Programmation objet

Cours

Capacités attendues	
Vocabulaire de la programmation objet : classes, attributs, méthodes, objets.	▪ Écrire la définition d'une classe. ▪ Accéder aux attributs et méthodes d'une classe.

Une **structure de données** est une façon spécifique d'organiser, de stocker et de gérer les données dans la mémoire d'un ordinateur pour qu'elles puissent être utilisées de manière efficace. Elle définit la manière dont les données sont agencées et les opérations qui peuvent être effectuées sur elles, comme la recherche, l'insertion, la suppression ou la modification.

Il existe différents types de structures de données, chacune adaptée à un usage spécifique en fonction des besoins du programme ou de l'algorithme :

Liste : Une collection d'éléments stockés de manière contiguë en mémoire. On accède aux éléments via un index.

Les dictionnaires : Une structure de données qui associe des clés à des valeurs.

Les tuples : permet de regrouper 2 ou plusieurs éléments de type distincts mais n'autorise pas la modification des éléments.

Pourquoi utiliser des structures de données ?

- Efficacité : Elles permettent de manipuler et de traiter des données rapidement et avec un minimum de ressources, notamment en termes de temps de calcul et d'espace mémoire.
- Organisation : Elles facilitent le rangement et l'accès aux données de manière logique, ce qui simplifie leur traitement.
- Adaptation à des besoins spécifiques : Différentes structures de données sont optimisées pour différents types d'opérations (recherche, insertion, suppression, tri, etc.).

Le paradigme de programmation objet qui est intégré à python et que nous allons présenter dans ce chapitre fournit une notion de classe qui permet à la fois de définir et nommer des structures de données composites et de structurer le code d'un programme.

1. Classe 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 nature variée. Chacune de ces composantes est appelée un *attribut* (on dit aussi un *champ* ou une *propriété*) et est dotée d'un nom.

Description d'une classe

Supposons que l'on souhaite manipuler des triplets d'entiers représentant des temps mesurés en heures, minutes et secondes. On appellera la structure correspondante chrono. Les 3 nombres pourront être appelés dans l'ordre : heures, minutes et secondes, et nous pourrions nous figurer le temps « 21heures, 34 minutes et 55 secondes » comme un triplet nommé correspondant à la représentation graphique 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 suivant :

```
class Chrono:
    """Une classe pour représenter un temps mesuré en
    heures, minutes, secondes
    """

    def __init__(self, h, m, s):
        self.heures = h
        self.minutes = m
        self.secondes = s
```

La définition d'une nouvelle classe est introduite par le mot-clé **class** suivie du nom choisi par la classe et d'un symbole : (deux points). Le nom de la classe commence par une lettre majuscule. Tout le reste de la définition est à leur placé en retrait. Veuillez noter la définition d'une fonction __init__ qui possède un premier paramètre appelé self, trois paramètres correspondant aux trois composantes de notre triplet, ainsi que trois instructions de la forme self.a = ...

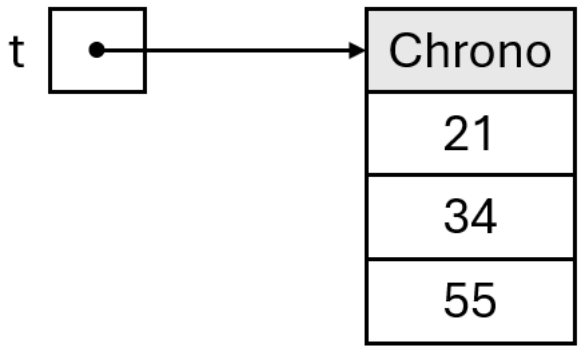
Création d'un objet

Une fois une telle 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. On peut ainsi définir et affecter à la variable t un objet représentant notre temps « 21 heures, 34 minutes et 55 secondes » avec la ligne suivante :

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

Tout s'écrit comme si le nom de la classe était également une fonction attendant trois paramètres et construisant l'élément correspondant.

Notez que comme c'était le cas pour les tableaux, la variable t ne contient pas à strictement parler l'objet qui vient d'être construit, mais un pointeur vers le bloc de mémoire qui a été alloué à cet objet. La situation correspond donc au schéma suivant :



A faire vous-même : Recopiez la classe Chrono et son instanciation dans : https://pythontutor.com. Constatez l'allocation mémoire correspondante.

