Intro_POO

November 1, 2020

1 Programmation orienté objet (POO)

1.1 Le concept d'objet

Dans la vie réelle, un objet possède des caractéristiques et nous permet de réaliser des actions. Un chat par exemple possède une taille, une couleur, un nom, etc. ce sont ses caractéristiques, il peut miauler, se déplacer etc., ce sont les actions qu'il peut réaliser.

Le concept d'objets en informatique s'inspire fortement de cette définition de la vie réelle : on va appeler "objet" un bloc cohérent de code qui possède ses propres variables ou attributs (équivalents des caractéristiques des objets de tous les jours) et fonctions ou méthodes (qui sont les actions que l'objet peut réaliser). Ces objets vont interagir entre eux et avec le monde exterieur.

En Python, tout est objet: int, float, dict etc. Nous allons apprendre à créer de nouvelles classes d'objet.

1.2 Création d'une première classe

Les classes sont les principaux outils de la POO. En python, on définit une classe en suivant la syntaxe: `class NomClasse:` Il est préférable d'utiliser pour des noms de classes la convention dite *Camel Case*.Cette convention n'utilise pas le signe souligné _ pour séparer les mots.

Le principe consiste à mettre en majuscule chaque lettre débutant un mot. Créons une classe **Point** définie à l'aide de ses coordonnées:

```
In [ ]: class Point:
            """
            définition d'un point géométrique
            """
            pass
```

Lorsqu'on crée un objet à partir d'une classe avec l'instruction `p1= Point()`, on dit que l'on instancie une classe (on crée une nouvelle instance de cette classe). Créez la classe point en recopiant le code ci-dessus, puis **instanciez** un objet point p1. Affichez-le et vérifiez son type.

Bien sûr, cette classe ne fait pas grand chose. Ajoutons deux variables sur cette instance: son abscisse et son ordonnée.

1.3 Attributs

On utilise la notation pointée que vous connaissez bien pour accéder aux attributs de l'objet que l'on peut aussi appeler variables d'instances:

```
In [4]: p1.x = 4
        p1.y = 10
        print(p1.x)
```

```
4
```

Ces attributs sont **encapsulés** dans l'objet et sont donc distincts d'autres variables pouvant porter le même nom. Nous pourrions écrire: x = p1.x sans que cela prête à confusion.

Les objets crées peuvent être passés comme argument dans une fonction; vous l'avez déjà fait avec les objets list, dict etc. De plus, ces objets pourront être réutilisés dans d'autres classes: cela s'appelle la composition. Une classe **Vecteur** pourrait avoir besoin d'une classe **Point**.

1.4 Méthodes

Les méthodes se définissent comme des fonctions, sauf qu'elles se trouvent dans le corps de la classe. Les attribut et les méthodes constituent donc l'objet et sont encapsulés dans cet objet. APr exemple, vous avez déjà utilisé la méthode append sur des objets de type list

```
In [5]: class Voiture():
            def rouler(self):
                print("VRrrooOmMM")
```

Les méthodes d'instance prennent toujours en premier paramètre self: référence à l'instance de l'objet manipulé car il faut toujours pouvoir désigner l'instance à laquelle la méthode sera associée. Ajoutez une méthode affichercouleur() qui affichera *la voiture est rouge* lors de son appel. Instanciez un objet de type Voiture et vérifiez que les deux méthodes fonctionnent.

Evidemment, vous vous dites que les instances de la classe Voiture ne seront pas toutes rouges. Nous voudrions donc pouvoir passer la couleur en paramètre à l'instanciation de l'objet. ### La méthode **init()** Une méthode d'instance spéciale est prévue pour cela: c'est le **constructeur de la classe** et elle porte toujours le nom réservé ***init***, le double underscore __ indique que la méthode est réservée.

