Aix*Marseille université
Initiative d'excellence
Année 2019-20

Diplôme Inter-Universitaire
Enseigner l'informatique au lycée

Bloc 4 : TP 6

Programmation orientée objet

1 Classe `Point`

1.1 Objectif

Le but de cet exercice est de vous familiariser avec les concepts de base de la programmation objet : *classe*, *instance*, *objet*, *attribut*, *méthode*, ...

On va se servir comme exemple du point en géométrie plane. Vous allez donc créer une classe permettant de représenter des points. En programmation objet, une classe permet de définir :

- une manière de créer des *objets* (pour cet exercice, des points dans un espace 2-dimensionnel),
- les services fournis par les objets, c'est-à-dire les fonctions (appelées *méthodes*) que l'on peut appeler sur les objets de la classe (pour cet exercice, des méthodes permettant de réaliser des opérations simples sur les points comme des translations, des rotations, des calculs de distance...),
- la structure des données des objets en définissant les *attributs* que possèdent les objets de la classe (pour cet exercice, les coordonnées en x et y des points) et
- la manière d'initialiser (on dit aussi *instancier*) les objets de la classe (pour cet exercice, le fait de pouvoir construire un point à partir de coordonnées).

1.2 Classe

On va donc commencer par définir une classe `Point`[1]. Pour cela, on va utiliser le mot-clé `class` suivi du nom de la classe et de `:`. Comme pour toute les autres syntaxes utilisant le `:` en Python, il doit y avoir ensuite un bloc d'instructions indenté contenant au moins une ligne (on peut écrire le code `pass` s'il n'y a finalement rien à faire dans ce bloc). Ce bloc nous servira à définir tout ce que contiendra la classe. Pour le moment, vous allez juste ajouter une documentation (texte entre triple guillemets) afin de décrire la classe que l'on vient de créer. Cela nous donne le code suivant :

```
class Point:
    """Un point en 2D"""
```

1.3 Instantiation

Vous venez donc de définir une classe `Point`. Vous pouvez à présent vous en servir pour créer des objets de cette classe, que l'on appellera aussi des *instances* de cette classe. Pour créer par exemple un nouvel objet `p`, il vous suffit de l'instancier en appelant `Point()` :

```
>>> p = Point()
```

1.3.1 Exercices

- Essayez d'afficher le point avec `print`. Qu'obtenez-vous ?
- Accédez à la documentation de votre point avec `p.__doc__`. Est-ce que cela correspond bien à la documentation que vous avez donnée à la classe ?
- Affichez le type de `p` avec la fonction `type`. Quel est le type de `p` ?

1. La convention en python est de commencer les noms des classes par une majuscule.

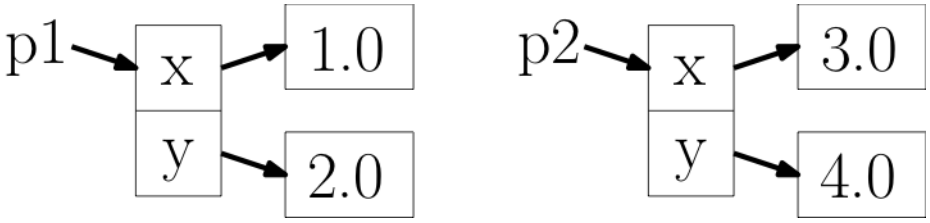

FIGURE 1 – État mémoire des points

1.4 Attributs

Une fois l'objet créé, il est possible de lui associer des données. Pour cela, on peut définir des attributs à l'intérieur des objets. La syntaxe pour accéder aux attributs d'un objet est `nom_objet.nom_attribut`. Pour créer/affecter un attribut, il suffit donc de mettre l'attribut dans le membre gauche d'une affectation avec cette syntaxe et la valeur que l'on souhaite affecter dans le membre droit. On peut aussi accéder à la valeur d'un attribut déjà affecté de la même manière qu'une variable à l'aide de la même syntaxe.

Le code suivant permet de créer deux points `p1` et `p2` et d'associer à chacun de ces points des attributs `x` et `y` de valeurs différentes. Vous pouvez faire cela avec le code suivant :

```
>>> p1 = Point()
>>> p1.x = 1.0
>>> p1.y = 2.0
>>> p2 = Point()
>>> p2.x = 3.0
>>> p2.y = 4.0
```

Vous pouvez remarquer que chaque instance (ici `p1` et `p2`) peut avoir des valeurs différentes pour ses attributs. Chaque instance de la classe `Point` contient donc des données qui lui sont propres comme l'illustre la figure qui donne le schéma mémoire après l'exécution du code ci-dessus.

