Programmation Orientée Objet

Sommaire

Sommaire

Introduction

- En programmation, on doit inévitablement pouvoir définir et manipuler des structures de données plus complexes que celles nativement fournies par le langage choisi (entiers, flottants, chaînes de caractères, listes,...).
- La Programmation Orientée Objet, ou P.O.O., s'inscrit dans une démarche de conception différente de celle de la programmation dite *structurée*, typique par exemple du langage Pascal.
- On passe ainsi de l'«équation de Wirth» :

  $$\text{Algorithmes} + \text{Structures de données} = \text{Programmes}$$

  à l'équation qui peut caractériser la P.O.O. :

  $$\text{Méthodes} + \text{Données} = \text{Objet}$$

- L'idée maîtresse est de rendre indissociable, dès la conception, les données des algorithmes agissant sur celles-ci. En fait, les données présupposent elles-mêmes quel type d'algorithme est susceptible d'agir sur elles, et doivent de ce fait fournir ces algorithmes comme une interface (principe d'encapsulation).

- Bénéfices escomptés de la P.O.O. : réutilisabilité du code, maintenance facilitée, extensibilité accrue,...
- Il existe des langages de P.O.O. «pure», où les principes fondant la P.O.O. sont obligatoirement (syntaxiquement) respectés, par exemple Eiffel. Python propose le paradigme de la P.O.O. [1], mais ne l'impose pas.
- La P.O.O. demande d'assimiler un certain vocabulaire et une approche particulière de la programmation. Le cours de NSI ne fait qu'effleurer le sujet...
- On verra l'utilité de l'approche orientée objet quand on traitera des algorithmes sur les graphes et les arbres.

1. comme la plupart des langages de haut niveau généralistes, par exemple C++, Java, C#,...

Sommaire

La notion de classe

- La notion de classe généralise celle de *type*, en particulier celle de type défini par l'utilisateur, comme on peut la rencontrer dans certains langages [2]
- L'idée est de regrouper dans une même structure plusieurs informations qui caractérisent chaque objet d'une certaine famille d'objets. Les informations à regrouper peuvent être de natures différentes.
- Exemples :
  - chaque personne d'une entreprise est caractérisée par son nom, son prénom, sa date de naissance, son sexe, le poste qu'il occupe, son salaire annuel, son ancienneté,...
  - chaque livre d'une bibliothèque peut être caractérisé par son titre, son auteur, son éditeur, sa date d'impression, son nombre de pages, son emplacement dans la bibliothèque,...
  - chaque point de l'espace euclidien à trois dimensions muni d'un repère peut être caractérisé par ses trois coordonnées : x (abscisse), y (ordonnée) et z (côte).

2. Par exemple les `struct` en C et les `record` en Pascal.

- En Python :

```
class Personne:
    """une classe regroupant les informations sur une personne
    dans un lycée"""
    def __init__(self, nom, prenom, annee):
        self.nom = nom
        self.prenom = prenom
        self.annee = annee  # année de naissance

pers1 = Personne("Dupond", "Paul", 2003)
pers2 = Personne("Durand", "Pierre", 2002)
print(pers1.nom, pers1.prenom, pers1.annee)
print(pers2.nom, pers2.prenom, pers2.annee)
pers2.nom = pers1.nom
print(pers2.nom)
pers2.prenom = pers1.prenom
pers2.annee = pers1.annee
print(pers1.nom == pers2.nom and
      pers1.prenom == pers2.prenom and
      pers1.annee == pers2.annee)
print(pers1 == pers2)
```

- Bien repérer la syntaxe propre à Python : mot clef **class**, suivi du nom de la classe, suivi du caractère : (deux points), indentation du corps du code de la classe.

