P.O.O. : Programmation Orientée Objet

Table des matières

1) Introduction

La programmation orientée objet, qui fait ses débuts dans les années 1960 avec les réalisations dans le langage *Lisp*, a été formellement définie avec les langages *Simula* (vers 1970) puis *SmallTalk*. Puis elle s'est développée dans les langages anciens comme le *Fortran*, le *Cobol* et est même incontournable dans des langages récents comme Java.

La programmation orientée objet (POO) permet de mieux structurer un programme. On l'utilise beaucoup lorsqu'un programme utilise des entités aux propriétés très semblables.

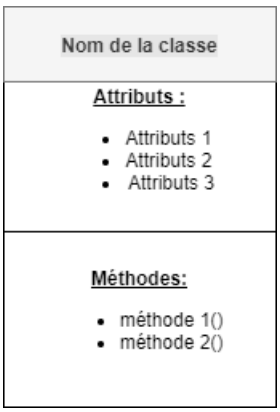
2) Concept de classe et d'encapsulation

Dans la programmation orientée objet, les différents objets utilisés peuvent être construits indépendamment les uns des autres (par exemple des programmeurs différents) sans qu'il n'y ait de risque d'interférence.

Ce résultat est obtenu grâce au principe d'**encapsulation** : la fonctionnalité interne et les variables qu'il utilise pour effectuer son travail sont, en quelque sorte, enfermées dans l'objet. Les autres objets et le monde extérieurs ne peuvent y accéder qu'à travers des procédures bien définies, c'est ce qu'on appelle l'interface de l'objet. Un système complexe a un grand nombre d'objet. Pour réduire cette complexité, on regroupe ces objets en classes.

Une classe est une description d'un ensemble d'objets ayant un structure de données commune (**attributs**) et pouvant réaliser des actions (**méthodes**). On considère en faite une classe comme un nouveau type de données. On appelle instance de la classe, l'objet (du type de la classe) qui la représente.

On peut représenter graphiquement une classe de la manière représentée ci-dessous :



3) Une première classe

Une *classe* définit et nomme une structure de données qui vient s'ajouter aux structures de base du langage. La structure définit par une classe peut regrouper plusieurs composantes de natures variées. Chacune de ces composantes est appelée un *attribut* (on dit un *champ* ou une *propriété*) et est doté d'un nom.

Supposons que l'on souhaite manipuler des entiers représentant des temps mesurés en heures, minutes et secondes. On appellera la structure correspondante `Chrono`. Les trois nombres pourront être appelés dans l'ordre `heures`,`minutes` et `secondes`. On peut représenter le temps "21 heures 34 minutes et 55 secondes" de la manière suivante :

	Chrono
heures	21
minutes	34
secondes	55

Python permet la définition de cette structure `Chrono` sous la forme d'une classe avec le code définit dans le programme 1.

```
1  class Chrono:
2      "une classe pour représenter le temps mesuré en heures, minutes et secondes"
3      def __init__(self,h,m,s):
4      "le constructeur prend comme arguments temps, minutes et secondes"
5          self.heures=h
6          self.minutes=m
7          self.secondes=s
8
9      def afficher(self):
10         return (str(self.heures)+"h"
11         +str(self.minutes)+"m"
12         +str(self.secondes)+"s")
```

Programme 1: class `Chrono`

Ce que contient la définition de la class `Chrono` :

- la définition d'une classe est introduite par le mot-clé `class` suivi du nom choisi pour la classe et d'un symbole : (deux points) ;
- le nom de la classe commence en général par une majuscule ;

- la chaîne de documentation ;
- tout le reste de la définition est placé en retrait.
- la définition d'une méthode particulière : le constructeur. En python, son nom est imposé : `__init__`. Un constructeur est une méthode invoquée lors de la création d'un objet. Cette méthode effectue les opérations nécessaires à l'initialisation d'un objet. Cette méthode n'a aucune valeur en retour, c'est l'objet qui est renvoyé.
- une méthode `affiche()` qui affiche l'heure.

Nous pouvons maintenant instancier notre classe. Pour créer un objet (une instance), on utilise la syntaxe :

```
1 t=Chrono(21,34,55)
```

Programme 2: instance de la class `Chrono`

L'appel à la méthode `afficher()` s'appliquant au chronomètre `t` est réalisé comme dans le programme 3

```
1 >>>t.affiche()
2 '21h 34m 55s'
```

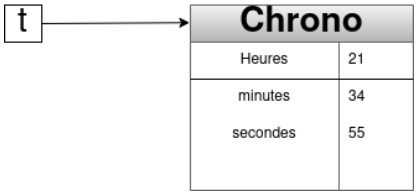
Programme 3: instance de la class `Appel des méthodes`

Remarque 1 : On peut accéder à la documentation de la classe et du constructeur grâce aux instructions suivantes.

```
1 t=Chrono(21,34,55)
2 print(t)
3 print(t.__doc__)
4 print(t.__init__.__doc__)
```

Programme 4: Accès à la documentation

Remarque 2 : La variable `t` ne contient à strictement parler l'objet qui vient d'être construit mais un pointeur vers le bloc mémoire qui a été alloué à cet objet. La situation correspond donc au schéma suivant.



Exercice 1 :

Écrire une classe `Dinosaure` dont les attributs sont :

- sa longueur ;

- sa hauteur ;
- son poids ;
- sa vitesse maximale

Vous construirez l'instance **triceratops** avec comme caractéristiques : 9 m de longueur, 3 m de hauteur, une masse de 9 tonnes environ, et une vitesse maximale donnée à 32 km/h. Puis l'instance Tyrannosaurus Rex. Celui-ci mesurait 13 m de longueur, pesait 8 t environ, 6 m de haut et devait courir à 27 km/h.

