La programmation objet

Dernière mise à jour le : 17/06/2024

Nous avons vu qu'il existait différentes manières de voir la programmation, on parle de différents *paradigmes de programmation*. L'un d'eux est le **paradigme objet**. Lorsque l'on programme en utilisant ce paradigme on parle de **programmation objet** ou de **programmation orientée objet** (abrégé *POO*, ou *OOP* en anglais pour « Object-oriented programming ».

> Nous nous limiterons cette année, comme le programme l'indique, à une brève introduction de la programmation objet.

Vocabulaire de la programmation objet

La programmation objet consiste à regrouper données et traitements dans une même structure appelée **objet**. Elle possède l'avantage de localiser en un même endroit toute l'implémentation d'une structure de données abstraite.

Objets, attributs, méthodes

Concrètement, un objet est une structure de données abstraite regroupant :

- des *données* associées à l'objet que l'on appelle des **attributs**.
- des *fonctions* (ou procédures) s'appliquant sur l'objet que l'on appelle **méthodes**.

Exemple : Prenons une voiture, on peut la considérer comme un objet. On peut lui associer des informations comme sa couleur, sa marque et sa catégorie : il s'agit des attributs de notre objet. On peut également définir des mécanismes concernant cet objet : démarrer, accélérer, freiner, klaxonner : il s'agit des méthodes s'appliquant sur notre objet.

Classe

Ces attributs et méthodes sont réunis dans ce qu'on appelle une **classe** qui est donc un *modèle décrivant un objet*. Les objets sont ensuite créés à partir de cette classe (ce modèle) qui est donc aussi une *machine à fabriquer des objets*.

Exemple : On peut définir une classe `Voiture` décrivant un modèle pour tous les objets "voitures". On peut la schématiser ainsi :

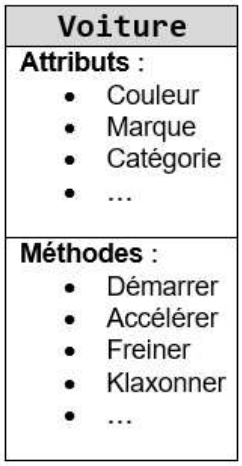

Schéma de la classe `Voiture`

On peut facilement créer des objets grâce à ce modèle. Il suffit de les construire en utilisant le nom de la classe (qui est aussi le nom du constructeur d'objets de cette classe). Chaque objet ainsi créé est une *instance* de la classe.

Exemple : En Python, on peut créer deux objets `clio` et `c3` (deux instances) de la classe `Voiture` en écrivant simplement les deux instructions

```
clio = Voiture()
c3 = Voiture()
```

Accès aux attributs et méthodes

Pour accéder aux attributs et aux méthodes d'une classe on utilise la **notation pointée**.

Exemple : L'instruction `clio.marque` renvoie la marque de l'objet (instance) `clio` et l'instruction `clio.klaxonner()` va faire klaxonner notre voiture (virtuelle). Nous reviendrons sur cela un peu plus loin.

Cette notation pointée ne vous dit rien ? Allons, vous l'avez déjà utilisée pourtant...

Classes et objets en Python

En Python, tout est objet !

> Vous ne le saviez sans doute pas, mais les objets vous connaissez déjà (et oui !)

Vous avez manipulé des objets depuis que vous programmez en Python, tout simplement car dans ce langage tout est objet. On peut le voir facilement.

```
>>> a = 5
>>> type(a)
<class 'int'>
```

```
>>> b = 2.1
>>> type(b)
<class 'float'>
```

```
>>> c = "nsi"
>>> type(c)
```

```
<class 'str'>
```

```
>>> d = (3, -4)
>>> type(d)
<class 'tuple'>
```

```
>>> e = [0, 1, 2, 3]
>>> type(e)
<class 'list'>
```

```
>>> f = {"a": 1, "b": 2}
>>> type(f)
<class 'dict'>
```

L'affichage montre que tous les types en Python sont des classes. Les entiers sont des objets de la classe `int`, les flottants sont des objets de la classe `float`, etc.

