3 – Programmation orientée objet

Notions introduites :

- Définition de classes
- Création et manipulation d'objets
- Attributs et méthodes

1 – Classes et attributs : structurer les données

Une classe va définir et nommer une structure de données, qui vient en plus des structures de base du langage. Une classe va pouvoir regrouper plusieurs composants, de différentes nature. C'est composants sont appelées attributs (on peut aussi voir les termes de champ ou encore propriété). Chaque attribut sera repéré dans le code, par son nom.

La description d'une classe

Imaginons que nous voulions manipuler un n-uplets d'entiers représentant le temps, il nous faudra donc heures, minutes et secondes. Nous avons donc un triplets pour représenter le temps, que l'on va inclure dans une classe que nous allons nommer Chrono.

Chrono	
18	heures
20	minutes
22	secondes

Le triplet va associer heures, minutes et secondes à un entier.

En Python, cette classe auraint le code suivant :

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

Une nouvelle classe va être définie par le mot-clé ***class***, puis le nom de celle-ci (ici Chrono), et ":" pour finir la ligne. Le nom d'une classe commence toujours par une lettre majuscule, et tout ce qui se trouve dans cette classe sera indenté (un rang vers la droite). Dans cet exemple, nous avons la définition d'une fonction d'initialisation : ***def __init__*** (on vera plus tard sa construction). Vous aurez remarqué qu'elle possède un premier paramètre qui s'appel self, puis 3 autres paramètres qui correspondent à notre triplet et pour finir, 3 instructions qui commencent pat self. qui vont aussi correspondre à notre triplet, et qui vont affecter des valeurs à celui-ci.

La création d'un objet

On va pouvoir utiliser notre classe en l'affectant à une variable. Cette variable deviendra un objet, qui sera une instance de la classe Chrono. On va définir cette instance de la façon suivante :

t = Chrono(18, 20, 22)

Cela ressemble à l'utilisation d'une fonction qui prendrait 3 paramètres.
Tout comme les tableaux, notre variable t ne contient pas notre objet, mais plutôt un pointeur vers le bloc mémoire qui est alloué à notre objet. On a donc le schéma suivant :

Chrono	<---------- ---------- t
18	heures
20	minutes
22	secondes

La manipulation des attributs

On va pouvoir accéder et/ou modifier les attributs de notre classe, en utilisant la notation t. suivis du nom de l'attribut souhaité. Par exemple :

```
print(t.secondes)
>>> 22
t.secondes += 1
print(t.secondes)
>>> 23
```

Chaque attributs d'objet est rattaché à une classe, mais chaque attributs d'un nouvel objet lui sera propre, pn parle d'attribut d'instance. Dans notre exemple, la classe Chrono contient 3 attributs, donc chaque nouvel objet créé à partir de cette classe contiendra aussi 3 attributs, mais lerus valeurs seront idépendantes. On peux donc créer notre objet t = Chrono(18, 20, 22) et un autre objet u = Chrono(5, 10, 15). Exemple :

Chrono	<---------- ---------- t
18	heures
20	minutes
22	secondes

Chrono	<---------- ---------- u
5	heures
10	minutes
15	secondes

Nous avons vu plus haut qu'il était possible de modifier la valeur d'un attribut, voyer le comme une évolution de cet objet. Imaginons que nous voulions avancer le temps en seconde de notre objet t, de 38 secondes, nous aurions donc la situation suivante :

Chrono	<---------- ---------- t
18	heures
21	minutes
0	secondes

Chrono	<---------- ---------- u
5	heures
10	minutes
15	secondes

Bien évidemment, on ne peut obtenir la valeur d'un attribut qui n'existe pas, exemple :

```
print(t.x)
>>>
Traceback (most recent call last):
        File "...", line 16, in <module>
                print(t.x)
AttributeError: 'Chrono' object has no attribute 'x'
```

Par contre, en Python, rien n'empêche d'affecter une valeur à un attribut qui n'existe pas dans notre classe. Exemple

```
t.x = 89
print(t.heures, t.minutes, t.secondes, t.x)
>>>21 34 55 89
```

Les spécificités des attributs en Python

En Python, la structure des objets n'est pas exactement comme dans d'autres langages. Dans le paradigme objet habituel, une classe comprends un ensemble d'attributs qui vont définir tous les attributs que contiendrons les instances cette classe (c'est ce qui sera utilisé pour ce cours).

