Chap. 3 — Programmation Orienté Objet (pa.dilla.fr/17)

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.

Exemple

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* :

Exemple

```
[ ]: 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).

ACTIVITÉ 1

Redéfinir les méthodes `avance`, `texte` (ou `__str__`), `egale` (ou `__eq__`) et `clone` en tenant compte de ce changement d'implémentation.

Ajouter une méthode `_conversion` qui extrait d'un temps le triplet `(h, m, s)` correspondant. **Utiliser** cette méthode auxiliaire pour simplifier les implémentations de `texte` et `clone`.

REMARQUE

Les méthodes possédant un nom commençant par _ sont des méthodes auxiliaires et les autres méthodes forment l'**interface** des objets de cette classe.

À retenir

La programmation orientée objet **structure** les programmes en regroupant dans une même entité des données et le code manipulant ces données.
Dans ce paradigme de programmation, on manipule des **objets** pouvant contenir plusieurs données sous la forme d'**attributs** à l'aide de fonctions particulières appelées **méthodes**. Chaque objet est une **instance** d'une **classe**, la classe définissant l'ensemble des attributs et méthodes que possèdent ses instances.

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

Exemple

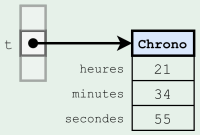
Ainsi, pour définir et affecter à la variable `t` un objet représentant notre temps *"21 heures, 34 mintes et 55 secondes"* on écrit :

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

REMARQUE

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.

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

t.secondes
```

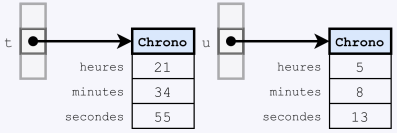
[ ]:
```
t.secondes = t.secondes + 1
t.secondes
```

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**.

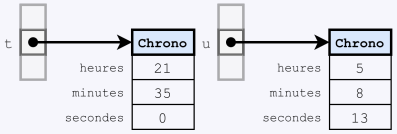
Exemple

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.

Exemple

Ainsi :

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



```
    ...
    def __len__(self):
        return self.taille
```

REMARQUE

Plusieurs remarques :

- L'appel à la fonction `cree()` est remplacée par un appel au constructeur `Ensemble()`.
- Les appels de fonctions `contient(s,x)` et `ajoute(s,x)` sont transformés en appels de méthodes `s.contient(x)` et `s.ajoute(x)`.

3 — Retour sur l'encapsulation

Dans la philosophie objet, l'interaction avec les objets se fait essentiellement **avec les méthodes**. Les attributs sont considérés par défaut comme relevant du *détail d'implémentation*.

Ainsi, pour la classe `Chrono`, il est essentiel de savoir qu'on peut afficher et faire évoluer les temps, mais l'existance des trois attributs `heures`, `minutes` et `secondes` est anecdotique (et peut être cachée à l'utilisateur).

Exemple

Par exemple, on peut simplifier la définition de la classe en ne définissant qu'un unique attribut `_temps` mesurant le temps en seconde.

```
class Chrono:
    def __init__(self, h, m, s):
        self._temps = 3600*h + 60*m + s
```

```
    elif t2.heures > t1.heures:
        return t2
    elif t1.minutes > t2.minutes:
        ...
```

Une instruction `Chrono.max(t,u)` renvoie donc l'objet (sa classe et son adresse mémoire) comme par exemple `<__main__.Chrono object at 0x10d8ac198>`

2.4 — Une classe pour les ensembles

Exemple

```
[ ]: class Ensemble:
    def __init__(self):
        self.taille = 0
        self.dates = [False] * 366

    def contient(self, x):
        return self.dates[x]

    def ajoute(self, x):
        if not self.contient(x):
            self.taille += 1
            self.dates[x] = True

def contient_doublon(t):
    s = Ensemble()
    for x in t:
        if s.contient(x):
            return True
        s.ajoute(x)
    return False
```

Le programme ci-dessus donne une adaptation du programme du *chapitre 2 — Modularité* sous la forme d'une classe.

Puisqu'une classe peut regrouper plusieurs données, nous en profitons pour mémoriser la taille de l'ensemble de dates. Cela permettrait par exemple une définition simple d'une méthode `__len__`

```
class Ensemble:
```



```
    ...
```

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 perseptible 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 calsse et sont appelée les **méthodes** de cette classe.

2.1 — Utilisation d'un 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.

Exemple

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

Exemple

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'`.

REMARQUE

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.

Exemple

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

éléments.

Par exemple les attributs `.x` ou `.y` peuvent être utilisés dans plusieurs classes différentes sans risque de confusion.

Méthodes de classe Il existe des attributs de classe et il existe aussi des **méthodes de classe**, dont la valeur ne dépend pas des instances mais est partagée au niveau de la classe entière.
Ces méthodes de classes sont appelées en programmation objet des **méthodes statiques**.

Exemple

Par exemple :

(1) méthode pertinente pour réaliser des fonctions auxilières ne travaillant pas directement sur les objets de la classe

```
def est_seconde_valide(s):
    return 0 <= s and s < 60
```

Une instruction `Chrono.est_seconde_valide(64)` renvoie donc `False`.

Exemple

(2) opérations d'appliquant à plusieurs instances aux rôles symétriques et dont aucune n'est modifiée (pas d'effets de bords) :

```
def max(t1, t2):
    if t1.heures > t2.heures:
        return t1
```

méthode	appel	effet
`__len__(self)`	`len(t)`	renvoie un nombre entier définissant la taille de `t`
`__contains__(self, x)`	`x in t`	renvoie `True` si et seulement si `x` est dans la collection `t`
`__getitem__(self, i)`	`t[i]`	renvoie le `i`-ième élément de `t`

Exemple

Par exemple, la méthode `texte` de la classe `Chrono` correspond exactement au rôle de la méthode `__str__`... mais ne bénéficie pas de la syntaxe allégée

Égalité entre deux objets Par défaut, la comparaison avec == entre deux objets ne renvoie `True` que lorsqu'elle est appliquée deux fois au même objet, ayant la même adresse mémoire.

Pour que la comparaison s'effectue en fonction de la valeur des attributs, il faut définir la méthode spéciale `__eq__(self, obj)`.

REMARQUE

On remarque que deux classes différents peuvent sans problème définir des attributs de même nom. Il n'y a aucun conflit dans ce cas là car une classe définit un *espace de noms*, c'est-à-dire une zone mémoire séparée des autres en ce qui concerne le nommage des variables et des autres

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

Exemple

Par exemple si pour la classe `Chrono` on admet l'existence des méthodes suivantes :

- `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'un 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`

Exemple

Ainsi, les fonctions `texte` et `avance` de la classe `Chrono` peuvent être implémentée 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

Il existe en Python d'autres *méthodes particulières* :

méthode	appel	effet
`__str__(self)`	`str(t)`	renvoie une chaîne de caractère décrivant `t`
`__lt__(self, u)`	`t < u`	revoie `True` si `t` est strictement plus petit que `True`
`__hash__(self)`	`hash(t)`	donne un code de hachage pour `t`, par exemple pour l'utiliser comme clé d'un dictionnaire `d`