Classes

Quand on s'y met, on a du mal à s'en passer...

Paradigmes de programmation

Programmation procédurale : Le programme est découpé en **fonctions** ou **procédures** (*càd fonction sans renvoi*) réalisant chacune un morceau de l'ensemble.

Les fonctions sont alors séparées des données traitées, toutes les données sont « au même niveau », elles subissent passivement.

Programmation Orientée Objet : POO : les données sont structurées en **objets** qui définissent des **méthodes** pouvant leur être appliquées, et des **attributs**. Un objet est défini par une **classe** qui est une sorte de manuel de fabrication et d'utilisation.

classe : sorte de notice qui détaille comment on fabrique l'objet et comment on l'utilise.
méthode : fonction définie dans une classe.
attribut : donnée membre de l'objet. Elle caractérise cet objet. *Exemple : objet Humain aurait les attributs genre, age...*

Apparu en 1960 avec **LISP**. Très répandu aujourd'hui. En **Java** (*1995*), *presque* tout est objet. La différence essentielle entre **C** (*1972*) et **C++** (*Bjarne Stroustrup 1983*) est la présence de classes en C++.

Quel problème rencontre-t-on sans classe ?

Dans le cours on donne l'exemple de structures représentant des triangles, des rectangles... :

1. Pas de façon simple pour regrouper toutes les données concernant un triangle,
2. pas de façon simple de savoir qu'une variable contient un triangle, et un triangle bien formé,
3. multiplication de fonctions en vrac. Par ex. il faut une fonction `aireTriangle(t)` pour un triangle `t` et une fonction `aireRectangle(r)` pour un rectangle `r`.

En quoi les classes apportent une solution ?

Point par point :

1. La classe définit un modèle qui définit précisément la façon de créer et d'utiliser des objets.
2. Dès lors qu'un objet est de type `Triangle` (*par ex.*), on est certain qu'il respecte les contraintes qu'on a fixé dans la classe.
3. Chaque classe définit ses propres fonctions (méthodes) de sorte que le code est mieux organisé. Par ex. `t.aire()` et `r.aire()`.
   Chacun appelle la fonction `aire` qui lui est propre, sans ambiguïté.

Par convention, le **nom d'une classe** utilise **UpperCamelCase** : première lettre en majuscule, majuscule à chaque changement de mot : `Triangle`, `TriangleIsocele`...

Comment procéder

- Identifier les objets et leurs attributs : quelles sont les données caractérisant les objets
- Définir l'interface : comment utilise-t-on l'objet, notamment depuis l'extérieur ?
- **visibilité** ? quels attributs / méthodes sont visibles par les autres ?
  attribut accessible = public – attribut non accessible = privé

Classe en Python

```
"""Commentaire décrivant le fichier.
De préférence : un fichier par classe.
"""

class MaClasse:
    x: float  # définit attribut x de type float
              # pas obligatoire mais plus lisible
    y:int = 0 # attribut y initialisé à 0
    def __init__(self, arg1, arg2):
        """Constructeur : méthode appelée
        à l'initialisation d'un objet.
        """
        self.x = arg1 + arg2 # initialisation de l'attribut x
                             # selon les arguments fournis

    def ma_fonction(self, arg1) -> int:
        """méthode recevant l'argument arg1
        self est toujours placé en premier argument
        self désigne l'objet appelant.
        -> int non obligatoire mais plus lisible.
        """
        return arg1 * self.x   # self.x = attribut x de l'objet
```

La commande `help(MaClasse)` permettra de lire la documentation (*les commentaires*).

Instanciation

`MaClasse` est un modèle. La définition de `MaClasse` ne crée pas d'objet.

```
mon_objet = MaClasse(5, 38)
```

Crée une **instance** d'objet `MaClasse` référencé par la variable `mon_objet`.

Autrement dit : *On peut définir ce qu'est une voiture et comment on fait une voiture dans un livre. Ce livre n'est pas une voiture. Quand on suit le livre et qu'on fabrique la voiture, on obtient une réalisation de voiture, c'est une* ***instance*** *de voiture.*

Une même classe peut avoir plusieurs instances.
On peut fabriquer plusieurs voitures, identiques ou pas.

- L'instanciation `mon_objet = MaClasse(...)` d'un objet provoque l'exécution du **constructeur**. Le constructeur réserve l'espace nécessaire en mémoire puis initialise l'objet avec la méthode `__init__(self, ...)`.
- En tant qu'objet `MaClasse`, `mon_objet` a des attributs et des méthodes.

  `mon_objet.x` désigne l'attribut `x` de `mon_objet`

  `mon_objet.ma_fonction` désigne la méthode `ma_fonction` de `mon_objet`.

- Dans la signature et le corps de la méthode `ma_fonction`, on peut voire apparaître le mot-clé `self`.

  La méthode `ma_fonction` ne peut être appelée que par un objet de type `MaClasse`, par exemple `mon_bjet.ma_fonction(45)`.
  Dans ce cas `self` désigne alors l'instance `mon_objet` lui.même.

```
objA.ma_fonction(10) # dans ce cas, self = objA
objB.ma_fonction(15) # dans ce cas, self = objB
```

- En Python, **lors de la définition** de la méthode `ma_fonction`, on est obligé de préciser l'argument `self` en première position.