Pour créer un entier ou une liste il suffit de les construire en utilisant le nom de leurs classes respectives.

```
>>> entier = int() # crée un entier (0 par défaut)
>>> liste = list() # crée une liste (vide par défaut), équivalent à liste = []
>>> type(entier), type(liste)
(<class 'int'>, <class 'list'>)
```

Interface d'une classe

En définissant une classe on définit un type abstrait de données (cf. chapitre en question). Comme tout type abstrait, une classe possède donc une **interface** qui décrit l'ensemble des méthodes auxquels on a accès pour manipuler les objets de cette classe.

On peut utiliser la fonction `dir` pour lister tous les attributs et méthodes d'un objet.

```
>>> L = [2, 3, 5]
>>> dir(L) # ou directement dir(list)
['__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']
```

On retrouve ici les méthodes applicables sur les objets du type prédéfini `list`. Son interface est disponible dans la documentation officielle. Vous noterez que l'interface ne précise pas la façon dont sont implémentées ces méthodes mais juste la façon de les utiliser, ce qui suffit largement généralement.

On constate aussi qu'il y a de nombreuses méthodes spéciales repérables par leur nom encadré d'un double **underscore** (`__`). Nous reviendrons sur quelques-unes d'entre elles un peu plus tard.

Par exemple, vous saviez déjà que pour ajouter un élément à une liste Python (qui est un objet) on pouvait utiliser la méthode `append` avec la notation pointée :

```
>>> L = [2, 3, 5]
>>> L.append(8)
```

```
>>> L
[2, 3, 5, 8]
```

Une première classe en Python

Nous allons voir comment implémenter une classe en Python.

Nous reprenons pour cela l'exemple du type abstrait `Rationnel` abordé dans un chapitre précédent. Pour rappel, on souhaitait pouvoir effectuer les opérations suivantes sur cette structure de données :

- Créer un rationnel
- Accéder au numérateur et au dénominateur d'un rationnel
- Ajouter, soustraire, multiplier, diviser deux rationnels
- Vérifier si deux rationnels sont égaux ou non

Nous avions déjà implémenté ce type abstrait (de plusieurs manières) dans le chapitre en question. L'objectif ici est de créer une classe appelée `Rationnel` dont le but est de pouvoir construire des objets de type `Rationnel` et les manipuler.

On déclare une classe en Python à l'aide du mot clé `class` :

```
class Rationnel:
    """Manipulation de rationnels définis par leurs numérateur et dénominateur"""
```

Par convention, les noms de classes en Python sont écrits en capitales (première lettre en majuscule). On a documenté notre classe avec une docstring qui sera accessible à quiconque souhaite utiliser notre classe.

Création et initialisation d'un objet en Python

La création standard d'un objet par la méthode de sa classe, crée des objets sans attributs. Pour que les méthodes puissent s'appliquer sans erreur, il faut cependant que tous les objets d'une classe possèdent certains attributs. C'est pour cette raison, qu'une méthode d'initialisation des objets créant des attributs est nécessaire. En Python, c'est la méthode spéciale `__init__`, si elle est définie dans la classe, qui est systématiquement appelée sur chaque nouvel objet juste après sa création.

Dans l'exemple de la classe `Rationnel`, les méthodes d'addition, de test d'égalité, etc. ne peuvent s'appliquer qu'à des objets ayant des attributs égaux au numérateur et au dénominateur. Pour initialiser les attributs `num` et `den` de chaque objet à sa création on utilise la méthode `__init__` de la façon suivante :

```
class Rationnel:
    """Manipulation de rationnels définis par leurs numérateur et dénominateur"""

    def __init__(self, numerateur, denominateur):
        """Initialise le rationnel avec les valeurs indiquées"""
        self.num = numerateur
        self.den = denominateur
```

Remarques importantes :

- En Python, le mot `self` n'est pas un mot clé. C'est une simple convention d'usage de nommer ainsi le premier paramètre d'une méthode qui désigne toujours l'objet auquel s'appliquera la méthode (l'objet précédant le `.`)
- C'est par la notation `self.num` que l'attribut `num` est créé pour l'objet sur lequel est appelée la méthode.

