TP n°3 *Définir des objets : l'exemple de la bibliothèque*

L'approche objet en programmation

En programmation, l'approche objet consiste à penser le système d'information, ou encore le modèle du monde, sous la forme d'un ensemble d'entités. Chaque entité a ses propres caractéristiques et ses propres fonctionnalités. Chaque entité est maîtresse de son évolution au cours du temps.

Dans la fiche de TP numéro 2, vous avez découvert la classe `str`, comment on instanciait un objet de cette classe, et comment on pouvait appliquer des méthodes à des objets de cette classe.

Dans ce TP, vous allez maintenant développer les classes pour gérer une bibliothèque. Vous allez donc créer vos propres classes. Pour cela, vous allez vous appuyer sur le TP numéro 1 (gestion d'une bibliothèque). Ce présent TP sera aussi l'occasion de revenir sur le vocabulaire associé à la programmation orentée objet.

Ma première classe

Dans le TP Bibliothèque, vous avez manipulé trois types d'entités : les usagers, les livres, et les emprunts.

Prenons l'exemple des usagers. Chaque usager est défini par les mêmes informations (nom, prénom, date de naissance, liste des emprunts), mais chaque usager a ses propres valeurs pour ces informations. En plus de ces caractéristiques, un usager est détenteur de plusieurs fonctionnalités comme "emprunter un livre".

Nous allons définir une classe des usagers, qui va déterminer ce qu'est un usager type, une sorte de squelette d'usager, une coquille vide. Il faudra aussi définir comment à partir de cette coquille vide, on construit un usager réel.

Voici votre première classe :

```
class Usager:
    def __init__(self,nom,prenom,naissance):
        self.nom = nom
        self.prenom = prenom
        self.date_naissance = naissance
        self.emprunts = []
```

Les attributs d'un objet

Le code Python ci-dessus définit la classe `Usager`. Cette définition précise qu'un usager est constitué d'un nom (`nom`), d'un prénom (`prenom`), d'une date de naissance (`date_naissance`) et de la liste de ses emprunts (`emprunts`). On appellera **attribut** chacune de ces caractéristiques. Donc, `nom`, `prenom`, `date_naissance`, `emprunts` sont les attributs de la classe `Usager`.

Le constructeur, et les composants d'une classe

La classe `Usager` contient une méthode, qui s'appelle `__init__`. Cette méthode construit un objet de la classe (ce qui inclut la réservation d'une zone mémoire dédiée). Toute classe doit contenir une telle méthode. Le terme **méthode** fait partie du vocabulaire de l'approche objet :

- Une **classe** est constituée d'**attributs** et de **méthodes**.
- Chaque individu représentant de la classe s'appelle un **objet**. On dit qu'il s'agit d'une **instance** de la classe.
- Pour construire un objet de la classe, il faut un **constructeur**. En Python, il s'agit de la méthode `__init__`.

La méthode `__init__` prend en argument au moins `self`, qui fait référence à l'objet en cours de construction. Ici, la ligne `self.nom = nom` se lit : "nom est un attribut de l'objet en cours de construction, et sa valeur initiale est donnée par le contenu du paramètre de la méthode `__init__` nommé `nom`. Comme on le remarque avec la ligne

```
self.date_naissance = naissance
```

le nom de l'attribut et celui de l'argument donnant la valeur initiale ne doivent pas nécessairement être les mêmes, mais c'est un usage courant que d'utiliser le même identifiant.

Construire un objet

Ici, nous n'avons fait que définir **comment** construire un objet de la classe `Usager`. Reste encore à utiliser cette définition dans un cas concret pour construire un objet (on dit aussi que l'on appelle le constructeur). Voici le code permettant d'instancier un usager particulier (`u1` est l'identifiant de l'objet construit) :

```
u1 = Usager("Nonyme","Albert","17/09/2000")
```

Notons qu'`u1` est une variable, et que la méthode `__init__` s'appelle via le nom de la classe (ici `Usager`). Lors de l'appel au constructeur, on passe des valeurs à chaque paramètre de la méthode `__init__` sauf le premier (`self`).