Par contre, en Python, les attributs ne sont pas réellement créée au niveau de la classe mais plutôt à la création de notre objet, c'est-à-dire, à chaque nouvelle instance de notre classe, on parlera alors de variables d'instance.

Il est donc, techniquement, possible que 2 objets venant de la même classe, puissent posséder des attributs qui n'ont pas de rapport les uns avec les autres. Il faut faire très attention car c'est la source d'un grand nombre d'erreurs. Pour rester dans le cadre de la programmation objet "habituelle", on conviendra de la chose suivante : nous ne pourrons utiliser que des attributs propre à la classe de l'objet créée, et qu'il ne sera pas possible d'en ajouter de nouveaux.

Les attributs de classe

Il est possible de créer des attributs de classe qui sont liés directement à la classe elle-même, et non à une méthode de cette classe. Exemple :

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

Nous avons aouté à notre classe, l'attribut heure_max, qui est définit à la valeur 24. Il est possible d'y accéder depuis n'importe où dans notre classe, mais aussi depuis une instance de celle-ci. Exemple :

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

Il est également possible de modifier cet attribut en passant directement par la classe, cette modification sera alors visible par toutes les instances, quelles soient présente ou futur. Exemple :

```
Chrono.heure_max = 12
print(t.heure_max)
>>>12
```

Cependant, un attribut comme celui-ci n'est pas vraiment destiné à être modifié depuis une instance (techniquement, il ne serait créée que dans une variable d'instance du même nom, et juste pour cette instance, qui n'aurait donc, plus aucun rapport avec l'attribut de notre classe).

2 – Méthodes : manipuler les données

En programmation objet, la notion de classe est souvent associé à la notion d'encaspulation, en manipulant un objet, nous ne somme pas censé accéder librement à la totalité du contenu de sa classe, une partie de ce contenu pouvant relever du "détail d'implémentation". Il est donc préférable de passer par une interface. Cette interface sera constitué de méthodes (fonctions dédiées à la manipulation de notre objet).

Utilisation d'une méthode

Ces méthodes servent à manipuler les objets de cette classe, et chaque appel à ces méthodes peuvent recevoir des paramètres. On pourrait imaginer une méthode ***texte***, qui s'appliquera au chrono t et renverras une chaine de caractères pour visualiser le temps. Exemple :

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

On peut remarquer que cet appel ressemble à celui d'un attribut, a un detail près, une paire de parenthèses, vide.

Imaginons maintenant que nous voulions créer une méthode qui nous permette d'avancer

le temps de notre chrono, cette fonction s'appelerait ***avance***, et demanderait 1 paramètre (le nombre de secondes à ajouter à notre chrono). Et bien l'appel se ferait de la même façon que pour une fonction. Nous aurions donc :

```
t.avance(5)
t.texte()
>>>21h 35m 0s
```

Nous avons déjà vue cette notation, par exemple, avec la fonction randint() de Rand.

Lors d'un appel *i.m(e1,...,en)* à une méthode m, l'objet i est appelé le *paramètre implicite* et les paramètres e1 à en, les *paramètres explicites*. Toutes les méthodes d'une classe attendent comme paramètre implicite un objet de cette classe. Les paramètres explicites, en revanche, de même que l'éventuel résultat de la méthode, peuvent être aussi bien des valeurs de base (nombre, chaînes de caractères, etc...) que des objets.

