NSI Terminale - Données

Programmation objet : résumé

qkzk

2020/07/09

Introduction à la programmation objet : Les grands principes

langage à objet

Alan Kay *SmallTalk*

- tout est objet
- chaque objet a un **type**
- chaque objet a sa propre mémoire, constituée d'autres objets
- tous les objets d'un type donné peuvent avoir les mêmes messages
- un programme est un regroupement d'objets qui interagissent par **envoi de messages**

type

c'est quoi un **type** ?

booléen, entier, Competiteur, Temps, List, Combattant, Voiture etc.

un **type** de données définit

- l'ensemble des valeurs possibles pour les données type
- les opérations applicables sur ces données

classes

une **classe** est un type d'objet

une classe définit

- la liste des **méthodes** et les traitements associés
  le comportement des objets
- la liste des **attributs** nécessaires à la réalisation des traitements
  l'**état** des objets
- les méthodes portent les traitements (comportement, actions)
- les attributs portent les données

classe = définition d'un modèle pour les objets de la classe
classe = abstraction (on programme des définitions)

instance

- une classe permet de **créer** des objets

on appelle **instance** un objet créé par une classe
tout objet est instance d'une classe

En Python, tout ce qu'on manipule est un objet :

```
>>> nom = 'Robert'          # une instance de la classe str
>>> age = 42                # une instance de la classe int
>>> taille = 1.80           # une instance de la classe float
>>> fort = True             # une instance de la classe bool
```

méthodes et attributs

méthode

- Une **méthode** est une fonction qui appartient à une classe. Ne peut être utilisée (*appelée, invoquée*) que par les instances de la classe qui la définit

attribut

- Un **attribut** est une donnée (*une variable*) qui appartient à un objet.

En Python

POO en Python

- On définit un type (*une classe*) avec `class`
- La méthode qui crée l'objet est `__init__`
- Dans la classe, `self` désigne l'objet lui même (`self` = soi même)
- `self` est toujours le premier paramètre d'une méthode d'objet
- on appelle un paramètre ou une méthode avec `self.parametre` ou `self.methode`

  Attention, il ne faut pas rappeler `self` dans les paramètres !
- On crée un objet avec `truc = NomClasse(parametre1, parametre2,...)`
- On utilise ensuite la notation pointée : `truc.methode(...)`, `truc.attribut`

Exemple

```
# CREATION DE LA CLASSE
class Voiture:                                  # nouveau type appelé "Voiture"

    def __init__(self, couleur, nb_roues):      # constuit un objet Voiture
        self.couleur = couleur                  # un attribut
        self.nb_roues = nb_roues
        self.compteur = 0

    def rouler(self, kilometres):               # méthode
        self.compteur += kilometres             # change l'attribut

    def get_nb_roues(self):                     # self est tjrs le 1er parametre
        return self.nb_roues



# UTILISATION DE LA CLASSE
ferrari = Voiture("rouge", 4)                   # instance d'un objet Voiture
ferrari.rouler(100)                             # appelle une méthode
type(ferrari) == Voiture                        # True
ferrari.nb_roues = 9                            # on peut changer directement les attributs
                                                # c'est une mauvaise pratique
```

- constructeur : `__init__`
  - initialisation de l'état (attributs)

- une classe est un type (`type()`, `isinstance()`)
- `self` : auto-référence = "l'objet dont on est en train de parler"
  ie. celui que l'on construit ou celui qui invoque (utilise) la méthode

  -> permet d'accéder aux attributs de l'objet : (cf. `__init__`, `rouler()`)
  - `self` n'est *pas* imposé en Python mais **très** fortement recommandé
  - `self` ne doit jamais être modifié

méthode d'objet *vs* méthode de classes

- **méthodes d'objets** : invoquée par l'objet
  = envoi de messages possibles
  - premier paramètre = `self` (cf `__init__`, `rouler`)
  - `self` est lié à l'objet utilisé pour invoquer la méthode
    notation pointée : `ferrari.rouler()` -> `self` lié à `ferrari`
  - permet d'accéder aux attributs de l'objet ou d'invoquer une méthode sur cet objet. cf `rouler`
- **méthode de classe** : méthode ne dépendant pas d'un objet mais **statique** appelée via la classe :

  On utilise alors un décorateur : `@classmethod` qui permet de retourner une instance.

  les attributs de classe sont également possibles.

méthodes spéciales

Permet de définir des "opérateurs" et de donner des propriétés aux objets :

Méthode spéciale	Usage
`__add__`	`+`
`__mul__`	`*`
`__sub__`	`-`
`__eq__`	`==`
`__ne__`	`!=`
`__lt__`	`<`
`__ge__`	`<=`
`__gt__`	`>`
`__ge__`	`>=`
`__repr__`	dans l'interpréteur `>>> obj`
`__str__`	`str(obj)`, `print(obj)`
`__getitem__`	`obj[i]`
`__iter__`	`for v in obj`

Exemple d'implémentation d'une méthode spéciale : `__add__`

```
class Vecteur:
  def __init__(self, x, y):
    self.__x = x
    self.__y = y

  def x(self):
    return self.__x

  def y(self):
    return self.__y

  def __add__(self, autre):
    return Vecteur(self.x() + autre.x(),
                   self.y() + autre.y())
```

Exemple d'utilisation de cette méthode spéciale

```
>>> u = Vecteur(1, 2)        # instance de la classe Vecteur
>>> v = Vecteur(3, 5)
>>> w = u + v                # utilise la méthode __add__ !!!!
>>> w.x()                    # méthode
4                            # 1 + 3 = 4
>>> w.y()
7                            # 2 + 5 = 7
```

encapsulation

coeur de la POO en NSI

encapsulation

Les données (attributs) sont regroupées avec les traitements qui les manipulent (méthodes)

- l'encapsulation implique le **masquage des données**
  - l'objet a la maîtrise de ses attributs *via ses méthodes*
  - seules les méthodes sont accessibles

règle d'or

les attributs sont déclarés privés = accessibles uniquement au sein de la classe

en Python, identifiant préfixé de `__`
on peut aussi définir des méthodes privées.

séparation de l'interface et de l'implémentation

- **interface publique d'une classe**
  = ensemble des méthodes *publiques* définies par la classe
  = ensemble des services que peuvent rendre les objets

intérêt ?

- la représentation des données utilisée n'a pas besoin d'être connue, elle pourra donc **évoluer** sans perturber l'existant "code client"
- ce qui compte c'est ce que l'on peut faire, pas comment on le fait
  en partant du principe que c'est bien fait.
- possibilité d'ajouter du contrôle
  - accès en lecture seulement d'un attribut
    `get_hours()` mais pas `set_hours()`
  - contrôle des valeurs classe `Person` avec attribut `__age`

    ```
    def set_age(self, new_age):
      if new_age < 0:             # on protège les valeurs
        new_age = 0
      self.__age = new_age
    ```

- lorsque l'on fait l'analyse objet d'un problème, on cherche à déterminer les **services** que doivent rendre les objets
  = les **méthodes**
- les attributs n'apparaissent que lorsque l'on se pose la question de la mise en oeuvre des méthodes, càd. de leur **implémentation**.

  un attribut existe parce qu'il permet l'implémentation d'une méthode