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. Les attributs, comme les cases d'un tableau sont mutables en Python : on peut non seulement consulter leur valeur mais aussi la modifier.

```
>>> t.secondes
55
>>> t.secondes = t.secondes + 1
>>> t.secondes
56
```

Notez que l'on a parlé *d'attributs d'un objet*. En effet bien que les noms des attributs soient attachés à une classe, chaque objet possède pour ces attributs des valeurs qui lui sont propres. C'est pourquoi on parle parfois aussi *d'attributs d'instance*. Ainsi chaque objet de la classe Chrono possède bien trois attributs : heures, minutes et secondes dont les valeurs sont **indépendantes** des valeurs des attributs de mêmes noms des autres instances.

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

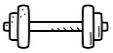
Les valeurs des attributs d'un objet pouvant varier, on les comprend parfois comme décrivant *l'état* de cet objet.
Un changement des valeurs des attributs d'un objet correspond alors à l'évolution de cet objet.

Exemple : Une avancée de 5 secondes du chronomètre t mènerait ainsi la situation suivante :

Erreurs

Il n'est évidemment pas possible d'obtenir la valeur d'un attribut inexistant :

```
>>> t.x
 Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
 AttributeError: 'Livre' object has no attribute 'x'
```

 **A faire vous-même : Créez deux instances de Chrono comme dans l'exemple précédent. Modifiez des attributs dans les deux objets. Constatez l'allocation mémoire correspondante via le site : https://pythontutor.com.**

Attributs de classes

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

```
class Chrono:

    heure_max = 24
```

On peut **consulter** de tels attributs depuis n'importe quelle instance, ou depuis la classe elle-même.

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

On peut également **modifier** cet attribut en y accédant via la classe elle-même pour que la modification soit perceptible par toutes les instances présentes ou futures.

```
>>> Chrono.heure_max = 12
>>> t.heure_max
12
```

En revanche, un tel attribut n'est pas destiné à être modifié depuis une instance (techniquement cela ne ferait que créer une variable d'instance de même nom, pour cette seule instance, qui serait donc décorrélée de l'attribut de classe).

2. Méthodes : manipuler les données

Dans le paradigme de la programmation objet, la notion de classe est souvent associée à la notion *d'encapsulation* : un programme manipulant un objet n'est pas censé accéder librement à la totalité de son contenu, une partie de ce contenu pouvant relever du « détail 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.

Utilisation d'une méthode

Les méthodes d'une classe servent à manipuler les objets de cette classe. Chaque appel de méthode peut recevoir des paramètres mais s'applique donc avant tout à un objet de la classe concernée.
L'appel à une méthode ***texte*** s'appliquant au chronomètre **t** et renvoyant une chaîne de caractères décrivant le temps représenté par **t** est réalisé ainsi :
>>> t.texte()
'21h 34m 55s'

Cette notation pour l'appel de méthode utilise la même notation pointée que l'accès aux attributs de t, mais fait apparaître en plus une paire de parenthèses, comme pour l'appel d'une fonction sans paramètres.
Lorsqu'une méthode dépend d'autres paramètres que cet objet principal **t**, ces autres paramètres apparaissent de la manière habituelle, entre les parenthèses et séparés par des virgules.

Lors d'un appel i.m(e_1, ..., e_n) à une méthode *m*, l'objet *i* est appelé *le paramètre implicite* et les paramètres e_1 à e_n *les paramètres explicites*.

Toutes les méthodes d'une classe attentent comme paramètre implicite un objet de cette classe. Les paramètres explicites, en revanche, de même que l'éventuel résultat de la méthode, peuvent être des valeurs Python arbitraires : on y trouvera aussi bien que des valeurs de base (nombre, chaînes de caractères, etc.) que des objets.

Définition d'une méthode

Une méthode d'une classe peut être vue comme une fonction ordinaire, pouvant dépendre d'un nombre arbitraire de paramètres, à ceci près qu'elle doit nécessairement avoir pour premier paramètre un objet de cette classe (le paramètre implicite). Une méthode ne peut donc pas avoir 0 paramètre.
La définition d'une méthode d'une classe se fait exactement avec la même notation que la définition d'une fonction. En Python, le paramètre implicite apparaît comme un paramètre ordinaire et prend la première position. Les paramètres explicites 1 à *n* prenant alors les positions 2 à n+1. Par convention, ce premier paramètre est systématiquement appelé ***self***. Ce paramètre étant un objet, notez que l'on va pouvoir accéder à ces attributs avec la notation **self.a** .

