan

Références

- Tutoriel officiel Python, chapitre 9 (Classes)
  `https://docs.python.org/fr/3/tutorial/classes.html`
- Guide (en anglais) des méthodes magiques :
  `https://rszalski.github.io/magicmethods/`
- Cours sur la POO en Python (en anglais) : `https://www.python-course.eu/python3_object_oriented_programming.php`