On peut ainsi imaginer dans notre classe Chrono une méthode ***egale*** s'appliquant à deux chronomètres (le paramètre implicite et un paramètre explicite) et testant l'égalité des temps représentés, et une méthode ***clone*** s'appliquant à un chonomètre t et renvoyant un nouveau chronomètre initialisé au même temps que t.

```
u = t.clone()
print(t.egale(u))
>>>True
t.avance(3)
print(t.egale(u))
>>>False
```

Définition d'une méthode

Une méthode ressemble a une fonction ordinaire, ayant un nombre arbitraire de paramètres, à ceci près qu'elle doit nécessairrement avoir pour premier paramètre un objet de cette classe (le paramètre implicite).

Une méthode ne peut donc pas avoir zéro paramètre !

La définition d'une méthode se fait comme la définition d'une fonction. Par convention, ce premier paramètre est systématiquement appelé ***self***. Etant un objet, on va pouvoir accéder à ses attributs avec la notation self. . ***texte*** et ***avance*** peuvent être définies de la manière suivante :

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

Erreurs :

Ne pas mettre de parenthèses après un appel de méthode ne déclenche pas son appel (même si la méthode n'attendait aucun paramètre explicite). Cet oublie ne provoque pas pour autant une erreur en Python, l'interprète construisant à la place un élément particulier représentant "la méthode associée à son paramètre implicite". On pourra ainsi observer la réponse suivante si l'on tente un appel incomplet dans la boucle interactive.

```
print(t.texte)
<bound method Chrono.texte of <__main__.Chrono object at 0x000001FFB1D0DFD0>>
```

L'appel n'ayant pas lieu, cet oubli se manifestera en revanche certainement plus tard, soit du fait que cette valeur spéciale produite n'est pas le résultat de la méthode sur lequel on comptait, soit plus sournoisement car l'objet n'a pas été mis à jour comme il aurait dû l'être.

En revanche, utiliser un attribut numérique comme une méthode déclenche une erreur immédiate.

```
print(t.heures())
Traceback (most recent call last):
    File "...", line 16, in <module>
        print(t.heures())
TypeError: 'int' object is not callable
```

Le 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, gérée directement par l'interprèteur ou le compilateur du langage.
2. 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 les exemples.

__init__ est comme une méthode ordinaire :
Son premier attribut est ***self*** (il représente l'objet auquel elle s'applique)
Ses autres paramètres donnés explicitement lors de la construction.
Sa particularité est la manière dont elle est appelée, directement par l'interprèteur Pyhton en réponse à une opération particulière.

D'autres méthodes particulières en Python

Il existe en Python un certain nombre de méthodes particulières, chacune avec un nom fixé et entouré de deux paires " _". Elles sont appelées par certaines opérations prédéfinies de Python, pemettant parfois d'alléger ou d'uniformiser la syntaxe.

Pour un usage général :

Méthode	Appel	Effet
__str__(self)	str(t)	Renvoie une chaine de caractères décrivant t
__lt__(self, u)	t < u	Renvoie True si t est strictement plus petit que u
__hash__(self)	hash(t)	Donne un code de hachage pour t, par exemple pour l'utiliser comme clé d'un dictionnaire d

Pour un usage spécifique à une certaine catégorie d'objet, ici 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

On peut remarquer que la méthode ***texte*** de notre classe Chrono correspond a la méthode __str__, mais ne bénéficie pas de la syntaxe allégée puisque nous n'avons pas utilisé le nom dédié à cela. On pourrait éventuellement ajouter la définition suivante.

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

Egalité 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, identifié 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.

Classes et espace de noms

Deux classes, même définies dans un même fichier, peuvent avoir des attributs ou des méthodes de même nom sans porter à confusion.

En effet, on accède toujours aux méthodes et attributs d'une classe via un objet de cette classe (voir dans certains cas via le nom de la classe elle-même) et l'identité de cet objet permet de résoudre toutes les ambiguïtés potentielles. Ainsi, des noms d'attributs courants comme x ou y pour des coordonnées dans le plan, ou des noms de méthodes généraux comme ***ajoute*** ou ***__init__*** peuvent être utilisés dans plusieurs classes différentes sans risque de confusion. On dit qu'une classe définit un *espace de noms (namespace)*, c'est-à-dire une zone séparée des autres en ce qui concerne le nommage des variables et des autres éléments.

