

Chap. 3 – Programmation Orienté Objet (pa.dilla.fr/10)

1 – Classes et attributs : structurer les données

Une **classe** définit et nomme une structure de données de base du langage qui peut regrouper plusieurs composantes de natures variées

Chacune de ces composantes est appelée un **attribut** et est doté d'un nom.

Par exemple, voici une manipulation de trois nombres entiers représentant des durées (heures, minutes, secondes).

On décide d'appeler la classe `Chrono` et de la munir de trois attributs : `heures`, `minutes` et `secondes`.

Voici alors comment on pourrait figurer le temps *21 heures, 34 minutes et 55 secondes* :

	Chrono
heures	21
minutes	34
secondes	55

1.1 – Description d'une classe

Voici comment définir cette structure sous la forme d'une *classe* :

```
[2]: class Chrono:
    """
    Une classe pour représenter un temps mesuré
    en heures, minutes et secondes. """
    def __init__(self, h, m, s):
        self.heures = h
        self.minutes = m
        self.secondes = s
```

Pour définir une nouvelle classe, on utilise :

1. le mot-clé `class`
2. suivi du nom choisi pour la classe et

3. suivi par les deux-points :.

Tout le reste de la définition est alors en retrait (indentation).

Par convention, le nom de la classe doit commencer par une **lettre majuscule**.

Suivent alors

- une documentation puis
- la définition d'une fonction `__init__` possédant
 - comme premier paramètre `self` puis ensuite
 - les trois paramètres correspondants aux trois composantes d'un objet admettant possédant la structure de `chrono`.

Les instructions de la forme `self.xxx =` correspondent aux affectations des valeurs aux trois attributs de la classe.

1.2 – Création d'un objet

Une fois la classe définie, un élément correspondant à la structure `Chrono` peut être construit avec une expression de la forme `Chrono(h, m, s)`.

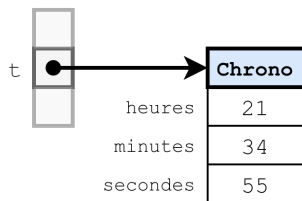
On appelle un tel élément un **objet** ou une **instance de la classe** `Chrono`.

Ainsi, pour définir et affecter à la variable `t` un objet représentant notre temps *"21 heures, 34 mintes et 55 secondes"* on écrit :

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

On remarque que, comme pour les tableaux, la variable `t` ne contient pas à strictement parler l'objet **mais** un pointeur vers le bloc de mémoire qui a été alloué à cet objet.

La situation correspond donc au schéma suivant :



1.3 – Manipulation des attributs

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.

Tout comme les cases d'un tableau, les attributs d'un objets sont mutables : on peut les consulter **et** les modifier.

```
[4]: t = Chrono(21, 34, 55)
t.secondes
```

```
4]: 55
```

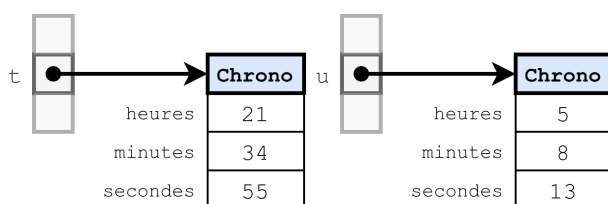
```
[5]: t.secondes = t.secondes + 1
t.secondes
```

```
5]: 56
```

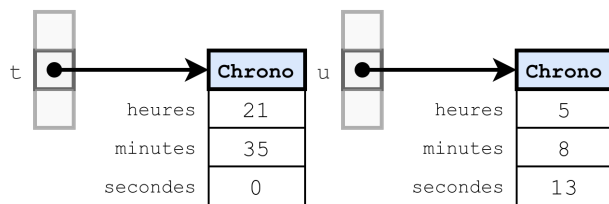
On parle bien d'**attribut d'un objet** car chaque objet possède pour ses attributs des valeurs qui lui sont propres. On parle alors aussi d'**attribut d'instance**.

Ainsi, chaque objet de la classe `Chrono` possède trois attributs dont les valeurs sont indépendantes des valeurs des attributs (de même nom) des autres instances.

Les définitions `t = Chrono(21, 34, 55)` et `u = Chrono(5, 8, 13)` conduisent donc à la situation suivante :



Une avancée de cinq secondes du chronomètre t mènerait ainsi à la situation suivante :



Une classe peut également définir des **attributs de classe**, dont la valeur est attachée à la classe elle même.

Ainsi :

```
class Chrono:
    heure_max = 24
    ...
```

On peut consulter de tels attributs depuis n'importe quelle instance avec `t.heure_max` ou depuis la classe elle même avec `Chrono.heure_max`.

