NSI Terminale - Données

Introduction à la programmation objet

qkzk

2020/07/10

Introduction à la programmation objet

Jusqu'ici les programmes que nous avons écrits utilisaient une approche procédurale :

- On définit les variables qui représentent ce qu'on souhaite modéliser
- On crée les fonctions qui vont changer l'état de ces variables
- L'exécution coordonnée de ces fonctions fait passer nos variables d'un état à l'autre.

Cette approche permet de résoudre tous les problèmes informatiques. On peut tout programmer de cette manière.

Néanmoins elle présente un défaut majeur : il est difficile de dégager une structure à notre code.

Cela le rend peu lisible et difficile à entretenir.

Autre inconvénient, nos programmes ne sont pas réexploitables facilement.

Pour réutiliser une fonction déjà écrite, la seule approche efficace est, pour l'instant, d'en faire un copier-coller.

Programmation objet : définir des *types*.

Les principes de la programmation objet vous sont familiers, vous les avez déjà rencontrés à chaque fois que vous avez crée une liste, un dictionnaire ou une fonction en Python.

Ils sont moins visibles lorsqu'on manipule des nombres directement mais, pourtant, les nombres en Python sont aussi des *objets*.

Qu'est-ce qu'un objet ?

Un objet, c'est simplement "quelque chose" qui respecte des règles préétablies.

Par exemple : un nombre est un objet sur lequel je peux faire des comparaisons :

```
>>> a = 2
>>> b = 3
>>> a < b
True
```

Des opérations :

```
>>> a + b
5
```

Une liste est un objet qui a une longueur, qui contient un nombre fini d'éléments auxquels je peux accéder, que je peux trier ou retourner :

```
>>> ma_liste = [1, 3, 2]
>>> len(ma_liste)
3
>>> ma_liste.append(4)
>>> ma_liste
[1, 3, 2, 4]
>>> ma_liste.sort()
>>> ma_liste
[1, 2, 3, 4]
```

```
>>> ma_liste.reverse()
>>> ma_liste
[4, 3, 2, 1]
```

Type d'un objet

Python propose d'accéder au type d'un objet de deux manière :

1. En le consultant avec la fonction `type`
2. En le vérifiant avec `isinstance`

```
>>> type(ma_liste)
<class 'list'>
>>> isinstance(ma_liste, list)
True
```

Cependant, quand nous créons un objet supposé représenter quelque chose du monde extérieur (réel : une voiture, un participant de la course au chicon) ou imaginaire (un Pokémon etc.), leur type n'est pas défini clairement

```
>>> competiteur = {"nom": "Duchmol", "prenom": "Robert", "age": 42}
>>> pikachu = {"nom": "Pikachu", "cri": "Pika ! Pika !", "type1": "Electrique",
               "type2": None, ...}
```

Certes, le programmeur pense ces objets là comme des compétiteurs ou des Pokémons, il peut même les dessiner à l'écran ou leur envoyer un email automatiquement... mais en pratique :

```
>>> type(competiteur)
<class 'dict'>
>>> type(pikachu)
<class 'dict'>
```

En pratique, ce sont toujours des dictionnaires.

On pourrait par exemple, créer des objets qui n'ont aucun sens :

```
>>> pikachu["telephone"] = "0678901234"
```

Aussi, rien n'est fait pour aider à la collaboration sur ce projet "pokémon".

Un nouveau développeur doit comprendre l'intégralité du code s'il veut utiliser certaines fonctions du programme.

Il serait bon qu'on ait une *interface* pour ces Pokémons.

Elle nous permettrait :

- d'accéder à leurs données (nom, points de vie, attaque etc.)
- de modifier ce qui peut l'être (perdre de la vie, gagner de l'expérience)
- d'exécuter leurs actions (attaquer, lancer une technique spéciale etc.)

Elle nous empêcherait aussi de faire n'importe quoi avec ces objets !

De la même manière qu'avec les `list` on a accès à certaines méthodes sans avoir à comprendre exactement ce qui est fait dans la machine.

Créer un nouveau type avec `class`

De la même manière que le mot clé `def` permet de créer une fonction et de l'affecter à une variable, le mot clé `class` permet de créer un type (une classe) et de l'affecter à une variable.