En effet, la méthode `__init__` de la classe `Usager` liste 4 arguments (`self`, `nom`, `prenom`, `naissance`), mais l'appel n'en donne que 3 (`nom`, `prenom`, `naissance`). En fait, le `self`, à l'appel du constructeur, est implicite : au moment de la construction, il est ajouté, et fait référence à l'objet en cours de construction.

Manipuler un objet

Maintenant, l'objet `u1` est construit, et on peut l'utiliser dans divers contextes comme par exemples :

```
print(u1.nom)
u1.nom = "Toto"
print(u1.nom)
u1.emprunts.append("23")
print(u1.emprunts)
```

L'opérateur `.` permet d'accéder à un attribut de l'objet. Si la valeur de l'attribut est elle-même un objet, l'opérateur `.` peut être "chainé" ; en effet, remarquez l'usage `u1.emprunts.append("23")` : le premier `.` permet d'accéder à la liste des emprunts de `u1`. Cette liste est un objet de type `List`, qui dispose d'une méthode `append`. Le deuxième `.` permet d'appliquer `apprend` à cette liste.

Finalement, le code de la classe `Usager` est le suivant (la première ligne est un commentaire indiquant à l'éditeur et à l'interpréteur python que le code source est encodé en UTF8, et le code contient dans un même fichier la *définition* et l'*utilisation* de la classe `Usager`) :

```
# coding utf-8

class Usager:
    def __init__(self,nom,prenom,naissance):
        self.nom = nom
        self.prenom = prenom
        self.date_naissance = naissance
        self.emprunts = []

if __name__ == '__main__':
    u1 = Usager("Nonyme","Albert","17/09/2000")
    print(u1.nom)
    u1.nom = "Toto"
    print(u1.nom)
    u1.emprunts.append("23")
    print(u1.emprunts)
```

Exercice 1. Créez un fichier nommé `usager.py`, et reprenez le code ci-dessus. Exécutez-le pour vérifier qu'on obtient le résultat attendu.

Exercice 2. Ajoutez à la classe `Usager` l'attribut `date_renouvellement`, qui est une date qui définit quand l'abonnement de l'usager se termine. Mettez à jour le constructeur (ajout d'un nouvel argument) et testez le en affichant la nouvelle information.

Ajout de méthode

Pour le moment, la classe `Usager` ne contient qu'une seule méthode : `__init__`. Nous allons maintenant ajouter une autre méthode (appelée `age`) :

```
from datetime import datetime

class Usager:
    ...

    def age(self):
        """ retourne l'age de l'usager """
        aujourdhui = datetime.now()
        dn = datetime.strptime(self.date_naissance, '%d/%m/%Y')
        resultat = aujourdhui.year - dn.year
        ## on enlève 1 an si la date actuelle est antérieure à
        ## celle de l'anniversaire
        if aujourdhui.month < dn.month:
            resultat -= 1
        else:
            if aujourdhui.month == dn.month and aujourdhui.day < dn.day:
                resultat -= 1
        return resultat
```

La fonction `age` est définie au sein de la classe `Usager`, comme l'indentation devrait vous le faire comprendre. Cette fonction ne prend qu'un seul argument, `self`, nécessaire afin qu'on puisse se référer à l'usager sur laquelle la fonction s'applique. `dn` et `aujourdhui` sont des objets de type `datetime` qui disposent des attributs `year`, `month`, et `day`. On ne commente pas plus ici l'algorithme de calcul de l'âge.

À noter, dans le module `datetime` on importe la classe du même nom.

Exercice 3. Ajoutez à la classe `Usager` une méthode nommée `renouvellement` qui renvoie `True` si la date de renouvellement de l'usager est dépassée, `False`, sinon.

Une fonction peut bien sûr modifier les valeurs des attributs de l'objet sur lequel elle s'applique. Par exemple, voici la méthode `renouveler`, qui renouvelle l'abonnement de l'usager. La date du prochain renouvellement est repoussée d'un an à partir de celle du renouvellement :

```
class Usager:
    ...

    def renouveller(self):
        """ décale la date de renouvellement d'un an """
        dr = self.date_renouvellement
        annee = int(dr[-4:])
        jour_mois = dr[:-4]
        self.date_renouvellement = jour_mois + str(annee + 1)
```

Exercice 4. Ajoutez à votre classe cette méthode, testez-la en affichant la date de renouvellement avant et après l'application de `renouveller`.

