Programmation orientée objet en Python

Programmation orientée objet

Objectifs.
Prérequis : fonctions en Python, notion de type de données
À l'issue de ce chapitre, vous saurez :

- définir une classe avec des attributs d'instance et des méthodes ;
- créer des objets et appeler leurs méthodes avec la notation pointée ;
- écrire un constructeur `__init__` pour initialiser les attributs ;
- utiliser les méthodes spéciales (`__repr__`, `__add__`, `__eq__`, etc.) pour personnaliser le comportement des objets.

Définitions

Jusqu'ici, nos programmes manipulaient d'un côté des données (nombres, listes, dictionnaires) et de l'autre des fonctions qui les traitent. La **programmation orientée objet** réunit les deux : un **objet** contient à la fois ses données et les opérations qui s'y appliquent. Tout ce qui concerne une même sorte de données se trouve ainsi écrit au même endroit.

- Les données d'un objet sont ses **attributs**.
- Les fonctions qui agissent sur un objet sont ses **méthodes**.
- Une **classe** décrit une sorte d'objets : quels attributs ils possèdent, quelles méthodes on peut leur appliquer. Elle sert aussi à **fabriquer** ces objets ; chaque objet fabriqué s'appelle une **instance** de la classe.

Un premier exemple : la classe `Point`

Un logiciel de dessin manipule des points. Chaque point possède deux attributs, ses coordonnées `x` et `y`, et une méthode `translater` qui le déplace.

- Pour un point `p`, on lit ses attributs avec la **notation pointée** : `p.x` et `p.y`.
- On applique une méthode avec la même notation : `p.translater(dx, dy)` déplace `p` de `dx` horizontalement et de `dy` verticalement.

Une première classe en Python

Une classe se définit avec le mot-clé `class`, suivi de son nom. L'usage veut que ce nom commence par une majuscule : `Point`, `CompteBancaire`.

Les fonctions écrites à l'intérieur de la classe sont ses **méthodes**. Leur premier paramètre, nommé `self`, reçoit l'objet auquel on applique la méthode.

```python
class Point:
    """Point du plan."""

    def translater(self, dx, dy):
        """Déplace le point de dx horizontalement et de dy verticalement."""
        self.x = self.x + dx
        self.y = self.y + dy
```

Appeler la classe comme une fonction, `Point()`, fabrique un nouvel objet. Dans l'appel `p.translater(1, 2)`, Python transmet `p` au paramètre `self`, puis `1` et `2` à `dx` et `dy`.

Remarque 1. `self` n'est pas un mot réservé de Python : c'est seulement le nom que tout le monde donne, par convention, au premier paramètre d'une méthode.

Ici, l'objet est créé vide : on lui donne ses coordonnées de l'extérieur avant de le déplacer.

```python
p = Point()
p.x = 3
p.y = 7
p.translater(1, 2)
print(f"{p.x} {p.y}")
```

```
4 9
```

Chaque objet a ses propres valeurs de `x` et de `y` : ce sont des **attributs d'instance**.

Remarque 2. Un attribut naît au moment où on lui affecte une valeur : `p.x = 3` crée l'attribut `x` de `p`, de même que `self.x = ...` dans une méthode le crée pour l'objet `self`.

Création et initialisation d'un objet en Python

Que se passe-t-il si l'on oublie de donner ses coordonnées à un point avant de le déplacer ?

```
>>> q = Point()
>>> q.translater(1, 2)
AttributeError: 'Point' object has no attribute 'x'
```

L'objet `q` vient d'être créé, il n'a encore aucun attribut : `translater` ne trouve pas `self.x`. Pour éviter cette erreur, chaque point doit recevoir ses coordonnées **dès sa création**.

