Programmation objet

L'année dernière, vous avez appris à représenter des données à l'aide de différents types construits : p-uplets, p-uplets nommés, tableaux et dictionnaires.
Le paradigme de *programmation objet* que nous allons présenter dans ce chapitre, fournit une notion de *classe* qui permet de définir des structures de données compososites et de structurer le code d'un programme.

1 Classes et attributs : structurer les données

Une **classe** définie et nomme une structure de données qui vient s'ajouter aux structures de base du langage. La structure définie par une classe peut regrouper plusieurs composantes de natures variées appelées *attributs* et chacune dotée d'un nom.

1.1 Description d'une classe

Supposons que l'on souhaite manipuler des triplets d'entier représentant le temps en heures, minutes, secondes.
On peut définir une structure `Chrono` contenant trois attributs appelés `heures`, `minutes` et `secondes`. :

Chrono
heures
minutes
secondes

Python permet la définition de cette structure `Chrono` sous la forme d'une classe avec le code suivant :

mot clef `class` suivi du nom et :

```
1 class Chrono:
2     '''une classe pour representer un temps mesure
3     en heures, minutes et secondes'''
4     def __init__(self, h, m, s):
5         self.heures = h
6         self.minutes = m
7         self.secondes = s
```

fonction `__init__`

affectation des valeurs aux attributs

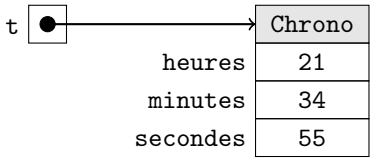
La définition d'une nouvelle classe est introduite par le mot clef **`class`**, suivi du nom de la classe et du symbole : . Le nom de la classe commence par une lettre majuscule.
La fonction `__init__`, dont nous détaillerons la construction par la suite, possède un premier paramètre appelé `self` ainsi que trois instructions de la forme `self.`a`=...` qui permettent d'affecter à chaque attribut sa valeur.

1.2 Création d'un objet

Une fois la classe `Chrono` définie, on peut créer un élément correspondant à cette structure appelé *objet* ou *instance* de la classe `Chrono`.
On peut par exemple définir et affecter à la variable `t` un objet représentant le temps « 21 heures, 34 minutes et 55 secondes » de la manière suivante :

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

Tout s'écrit comme si le nom de la classe était une fonction.
La variable `t` contient un pointeur vers le bloc mémoire qui a été alloué à cet objet. On obtient la situation suivante :



1.3 Manipulation des attributs

On peut accéder aux attributs d'un objet `t` avec la notation `t.a` où `a` désigne le nom de l'attribut visé. Les attributs en Python sont mutables.

```
>>> t.heures
21
>>> t.heures += 1
>>> t.heures
22
```

Une classe peut également définir des *attributs de classe* , dont la valeur est attachée à la classe elle-même. On peut consulter de tels attributs depuis n'importe quelle instance, ou depuis la classe elle-même.

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

>>> t = Chrono(21, 34, 55)
>>> (t.heure_max, Chrono.heure_max)
(24, 24)
```

2 Méthodes : manipuler les données

Dans le paradigme de la programmation objet, un programme manipulant un objet n'est pas censé accéder à la totalité de son contenu mais uniquement a une interface constitué de fonctions dédiées appelées les *méthodes* de cette classe.

2.1 Utilisation d'une méthode

L'appel à une méthode `texte` s'appliquant à l'objet `t` qui affiche une chaîne de caractère décrivant le temps représenté est réalisé ainsi :

```
>>> t.texte()
21h 34m 55s
```

Cette notation utilise la même notation pointé que l'accès aux attributs de `t`, mais fait apparaître une paire de parenthèses comme pour l'appel d'une fonction sans paramètre.
L'appel à une méthode `avance` faisant avancer `t` d'un certains nombre de secondes passé en paramètre s'écrit de la manière suivante :

```
>>> t.avance(5)
>>> t.texte()
21h 35m 00s
```

Comme pour les fonctions, les paramètres autre que l'objet principal `t` apparaissent entre parenthèses et séparés par des virgules.

2.2 Définition d'un méthode

Une méthode d'une classe peut être vu comme une fonction ordinaire à ceci près qu'elle doit nécessairement avoir pour premier paramètre un objet de cette classe. Par convention, ce premier paramètre est systématiquement appelé `self`. Les méthodes `texte` et `avance` peuvent être définies de la manière suivante :

```
    def texte(self):
        return (str(self.heures)+'h'
                +str(self.minutes)+'m'
                +str(self.secondes)+'s')

    def avance(self,s):
        self.secondes += s
        # depassement secondes
        self.minutes += self.secondes//60
        self.secondes = self.secondes%60
        #depassement minutes
        self.heures += self.minutes//60
        self.minutes = self.minutes%60
```

2.3 Méthodes particulières

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

1. la création de l'objet lui-même
2. l'appel à une méthode spéciale, appelée *constructeur*, chargé d'initialiser les valeurs des attributs. En Python, il s'agit de la méthode **`__init__`**.

La particularité de cette méthode est la manière dont elle est appelée, directement par l'interprète Python en réponse à une opération particulière.

Autres méthodes particulières en Python
Il existe en Python un certain nombre d'autres méthodes particulières, chacune avec un nom fixé et entouré de deux paires de _. Elles permettent d'alléger ou d'uniformiser la syntaxe. Le tableau suivant en donne quelques exemples.

méthode	appel	effet
`__str__(self)`	**`str`**`(t)` ou **`print`**`(t)`	renvoie une chaîne de caractère décrivant `t`
`__lt__(self,u)`	`t < u`	revoie `True` si `t` est strictement plus petit que `u`
`__len__(self)`	**`len`**`(t)`	renvoie un nombre entier définissant la taille de `t`
`__contains__`	`x` **`in`** `t`	revoie `True` si `x` est dans la collection `t`
`__getitem__(self,i)`	`t[i]`	renvoie le `i`-ième élément de `t`

On peut alors ajouter la définition suivante à la classe `Chrono`

```
    def __str__(self):
        return self.texte()
```

3 Encapsulation

Dans la philosophie objet, l'interaction avec les objets d'une classe se fait avec les méthodes de l'interface. L'utilisateur extérieur n'est pas censé accéder aux attributs ou aux autres méthodes.

Dans certains langage de programmation, comme le C++ ou le Java, on peut empêcher cet accès. En Python, on précède les attributs ou les méthodes qui ne font pas parties de l'interface par `_` mais rien n'empêche que l'utilisateur extérieur y accèdent.

Dans notre classe `Chrono`, il vaut donc mieux appeler nos trois attributs `_heures`, `_minutes` et `_secondes`.

```
class Chrono:
    def __init__(self,h,m,s):
        self._heures = h
        self._minutes = m
        self._secondes = s
```

On peut alors modifier structure de la classe tant que l'interface reste inchangée. En l'occurence, on pourrait modifier la classe `Chrono` et se contenter d'un attribut `_temps` mesurant le temps en secondes.

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

Les opérations comme `avance` sont alors simplifiées :

```
    def avance(self,s):
        self._temps += s
```

On pourra alors introduire une méthode qui ne fait pas partie de l'interface mais destinée à être utilisée par les méthodes principales :

```
    def _conversion(self):
        s = self._temps
        return (s//3600, (s//60)%60, s%60)

    def texte(self):
        h,m,s = self._conversion()
        return str(h)+'h '+str(m)+'m '+str(s)+'s'
```