TP N°5 : Programmation Orientée Objet

1 Attributs d'instances

Les attributs d'instances sont des attributs déclarés à l'intérieur du constructeur de la classe, accessibles via le nom de l'instance (objet) et leurs valeurs ne sont pas partagés entre les instances de cette classe. Il est toujours possible en Python, après la définition d'une classe, d'ajouter des attributs pour un objet instance de cette classe.
Exemple :

```
class etudiant:
    def __init__(self, prenom, nom):
        self.prenom = prenom
        self.nom = nom

e1 = etudiant("ali","suissi")
e1.cin = 12345678 # ajouter l'attribut cin à l'instance e1
print(e1.__dict__)
e2 = etudiant("jihen","suissi")
print(e2.cin) # AttributeError: 'etudiant' object has no attribute 'cin'
print(e2.__dict__)
```

2 Attributs de classes

Les attributs de classes sont des attributs déclarés en dehors du constructeur, accessibles via le nom de la classe et leurs valeurs sont partagés entre tous les objets de cette classe.
Exemple :

```
class etudiant:
    institut = "ipein" # attribut de classe
    def __init__(self, prenom, nom):
        self.prenom = prenom # attribut d'instance
        self.nom = nom # attribut d'instance

print("institut" in etudiant.__dict__)
print(etudiant.institut)
e1 = etudiant("ali","suissi")
e1.cin = 12345678 # ajouter l'attribut cin à l'instance e1
print(e1.__dict__) #{'cin': 12345678, 'nom': 'suissi', 'prenom': 'ali'}
print(e1.institut) # accès à l'attribut de classe
e1.institut = "enit" # ajouter l'attribut institut à l'instance e1
print(e1.__dict__) #{'prenom': 'ali', 'cin': 12345678, 'nom': 'suissi', 'institut':
 ↪  'enit'}

e2 = etudiant("jihen","suissi")
print(e2.institut) # accès à l'attribut de classe
print(e2.__dict__)#{'nom': 'suissi', 'prenom': 'jihen'}

#modifier l'attribut de classe
etudiant.institut = "ensi"
print(etudiant.institut)
print(e1.institut)
print(e2.institut)
```

Exercice 1

Soit les classes A et B suivantes. Quelle sera la sortie du programme suivant?

```
class A:
    """ Définition de la classe A """
        i = 1
        k = 3
        def __init__(self):
            self.i = 2
        def f(self):
            return self.g()
        def g(self):
            return 'A'
class B(A):
    def g(self):
         return 'B'

a = A(); b = B() ;
type(a) ; help(a) ;
print(a.g(), b.g()) # affiche :
print(a.f(), b.f()) # affiche :
print(A.i) # affiche :
print(A().i) # affiche :
a.j = 0 #Ajouter un attribut 'j' à l'objet a
print(a.j) # affiche :
print(b.j) # affiche :
b = a ; print(b.j)  # affiche :
b.j = -1 ; print(a.j) # affiche :
```

3 Attributs privées

Un attribut d'instance privé est accessible uniquement à l'intérieur de la classe où il est définie.
Syntaxe : `self.__attribut = valeur initiale`
Exemple :

```
class etudiant:
    def __init__(self, prenom, nom, cin):
        self.prenom = prenom # attribut d'instance publique
        self.nom = nom # attribut d'instance publique
        assert str(cin).isdigit() == True and len(str(cin)) == 8, "cin invalide"
        self.__cin = str(cin)  # attribut d'instance privé

e1 = etudiant("ali","suissi","12345678")
print(e1.cin) # AttributeError: 'etudiant' object has no attribute 'cin'
print(e1.__dict__) #{'_etudiant__cin': '12345678', 'nom': 'suissi', 'prenom': 'ali'}
e1.cin = "1234abcd" # erreur 1 : cin invalide (pas de contrôle de saisie)
                    # erreur 2 : ajout d'un attribut d'instance cin
print(e1.__dict__) #{'prenom': 'ali', 'cin': '1234abcd', 'nom': 'suissi',
 ↪  '_etudiant__cin': '12345678'}
print(e1._etudiant__cin)
```

4 Méthodes privées

Une méthode privée est accessible uniquement à l'intérieur de la classe où elle est définie.
Syntaxe : `def __méthode(self):`
Exemple :

```
class etudiant:
    def __init__(self, prenom, nom, cin):
        self.prenom = prenom # attribut d'instance publique
        self.nom = nom # attribut d'instance publique
        assert self.__verifier_cin(cin) == True , "cin invalide"
        self.__cin = str(cin)  # attribut d'instance privé

    def __verifier_cin(sef,cin): # méthode privée
        return str(cin).isdigit() == True and len(str(cin)) == 8

    def modifier_cin(self,cin): # méthode pubique
        assert self.__verifier_cin(cin) == True , "cin invalide"
        self.__cin = cin

e1 = etudiant("ali","suissi","12345678")
cin = input("Nouveau cin:")
e1.modifier_cin(cin) # appel à une méthode publique
# appel à une méthode privée : obj._classe__methode()
print(e1._etudiant__verifier_cin("1234abcd"))
```

Exercice 2

1. Créer une classe Intervalle possédant une méthode __init__ permettant d'initialiser une borne inférieure (borne_inf) et une borne supérieure (borne_sup) pour un objet de type Intervalle. Vérifier que les bornes sont numériques, positives, non nulles et placées dans le bon ordre, sinon générer une exception de type « IntervalError » affichant le message d'erreur « Erreur : Bornes invalides ! ». Le type « IntervalError » est une Exception à définir.
   Stocker dans les variables a, b et c, respectivement, les intervalles [1,6], [4,8] et ["0.35","0.8"].
2. En effet, avec les contrôles définis dans la méthode __init__, on ne peut plus créer un intervalle mal formé. Toutefois, il est toujours possible d'écrire directement a.borne_sup = -2, ce qui mettra - 2 dans la borne supérieure de l'intervalle a.
   Modifier la portée des attributs borne_inf et borne_sup afin qu'ils ne seront visibles que depuis les méthodes de la classe, mais pas de l'extérieur. Essayer de modifier la valeur de la borne supérieure de a à -5.
3. Pour modifier une valeur de l'intervalle, écrire maintenant dans la classe Intervalle une méthode modif_borne_sup qui permettra de protéger la borne supérieure en ne pouvant y écrire que des nombres supérieurs à la borne inférieure.
4. Ajoutez une méthode modif_borne_inf à la classe Intervalle. Faites attention à ce qu'une valeur négative ne puisse pas être enregistrée.
5. Écrivez deux méthodes d'accès borne_inf(self) et un borne_sup(self)qui retourneront les valeurs des bornes.
6. Écrire une méthode spéciale __str__(self) permettant de retourner une chaine indiquant les valeurs des deux bornes de l'intervalle. Afficher l'intervalle a.
7. Écrire une méthode spéciale __contains__(self, val) qui teste si une valeur val appartient ou non à l'intervalle (cette méthode remplace l'opérateur in).
8. Écrire une méthode spéciale __add__(self, autre) qui retourne un nouvel Intervalle addition des deux intervalles. Exemple : [2,5] + [3,4] = [5,9]
9. Écrire une méthode spéciale __and__(self, autre) qui retourne l'intersection des deux intervalles et « None » si leur intersection est vide. Exemple : [2,5] [3,6] =[3,5]