- La fonction `__init__` est un *constructeur*, automatiquement appelé à la création de l'objet (par l'instruction `pers1 = Personne(...)`), qui permet de créer et d'initialiser ses données (en passant leur valeur en argument au constructeur).
- Vocabulaire : on dit que `pers1` est une *instance* de la classe `Personne` et que `nom`, `prenom` et `annee` sont des *champs* (ou *attributs*, ou *propriétés*) de la classe `Personne`.
- Noter l'accès aux champs d'un objet de la classe `Personne` par le point [3] : `pers1.nom`
- Dans le constructeur, `self` est une référence à l'objet courant. Il n'y a pas d'ambiguïté entre `self.nom` et `nom`.
- Remarquer la convention de nommage [4], qui veut qu'on commence les noms de classe par une majuscule.
- Attention : comme pour les tableaux (listes), `pers1` contient une *référence* à l'objet construit.

Ex. 1 Prévoir ce que va afficher le code source précédent. Vérifier à l'aide d'un IDE ou en console.

3. On parle de *notation pointée*.
4. On parle de CamelCase en anglais.

Sommaire

Les méthodes d'une classe

- Les *méthodes* d'une classe sont les fonctions par lesquelles on va pouvoir manipuler les attributs des instances de cette classe.
- Le respect du principe d'encapsulation demande de ne manipuler les données (attributs) internes que par le biais de méthodes de la classe (y-compris pour la simple lecture ou écriture de ces données).
- Exemple :

```
class Personne:
    """une classe regroupant les informations sur une personne
    dans un lycée"""
    def __init__(self, nom, prenom, annee):
        self.nom = nom
        self.prenom = prenom
        self.annee = annee  # année de naissance

    def set_nom(self, nom):
        """modifie l'attribut nom"""
        self.nom = nom

    def get_nom(self):
        """récupère l'attribut nom"""
        return self.nom
```

```
pers1 = Personne("Dupond", "Paul", 2003)
print(pers1.get_nom())
pers1.set_nom("Dupont")
print(pers1.get_nom())
```

- Un autre exemple avec une méthode permettant de modifier les attributs d'une instance de la classe :

```
class Point:
    """Un point du plan repéré par son abscisse x et son ordonnée y"""
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def affiche(self):
        """affichage des coordonnées du point"""
        print("abscisse: " + self.x + ", ordonnée: " + self.y)

    def set_xy(self, x, y):
        """modification des coordonnées du point"""
        self.x = x
        self.y = y

    def deplace(self, dx, dy):
        """déplacement du point de dx en x et de dy en y"""
        self.x += dx
        self.y += dy
```

Sommaire

Ex. 2 On donne ci-dessous (page suivante) le code en Python d'une classe nommée `Toto`. Chaque objet de cette classe a 3 attributs de type entier et dispose de 3 méthodes. Le code d'une de ces méthodes n'est pas complètement écrit.

1. Écrire la ligne de code qui crée un objet de nom `toto` de la classe `Toto`, dont les attributs `att1`, `att2` et `att3` contiennent respectivement 1, 2 et 3.
   De même créer un objet de nom `titi` de la même classe et contenant 2, 1 et 6 dans ses attributs `att1`, `att2` et `att3`.
2.
   1. Quel est le nom de la méthode de la classe `Toto` qui échange les attributs `att1` et `att2` ?
   2. Compléter le code de la méthode `meth3` afin qu'elle prenne en argument l'entier `c` et qu'elle multiplie l'argument `att3` par `c`.
3. Écrire la ligne de code permettant, à l'aide des méthodes de la classe `Toto` d'échanger les attributs `att1` et `att2` de `titi`.
4. Écrire la ligne de code permettant, à l'aide d'une méthode de la classe `Toto` de multiplier par 2 l'attribut `att3` de `toto`.
5. Toutes les lignes de codes précédentes ayant été exécutées, que va afficher l'instruction `print(toto == titi)` ? Expliquer.

```python
class Toto:
    def __init__(self, i, j, k):
        self.att1 = i
        self.att2 = j
        self.att3 = k

    def meth1(self):
        tmp = self.att1
        self.att1 = self.att2
        self.att2 = tmp

    def meth2(self):
        return self.att3

    def meth3(self, c):
        # code à compléter
```

Ex. 3 Définir une classe `Fraction` pour représenter un nombre rationnel[5]. Cette classe possède deux attributs, `num` et `denom`, qui sont des entiers et désignent respectivement le numérateur et le dénominateur. On demande que le dénominateur soit plus particulièrement un entier strictement positif.