C'est le rôle de la méthode spéciale `__init__`. Quand on écrit `Point(12, 6)`, Python crée un objet vide, puis appelle aussitôt `__init__` sur cet objet, avec les arguments `12` et `6` ; `__init__` crée alors les attributs.

```python
class Point:
    """Point du plan."""

    def __init__(self, abscisse, ordonnee):
        """Crée les attributs x et y du point."""
        self.x = abscisse
        self.y = ordonnee

    def translater(self, dx, dy):
```

```
        """Déplace le point de dx horizontalement et de dy verticalement."""
        self.x = self.x + dx
        self.y = self.y + dy
```

Un point ne peut plus être créé sans ses coordonnées : tous les points possèdent donc `x` et `y`, et `translater` s'applique sans erreur.

```
p = Point(12, 6)
p.translater(1, 2)
print(f"{p.x} {p.y}")
```

```
13 8
```

Activité. Créer une classe `Temps` avec des attributs d'instance `h`, `min` et `s`, une méthode d'initialisation et une méthode `addition`.

Interface et encapsulation

L'**interface** d'une classe est l'ensemble des méthodes que l'utilisateur de la classe peut appeler, avec leur spécification (paramètres, valeur renvoyée, effet). C'est le « mode d'emploi » de la classe : pour utiliser un objet `Point`, il suffit de connaître `Point(x, y)` et `translater(dx, dy)`, sans lire le code des méthodes. L'addition de deux points, ajoutée plus bas, viendra compléter cette interface.

L'**encapsulation** consiste à regrouper les données (attributs) et les traitements (méthodes) dans l'objet, et à n'accéder aux données **qu'à travers les méthodes** de l'interface. L'état interne de l'objet reste ainsi cohérent : on ne peut pas mettre un compte bancaire à découvert si le solde n'est modifié que par `deposer` et `retirer`, qui vérifient les règles.

```
class CompteBancaire:
    """Un compte dont le solde ne peut pas devenir négatif."""

    def __init__(self, titulaire):
        self.titulaire = titulaire
        self._solde = 0          # attribut interne : ne pas y toucher directement

    def deposer(self, montant):
        assert montant > 0, "le montant doit être positif"
        self._solde = self._solde + montant

    def retirer(self, montant):
        assert 0 < montant <= self._solde, "solde insuffisant"
        self._solde = self._solde - montant

    def solde(self):
        return self._solde
```

Python ne garantit pas l'encapsulation : rien n'interdit d'écrire `compte._solde = -1000`. La convention est claire : un nom qui commence par `_` fait partie de l'**implémentation** et non de l'interface, on ne l'utilise pas de l'extérieur. Séparer interface et implémentation permet de modifier l'intérieur d'une classe (représentation des données, algorithme) sans changer les programmes qui l'utilisent : c'est le principe que l'on retrouvera avec les structures de données, dont l'interface (pile, file, arbre) ne dépend pas de l'implémentation choisie.

Classes et objets en Python

Les classes ne sont pas réservées aux programmes que nous écrivons : en Python, **toutes** les valeurs sont des objets. Une liste, une chaîne de caractères ou un dictionnaire sont des instances des classes `list`, `str` et `dict`, et `L.append(4)` est l'appel d'une méthode.

La fonction `dir` donne la liste des noms d'attributs et de méthodes d'un objet. Gardons ceux qui ne commencent pas par `__` :

```
p = Point(12, 6)
print([nom for nom in dir(p) if not nom.startswith("__")])
```

```
['translater', 'x', 'y']
```

```
L = [1, 8, 7]
print([nom for nom in dir(L) if not nom.startswith("__")])
```

```
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', '
   reverse', 'sort']
```

On retrouve pour `p` les attributs et la méthode de la classe `Point`, et pour `L` les méthodes des listes. Sans le filtre, `dir` affiche aussi des dizaines de noms encadrés par deux tirets bas, comme `__init__` : ce sont les **méthodes spéciales**.

Méthodes spéciales en Python

Une méthode spéciale n'est presque jamais appelée par son nom : Python l'appelle tout seul quand on utilise une fonction ou un opérateur du langage. En la définissant dans une classe, on choisit ce que fait cette fonction ou cet opérateur pour les objets de la classe.