On peut désormais créer un objet `r` par appel du constructeur en fournissant les valeurs des paramètres prévus dans la méthode spéciale d'initialisation. On peut accéder aux attributs de l'objet en utilisant la notation pointée.

```
>>> r = Rationnel(3, 4)
>>> r.num, r.den
(3, 4)
```

On peut modifier les attributs d'un objet en les redéfinissant.

```
>>> r.num = 2  # modification de la valeur du numérateur
>>> r.num, r.den
(2, 4)
```

Les attributs `num` et `den` sont propres à chaque objet. Dans la terminologie des langages à objet, on parle d'*attributs d'instance*.

Notre objet est bien du type abstrait de données `Rationnel` que l'on vient de créer en définissant notre classe.

```
>>> type(r)
__main__.Rationnel
```

> On peut observer avec python tutor la création des différentes instances d'une classe, l'initialisation de leurs attributs et les modifications d'attributs.

Ecriture des méthodes dédiées

Si on veut pouvoir manipuler nos objets, il faut ajouter à notre classe les méthodes souhaitées. Par exemple, on ajoute les méthodes `ajouter` et `egal` en définissant deux fonctions dans notre classe.

```
class Rationnel:
    """Manipulation de rationnels définis par leurs numérateur et dénominateur"""

    def __init__(self, numerateur, denominateur):
        """Initialise le rationnel avec les valeurs indiquées"""
        self.num = numerateur
        self.den = denominateur

    def ajouter(self, other):
        """Renvoie un nouveau rationnel égal à la somme"""
        import math
        num = self.num * other.den + other.num * self.den  # calcul du numerateur
        den = self.den * other.den  # calcul du dénominateur
        d = math.gcd(num, den)  # calcul du pgcd pour simplifier le rationnel
        return Rationnel(num // d, den // d)  # on renvoie un nouvel objet 'Rationnel'

    def egal(self, other):
        """Renvoie Vrai si les deux rationnels sont égaux, Faux sinon."""
        return self.num == other.num and self.den == other.den
```

On peut alors accéder à ces méthodes en utilisant également la notation pointée sur l'objet auquel s'applique la méthode.

```
>>> r1 = Rationnel(1, 4)
>>> r2 = Rationnel(1, 2)
>>> r3 = r1.ajouter(r2)  # on ajoute r2 à r1
```

```
>>> r3.num, r3.den
(3, 4)
```

```
>>> r4 = Rationnel(3, 4)
>>> r3.egal(r4)  # pour vérifier si r3 = r4
True
```

Vous noterez que l'on fournit toujours un paramètre de moins lors de l'appel à une méthode que dans la définition de la méthode. En effet, le paramètre `self` n'est pas utilisé car il désigne la référence à l'objet auquel s'applique la méthode.

```
>>> r = Rationnel(5, 3)
>>> dir(r)
['__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__', 'ajouter', 'den', 'egal', 'num']
```

On constate qu'il y a de nombreuses méthodes spéciales repérables par leur nom encadré de `__`. Ces méthodes sont appelées dans des contextes particuliers et peuvent être redéfinies par le programmeur pour une classe particulière.

L'usage de ces méthodes spéciales n'est pas un attendu du programme mais cela peut se révéler très utile. C'est pourquoi nous en présenterons quelques-unes.

Méthodes spéciales en Python

Nous nous contenterons ici de présenter trois méthodes spéciales (quelques autres seront évoquées dans les activités) :

- la méthode `__repr__(self)` est appelée pour calculer la représentation *officielle* en chaîne de caractères d'un objet (c'est cette méthode qui est appelée lorsque l'on veut évaluer un objet) ;
- la méthode `__str__(self)` est appelée pour calculer une chaîne de caractères *informelle* ou joliment mise en forme de représentation de l'objet (c'est cette méthode qui est appelée par la fonction `print()`) ;
- la méthode `__eq__(self, other)` est appelée pour tester l'égalité entre deux objets.

