Un peu de POO

Laurent Signac

La POO est un paradigme de programmation qui place les données au centre de la réflexion (plutôt que les opérations à effectuer).

- Un programme est formé d'**objets** qui interagissent.
- Ces objets sont des instances d'une **classe**

Exemple :

On veut manipuler des segments du plan, calculer leur intersection, leur milieu...

Vision procédurale

```
def intersections(s1:list, s2:list)->list:
     ...
     return p

def milieu(s:list)->list:
     p = []
     p.append((s[0] + s[2])/2)
     p.append((s[1] + s[3])/2)
     return p

s1 = [0, 0, 3, 5]
p = milieu(s1)
print(p)
```

Vision objet

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

class Segment:
    def __init__(self, p1, p2):
        ...

    def milieu(self)->Point:
        x = (self.p1.c[0] + self.p2.c[0]) / 2
        y = (self.p1.c[1] + self.p2.c[1]) / 2
        return Point(x, y)

    def intersection(self, s)->Point:
        ...
        return p

p1 = Point(0, 0)
p2 = Point(3, 5)
s = Segment(p1, p2)
p3 = s.milieu()
print(p3)
```

De quoi est fait un objet ?

Un objet est composé :

- d'attributs (`x` et `y` pour `Point`)
- de méthodes (`milieu` pour `Segment`)

Les **attributs** représentent les caractéristiques des l'objet (ses coordonnées pour un point, ses extrémités pour un segment...) et son état interne.

Les **méthodes** sont les *services* que peut rendre l'objet, les *actions* qu'il peut faire.

Principe d'encapsulation

L'encapsulation est le principe selon lequel l'état interne et les caractéristiques d'un objet (donc ses attributs) ne **doivent pas être manipulés directement**, mais par le biais d'une interface (les méthodes).

- Le programmeur se protège contre lui même et protège son équipe
- Il est possible de modifier le fonctionnement interne de l'objet (et donc ses attributs) sans modifier le code qui utilise l'objet.

Exemple

Je décide de modifier la classe Point 🤔 et stocke les coordonnées **polaires** dans le *tuple* `c` :

```
class Point:
    def __init__(self, x, y):
        r = (x**2 + y**2) ** 0.5
        self.c = (r, math.atan2(y, x))
```

Conséquence : la méthode `milieu` ne marche plus 😖

```
    def milieu(self)->Point:
        x = (self.p1.c[0] + self.p2.c[0]) / 2
        y = (self.p1.c[1] + self.p2.c[1]) / 2
        return Point(x, y)
```

Solution

```
class Point:
    def __init__(self, x, y):
        r = (x**2 + y**2) ** 0.5
        self.c_ = (r, math.atan2(y, x))
    def x(self):
        return self.c_[0] * math.cos(self.c_[1])
    def y(self):
        return self.c_[0] * math.sin(self.c_[1])

class Segment:
        def milieu(self)->Point:
        x = (self.p1.x() + self.p2.x()) / 2
        y = (self.p1.y() + self.p2.y()) / 2
        return Point(x, y)
```

Dans la classe `Segment`, on n'accède plus aux attributs d'un point, mais on utilise son interface `x()` et `y()` 😇

Pourquoi utiliser la programmation orientée objets ?

- Selon les modules ou bibliothèques utilisées, vous pouvez y être obligé (PyQt ou Kivy en Python par ex.)
- En *pensant* objet, vous pouvez trouver un design intéressant pour ce que vous développez
- Tous les aspects de la POO ne sont pas présents en Python et Javascript. Vous pouvez apprendre au fur et à mesure (demandez !)

Méthodes magiques

Ce sont les méthodes encadrées par `__`. Elles sont appelées dans des circonstances particulières.

Exemple : je veux manipuler des polygones.

J'ai envie de pouvoir écrire :

```
po = Polygone([p1, p2, p3])
print("Nb sommets: ", len(po))
for p in po:
    print(p)
```

👉 `len`, itérateur, conversion en chaîne.

```
class Point:
    def __str__(self):
        return "({},{})".format(self.x(), self.y())

class Polygone:
    def __init__(self, listepoints):
        self.lp_ = listepoints[:] # !! Copie superf.
    def __len__(self):
        return len(self.lp_)
    def __iter__(self):
        for p in self.lp_:
            yield p

po = Polygone([p1, p2, p3])
print("Nb sommets: ", len(po))
for p in po:
    print(p)
```

```
Nb sommets:  3
(0.0,0.0)
(2.0,5.0)
(6.0,2.5)
```

Étendre un objet

Lors de la conception d'une interface on a souvent à prendre un Widget existant, puis à lui ajouter des fonctionnalités spécifiques qui correspondent à ce qu'on veut faire.

Exemple :

Création d'un Canvas (zone de dessin tkinter), qui affiche automatiquement un compte à rebours. Chaque fois qu'on appelle la méthode `decr`, le compte à rebour diminue :

```
class Decompte(tk.Canvas):

    def __init__(self, parent, width, height, value):
        super().__init__(parent, width=width,
                         height=height, bg="yellow")
        self.value = value
        self.initvalue = value
        self.txt = self.create_text(width//2,
                                    height//2, text="")
        self.update_()

    def decr(self):
        self.value -= 1
        if self.value < 0:
            self.value = self.initvalue
        self.update_()

    def update_(self):
        self.itemconfigure(self.txt,
                           text=str(self.value))
```

Un objet de la classe `Decompte` **est** un Canvas, mais il intègre la logique du compte à rebours. L'utiliser est simplissime :

```
class Fenetre(tk.Frame):
    def __init__(self, root):
        super().__init__(root)
        root.title("Compte à rebours")
        self.pack(fill=tk.BOTH, expand=1)

        self.compte = Decompte(self, width=180,
                               height=180, value=10)
        self.compte.pack()
        tk.Button(self, text="Decompte",
                  command=self.action).pack()

    def action(self):
        self.compte.decr()
```

Utilisation de la POO en projet

J'ai largement utilisé les méthodes magiques, et
les itérateurs en général, pour écrire la classe Graphe

Dans le client Python relatif au Pendu, la zone de dessin
est un Canvas personalisé.