Exercice 5. En réfléchissant, on se dit qu'il aurait plus utile de fixer la date de renouvellement automatiquement lors de la création de l'objet (à date du jour + un an). Mettez à jour `init` pour ce faire. Mettez aussi à jour la partie test.

Jusqu'à maintenant, dans nos exemples, une méthode ne prend qu'un seul argument, nécessaire : `self`. Mais, on peut bien sûr créer des méthodes avec plus d'arguments. Par exemple, dans le TP numéro 1, un usager pouvait changer de nom. Reproduisons cette possibilité en ajoutant la méthode `changer_nom` à la classe `Usager` :

```
class Usager:
    ...

    def changer_nom(self,nvnom):
        self.nom = nvnom
```

Et on peut utiliser la nouvelle méthode :

```
u1.changer_nom("Dupont")
```

Remarquez bien que la définition liste 2 arguments, mais que l'appel n'en fournit qu'un : encore une fois, le `self` est automatiquement lié à `u1` au moment de l'appel à `changer_nom`.

Mais, pourquoi avoir utilisé une méthode pour changer le nom d'un usager ? Après, tout, autant opérer directement l'affectation que d'appeler la méthode. Certes, mais passer par une méthode de l'objet qui se charge de l'affectation a un avantage : l'objet maîtrise les conditions de ses modifications. Imaginons que un usager est un attribut `ancien_nom`. Alors, la modification de `nom` devrait impliquer automatiquement la modification de `ancien_nom` afin de maintenir la cohérence de l'usager. Faire directement un `u1.nom = "Dupont"` n'assure pas ce maintient, car on intervient sur l'objet sans qu'il puisse maîtriser cette modification. Dans l'idéal, pour chaque attribut, il faudrait créer une méthode de lecture (on parler de **getter**) et une méthode de modification (on parle de **setter**). Mais cela est laborieux, et on ne le fait pas systématiquement. Notez que des environnements de programmation évolués génèrent automatiquement les getter et les setter quand on crée un attribut.

Exercice 6. Créez une méthode qui ajoute un emprunt à la liste des emprunts d'un usager. Un emprunt est donné par une variable. Inutile (pour le moment) de déterminer le type de cette variable (puisque le typage est dynamique en python), nous allons voir qu'il s'agira en fait aussi d'un objet.

L'ensemble des usagers

On pourrait continuer à modéliser l'ensemble des usagers sous la forme d'un dictionnaire. Mais il s'agirait alors d'un dictionnaire d'instances de la classe `Usager` :

```
usagers = {}
usagers["u1"] = Usager("Nonyme","Albert","17/09/2000")
usagers["u2"] = Usager("Mini","Cathy","12/12/1995")
```

Mais nous allons plutôt définir une classe `Usagers`, qui contiendra ce dictionnaire des usagers, et des fonctions (méthodes) permettant d'agir sur ces usagers. L'idée est de rendre indépendant l'utilisation de la bibliothèque, de sa structure interne (un programme utilisateur de la bibliothèque ne doit pas avoir besoin de savoir quelles sont les structures utilisées en interne, il fonctionnera qu'il s'agisse de listes, dictionnaires ou autre).

Exercice 7. Créez la classe `Usagers` dans un fichier `usagers.py`. Cette classe contient un attribut nommé `usagers` (le dictionnaire des usagers). La constructeur prend en argument une chaîne de caractères qui est le nom du fichier csv contenant la liste des usagers. reprenez la fonction `charger_usagers`, adaptez-la pour créer le constructeur.

Exercice 8. Créez une méthode `changer_nom_usager` qui prend en argument un identifiant d'usager, et qui modifie le nom de l'usager correspondant. Il faut donc recherche l'usager dans la dictionnaire, et appliquer la méthode `changer_nom` sur cet usager.

Exercice 9. De même, adaptez les fonctions `ajouter_mot_cle`, `lister_usagers_majeurs` en ajoutant des méthodes à la classe `Usagers`. Il vous faudra sans doute ajouter la méthode `majeur` à la classe `Usager`.

Exercice 10. Écrivez une fonction `majeurs` qui retourne la liste des usagers majeurs. Cette fonction devra pouvoir être appelée par exemple comme suit (`users` est une variable désignant l'ensemble des usagers) :

```
print(users.majeurs())
```