1. Écrire le constructeur de cette classe. Le constructeur doit lever une `ValueError` si le dénominateur fourni n'est pas strictement positif.
2. Ajouter une méthode `__str__` qui renvoie une chaîne de caractères de la forme `"12 / 35"`, ou simplement `"12"` lorsque le dénominateur vaut 1.
3. Ajouter des méthodes `__eq__` et `__lt__` qui reçoivent une deuxième fraction en argument et renvoient `True` si la première fraction représente respectivement un nombre égal ou un nombre strictement inférieur à la deuxième fraction.
4. Ajouter des méthodes `__add__` et `__mul__` qui reçoivent une deuxième fraction en argument et renvoient une nouvelle fraction représentant respectivement la somme et le produit des deux fractions.

5. Rappel : l'ensemble $\mathbb{Q}$ des nombres rationnels est l'ensemble des nombres de la forme $\frac{a}{b}$, où a et b sont deux entiers ($b \neq 0$).

4. Tester ces opérations. Bonus : s'assurer que les fractions sont toujours représentées sous forme réduite.

Ex. 4 Dans certains langages de programmation comme Pascal ou Ada, les tableaux ne sont pas nécessairement indexés à partir de 0. C'est le programmeur qui choisit sa plage d'indices. Par exemple, on peut déclarer un tableau dont les indices vont de -9 à 10 si on le souhaite. Dans cet exercice, on se propose de construire une classe `Tableau` pour réaliser de tels tableaux. Un objet de cette classe aura deux attributs, un attribut `premier` qui est la valeur de premier indice et un attribut `contenu` qui est un tableau Python contenant les éléments. Ce dernier est un vrai tableau Python, indexé à partir de `0`.

1. Écrire un constructeur `__init__(self, imin, imax, v)`, où `imin` est le premier indice, `imax` le dernier indice et `v` la valeur utilisée pour initialiser toutes les cases du tableau. Ainsi on peut écrire `t=Tableau(-10, 9, 42)` pour construire un tableau de 20 cases indexées de -10 à 9 et toutes initialisées avec la valeur 42.
2. Écrire une méthode `__len__(self)` qui renvoie la taille du tableau.

3. Écrire une méthode `__getitem__(self, i)` qui renvoie l'élément du tableau `self` d'indice `i`. De même , écrire une méthode `__setitem__(self, i, v)` qui modifie l'élément du tableau `self` d'indice `i` pour lui donner la valeur `v`. Les deux méthodes précédentes doivent vérifier que l'indice `i` est bien valide, et dans le cas contraire lever une exception de type `KeyError` avec un message explicite.
4. Enfin, écrire une méthode `__str__(self)` qui renvoie une chaîne de caractères décrivant le contenu du tableau.

Ex. 5 Définir une classe `Intervalle` représentant des intervalles de nombres. Cette classe possède deux attributs `a` et `b` représentant respectivement la borne inférieure et la borne supérieure de l'intervalle. On supposera que les bornes sont incluses dans l'intervalle [6]. Tout intervalle avec `b < a` représente l'intervalle vide.

1. Écrire le constructeur de la classe `Intervalle` et une méthode `est_vide` renvoyant `True` si l'intervalle est vide et `False` sinon.
2. Ajouter des méthodes `__len__`renvoyant la longueur de l'intervalle (l'intervalle vide a une longueur 0) et `__contains__` testant l'appartenance d'un élément `x` à l'intervalle.
3. Ajouter une méthode `__eq__` permettant de tester l'égalité de deux intervalles avec `==` et une méthode `__le__` permettant de tester l'inclusion d'un intervalle dans un autre avec `<=`. Attention : toutes les représentations de l'intervalle vide doivent être considérées égales et incluses dans tout intervalle.
4. Ajouter des méthodes `intersection` et `union` calculant respectivement l'intersection de deux intervalles et le plus petit intervalle contenant l'union de deux intervalles. Ces deux fonctions doivent renvoyer un nouvel intervalle sans modifier leurs paramètres.