4) Encapsulation et attributs

4.1) Le problème des accès aux données

On peut accéder aux attributs d'un objet `t` de la classe `Chrono` avec la notation `t.a` où a désigne le nom de l'attribut visé. Par exemple, le code d'accès à l'attribut `secondes` s'écrit comme dans le programme 5.

```
1 >>>t.secondes
2 55
```

Programme 5: Accès aux attributs

4.2) Les accesseurs ou "getters"

On ne va généralement pas utiliser la manière de procéder précédente `nom_objet.nom_attribut` permettant d'accéder aux valeurs des attributs car on ne veut pas forcement que l'utilisateur ait accès à la représentation interne des classes. Pour utiliser ou modifier les attributs, on utilisera de préférence des méthodes dédiées dont le rôle est de faire l'interface entre l'utilisateur de l'objet et la représentation interne de l'objet (ses attributs). Les attributs sont alors en quelque sorte encapsulés dans l'objet, c'est à dire non accessibles directement par le programmeur qui a instancié un objet de cette classe.

Pour obtenir la valeur d'un attribut nous utiliserons la méthode des **accesseurs** (ou "getters") dont le nom est généralement : `getNom_attribut()`. Par exemple, pour afficher la valeur des heures, on peut employer la méthode `getHeure()` comme dans le programme 6

4.3) Modifications contrôlées des valeurs des attributs : les mutateurs ou "setters"

En cas de besoin, on peut modifier les valeurs attribuées aux attributs. Pour cela, on passe par des méthodes particulières appelées mutateurs (ou "setters") qui vont modifier la valeur d'une propriété d'un objet. Le nom d'un mutateur est généralement : `setNom_attribut()`.

```
1 def setHeure(self,h2):
2     self.heures=h2
```

Exercice 2 :

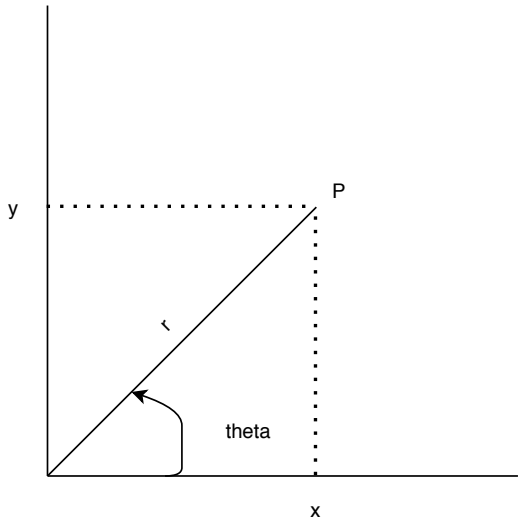
On souhaite caractériser informatiquement la notion de point telle qu'elle existe en 2 dimensions : aussi bien en coordonnées cartésiennes (x, y) qu'en coordonnées polaires (r, θ).

```python
class Chrono:
    "une classe pour représenter le temps mesuré en heures, minutes et secondes"
    def __init__(self,h,m,s):
    "le constructeur prend comme arguments temps, minutes et secondes"
        self.heures=h
        self.minutes=m
        self.secondes=s

    def affiche(self):
    "affiche l'heure "
        return (str(self.heures)+"h"
        +str(self.minutes)+"m"
        +str(self.secondes)+"s")

    def getHeure(self):
    "retourne la valeur des heures"
        return self.heures
```

Programme 6: class `Exemple de getter`



1. Remplir le modèle de la classe Point ci-dessous en listant les attributs privés et les actions permettant seulement d'y accéder (accesseurs) :

Point
Attributs - - - -
Méthodes -Construire (abs,ord) - - -

2. Inplémenter cette classe en python. On utilisera le module math. On rappelle que dans le l'intervalle $]-\pi,\pi[$, on a :

$$\theta(rad) = \begin{cases} Arctan(y/x) & si \quad x > 0 \\ Arctan(y/x) + \pi & si \quad x < 0 \ et \ y \geq 0 \\ Arctan(y/x) - \pi & si \quad x < 0 \ et \ y < 0 \\ \pi/2 & si \quad x = 0 \ et \ y > 0 \\ -\pi/2 & si \quad x = 0 \ et \ y < 0 \\ 0 & si \quad x = 0 \ et \ y = 0 \end{cases}$$

3. A l'aide de classe, on donnera les coordonnées polaires des 4 points suivants :

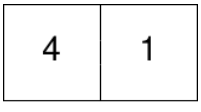
   A(-2,5) ; B(5,5) ; C(-2,-2) ; D(5-2).

Exercice 3 :

Le domino est un jeu très ancien constitué de 28 pièces toute différentes. Sur chacune de ces pièces, il y a deux côtés qui sont constitués de 0 (blanc) à 6 points noirs. Lorsque les deux côtés possèdent le même nombre de points, on l'appelle domino double.

1. Proposer une classe Domino permettant de représenter une pièce. Les objets seront initialisés avec les valeurs des deux côtés (gauche et droite). On définira des méthodes pour tester si le domino est double ou blanc. On implémentera également une méthode pour compter le nombre de points sur un domino.

   On ajoutera également une méthode qui affiche les valeurs des deux faces de manière horizontale pour un domino classique et de manière verticale pour un domino double comme le montre la figure ci-dessous.



2. Proposer un classe JeuDeDomino permettant de manipuler le jeu de domino complet. On créera une méthode pour mélanger le jeu et pour distribuer selon deux joueurs ou plus.

3. En utilisant cette classe, on affichera le jeu de deux joueurs ainsi que le jeu restant (la pioche). Pour chaque joueur, on affichera le nombre de points dans son jeu.