On écrit	Python appelle
`Point(12, 6)`	`__init__(self, ...)`, juste après la création de l'objet
`str(p)` ou `print(p)`	`__str__(self)`, qui renvoie la chaîne à afficher
`repr(p)`	`__repr__(self)` ; `str` et `print` s'en servent aussi quand `__str__` n'est pas définie
`p == q`	`__eq__(self, other)`
`p + q`	`__add__(self, other)`
`len(p)`	`__len__(self)`

La documentation de Python les décrit toutes.

Suite de l'exemple de la classe `Point`

Donnons aux points deux affichages, l'un pour `print`, l'autre pour `repr`, et une addition qui ajoute les coordonnées une à une :

```python
class Point:
    """Point du plan."""

    def __init__(self, abscisse, ordonnee):
        """Crée les attributs x et y du point."""
        self.x = abscisse
        self.y = ordonnee

    def translater(self, dx, dy):
        """Déplace le point de dx horizontalement et de dy verticalement."""
        self.x = self.x + dx
        self.y = self.y + dy

    def __str__(self):
        """Renvoie la chaîne affichée par print : (x, y)."""
        return f"({self.x}, {self.y})"

    def __repr__(self):
        """Renvoie l'instruction qui recrée le point : Point(x, y)."""
        return f"Point({self.x}, {self.y})"

    def __add__(self, other):
        """Renvoie le point dont les coordonnées sont les sommes des coordonnées."""
        return Point(self.x + other.x, self.y + other.y)

    def __eq__(self, other):
        """Renvoie True si les deux points ont les mêmes coordonnées."""
        return self.x == other.x and self.y == other.y
```

```python
p = Point(12, 6)
q = Point(3, 4)
print(p)
print(repr(p))
print(p + q)
print([p, q])
print(p == Point(12, 6))
```

```
(12, 6)
Point(12, 6)
(15, 10)
[Point(12, 6), Point(3, 4)]
True
```

`print(p)` appelle `__str__`, qui écrit le point comme en mathématiques ; `repr(p)` appelle `__repr__`, qui reproduit l'instruction permettant de recréer le point. L'expression `p + q` appelle `p.__add__(q)`, qui fabrique un nouveau point ; `print` l'affiche ensuite grâce à `__str__`. Pour afficher une liste, Python appelle `__repr__` sur chaque élément, et non `__str__`. Enfin, `p == Point(12, 6)` appelle `p.__eq__(Point(12, 6))`, qui compare les coordonnées : sans cette méthode, `==` vérifierait seulement si les deux points sont le même objet, et donnerait `False`.

Remarque 3. `__str__` et `__repr__` doivent **renvoyer** leur chaîne avec `return`, sans l'afficher : c'est `print` qui l'affiche ensuite. Avec `print` à la place de `return`, la méthode renvoie `None`, et `print(p)` s'arrête sur l'erreur `TypeError: __str__ returned non-string (type NoneType)`.

À retenir.

- La **POO** regroupe données (attributs) et traitements (méthodes) dans un même objet, facilitant l'organisation et la réutilisation du code.
- Une **classe** définit le modèle ; un **objet** (ou instance) est une réalisation concrète de ce modèle, avec ses propres valeurs d'attributs.
- La méthode `__init__` initialise chaque objet à sa création ; `self` désigne l'objet courant.
- L'**interface** d'une classe (ses méthodes publiques et leur spécification) est ce que l'utilisateur doit connaître ; l'**encapsulation** impose de passer par ces méthodes pour lire ou modifier les attributs, dont l'implémentation peut changer sans toucher au reste du programme.
- Les **méthodes spéciales** (`__repr__`, `__add__`, `__eq__`, etc.) permettent de personnaliser l'affichage, les opérateurs et les comparaisons.
- La POO est la brique de base pour implémenter les structures de données du programme (listes chaînées, piles, files, arbres).