On y définit les attributs d'une instance en suivant cette syntaxe : self.nom_attribut = valeur.

```
In [ ]: def __init__(self, couleur= "non renseignée", marque="non renseignée"):
            self.couleur = couleur
            self.marque = marque
```

In [ ]: Insérez cette méthode en haut de la classe `Voiture` puis instanciez une voiture de la couleur et de la marc

1.4.1 La méthode str()

On remarque que l'affichage avec print n'est pas vraiment convivial, on peut modifier cela avec une autre méthode réservée:la méthode __str__ . Lorsque l'on passe des paramètres à la fonction print, celle-ci appelle la fonction str() pour les convertir en chaînes de caractères. Définir cette méthode à l'intérieur d'une classe permet d'utiliser cette fonction str() sur les instances de l'objet. **str** doit se comporter comme str() et donc renvoyer une chaine de caractère. Ajoutez cette méthode à la classe avec les lignes suivantes et vérifiez que la fonction print se comporte comme attendu.

```
In [7]: def __str__(self):
            return str(f"La voiture est une {self.marque} {self.couleur}")
```

Note: il existe bien sûr d'autres méthodes réservées qu'il peut être utile de connaitre comme la méthode __add__ qui permet d'adapter le comportement de l'opérateur + sur les objets créés ou __repr__ qui fonctionne un peu comme __str__ mais pour la représentation de l'objet dans le shell.

1.5 Composition

Ecrivons une classe Garage qui comportera plusieurs éléments de type Voiture.

```
In [11]: class Garage():
             def __init__(self):
                 self.stock = [Voiture("Noire", "Lotus"), Voiture("Bleue", "Peugeot"),
                               Voiture("Verte", "Renault"), Voiture("Grise", "DeLorean" )]
```

Nous avons composé un nouvel objet Garage en prenant des objets Voiture comme attributs, ce procédé s'appelle la **composition**.

1.6 Espace de noms

Comme dejà vu pour les fonctions, chaque classe possède son propore espace de noms: 1. Chaque classe possède son propre espace de noms. Les variables qui en font partie sont appelées variables de classe ou attributs de classe. 2. Chaque objet instance (créé à partir d'une classe) obtient son propre espace de noms. Les variables qui en font partie sont appelées variables d'instance ou attributs d'instance. 3. Les classes peuvent utiliser (mais pas modifier) les variables définies au niveau principal. 4. Les instances peuvent utiliser (mais pas modifier) les variables définies au niveau de la classe et les variables définies au niveau principal.

1.7 A faire

1.7.1 Exercice

Définissez une classe Domino() qui permette d'instancier des objets simulant les pièces d'un jeu de dominos. Le constructeur de cette classe initialisera les valeurs des points présents sur les deux faces A et B du domino (valeurs par défaut = 0). Deux autres méthodes seront définies : 1. une méthode affiche_points qui affiche les points présents sur les deux faces 2. une méthode valeur() qui renvoie la somme des points présents sur les 2 faces.

Exemples d'utilisation de cette classe :

```
>>> d1 = Domino(2,6)
>>> d2 = Domino(4,3)
>>> d1.affiche_points()
face A : 2 face B : 6
>>> d2.affiche_points()
face A : 4 face B : 3
>>> print "total des points :", d1.valeur() + d2.valeur()
15
```

1.7.2 Exercice

Définissez une classe CompteBancaire(), qui permette d'instancier des objets tels que compte1, compte2, etc. Le constructeur de cette classe initialisera deux attributs d'instance nom et solde, avec les valeurs par défaut 'Dupont' et 1000. Trois autres méthodes seront définies : - deot(somme) permettra d'ajouter une certaine somme au solde - rtrait(somme)permettra de retirer une certaine somme du solde - affiche()permettra d'afficher le nom du titulaire et le solde de son compte. **Exemples d'utilisation de cette classe :**

```
>>> compte1 = CompteBancaire('Haddock', 800)
>>> compte1.depot(350)
>>> compte1.retrait(200)
>>> compte1.affiche()
Le solde du compte bancaire de Haddock est de 950 euros.
>>> compte2 = CompteBancaire()
>>> compte2.depot(25)
>>> compte2.affiche()
Le solde du compte bancaire de Dupont est de 1025 euros.
```