1.4.1 Exercices

- Affichez les coordonnées des points `p1` et `p2`.
- Créez un point `p3` ayant les mêmes coordonnées que le point `p1`.
- Testez l'égalité entre `p1` et `p3` avec l'opérateur `==`. Que constatez-vous ? Comment est-ce que vous expliquez ce résultat ?

1.5 Méthodes

Une des notions clés de la programmation objet est la notion de méthodes. Ce sont des fonctions définies au sein d'une classe et qui s'appellent généralement avec une instance de la classe (un objet `Point` pour cet exercice). Vous allez commencer par ajouter la méthode suivante à votre classe `Point` (n'oubliez pas de mettre la ligne `import math` au début de votre fichier pour avoir la fonction `sqrt`) :

```
def distance_to_origin(self):
    return math.sqrt(self.x ** 2 + self.y ** 2)
```

On peut remarquer que cette méthode a un seul argument (`self` qui correspond à l'objet `Point` avec lequel la méthode va être appelée). Cette méthode prend donc un point en argument et renvoie la distance entre ce point et l'origine du système de coordonnées. On accède à une méthode de manière similaire qu'un attribut `nom_objet.nom_méthode(arguments)`. Il n'y a donc pas d'arguments entre les parenthèses de l'appel de cette méthode comme le montre le code ci-dessous que vous pouvez exécuter dans la console. En effet, le seul argument

à la déclaration de la méthode (`self`) correspond au point que l'on place avant le nom de la méthode lors de l'appel.

```
>>> p1 = Point()
>>> p1.x = 1.
>>> p1.y = 1.
>>> p1.distance_to_origin()
1.4142135623730951
```

1.5.1 Exercices

- Créez un point `p2` sans lui définir d'attributs `x` ou `y`. Appelez la méthode `distance_to_origin` avec `p2`. Que se passe-t-il ?
- Définissez une méthode `distance_to(self, other_point)` qui calcule la distance entre le point `self` et un autre point `other_point`.
- Créez deux points `p3` et `p4` puis donnez-leur des attributs `x` et `y` puis faites un appel `p3.distance_to(p4)`. Est-ce que le retour de la méthode correspond au résultat que vous espériez ?
- Écrivez une méthode `translate(self, dx, dy)` qui opère une translation du point de vecteur (dx, dy). Testez cette méthode sur les points.
- Écrivez une méthode `__repr__(self)` qui renvoie une chaîne de caractères représentant le point. Pour un point ayant des coordonnées `x` et `y` égales à `1.0` et `2.0` respectivement, la méthode `__repr__` devra renvoyer la chaîne de caractères `"(1.0, 2.0)"`.
- Affichez un point à l'aide de `print`, qu'observez-vous ? Qu'est-ce qui se passe selon vous ?
- Écrivez une méthode `__eq__(self, other)` qui teste si `self` et `other` sont des points ayant les mêmes coordonnées. L'argument `self` étant forcément un point, il suffit de renvoyer vrai si `other` est un point ayant les deux mêmes coordonnées et faux sinon. Pour tester si un objet est d'un certain type, vous pouvez utiliser la fonction `isinstance(object, Type)` qui renvoie vrai si `object` est une instance de `Type` et faux sinon.
- Retestez l'égalité entre deux points ayant les mêmes coordonnées. Que constatez-vous ? Qu'est-ce qui se passe selon vous ?

1.6 Constructeur

Votre classe `Point` est loin d'être complète. En effet, même si vous pouvez instancier des objets de type `Point`, ceux-ci sont créés sans aucune donnée (pas de valeurs de coordonnées). Il vous faut à chaque fois créer des attributs après avoir instancié l'objet car si vous ne le faites pas, il y a un risque d'erreur (comme l'a montré une question précédente). Pour pallier ce problème, vous allez ajouter un constructeur à votre classe `Point` qui va vous permettre d'*instancier* un point avec les bons attributs. En Python, le constructeur est une méthode nommée `__init__`. Elle prend au moins un argument appelé `self` qui est l'objet qu'on instancie (comme toute méthode d'instance) plus des arguments utiles pour instancier l'objet (comme les valeurs des coordonnées du point qu'on souhaite créer dans notre cas). Le rôle du constructeur en python est de créer les attributs de l'objet et de leur donner des valeurs.