Une classe est un modèle qui permet de créer des objets de ce type.

- Les objets de cette classe sont appelés *instances*.
- Les variables propres à chaque éléments de cette classe sont appelés *attributs*.
- Les fonctions propres à chaque élément de cette classe sont appelées *méthodes*.

L'ensemble des méthodes accessibles à un utilisateur définit *l'interface* de la classe.

Exemple de classe : `Combattant`

```
1  class Combattant:
2
3      def __init__(self, vie, attaque):
4          self.vie = vie
5          self.attaque = attaque
6          self.vivant = True
7
8      def perdre_vie(self, points):
9          self.vie = self.vie - points
10         if self.vie <= 0:
11             self.vivant = False
12             self.vie = 0
13
14     def attaquer(self, autre):
15         autre.perdre_vie(self.attaque)
```

Commentaires sur le code de la classe.

- Ligne 1. On utilise une majuscule pour un nom de classe.
- Lignes 2 à 15. La définition de la classe se termine quand on revient à l'indentation de départ.
- Ligne 3, ligne 8, ligne 15. définition d'une *méthode*
- La variable `self` est toujours le premier paramètre d'une méthode. Il désigne l'objet lui même.
- Ligne 4, 5, 6 : des *attributs*.
- Ligne 4 à 6, ligne 9 : *dans* la classe on utilise `self.qqchose` pour accéder à un attribut ou une méthode.
- La méthode `__init__` est appelée "constructeur", c'est elle qui est exécutée quand vous exécutez l'instruction `Combattant(10, 2)`. Elle crée un objet du type `Combattant`

Attention ! Python n'impose pas le nom `self`... ce n'est qu'un *usage*.
Cependant, *je vous l'impose.* Si vous écrivez autre chose que `self` :

1. personne ne comprendra,
2. vous aurez 0 à l'exo.

Utilisation de la classe `Combattant`

```
>>> guerrier = Combattant(10, 2)     # on crée une instance. Utilise '__init__'
>>> type(guerrier)                   # type d'une instance = nom de la classe
<class 'Combattant'>
>>> magicien = Combattant(5, 4)      # une autre instance
>>> guerrier.vie                     # chaque instance a ses valeurs d'attributs
10
>>> guerrier.vivant
True
>>> magicien.vie                     # != guerrier.vie
5
>>> magicien.attaquer(guerrier)      # utilisation d'une méthode
>>> guerrier.vie
6
>>> magicien.attaquer(guerrier)
>>> guerrier.vie
2
>>> magicien.attaquer(guerrier)
>>> guerrier.vie
0
```

Dans la suite, nous introduirons peu à peu les bonnes pratiques. Par exemple, rien n'empêche de faire ceci :

```
>>> guerrier.vivant
False
>>> guerrier.vivant = True
>>> guerrier.attaque = 1000000000000000000000000
>>> guerrier.attaque(magicien)
```

Ce qui ne semble pas très cohérent. On pourrait même écrire des choses absurdes :

```
>>> guerrier.vie = "BONJOUR !"
```

ce qui va générer des erreurs par la suite.

Grand principe de la POO : le passage de *messages*

Revenons sur le code de la méthode combattant, en particulier sur attaquer :

```
class Combattant:
    # debut du code...

    def perdre_vie(self, points):
        self.vie = self.vie - points
        if self.vie <= 0:
            self.vivant = False
            self.vie = 0

    def attaquer(self, autre):
        autre.perdre_vie(self.attaque)
```

La méthode `attaquer` prend en paramètre un objet `autre` qui est lui même un objet du type `Combattant`.

Elle exécute une méthode de cet objet (`autre.perdre_vie(...)`) , c'est un *message* !

Ce n'est pas le seul usage envisageable de la méthode `perdre_vie`, plus tard dans le développement du jeu, on peut imaginer qu'un Combattant perde de la vie en marchant dans la lave ou sur un piège...

Interface

- Certaines méthodes sont internes à la classe, comme `__init__` : on n'exécute pas `__init__` directement. On parle généralement de méthodes *privées*.
- D'autres sont *publiques* comme `list.reverse()` ou `Combattant.attaquer(...)`

L'interface d'un objet est définie par les méthodes qu'il expose, celles qui sont publiques.