Attention, une classe donnée ne peut pas contenir un attribut et une méthode de même nom.

Accès direct aux méthodes

Dans un style de programmation objet ordinaire, l'appel de méthode se fait exclusivement avec la notation t.m(e1, ..., en) que nous présentons ici. En Python, il reste toutefois possible d'accéder directement à une méthode m d'une classe C et de l'appeler comme une fonction ordinaire. Il faut, dans ce cas, passer le paramètre implicite comme les autres : C.m(t, e1, ..., en)

Méthodes 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 appelée ***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 symétrique et dont aucune n'est modifié :

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

Pour appeler de telles méthodes, on peut utiliser la notation d'accès direct avec le nom de la classe.

```
print(Chrono.est_seconde_valide(64):)
False
print(Chrono.max(t, u))
<__main__.Chrono object at 0x10d8ac198>
```

Notez que de telles méthodes sont équivalentes à des fonctions qui seraient définies à l'exterieur de la classe. Cette notion n'est donc pas crucial en Python. Sachez simplement que définir comme ici une méthode sans paramètre self suffit à créer l'effet simple que nous venons de décrire, sans correspondre exactement à la notion de méthode statique de Python.

3 – Retour sur l'encapsulation

Dans la philosophie objet, l'intéraction avec les objets d'une classe se fait essentiellement avec les méthodes, et les attributs sont considérés par défaut comme relevant de détail d'implémentation. Ainsi, concernant la classe Chrono, il est fondamental de savoir que l'on peut afficher et faire évoluer les temps, mais l'existance des trois attributs heures, minutes et secondes est anecdotique.

En l'occurence, il serait certainement bienvenu de modifier la structure de cette classe pour simplifier toutes les opérations arithmétiques sur les temps. On pourrait ainsi se contenter d'un unique attribut ***_temps*** mesurant le temps en secondes.

```
class Chrono:

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

Les opérations arithmétiques modifieraient alors cet attribut sans besoin de se soucier des dépassements comme c'était le cas dans notre première version de la méthode ***avance***.

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

En contrepartie, nous devons adapter le code de certaines méthodes pour qu'elles assurent la conversion entre les secondes et les triplets "heures, minutes, secondes".

```
    def texte(self):
        return (str(self._temps // 3600) + 'h '
                + str((self._temps // 60) % 60) + 'm '
                + str(self._temps % 60) + 's')
```

Dans certains cas, on pourra introduire également des méthodes qui ne sont pas destinées à faire partie de l'interface. Par exemple ici, on peut ajouter une méthode ***_conversion*** qui extrait d'un temps le triplet (h, m, s) correspondant, destinée à être utilisé par les méthodes principales comme texte et clone. Le programme suivant donne une version complète de la classe Chrono qui inclut cette méthode auxiliaire.

```
class Chrono:

