Programmation orientée objet en python

Institut supérieur du numérique
(SupNum)

April 8, 2023

Approche de la POO

- La programmation orientée objet (POO) est une approche qui modélise le monde réel à travers des objets et des classes.
- Toutes les variables en POO sont des objets associés à une classe.
- Une classe est un type complexe qui peut contenir des données membres (attributs) et des fonctions membres (méthodes).

Définition d'une classe

- Une classe est utilisée pour instancier des objets du même type
- Pour définir une classe en Python, on utilise le mot clé "class", suivi du nom de la classe et des deux points. Par exemple :

```
1 class MaClasse:
```

- il est possible d'ajouter des données membres et des fonctions membres:

```
1 class MaClasse:
2     # données membres
3     # fonctions membres
4     def fonctionMembre(self):
5         print("Ceci est une fonction membre")
```

Notre nouvelle classe MaClasse possède maintenant une méthode fonctionMembre applicable sur tous ses objet

Définition d'un objet

- Un objet est une instance de classe, c'est-à-dire une collection de
  - données membres représentent les attributs ou les propriétés spécifiques à l'objet
  - fonctions membres, appelées méthodes, permettent de manipuler ces données membres
- Un objet est appelé également une instance d'une classe
- En Python, tout est objet
- On peut créer autant d'objets de classe que nécessaire

Création d'un objet

- Une fois que nous avons défini la classe, nous pouvons instancier des objets à partir de cette classe en utilisant la syntaxe suivante :

```
1 monObjet=MaClasse()
```

- Ceci crée une instance de la classe "MaClasse" et l'assigne à la variable "monObjet".
- Nous pouvons maintenant appeler les fonctions membres de cette instance en utilisant la syntaxe suivante :

```
1 monObjet.fonctionMembre() #affiche Ceci est fonction membre
```

Référence à l'instance (self)

- Vous avez peut-être remarqué la présence du paramètre spécial "self" dans la définition de la fonction.

```
1  def fonctionMembre(self):
```

- Ce paramètre est utilisé pour faire référence à l'instance de la classe sur laquelle la fonction est appelée.
- Cependant, lors de l'appel de la fonction membre, vous n'avez pas besoin de spécifier le paramètre "self".
- En effet, Python ajoute automatiquement l'instance de la classe en tant que premier paramètre de la fonction.
  - L'appel

```
1  monObjet = MaClasse()
2  monObjet.fonctionMembre()
```

  - Traduit en

```
1  MaClasse.fonctionMembre(monObjet)
```

Référence à l'instance (self)

- Toute fonction membre d'une classe a au moins un paramètre : l'instance. Par convention on le nomme self
- A l'intérieur de la fonction membre, self désigne l'instance sur laquelle la fonction est appelée, par exemple

```
1  class MaClasse:
2      maVariable = 0
3      def fonctionMembre(self):
4          print("La valeur de maVariable est :", self.maVariable)
5
```

- Nous pouvons ensuite instancier un objet de cette classe et accéder à sa variable de données membres à travers sa fonction membre

```
1  mon_objet = MaClasse()
2  mon_objet.maVariable = 42
3  mon_objet.fonctionMembre()
4  #affiche le message "La valeur de maVariable est : 42"
```

Constructeur (1/2)

- Le constructeur est une fonction membre spéciale d'une classe qui est appelée automatiquement lorsqu'un nouvel objet de la classe est créé.
- Le constructeur est généralement utilisé pour initialiser les données membres de la classe avec des valeurs par défaut.
- Il est défini avec la fonction membre spéciale "init".
- Il peut prendre des paramètres, qui sont utilisés pour initialiser les données membres de la classe.

```
1 class Point:
2     def __init__(self, x=0, y=0):
3         self.x = x
4         self.y = y
```

Constructeur (2/2)

- Dans cette classe, le constructeur prend deux paramètres optionnels "x" et "y", qui sont utilisés pour initialiser les données membres "x" et "y" de la classe.
- Si aucun paramètre n'est fourni, les valeurs par défaut pour "x" et "y" sont toutes deux définies à 0.
- Nous pouvons créer un objet de cette classe en utilisant la syntaxe suivante :

```
1 monPoint = Point(3, 4)
```

- Nous pouvons accéder aux données membres de l'objet en utilisant la notation pointée, comme suit :

```
1 print("La valeur de x est :", monPoint.x)
2 print("La valeur de y est :", monPoint.y)
```

Encapsulation (1/3)

- L'encapsulation sert à protéger les attributs et les méthodes d'une classe de l'accès non autorisé.
- En Python, l'encapsulation est réalisée en utilisant des attributs et des méthodes privés.
- Un attribut ou une méthode privé(e) commence par deux tirets bas (__).
- Les attributs privés ne peuvent pas être accédés directement depuis l'extérieur de la classe, tandis que les méthodes privées ne peuvent être appelées que depuis l'intérieur de la classe

Encapsulation (2/3)

- L'encapsulation est implémentée à l'aide de l'utilisation de méthodes d'accès, également appelées "getters" et "setters".
- Les "getters" permettent de récupérer la valeur d'une variable, tandis que les "setters" permettent de modifier cette valeur.
- En encapsulant les variables et en n'autorisant l'accès qu'aux "getters" et "setters", on peut mieux contrôler l'utilisation des données et éviter les erreurs.

Encapsulation (3/3)

```
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    def get_name(self):
        return self._name

    def set_name(self, name):
        self._name = name

    def get_age(self):
        return self._age

    def set_age(self, age):
        if age < 0:
            print("l'age doit être un entier positif")
        self._age = age
```

Encapsulation

- En utilisant les méthodes d'accès, nous pouvons accéder aux variables protégées de la classe Person de la manière suivante :

```
1  p = Person("Amadou", 25)
2  print(p.get_name())   # affiche "Amadou"
3  p.set_name("Fatma")
4  print(p.get_name())   # affiche "Fatma"
5  print(p.get_age())    # affiche 25
6  p.set_age(30)
7  print(p.get_age())    # affiche 30
```

Encapsulation: exemple

```
class Etudiant:
    def __init__(self, matricule, nom, cours=None):
        self._nom = nom
        self._matricule = matricule
        self._cours = cours or []

    def get_nom(self):
        return self._nom

    def set_nom(self, nom):
        self._nom = nom

    def get_matricule(self):
        return self._matricule

    def set_matricule(self, matricule):
        self._matricule = matricule
```

Encapsulation: exemple

```
    def ajouter_cours(self, cours):
        self._cours.append(cours)
    def supprimer_cours(self, cours):
        if cours in self._cours:
            self._cours.remove(cours)
        else:
            print("Cours n'existe pas.")

    def get_cours(self):
        return self._cours
```

Encapsulation: exemple

```
etudiant = Etudiant("22003","Mariem")
etudiant.ajouter_cours("Python")
etudiant.ajouter_cours("C++")
print(etudiant.get_nom())     # affiche "Mariem"
print(etudiant.get_matricule())       # affiche "22003"
print(etudiant.get_cours()) # affiche ["Python", "C++"]
etudiant.supprimer_cours("c++")
etudiant.set_nom("Ahmed")
etudiant.set_matricule("22002")
print(etudiant.get_nom())     # affiche "Ahmed"
print(etudiant.get_matricule())       # affiche "22002"
```