On pourrait être tenté d'afficher une instance ou de tester l'égalité entre deux instances d'une même classe. Par exemple, avec notre classe `Rationnel` on aimerait écrire.

```
>>> r1 = Rationnel(1, 2)
>>> r2 = Rationnel(1, 2)
```

```
>>> r1  # évaluation
<__main__.Rationnel at 0x28d4eff47c8>
```

```
>>> print(r1)  # affichage
  <__main__.Rationnel object at 0x0000028D4EFF47C8>
```

```
>>> r1 == r2 # test d'égalité
False
```

Nous ne pouvons nous satisfaire des résultats. L'évaluation d'un objet, son affichage et le test d'égalité (avec les notations habituelles) font appel respectivement aux méthodes `__repr__`, `__str__` et `__eq__` que nous avons besoin de redéfinir pour obtenir des résultats cohérents.

```python
class Rationnel:
    """Manipulation de rationnels définis par leurs numérateur et dénominateur"""

    def __init__(self, numerateur, denominateur):
        """Initialise le rationnel avec les valeurs indiquées"""
        self.num = numerateur
        self.den = denominateur

    def ajouter(self, other):
        """Renvoie un nouveau rationnel égal à la somme"""
        import math
        num = self.num * other.den + other.num * self.den  # calcul du numerateur
        den = self.den * other.den  # calcul du dénominateur
        d = math.gcd(num, den)  # calcul du pgcd pour simplifier le rationnel
        return Rationnel(num // d, den // d)  # on renvoie un nouvel objet 'Rationnel'

    def egal(self, other):
        """Renvoie Vrai si les deux rationnels sont égaux, Faux sinon."""
        return self.num == other.num and self.den == other.den

    def __repr__(self):
        return "Rationnel(" + str(self.num) + ", " + str(self.den) + ")"  # ou f"Rationnel({str.nu

    def __str__(self):
        return str(self.num) + " / " + str(self.den)  # ou f"{self.num} / {self.den}"

    def __eq__(self, other):  # on pourrait aussi écrire simplement __eq__ = egal
        return self.num * other.den == other.num * self.den
```

On peut désormais utiliser les instructions classiques d'évaluation (ou d'affichage) et de test d'égalité avec les objets de notre classe.

```
>>> r1 = Rationnel(1, 2)
>>> r2 = Rationnel(1, 2)
>>> r1
Rationnel(1, 2)
```

```
>>> print(r1)
1 / 2
```

```
>>> r3 = Rationnel(1, 4)
>>> r4 = r3.ajouter(r1)
>>> r4
Rationnel(3, 4)
```

```
>>> r1 == r2
True
```

Que se passe-t-il pour la dernière instruction ?

Python reconnaît qu'il doit tester l'égalité entre deux instances de la classe `Rationnel`. Ce test (`==`) invoque la méthode spéciale `__eq__` de la classe `Rationnel`. Plus précisément, `r1 == r2` appelle `r1.__eq__(r2)` et comme nous venons de définir cette méthode, le résultat est cohérent. On peut désormais tester l'égalité de deux rationnels sans utiliser l'instruction un peu plus lourde `r1.egal(r2)`.

Bilan

La *paradigme objet* est une autre façon de voir la programmation qui consiste à utiliser un structure de donnée appelée *objet* qui réunit des données et des fonctionnalités. Les données sont appelées **atributs** et les fonctionnalités sont appelées **méthodes**.

Une **classe** permet de définir un modèle d'objet en spécifiant des attributs et des méthodes. On peut ensuite utiliser cette classe pour fabriquer des objets selon ce modèle.

En Python, on utilise le mot clé `class` pour définir une classe qui devient alors un nouveau type abstrait de données. On peut alors créer de nouveaux objets en appelant le constructeur qui porte le nom de la classe. Les objets ainsi créés s'appellent des *instances* de la classe.

En Python, la méthode spéciale `__init__` est appelée à la construction d'un nouvel objet. C'est dans cette méthode que l'on définit les attributs de nos objets.

Les attributs et méthodes d'une instance de classe sont accessibles en utilisant la notation pointée : `objet.attribut` et `objet.methode(arguments)`.

Références :

- Documents ressources de l'équipe éducative du DIU EIL, Université de Nantes, Christophe DECLERCQ.
- Documentation officielle de Python sur les classes.
- Documentation officielle de Python sur les méthodes spéciales.

Germain BECKER, Lycée Mounier, ANGERS