Activités - Programmation objet

Exercice 1

Question 1 :

1. Écrire une classe `Eleve` dont les objets possèdent les attributs `prenom`, `classe`, `age`.
2. Instancier trois élèves de cette classe.
3. Ajouter à cette classe une méthode `est_majeur` qui renvoie `True` si l'élève est majeur et `False` sinon. Testez si tout fonctionne !

In [ ]:
```
# à vous de jouer !
```

Question 2 : Écrire une fonction `compare(eleve1: Eleve, eleve2: Eleve) -> str:` qui renvoie le nom de l'élève le plus âgé.

In [ ]:
```
# à vous de jouer !
```

Question 3 : Écrire une fonction `eleves_de_la_classe(classe: str, liste_eleves: list) -> list:` qui renvoie une liste avec le nom de tous les élèves de `liste_eleves` qui sont dans la classe `classe`.

Vous testerez en créant quelques élèves supplémentaires.

In [ ]:
```
# à vous de jouer !
```

Exercice 2 : La classe `Point`

On souhaite créer une classe `Point` permettant de construire des objets de type `Point` définis par une abscisse et une ordonnée.

Création de la classe

Question 1 : Créez une telle classe en Python et définissez la méthode spéciale `__init__` pour déclarer les attributs `x` (abscisse) et `y` (ordonnée) d'un objet `Point`.

In [ ]:
```
# à vous de jouer !
```

Question 2 : Créez deux instances `p1` et `p2` de cette classe ayant pour coordonnées respectives (2; 3) et (-1; 4).

In [ ]:
```
# à vous de jouer !
```

Question 3 : Quelles instructions permettent d'accéder aux attributs de l'objet `p1` ?

In [ ]:
```
# à vous de jouer !
```

Question 4 : Le point `p1` a changé de coordonnées qui sont maintenant (3, 4). Modifiez les attributs du point `p1`, en utilsant la notation pointée, afin qu'elles correspondent à ce changement.

In [ ]:
```
# à vous de jouer !
```

On souhaite également pouvoir translater un point et vérifier si deux points sont égaux. Il est donc nécessaire d'ajouter des méthodes à notre classe.

Question 5 : Compléter les méthodes `translater` et `egal` dans la classe `Point`.

In [ ]:
```
class Point:
    """Manipulation de points"""

    def __init__(self, abscisse, ordonnee):
        self.x = abscisse
        self.y = ordonnee

    # à compléter
```

Question 6 : Vérifiez que l'implémentation est correcte sur des exemples.

In [ ]:
```
# à vous de jouer !
```

Améliorations

Une représentation d'un point en chaîne de caractères

On souhaiterait obtenir une représentation "officielle" de notre objet de type `Point` sous la forme d'une chaîne de caractères *évaluable* de la forme `Point(x, y)`. On rappelle qu'il faut redéfinir la méthode `__repr__(self)` puisque c'est elle qui est appelée dans ce cas.

Question 7 : Définissez la méthode `__repr__(self)` afin d'obtenir la représentation demandée de nos points. Vous testerez ensuite si l'affichage est correct.

Exemple :

```
>>> p3 = Point(2, 3)
>>> p3.translater(1, -1)
>>> p3  # l'évaluation de p3 appelle la méthode __repr__
Point(3, 2)
```

In [ ]:
```
class Point:
    """Manipulation de points"""

    def __init__(self, abscisse, ordonnee):
        self.x = abscisse
        self.y = ordonnee

    def translater(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy

    def egal(self, other):
        return self.x == other.x and self.y == other.y
```

```
    # à compléter
```

> **Remarque** : comme la méthode spéciale `__str__` n'est pas définie, la fonction `print` appelle également la méthode `__repr__`. Vous pouvez essayer !

Un test d'égalité avec `==`

On rappelle que si on veut tester l'égalité entre deux objets d'une classe avec la notation habituelle (`objet1 == objet2`) il est nécessaire de définir la méthode spéciale `__eq__` de la classe car c'est elle qui est appelée dans ce cas.

Question 8 : Compléter l'implémentation de la classe par la méthode `__eq__`. Vous testerez ensuite si votre implémentation est correcte.

In [ ]:
```
# à vous de jouer !
```

Accès en lecture/écriture aux attributs

Ce n'est pas le cas en Python, mais dans beaucoup d'autres langages les attributs d'un objet sont privés, c'est-à-dire qu'il n'est pas possible d'y accéder ou de les modifier sans faire appel à des méthodes (publique) de la classe.

Ainsi, dans d'autres langages, on ne pourrait plus écrire `p.x` pour accéder à la valeur de l'abscisse du point `p` ou écrire `p.x = 5` pour la modifier car l'attribut `x` n'est pas accessible directement.

Pour accéder et modifier les valeurs des attributs on utilise alors des méthodes appelées :

- des *accesseurs* (ou *getters*) qui fournissent des informations relatives à l'état d'un objet, c'est-à-dire aux valeurs de certains de ses attributs (généralement privés) sans les modifier ;
- des *mutateurs* (ou *setters*) qui modifient l'état d'un objet, donc les valeurs de certains de ses attributs.

Par exemple, pour accéder à l'attribut `x` (abscisse) d'un point il suffirait d'ajouter la méthode

```
def lire_abscisse(self):
    return self.x
```

Question 9 : Implémentez les méthodes `lire_abscisse`, `lire_ordonnee`, `modifier_abscisse` et `modifier_ordonnee` de la classe `Point` pour lire et modifier les attributs d'un objet. Vérifiez ensuite sur des exemples.