Les attributs devraient tous être *privés* et ne servir qu'aux méthodes.

Aussi, lorsque nous faisons :

```
>>> guerrier.vie
5
```

On consulte un attribut directement et c'est une mauvaise pratique.

Encapsulation

Le principe de l'encapsulation est de protéger les attributs et de n'exposer que l'interface de la classe.

Python permet (plus ou moins) de protéger les attributs en leur donnant un nom qui commence par `__`.

```
class Voiture:
    def __init__(self, couleur):
        self.__couleur = couleur
```

Lorsqu'on crée un attribut (ou une méthode) dont le nom commence par `__` il n'est plus accessible directement.

setter et getter

Si on veut consulter ou modifier la couleur de la voiture, il faut créer des méthodes qui le permettent :

```
class Voiture:
    def __init__(self, couleur):
        self.__couleur = couleur

    def get_couleur(self):
        return self.__couleur
```

```
def set_couleur(self, couleur):
    self.__couleur = couleur
```

Ici, cela peut sembler inutile mais on pourrait, dans la méthode `set_couleur` restreindre les choix de couleurs à ceux qui la marque propose.

Ou vérifier que la nouvelle valeur d'un numéro de téléphone est valide, qu'un age est positif, qu'un compte bancaire bloqué n'est pas à découvert etc.

---

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
- 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:
        new_age = 0
      self.__age = new_age
    ```

    classe `BankAccount`, accès au solde `get_balance()` contrôlé par code
- 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

exemple : les disques

On doit représenter des **disques**. On a besoin de connaître le rayon, diamètre, aire, périmètre.

- classe `Disc` + méthodes

  `get_rayon(), get_diametre(), get_perimetre()`

- attributs ?
  dépendent de choix d'implémentation...

---

implémentation avec "rayon"

```
class Disque:
  def __init__(self, rayon):
    self.__rayon = rayon

  def rayon(self):
    return self.__rayon

  def diametre(self):
    return 2 * self.__rayon

  def perimetre(self):
    return 2 * math.pi * self.__rayon
```

implémentation avec "diamètre"

```
class Disque:
  def __init__(self, diametre):
    self.__diametre = diametre

  def rayon(self):
    return self.__diametre / 2

  def diametre(self):
    return 2 * self.__diametre

  def perimetre(self):
    return math.pi * self.__diametre
```

Qu'est ce qui change ?

- Pour le **développeur**, s'il défini ses disques avec le diamètre, le constructeur et les méthodes changent.
- Pour **l'utilisateur**, rien ne change. Qu'il utilise l'un ou l'autre, il obtient le même résultat.

  Inutile pour lui de savoir quelle formule on a employé.

méthode d'objet *vs* méthode de classes

Attention cependant, il est toujours possibles de créer des attributs et des méthodes *pour la classe entière* plutôt que pour un objet particulier.

- **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.

C'est une nuance importante, vous devez savoir que ça existe, je ne pense pas qu'on puisse exiger que vous sachiez créer ce genre de méthode en Python.

Exemple de méthode de classe

Nous allons créer une classe dont les instances peuvent être définies de deux manières différentes :

```
class Contact:
    def __init__(self, nom, prenom, telephone):
        self.__nom = nom
        self.__prenom = prenom
        self.__telephone = telephone

    # ici les méthodes publiques

    def get_prenom(self):
        return self.__prenom

    @classmethod
    def depuis_dictionnaire(cls, dictionnaire):
        nom = dictionnaire["nom"]
        prenom = dictionnaire["prenom"]
        telephone = dictionnaire["telephone"]
        return cls(nom, prenom, telephone)
```

La méthode `depuis_dictionnaire` définit un second constructeur (ce n'est pas le seul usage mais le plus courant. Le premier paramètre est un nom de classe, identifié par `cls` il fait référence à la classe dans laquelle il est appelé. Comme pour `self` ce n'est qu'un usage.)

