Paradigme de Programmation Orientée Objet (POO)

Concepts

Le *paradigme* de Programmation Orientée Objet (POO) propose une organisation du code autour du concept d'**objet**.

Un **objet** regroupe pour une même structure de données :

- ses informations stockées sous forme d'**attributs** qui sont des *variables*
- les fonctions permettant de manipuler ses informations sous forme de **méthodes** qui sont des *fonctions*

Chaque **objet** est fabriqué à l'aide d'une **classe**. Une **classe** est un nouveau type de données qui est défini par le programmeur.

La classe est le *moule* de l'objet, un même moule peut servir à fabriquer plusieurs objets.
On dit qu'un objet est une **instance** de sa classe.

Intérêt du paradigme Objet

En regroupant tout le code d'une structure de données dans une classe, le *paradigme* de Programmation Orientée Objet (POO) permet :

- d'améliorer la *lisibilité* du code
- de cloisonner les *espaces de nommage* : un même nom de méthode ou d'attribut peut être utilisé dans plusieurs classes
- de faciliter la *maintenance* et la *réutilisatibilité* du code en implémentant le principe d'**encapsulation** : on n'a pas besoin de connaître les détails d'implémentation interne d'un objet pour l'utiliser, l'interface publique offerte par ses méthodes doit suffir. L'application stricte de ce principe conduit à distinguer des niveaux d'accès *public* (depuis l'extérieur de la classe) ou *privé* (depuis l'intérieur de la classe) pour les attributs et méthodes d'une classe. C'est le cas en Java mais pas en Python.

Interface d'une classe

Une classe est déterminée par son **interface** :

- la liste des *attributs* avec leur *type* et leur *signification*
- la liste des *méthodes* avec leur *signature* et leur *spécification*

Implémentation en Python

Voici l'exemple de l'interface d'une classe `Point` permettant de créer des objets représentant des points du plan.

- **Attributs :**

Nom de l'attribut	Type	Signification
`x`	`float`	abscisse du point
`y`	`float`	ordonnée du point

- **Méthodes :**

Nom de la méthode	signature	Sprécification
`__init__`	`__init__(self, x, y)`	construit un point de coordonnées x et y
`distance`	`distance(self, autre)`	distance entre le point courant et un autre point

On donne ci-dessous une implémentation en Python de cette interface.

Dans la syntaxe, on distingue les phases de définition de la classe et de manipulation d'un objet instancié.

Définition de la classe

Action	Syntaxe

Action	Syntaxe
Définition d'une classe	`class Maclasse:# bloc indenté`
Référence à l'objet courant depuis l'intérieur de la classe	`self`
Définition d'une méthode comme une fonction, `self` obligatoire comme premier paramètre	`def methode(self, paramètre): # bloc`
Initialisation des attributs	dans la méthode spéciale `__init__`
Accès à un attribut depuis l'intérieur de la classe	`self.attribut`

Instanciation et manipulation d'un objet

On crée ou *instancie* un objet en utilisant le nom de la classe comme une fonction à laquelle on passe les valeurs par défaut des attributs. L'objet est créé et la méthode spéciale `__init__` est appelée pour initialiser les attributs.

On manipule ensuite les **attributs** comme des variables et les **méthodes** comme des fonctions avec la notation pointée `objet.attribut` ou `objet.methode(paramètres)` .

Action	Syntaxe
Instanciation/Création d'un objet	`objet = Maclasse(valeurs_attributs)`
Appel de méthode sur l'objet	`objet.methode(paramètres)`
Accès aux attributs depuis l'extérieur de la classe	`objet.attribut`

> 💡 Si on veut respecter le *principe d'encapsulation*, il ne faut pas accéder directement aux attributs mais le faire à travers des méthodes appelées `getter` en *lecture* et `setter` en *écriture*.

Définition d'une classe :

```
import math

class Point:
    """Classe de fabrication d'un point du plan"""

    def __init__(self, x, y):
        """Constructeur d'un point à partir de ses coordonnées"""
        self.x = x
        self.y = y

    def distance(self, autre):
        """Méthode qui renvoie la distance d'un point à un autre point"""
        return math.sqrt((self.x - autre.x) ** 2 + (self.y - autre.y) ** 2)
```

Création des objets :

```
# code client
>>> p1 = Point(10, -4)  # construction d'un premier point
>>> p2 = Point(-2, 3)    # construction d'un second point
```

L'affichage par défaut d'un objet n'est pas explicite c'est pourquoi on peut vouloir définir une méthode spéciale `__str__` pour l'affichage qui sera appelée de façon simplifée avec `str(objet)` :

```
>>> p2
<__main__.Point at 0x7f42b8f21a30>
```

Accès aux attributs :

```
>>> p1.x
10
>>> p1.y
-4
```

Appel de méthode :

```
d12 = p1.distance(p2) # distance entre p1 et p2
```