In [ ]:
```
class Point:
    """Manipulation de points"""

    def __init__(self, abscisse, ordonnee):
        self.x = abscisse
        self.y = ordonnee

    def translater(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy

    def egal(self, other):
```

```
        return self.x == other.x and self.y == other.y

    def __repr__(self):
        return "Point(" + str(self.x) + ", " + str(self.y) + ")"

    __eq__ = egal  # pour aller plus vite, sinon, on remplace la méthode egal ci-dessus

    # méthodes à compléter
```

Exercice 3 : Une classe `Temps`

On souhaite créer une classe `Temps` définissant des objets de type `Temps` définis par des heures, minutes, secondes. On souhaite également pouvoir :

- obtenir une représentation officielle d'un temps sous la forme du chaîne de caractères évaluable de la forme `Temps(h, m, s)` (c'est la méthode `__repr__` qu'il faut définir correctement)
- obtenir un affichage d'un temps sous la forme d'une chaîne de caractères de la forme `h:m:s` lorsque l'on utilise la fonction `print()` (c'est la méthode `__str__` qu'il faut définir correctement)
- ajouter deux temps

Implémentation de la classe

Question 1 : Implémentez cette classe. Vous vérifierez ensuite si votre implémentation est correcte.

In [ ]:
```
# à vous de jouer !
```

Amélioration

On aimerait pouvoir ajouter deux instances `t1` et `t2` de notre classe `Temps` en utilisant le symbole usuel `+`.

Concrètement, on veut pouvoir ajouter deux temps en écrivant

```
t1 + t2
```

au lieu de

```
t1.ajouter(t2)
```

Question 2 : Lorsque l'on écrit `t1 + t2`, c'est la méthode spéciale `__add__` qui est invoquée. Définissez cette méthode dans la classe `Temps` et vérifiez si votre implémentation est correcte.

In [ ]:
```
# à vous de jouer !
```

Exercice 4

Écrire une classe `Player` qui :

- ne prendra aucun argument lors de son instanciation.
- affectera à chaque objet créé un attribut `energie` valant 3 par défaut.

- affectera à chaque objet créé un attribut `en_vie` valant True par défaut.
- fournira à chaque objet une méthode `blessure()` qui diminue l'attribut `energie` de 1.
- fournira à chaque objet une méthode `soin()` qui augmente l'attribut `energie` de 1.
- si l'attribut `energie` passe à 0, l'attribut `en_vie` doit passer à `False` et ne doit plus pouvoir évoluer.

Exemple d'utilisation de la classe :

```
>>> mario = Player()
>>> mario.energie
3
>>> mario.soin()
>>> mario.energie
4
>>> mario.blessure()
>>> mario.blessure()
>>> mario.blessure()
>>> mario.alive
True
>>> mario.blessure()
>>> mario.alive
False
>>> mario.soin()
>>> mario.alive
False
>>> mario.energie
0
```

In [ ]:
```
# à vous de jouer !
```

Exercice 5 : Une classe `Liste`

Interface et implémentation

L'objectif ici est de créer une classe `Liste` définissant des objets de type `Liste` à l'aide des 5 opérations primitives de cette structure de données.

Dans toute la suite on désignera par *liste*, des objets de la classe `Liste` dont voici l'interface :

- **`Liste()`** permet de créer une liste vide
- `Liste` **`.ajouter_en_tete(e)`** ajoute l'élément `e` en tête d'une liste
- `Liste` **`.premier()`** renvoie le premier élément d'une liste (sa tête)
- `Liste` **`.reste()`** renvoie le reste d'une liste (sa queue)
- `Liste` **`.est_vide()`** renvoie Vrai si la liste est vide et Faux sinon

Question 1 : Quelle suite d'instructions faut-il écrire pour créer une liste vide `L`, lui ajouter en tête successivement les valeurs 3, 1, 2 et enfin accéder à son premier élément et à son reste ?

Réponse :

Question 2 : Implémentez la classe `Liste` correspondante en utilisant le type prédéfini `list` de Python. *La méthode* `__init__` *créera une liste vide.*

In [ ]:
```
# à vous de jouer !
```

Question 3 : Vérifiez que l'implémentation est correcte en testant par exemple votre réponse à la première question.

In [ ]:
```
# à vous de jouer !
```

Améliorations

Une représentation sous forme d'une chaîne de caractères

On souhaite représenter nos objets de type `Liste` sous la forme d'une chaîne de caractères formée des éléments de la liste séparés par des virgules. Par exemple, la liste contenant les éléments 2, 1, 3 doit pouvoir être représentée par la chaîne `2, 1, 3`.

Question 4 : Définissez la méthode spéciale `__repr__` de la classe `Liste` pour obtenir cette représentation.

In [ ]:
```
# à vous de jouer !
```

Concaténation

On souhaite pouvoir concaténer deux instances de la classe `Liste`. Pour cela, on pourrait définir une nouvelle méthode `concatene` mais l'idée ici est de pouvoir concaténer deux listes en utilisant l'opérateur `+` usuel. Pour cela, il faut définir la méthode spéciale `__add__` de la classe `Liste` puisque c'est elle qui est appelée lorsque l'opérateur `+` est utilisé.

Question 5 : Implémentez cette méthode spéciale puis vérifiez que l'implémentation est correcte. **Attention** : cette méthode doit renvoyer un nouvel objet de type `Liste` (et pas un objet du type prédéfini `list` de Python !).

In [ ]:
```
# à vous de jouer !
```

Références :

- Documents ressources de l'équipe éducative du DIU EIL, Université de Nantes, Christophe DECLERCQ & Christophe JERMANN.
- Documentation officielle de Python sur les méthodes spéciales.
- Exercices 1 et 4 : exercices proposés par de Gilles Lassus : ici

---

Germain BECKER, Lycée Mounier, ANGERS