Constructeur

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, gérer directement par l'interprète ou le compilateur du langage,
2. L'appel à une méthode spéciale chargé d'initialiser les valeurs des attributs. Cette méthode appelée constructeur est définie par les programmeurs. En Python, il s'agit de la méthode __init__ que nous avons pu observer dans les exemples.

La définition de la méthode spéciale __init__ ne se distingue en rien de la définition d'une méthode ordinaire : son premier attribut est self et représente l'objet auquel elle s'applique, et ses autres paramètres sont les paramètres donnés explicitement lors de la construction.

Autres méthodes particulières en Python

Il existe en Python un certain nombre d'autres méthodes particulières, chacune avec un nom fixé et entouré comme pour __init__ de deux paires de symboles _ .

Il existe plusieurs méthodes d'usage général comme les exemples donnés dans le tableau suivant :

Méthode	Appel	Effet
__str__(self)	**str**(t)	Renvoie une chaîne de caractères décrivant t
__lt__(self, u)	t < u	Renvoie True si t est strictement plus petit que u.

D'autres sont spécifiques à des classes représentant certaines catégories d'objets. Ci-dessous trois méthodes pour une classe représentant une collection :

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

Il existe beaucoup d'autres méthodes spéciales :

- Pour le inférieur ou égal (<=) : object.__le__(self, other) permet d'écrire ensuite object <= other
- Pour créer une éventuelle égalité (==) : object.__eq__(self, other) permet d'écrire ensuite object == other
- Pour créer un test de différence (!=) : object.__ne__(self, other) permet d'écrire ensuite object != other
- Pour gérer le supérieur (>) : object.__gt__(self, other) permet d'écrire ensuite object > other
- Pour gérer le supérieur ou égal (>=) : object.__ge__(self, other) permet d'écrire ensuite object >=

other
- Pour gérer l'opposé (-) : object.__neg__(self) permet d'écrire ensuite -object
- Pour gérer l'addition (+) : object.__add__(self, other) permet d'écrire ensuite object + other
- Pour gérer la soustraction (-) : object.__sub__(self, other) permet d'écrire ensuite object - other
- Pour gérer la multiplication (*) : object.__mul__(self, other) permet d'écrire ensuite object * other
- Pour gérer la division (/) : object.__div__(self, other) permet d'écrire ensuite object / other

Égalité entre objets

Par défaut, la comparaison entre deux objets avec == ne considère pas comme égaux deux objets avec les mêmes valeurs pour chaque attribut : elle ne renvoie True que lorsqu'elle est appliquée deux fois au même objet, identifiée par son adresse en mémoire.
Pour que cette comparaison caractérise les objets qui, sans être physiquement les mêmes, représentent la même valeur, il faut définir la méthode spéciale __eq__(self, other). On peut à cette occasion soit simplement comparer les valeurs de chaque attribut, soit appliquer un critère plus fin adapté à la classe représentée.

Présentation des concepts par l'exemple :
https://capytale2.ac-paris.fr/web/c/885b-4244176

Méthode de classe

On a vu que pouvaient exister des attributs de classe, dont la valeur ne dépend pas des instances mais est partagée au niveau de la classe entière. De même, la programmation objet connaît une notion de *méthode de classe*, aussi appelé *méthode statique*, qui ne s'applique pas à un objet en particulier. Ces méthodes sont parfois pertinentes pour réaliser des fonctions auxiliaires ne travaillant pas directement sur les objets de la classe :

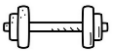
```
def est_seconde_valide(s) :
    return 0 <= s and s < 60
```

Ou des opérations s'appliquant à plusieurs instances aux rôles symétriques et dont aucune n'est modifiée :

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

Pour rappeler de telles méthodes on peut utiliser la notation d'accès direct avec le nom de la classe :

```
>>> Chrono.est_seconde_valide(64)
False
```

 **Préparer une carte mentale de la leçon.**