6. Donc qu'il s'agit de l'intervalle *fermé* $[a; b]$.

5. Tester ces méthodes.

Ex. 6 Créer une classe nommée `CompteBancaire` qui représente un compte bancaire, ayant pour attributs `numeroCompte` (de type numérique), `nom` (nom du propriétaire du compte du type chaîne de caractères), `solde` (de type numérique).

1. Créer un constructeur ayant comme paramètres : `numeroCompte`, `nom`, `solde`.
2. Créer une méthode `versement(somme)` qui gère les versements. Cette méthode devra vérifier que `somme` est positif et lever une `ValueError` si ce n'est pas le cas.
3. Créer une méthode `retrait(somme)` qui gère les retraits. Cette méthode devra vérifier que `somme` est positif et et lever une `ValueError` si ce n'est pas le cas.

1. Créer une méthode `agios(nb_jours)` renvoyant le montant des agios à retirer du solde du compte si celui-ci est débiteur depuis `nb_jours`. On supposera que le montant des agios est égal au prorata annuel de 5 % du solde si celui-ci est débiteur, 0 sinon [7].
2. Créer une méthode `afficher()` permettant d'afficher les détails sur le compte.

7. Cette méthode n'est pas utilisable pratiquement, puisqu'il faudrait ajouter à une instance de `CompteBancaire` des données temporelles. Cela supposerait aussi d'associer une date à chaque opération (versement/retrait).

Ex. 7 Définir une classe `Angle` pour représenter un angle en degrés. Cette classe contient un unique attribut, `angle`, qui est un entier. On demande que, quoi qu'il arrive, l'égalité $0 \leqslant$ `angle` < 360 reste vérifiée.

1. Écrire le constructeur de cette classe.
2. Ajouter une méthode `__str__` qui renvoie une chaîne de caractères de la forme `60 degrés`. Observer son effet en construisant un objet de la classe `Angle`, puis en l'affichant avec `print`.
3. Ajouter une méthode `ajoute` qui reçoit un autre angle en argument (un objet de de la classe `Angle`) et l'ajoute au champ `angle` de l'objet. Attention à ce que la valeur d'`angle` reste bien dans le bon intervalle.
4. Ajouter deux méthodes `cos` et `sin` pour calculer respectivement le cosinus et le sinus de l'angle. On utilisera pour cela les fonctions `cos` et `sin` de la bibliothèque `math`. Attention, il faut convertir l'angle en radians (en le multipliant par $\pi/180$) avant d'appeler les fonctions `cos` et `sin`.
5. Tester les méthodes `ajoute`, `cos` et `sin`.

Ex. 8 Définir une classe `Date` pour représenter une date, avec trois attributs `jour`, `mois` et `annee`, chacun de type entier.

1. Écrire son constructeur. Celui veillera à ce que chaque attribut soit du type entier et dans l'intervalle adéquat. Dans le cas contraire il lèvera l'exception appropriée.
2. Ajouter une méthode `__str__` qui renvoie une chaîne de caractères de la forme `"8 mai 1945"`. On pourra se servir d'un *attribut de classe* qui est un tableau donnant les noms des douze mois de l'année. Tester en construisant des objets de la classe `Date` puis en les affichant avec `print`.
3. Ajouter une méthode `__lt__` qui permet de déterminer si une date `d1` est antérieure à une date `d2` en écrivant `d1 < d2`. La tester.

Ex. 9 Définir une classe `Chrono` pour représenter des temps mesurés en heures, minutes et secondes. Chaque objet de la classe `Chrono` a trois attributs : `heures`, `minutes` et `secondes`.

1. Écrire son constructeur. Veiller à ce que `heures` soit un entier compris entre 0 et 23 (inclus) et `minutes` et `secondes` des entiers compris entre 0 et 59.