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

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

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

    def egale(self, u):
        return self._temps == u._temps

    def clone(self):
        h, m, s = self._conversion()
        return Chrono(h, m, s)
```

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

La présence du symbole "_" n'est qu'une déclaration d'intention en Python : il rappel à un utilisateur extérieur que celui-ci n'est pas censé utiliser la méthode ***_conversion***, sans que rien dans le langage ne l'empêche réellement de le faire.

Notez que, cette remarque étant faite sur la philosophie de la programmation objet, il aurait mieux valu appeler nos trois attributs ***_heures***, ***_minutes*** et ***_secondes*** dans la première version de la classe Chrono, c'est-à-dire préfixer leur nom d'un symbole "_" soulignant leur caractère interne. En revanche, les méthodes conservent bien , sauf exception, leur nom tel quel : elles forment ***l'interface*** des objets de cette classe.

4 – Héritage

L'héritage est hors programme, mais étant l'un des principaux éléments distinctifs de la programmation orientée objet, il faut l'aborder, brievement, pour donner une présentation équilibrée de ce paradigme de programmation.

Il est possible de définir une nouvelle classe comme une extension d'une classe existante. Dans ce contexte, la classe d'origine est appelée classe de base ou classe mère et la nouvelle, classe fille. Dans une telle situation d'extension, la classe fille possède automatiquement tous les attributs et méthodes de la classe de base (on dit qu'elle en hérite), et peut en outre ajouter à sa définition de nouveaux attributs et de nouvelles méthodes qui lui sont spécifiques. Il faut comprendre la classe fille comme définissant un cas particulier de la structure générale décrite par la classe mère. On dit donc aussi que la classe fille est une spécialisation de la classe mère et que la classe mère est une généralisation de la classe fille.

Ainsi, une structure CompteARebours peut être définie comme un Chrono qui posséderait, en plus de ce qui caractèrise un chrono ordinaire, la capacité de faire évoluer son temps à reculons. La définition à écrire pour cela est la suivante.

```
class CompteARebours(Chrono):
    def tac(self):
        self._temps -= 1
```

A la première ligne de la définition, le nom de la nouvelle classe suit directement le mot-clé class et le nom de la classe de base est fourni entre parenthèses. La définition du contenu de la classe ne mentionne ensuite que ce qui est spécifique à la classe fille, ici la nouvelle méthode tac. Toutes les méthodes déjà présentes dans Chrono, comme __init__ et texte, sont héritées : elles sont présentes sans qu'on les ait mentionnées à nouveau.

```
>>> c = CompteARebours(0, 1, 1)
>>> c.tac()
>>> c.tac()
>>> c.texte()
'0h 0m 59s'
```

Dans une situation d'héritage, une instance de la classe fille possède tous les attributs et méthodes de la classe mère. Il s'ensuit que tout programme destiné à manipuler une instance de la classe mère arrivera tout aussi bien à manipuler une instance de la classe fille : un objet de la classe fille peut être vu comme un objet de la classe mère.

Il se trouve que la spécialisation que représente une classe fille va plus loin que le seul ajout de nouveaux attributs ou de nouvelles méthodes : certaines méthodes de la classe mère peuvent également être redéfinies pour être mieux adaptées au cas particulier défini par la classe fille. Pour cela, il suffit de définir à nouveau une méthode du même nom qu'une méthode déjà existante.

On peut par exemple définir dans la classe CompteARebours étendant Chrono une nouvelle version de la méthode texte.

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

Notez que ce code réalise au passage un appel self._conversion(), qui fait référence à la méthode _conversion héritée de la classe Chrono.

Héritage et réutilisation du code

Grace à l'héritage, la réutilisabilité du code et démultiplié, surtout si l'on a besoin de réecrire plusieurs fonctions qui réutilise une partie du même code. On pourrais donc créer une classe mére qui regrouperait toutes les fonctions communes, puis créer les classes filles depuis la classe mére, et compléter les classes fille. Voir aussi la substitution.

Héritage multiple

Sachez qu'il est aussi possible de faire hérité une classe fille, de plusieurs classe mère. Cependant, cela pose beaucoups de questions, notamment si, au sein des classe mère, se retrouve les mêmes nom de fonction.

Accès aux méthodes de la classe mère

Il est possible d'accéder directement aux méthodes d'une classe mère, en utilisant la fonction spéciale ***super***, qui donne accès aux méthodes de la classe mère, tel quelles sont définit. On pourrait, notamment, redéfinir notre méthode texte comme ceci :

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

"La programmation orientée objet, structure les programmes, regroupe dans une même entité des données ainsi que le code permettant de les manipuler. Ces données sont appelées ***attributs****, que l'on manipule via des* ***méthodes.*** *Chaque objet sera une* ***instance*** *de cette* ***classe."***

Cours grandement inspiré du livre Numérique et Sciences Informatique, aux éditions Ellipses.

Programmer avec des objets