Cours : Programmation Orientée Objet (POO)

Thème 1 - Structure de données

06 Cours : Programmation Orientée Objet (POO)

Le paradigme objet

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.

1. 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.

1.1. 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**.

2. Classes et objets en Python

2.1. En Python, tout est objet !

> Vous ne le saviez sans doute pas, mais les objets vous connaissez déjà.

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.

Script Python

```
m=[4,5,8,2]
type(m)
```

Texte

```
<class 'list'>
```

`m` est une liste, ou plus précisément un **objet** de type `list`. Et en tant qu'objet de type `list`, il est possible de lui appliquer certaines fonctions prédéfinies (qu'on appelera **méthodes**) :

Script Python

```
m.reverse()
m
```

Texte

```
[2, 8, 5, 4]
```

La syntaxe utilisée (le . après le nom de l'objet) est spécifique à la POO. Chaque fois que vous voyez cela, c'est que vous êtes en train de manipuler des objets.

Nous ne sommes pas surpris par ce résultat car la personne qui a programmé la méthode `reverse()` lui a donné un nom explicite.
Comment a-t-elle programmé cette inversion des valeurs de la liste ? Nous n'en savons rien et cela ne nous intéresse pas. Nous sommes juste utilisateurs de cette méthode. L'objet de type `list` nous a été livré avec sa méthode `reverse()` (et bien d'autres choses) et nous n'avons pas à démonter la boîte pour en observer les engrenages : on parle de principe d'**encapsulation**.

On peut obtenir la liste de toutes les fonctions disponibles pour un objet de type `list`, par la fonction `dir` :

Script Python

```
dir(m)
```

Texte

```
['__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']
```

Les méthodes encadrées par un double underscore __ sont des méthodes **privées**, a priori non destinées à l'utilisateur. Les méthodes **publiques**, utilisables pour chaque objet de type `list`, sont donc `append`, `clear`, ...

Comment savoir ce que font les méthodes ? Si elles ont été correctement codées (et elles l'ont été), elles possèdent une *docstring*, accessible par :

Script Python

```
m.append.__doc__
```

Texte

```
'Append object to the end of the list.'
```

Script Python

```
m.index.__doc__
```

Texte

```
'Return first index of value.\n\nRaises ValueError if the value is not present.'
```

Script Python

```
help(m.index)
```

Texte

```
Help on built-in function index:

index(value, start=0, stop=2147483647, /) method of builtins.list instance
    Return first index of value.

    Raises ValueError if the value is not present.
```

3. Mise en pratique

3.1. Une première classe en Python

Nous allons voir comment implémenter une classe en Python.

> **Vocabulaire :**
>
> 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.

Script Python

```
class Voiture :
    def __init__(self, annee, coul, vmax) :
        self.annee = annee
        self.couleur = coul
```

```
        self.vitesse_max = vmax
        self.age = 2020 - self.annee
```

3.2. Création et initialisation d'un objet en Python

A retenir : la méthode constructeur

La **méthode constructeur**, toujours appelée `__init__()`, est une méthode (une «def») qui sera automatiquement appelée à la création de l'objet. Elle va donc le doter de tous les attributs de sa classe.

Script Python

```
class Voiture :
    def __init__(self, annee, coul, vmax) :
        self.annee = annee
        self.couleur = coul
        self.vitesse_max = vmax
        self.age = 2022 - self.annee
```

- le mot-clé `self`, omniprésent en POO (d'autres langages utilisent `this`), fait référence à l'objet lui-même, qui est en train d'être construit.
- pour construire l'objet, 3 paramètres seront nécessaires : `annee`, `coul` et `vmax`. Ils donneront respectivement leur valeur aux attributs `annee`, `couleur` et `vitesse_max`.
- dans cet exemple, les noms `coul` et `vmax` ont été utilisés pour abréger `couleur` et `vitesse_max`, mais il est recommandé de garder les mêmes noms, même si ce n'est pas du tout obligatoire.

Construisons donc notre première voiture !

Script Python

```
ma_voiture = Voiture(2012, "Grise", 180)
```

`mon_bolide` possède 4 attributs : - `annee`, `couleur` et `vitesse_max` ont été donnés par l'utilisateur lors de la création. - `age` s'est créé «tout seul» par l'instruction `self.age = 2022 - self.annee`.

Script Python

```
type(ma_voiture)
```

Texte

```
<class '__main__.Voiture'>
```

Script Python

```
print(ma_voiture.annee)
print(ma_voiture.couleur)
print(ma_voiture.vitesse_max)
print(ma_voiture.age)
```

Texte

```
2012
Grise
180
8
```

Bien sûr, on peut créer aytant de voitures que l'on veut en suivant le même principe :

Script Python

```
batmobile = Voiture(2036, "noire", 525)
batmobile.couleur
```

Texte

```
'noire'
```

Exercice : Reprendre l'exemple du type abstrait `Rationnel`

Nous reprenons pour cela l'exemple du type abstrait `Rationnel` abordé dans le 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` :

Script Python

```
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.

Script Python

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

    def __init__(self, numerateur, denominateur):
        """Initialise le rationnel avec les valeurs indiquées"""
        pass
```

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.

Script Python

```

```

Texte

```
3
4
```

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

Script Python

```
r.num = 2  # modification de la valeur du numérateur
r.num, r.den
```

Texte

```
(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.

Script Python

```
type(r)
```

Texte

```
<class '__main__.Rationnel'>
```

3.3. 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.

Script 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 = ...  # calcul du numerateur
        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 ... and ...
```

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

Script Python

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

Script Python

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

> **Remarque** : 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.

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

Script Python

```
r= Rationnel(5, 3)
dir(r)
```

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.

3.4. Méthodes spéciales en Python : Hors programme

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.

Script Python

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

Script Python

```
r1  # évaluation
```

Script Python

```
print(r1)  # affichage
```

Script Python

```
r1 == r2 # test d'égalité
```

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.

Script 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.num}, {str.den})"

    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.

Script Python

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

Script Python

```
print(r1)
```

Script Python

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

Script Python

```
r1 == r2
```

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)`.