Il vous faut donc ajouter le code suivant à la classe `Point` :

```
def __init__(self, x, y):
    self.x = x
    self.y = y
```

Pour appeler le constructeur, il vous suffit de mettre les coordonnées du point que vous souhaitez instancier entre les parenthèses. Cela se fait avec le code suivant :

```
>>> p1 = Point(1., 1.)
```

1.6.1 Exercices

— Créez deux points avec ce nouveau constructeur.
— Appelez les méthodes `distance_to_origin` et `distance_to` avec ces points.
— Essayez de construire un point en ne mettant aucun argument entre les parenthèses. Que se passe-t-il ?

2 Structure de formule

2.1 Objectif

Le but de cet exercice est de voir en action la puissance du paradigme objet et surtout de voir comment on peut utiliser des classes partageant une même interface (ensemble de méthode disponibles) pour créer facilement et proprement des programmes.

Dans cet exercice, vous allez implémenter des classes pour générer des formules. Chaque classe correspondra à un type de formule.

2.2 Constante et somme

Vous allez commencer par définir les classes :
— `Constant` pour les littéraux (constante),
— `Sum` pour la somme de deux formules (qui peuvent être pour le moment des constantes ou bien des sommes).

Vous allez donc créer un fichier `formula.py` qui contiendra au début le code suivant :

```
class Constant:
    def __init__(self, value):
        self.value = value

class Sum:
    def __init__(self, left_member, right_member):
        self.left_member = left_member
        self.right_member = right_member
```

2.2.1 Exercices

— Ajoutez dans les classes `Sum` et `Constant` une méthode `eval(self)` afin de pouvoir obtenir l'exécution suivante en terminal :

```
>>> formula = Sum(Constant(2), Sum(Constant(3), Constant(4)))
>>> formula.eval()
9
```

— Ajoutez une méthode `__repr__(self)` dans chacune des classes pour permettre l'exécution suivante :

```
>>> formula
(2+(3+4))
```

Comme vous l'avez vu à l'exercice précédent, la méthode `__repr__` doit renvoyer un chaîne de caractères qui correspond à une représentation de l'objet.

2.3 Ajout d'opérations

On va définir de nouvelles classes afin de pouvoir définir de nouveaux opérateurs et fonctions dans les formules.

2.3.1 Exercices

Implémentez les classes suivantes :
— `Multiplication` (multiplication de deux formules),
— `Division` (division d'une formule par une autre),
— `Opposite` (opposé d'une formule),
— `Cosinus` (cosinus d'une formule),
— `Sinus` (sinus d'une formule),
— `Exponential` (fonction exponentielle d'une formule) et
— `SquareRoot` (fonction racine carrée d'une formule) de manière à ce que tout puisse s'évaluer et s'afficher correctement.

2.4 Ajout de variable et dérivée

On souhaite maintenant introduire la notion de variable qui a un nom et une valeur. Pour cela, vous allez définir la classe suivante :

```
class Variable:
    def __init__(self, name, value):
        self.name = name
        self.value = value

    def eval(self):
        return self.value

    def __repr__(self):
        return self.name
```

On veut pouvoir dériver par rapport à une variable, on va donc rajouter une méthode `derivative` à toutes les classes correspondant à des formules qui renverra la formule dérivée. Pour les classes `Variable` et `Sum`, cela nous donne le code suivant :

```
class Variable:
    def derivative(self, var_name):
        if self.name == var_name:
            return Constant(1)
        return Constant(0)

class Sum :
    def derivative(self, var_name):
        return Sum(self.left_member.derivate(var_name),
            self.right_member.derivate(var_name))
```

2.4.1 Exercices

— Ajoutez la classe `Variable` ci-dessus à votre code.
— Ajoutez les méthodes `derivate` ci-dessus pour les classes `Variable` et `Sum`.
— Implémentez `derivate` pour l'ensemble des autres classes.
— Une formule est constante si elle ne contient pas de variable. Implémentez les méthodes `simplify` dans vos formules qui remplace toute sous-formule constante par une constante de sa valeur.