2. Ajouter une méthode `__str__` qui renvoie une chaîne de caractères de la forme `"21h 34m 55s"`. Tester en construisant des objets de la classe `Chrono` puis en les affichant avec `print`.
3. Ajouter une méthode `avance` qui reçoit un nombre de secondes en argument et avance l'heure associée à l'objet de la classe `Chrono`.
4. Ajouter une méthode `__eq__` qui reçoit un deuxième chrono en argument et teste l'égalité des heures associées aux deux chronos.
5. Réimplémenter les *mêmes* méthodes de la classe `Chrono` sans les attributs `heures`, `minutes` et `secondes`, mais à l'aide d'un unique attribut `temps`.
6. Créer une classe `CompteARebours`, qui hérite de la classe `Chrono`, et qui ajoute une méthode `tac`, diminuant le chrono d'une seconde à chaque appel.
7. Redéfinir la méthode `__str__` pour que l'affichage d'un objet de la classe `CompteARebours` soit : `"plus que 5h 23m 14s"`.

Ex. 10 Le but de cet exercice est d'écrire un programme capable de résoudre le problème des huit reines. Il s'agit, sur un échiquier classique (à 8x8=64 cases) de parvenir à placer 8 reines de telle sorte qu'aucune reine ne se trouve attaquée par aucune autre. On peut dans un premier temps se contenter de trouver (à l'aide d'un algorithme) s'il existe une solution. Une version étendue est d'énumérer toutes les solutions. Une version généralisée, est de rechercher toutes les solutions pour N reines (sur un échiquier à $N \times N$ cases).
Deux remarques permettent de simplifier le problème :

- deux reines sont forcément sur deux colonnes distinctes. On pourra donc attribuer à chaque reine une colonne. Le problème est de placer chaque reine sur une ligne adéquate, de sorte qu'aucune reine n'en attaque aucune autre.
- Il faut et il suffit, pour être dans une situation qui résout le problème, qu'aucune reine ne se trouve en position d'être attaquée par une de ses «voisines» de gauche (c'est à dire les reines sur des colonnes de numéro inférieur, en supposant qu'on numérote les huit colonnes de gauche à droite)[8].

8. Pour le démontrer : la CN est évidente, quand à la CS il suffit de raisonner par l'absurde.

On se propose de résoudre ce problème avec une approche orientée objet et récursive.

1. Écrire le constructeur d'une classe `Reine` ayant trois attributs : `ligne`, `colonne` et `voisine` : le premier est la colone de la reine courante (qui ne changera pas), le second est sa ligne, le dernier est le numéro de la ligne de sa voisine située immédiatement à sa gauche (pour la reine sur la première colonne, cet attribut sera égal à `None`). Le constructeur devra initialiser systématiquement à 1 l'attribut `ligne` de la reine.
2. Écrire une méthode `peutAttaquer(ligne, colonne)`. Cette méthode doit renvoyer `True` si la reine, ou l'une de ses voisines de gauche, peut attaquer la position repérée par la ligne `ligne` et la colonne `colonne`. Sinon, elle renvoie `False`.

3. Écrire une méthode `avance()`, sans arguments, qui doit renvoyer `True` s'il est possible d'avancer la reine courante, ou l'une de ses voisines de gauche, jusqu'à trouver une configuration «acceptable» (dans laquelle la reine courante et aucune de ses voisines de gauche ne s'attaquent). Si ce n'est pas possible, `avance()` doit renvoyer `False`. Dans le cas où c'est possible, `avance` doit modifier l'état de la (ou des) reine(s) jusqu'à cette configuration acceptable.
4. Écrire une méthode `trouveSolution()`, sans arguments, chargée de renvoyer `True` si la configuration actuelle de la reine courante et de ses voisines de gauche est acceptable ou s'il en existe une obtenue en avançant une (ou plusieurs) reine(s) verticalement. Sinon, `trouveSolution` doit renvoyer `False`. On remarquera que les méthodes `avance()` et `trouveSolution` peuvent être définies par des appels récursifs mutuels.
5. Écrire un programme principal qui construit les huit reines et détermine la première solution acceptable et l'affiche.