On peut modifier cet attribut de classe en y accédant via la classe elle même pour que la modification soit perceptible par toutes les instances présentes et futures :

```
Chrono.heure_max = 12
```

2 – Méthodes : manipuler les données

Dans le *paradigme* de la programmation objet, la notion de classe est associée à la notion d'*encapsulation* : un programme manipulant un objet n'est pas censé accéder librement à la totalité de son contenu.

L'utilisateur n'a pas à savoir ou à accéder aux détails d'implémentation.

La manipulation de l'objet passe donc de préférence par une interface constituée de fonctions dédiées qui font partie de la définition de la classe et sont appelées les **méthodes** de cette classe.

2.1 – Utilisation d'une méthode

Les méthodes d'une classe servent à manipuler les objets de cette classe. Même si les méthodes sont des fonctions qui peuvent recevoir des paramètres, chaque appel de méthode s'applique avant tout à un objet de la classe concerné.

L'appel d'une méthode `texte` s'appliquant au chronomètre `t` et renvoyant une chaîne de caractères décrivant le temps représenté par `t` est réalisé par l'instruction `t.texte()` et elle pourra renvoyer la chaîne de caractère `'21h 34h 55s'`.

Cette notation pour l'appel de méthode est la même notation pointée que l'accès aux attributs de `t`. **Mais** la paire de parenthèse fait bien apparaître une méthode, comme pour une fonction sans paramètre.

Lorsqu'une méthode dépend d'autres paramètres que cet objet principal `t`, ces autres paramètres apparaissent de la manière habituelle.

Par exemple, s'il existe une méthode `avance` faisant avancer le chronomètre `t` d'un *certain* nombre de secondes passé en paramètre, on écrira pour avancer le chronomètre de 5 secondes :

```
t.avance(5)
```

Ainsi un nouvel appel à `t.texte()` renverra cette fois-ci `'21h 35m 0s'`.

Comme le montre l'exemple, on ne manipule pas directement les attributs d'un objet. On utilise pour cela des méthodes ce qui préserve l'encapsulation du code.

On a déjà rencontré des notations de ce type comme par exemple `tab.append(42)` pour ajouter le nombre 42 au tableau `tab`.

Les paramètres des méthodes peuvent être aussi bien des valeurs de base (nombre, chaîne de caractère, tableau, etc.) que des objets.

Par exemple si pour la classe `Chrono` on admet l'existence des méthodes sui-

vantes :

- `egale` s'appliquant à deux chronomètres pour tester l'égalité des temps représentés
- `clone` s'appliquant à un chronomètre `t` et renvoyant un nouveau chronomètre initialisé au même temps que `t`

On pourra alors écrire l'instruction suivante

```
u = t.clone()  
t.egale(u)
```

qui nous renverra `True`.

Puis après `t.avance(3)`, l'instruction `t.egale(u)` nous renverra alors `False`.

2.2 – Définition d'une méthode

Comme nous venons de le voir, une méthode d'une classe peut être vue comme une fonction ordinaire, pouvant dépendre d'un nombre de paramètres arbitraire **sauf** qu'elle doit avoir obligatoirement pour **premier paramètre** un objet de la classe. Une méthode ne peut donc pas avoir zéro paramètre.

La définition d'une méthode de classe se fait avec la même notation que la définition d'une fonction. Le premier paramètre est systématiquement appelé `self`. Comme ce paramètre est un objet, on pourra accéder à ses attributs avec la notation `self.a`

Ainsi, les fonctions `texte` et `avance` de la classe `Chrono` peuvent être implémentées de la façon suivante :

```
class Chrono:  
    ...  
    def texte(self):
```

```
    return ( str(self.heures)    + 'h '
            + str(self.minutes)  + 'min '
            + str(self.secondes) + 's'    )

def avance(self, s):
    self.secondes += s

    # dépassement secondes
    self.minutes += self.secondes // 60
    self.secondes = self.secondes % 60

    # dépassement minutes
    self.heures += self.minutes // 60
    self.minutes = self.minutes % 60
```

2.3 – Constructeur

La construction d'un nouvel objet avec une expression comme `Chrono(21, 34, 55)` déclenche deux choses :

- la création de l'objet lui même
- l'appel à une méthode spéciale chargée d'initialiser les valeurs des attributs. Cette méthode, appelée **constructeur**, est définie par le programmeur. En Python, il s'agit de la méthode `__init__` que nous avons pu observer dans le premier exemple.

La définition de la méthode spéciale `__init__` ne se distingue pas des autres méthodes ordinaires : son premier paramètre est `self` et représente l'objet auquel elle s'applique. Les autres paramètres sont donnés explicitement lors de la construction.

2.4 – Autre méthodes particulières en Python