On peut maintenant créer des contacts :

```
>>> robert = Contact("Duchmol", "Robert", "0678901234")    # directement
>>> dict_martin = {"nom": "Martin", "prenom": "Patrick", "telephone": "0789012345"}
>>> martin = Contact.depuis_dictionnaire(dict_martin)     # avec la méthode de classe
>>> type(martin)
<class 'Contact>'
>>> martin.get_prenom()
"Patrick"
```

Lister les méthodes et attributs

Python propose la fonction `dir` qui permet de lister les méthodes et attributs définis pour un objet :

```
>>> ma_liste = [1, 2, 3]
>>> dir(ma_liste)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

Cela fonctionne aussi pour les objets que nous créons nous mêmes :

```
>>> robert = Contact("Duchmol", "Robert", "0678901234")
>>> dir(robert)
['_Contact__nom', '_Contact__prenom', '_Contact__telephone', '__class__',
'__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'depuis_dictionnaire',
'get_prenom', 'get_nom', 'get_telephone']
```

Remarquons que :

- certaines méthodes ont un nom “normal” comme `append` ou `get_telephone`
  Ce sont les méthodes *publiques*.
- d’autres ont un nom qui commence par `__` et se termine par `__`
  Ce sont les méthodes *spéciales*.
- certains attributs ont un nom de la forme `_Contact__telephone`
  Ce sont les attributs et méthodes *privés*. Python a changé leur nom.

méthodes spéciales

Lorsqu’on utilise une fonction courante de Python sur un objet, il appelle une méthode spéciale de cet objet :

```
>>> 1 + 1                    # appelle la méthode spéciale '__add__' des 'int'
>>> ma_liste = [1, 2, 3]     # appelle la méthode spéciale '__init__' des 'list'
>>> ma_liste[1]              # appelle la méthode spéciale '__getitem__' des 'list'
2
>>> ma_liste                 # appelle la méthode spéciale '__repr__' des 'list'
[1, 2, 3]
```

On peut implémenter nous mêmes ces méthodes spéciales.

Quelques méthodes spéciales courantes :

Elles permettent 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 `__repr__` :

Pour l’instant lorsqu’on exécute :

```
>>> martin
<__main__.Contact object at 0x7fc027ddd040>
```

Pas génial . . . On implémente maintenant une méthode `__repr__` :

```
class Contact():

    # le début du code de la classe
    # ...
    # ...

    def __repr__(self):
        return f'Contact("{self.get_nom()}", "{self.get_prenom()}", "{self.get_telephone()}")'
```

Et maintenant :

```
>>> martin
Contact("Martin", "Patrick", "0789012345")
```

C’est beaucoup plus pratique !

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
```

Polymorphisme / héritage

Polymorphisme / héritage

- **hors programme**, donc pas abordé ici
- idée générale : un objet fils hérite des propriétés d'un objet parent :
  - *parent* : rectangle : défini avec (`x, y, l, h`).
    - méthodes : aire, périmètre, contient un point ? etc.
  - *enfant* : carré : défini avec (`x, y, c`)
    l'enfant hérite aussi des méthodes du parent !
- permet de créer des objets répondant à des contraintes succeptibles *d'évoluer*...
  donc de *maintenir* du code.
- J'ai des sources si ça vous intéresse.

En pratique... en Python, toutes les classes héritent de `object` qui est un format général de classe.

Bref historique de la POO

- Introduite dans Simula en 1962
- Vraiment passée dans l'usage courant en 1972 avec SmallTalk, grâce à Alan Kay (prix Turing en 2003)
- Généralisée à de nombreux langages dans les années 1970 et 1980 : C++ ajoute la programmation objet au langage le plus populaire de l'époque : C.
- Depuis les années 90, tous les langages "impératifs" proposent de la POO :
  python, java, javascript, ruby, C# (se prononce *C SHARP*), OCaml etc.

La programmation objet est enseignée dans tous les parcours informatique. La grande majorité du code est écrit en POO...

avec quelques exceptions notable de :

- C : le langage C permet de définir des "structures" mais pas de méthodes (d'où C++). C est toujours employé pour écrire des logiciels "proches du métal" (pilotes matériels) ou qu'on souhaite très rapides (fonctions de base d'un OS).
- les langages fonctionnels purs comme Haskell,
- Golang : (le langage en vogue pour écrire des services dans le cloud). Un peu de POO mais pas tout le pataquès.