  Au moment de l'appel, on ne précise pas `self`. On se contente de (par ex) `objA.ma_fonction(12)`. Tout se passe comme si `objA` était premier argument, donc `self = objA`.

À `self`, d'autres langages préfèrent `this`.

Sans classe, on faisant `aireTriangle(triangle)` où `triangle` subissait la fonction passivement.

Maintenant, `triangle` est un objet doté de méthodes et on écrit `triangle.aire()`. L'objet est **responsable** du calcul.

Chaque type d'objet définit ses propres méthodes. Le code devient plus clair : `triangle.aire()`, `rectangle.aire()`. C'est léger, lisible, bien organisé.

visibilité : public / privé

Supposons qu'on se soit fixé la contrainte `x != 0` dans la classe `MaClasse`.

Lors de la création `mon_objet = MaClasse(a,b)`, la fonction `__init__` s'exécute. On peut ajouter dans `__init__` des vérifications pour s'assurer que `self.x != 0`

Problème : bien que, à la création, on se soit soit assuré que `mon_objet.x != 0`, rien n'empêche plus tard d'écrire l'affectation `mon_objet.x = 0`. Comment empêcher que l'on modifie n'importe comment les attributs ?

Solution : On peut rendre l'attribut **privé** de façon qu'il ne soit plus modifiable que depuis l'intérieur de la classe.

Nous en parlons car c'est un grand intérêt des classes. Mais ne vous en soucier pas trop : nous pouvons ignorer la visibilité pour cette introduction.

Bonne pratique : Classes et sous-classes

On cherche à éviter tout bloc de code trop compliqué.

On n'hésite pas à créer une classe, même pour quelque chose de très simple. Si une classe est compliquée, c'est qu'elle est responsable de trop de choses. On la découpe au maximum en sous-structures plus simples.

Exemple :

- Au lieu de : Classe `Triangle` ayant 6 attributs `x1, y1, x2, y2, x3, y3`
- On préfère : Classe `Point` ayant 2 attributs `x, y` et la classe `Triangle` ayant les 3 attributs `p1, p2, p3` de type `Point`.

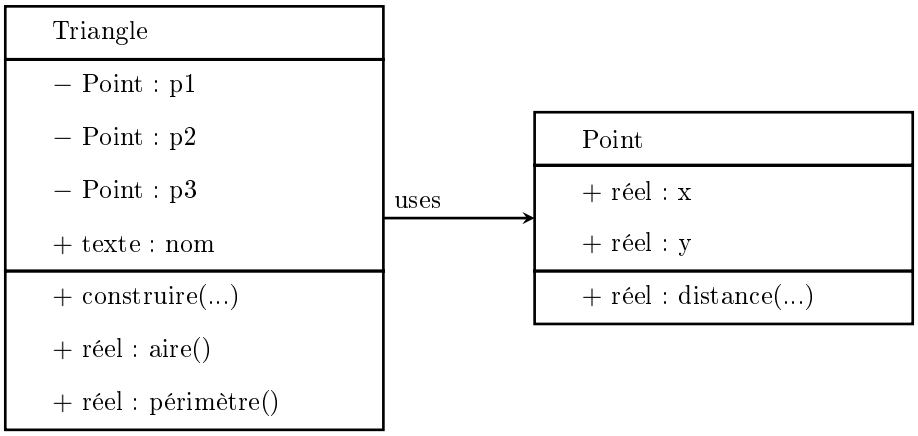
```
>>> p1 = Point(2,3)
>>> p2 = Point(2,3)
>>> p1 == p2 # la réponse est False !
```

Les deux points ont les mêmes coordonnées mais ce sont néanmoins deux points différents.

On peut imaginer deux voitures de même marque, même modèle, produite dans le même atelier le même jour... et pourtant ce sont deux voitures distinctes.

Diagramme UML - Triangle utilise Point

Le diagramme UML permet de représenter la définition des différentes classes ainsi que les dépendances entre elles. Ici, la classe `Triangle` utilise la classe `Point`.



On utilise – pour attribut / méthode privé, + pour public.

Méthodes magiques

On peut configurer les classes pour qu'elle réagissent aux différentes situations de programmation.

Exemple : Quoi faire dans le cas où on additionne : `obj1 + obj2` ?
En Python, on peut définir dans la classe la **méthode magique** `__add__` qui définira le comportement attendu lors d'une addition.

Les méthodes magiques sont nombreuses. La plus utile dans un premier temps est celle qui permet de **tanstyper** l'objet vers `str`. C'est la fonction `__str__`.
Exemple :

```
class MaClasse
    ...
    def __str__(self):
        return "Je suis un objet MaClasse avec x = {}".format(self.x)
```

De sorte que :

```
>>> obj = MaClasse(10.0, 15.2) # donc on aura obj.x = 25.2
>>> print(obj) # affiche le message défini dans __str__
```

L'appel `print(...)` nécessite un argument de type `str`. Donc `obj` est automatiquement transtypé en utilisant la méthode `__str__`.