Spécialité NSI en terminale
TP 4 : programmation orientée objet

La programmation orientée objet permet de représenter les objets du réel ou des objets abstraits avec un code en général plus stable et plus facile à maintenir que dans d'autres styles. Dans un lycée on peut par exemple représenter les personnes (élèves, professeurs, personnel administratif ou technique) avec leur nom et prénom ainsi que des informations diverses. On complète ensuite avec des fonctions permettant d'obtenir de nouvelles informations (moyenne par élève, nombre de classe par enseignant, etc).

De manière générale, on peut peut créer une classe `Personne` avec le nom, le prénom et la date de naissance, ainsi :

```
classe Personne {
    nom : str
    prenom : str
    date : str
    }
```

Le nom, le prénom et la date de naissance sont de type `str`. Mais il y plusieurs manières d'écrire une date, donc on pourrait créer une classe `Date` avec différentes écritures possibles : format anglais, format américain, format dit hongrois. Et dans ce cas, on pourrait remplacer le type `str` de la variable `date`, par le type `Date`.

1 Classe, attributs, méthodes, objets

Nous allons définir une *classe* d'*objets*, qui permet de regrouper dans une même entité des données appelées *attributs*, et des fonctions appelées *méthodes*. On parle d'encapsulation. Un objet est un élément de la classe, ou une *instance* de la classe. Une classe d'objets définit un type composé. Les différents attributs ont un type simple (ou primitif, comme entier, flottant, chaîne de caractères) ou un type composé. Les méthodes sont des opérations sur les objets.

Nous commençons par un exemple simple en définissant une classe `Point()`. Nous utilisons le mot clé `class`. Nous créons ensuite un objet de cette classe.

```
class Point:
    "Définition d'un point dans le plan"

p = Point()
```

Entre guillemets, nous avons un commentaire sur la classe.
Après l'exécution dans l'interpréteur :

- l'écriture `type(p)` affiche `<class '__main__.Point'>`;
- l'écriture `p.__doc__` affiche `"Définition d'un point dans le plan"`;
- les écritures `help(Point)` ou `help(p)` affichent toutes les informations disponibles sur la classe.

Dans le plan rapporté à un repère orthonormé un point `p` est défini par ses deux coordonnées x et y qui sont des nombres réels.
On définit la distance entre ce point et l'origine du repère par $\sqrt{x^2+y^2}$.
Nous complétons le code définissant la classe `Point` avec trois méthodes :

```
class Point:
    "Définition d'un point dans le plan"
    def __init__(self, a = 0, b = 0):
        self.x = a
        self.y = b

    def distance(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5

    def __repr__(self):
        return "(" + str(self.x) + "; " + str(self.y) + ")"
```

Dans cet exemple, un point a deux attributs (`x` et `y`) et trois méthodes.

Le mot `self` représente une instance de la classe. Autrement dit, une méthode d'un objet a toujours en premier paramètre l'objet lui-même.

Il y a plusieurs manières de procéder pour définir les coordonnées x et y. La manière la plus « propre » est d'écrire une méthode `__init__`.

La méthode `__init__` (avec deux tirets de soulignement de chaque côté de init) est appelée un *constructeur*. Un point a par défaut les coordonnées $(0\,;\,0)$. Cette méthode est exécutée automatiquement lorsqu'on crée un nouvel objet de la classe.

La méthode `__repr__` renvoie une chaîne de caractères. Elle permet un affichage correct dans l'interpréteur.

La notation pointée permet d'accéder aux membres d'un objet, les attributs et les méthodes.

```
>>> p1 = Point()
>>> p2 = Point(4, 3)
>>> p2.x
>>> 4
>>> p1.distance() # la méthode distance a p1 comme paramètre
0.0
>>> Point.distance(p2)
5.0
>>> p2
(4; 3)
>>> print(p2)
(4; 3)
```

Si nous écrivons `p2 = Point(4, 3)`, puis `p3 = Point(4, 3)`, alors l'expression `p2 == p3` a la valeur `False`. Dans ce cas, `p2` et `p3` sont deux instances de la classe `Point`, deux objets à priori distincts.

```
>>> p2 = Point(4, 3)
>>> p3 = Point(4, 3)
>>> p2 == p3
False
```

Nous pouvons modifier ce comportement et obtenir que deux points sont égaux s'ils ont les mêmes coordonnées.

Nous ajoutons donc une méthode pour définir l'égalité de deux points et ajoutons aussi une méthode pour obtenir la distance entre deux points.

```
class Point:
    ...

    def __eq__(self, p):
        return self.x == p.x and self.y == p.y

    def distAB(self, p):
        return ((p.x - self.x) ** 2 + (p.y - self.y) ** 2) ** 0.5
```

Testons les deux nouvelles méthodes :

```
>>> p2 = Point(4, 3)
>>> p3 = Point(4, 3)
>>> p2 == p3
True
>>> p1 = Point()
>>> p1.distAB(p2)
5.0
>>> Point.distAB(p1, p2)
5.0
```

Exercice

Créer une classe `Eleves` avec pour attributs le nom, le prénom, la date de naissance. Ces trois attributs sont de type str.

Créer quelques élèves.

Ajouter un attribut `notes` de type list et une méthode `ajoute` pour ajouter une note à la liste des notes d'un élève.

Compléter avec des notes pour tous les élèves créés.

Ajouter une méthode `moyenne` qui calcule la moyenne d'un élève.

Écrire une fonction `moyenne` qui calcule la moyenne de tous les élèves.