2 : Notions de classe et d'objet

2.1 : Principe de la programmation orientée objet

La **programmation orientée objet** consiste à écrire un programme en décomposant :

- la définition de structures de données appelées **objets**,
- de l'interaction entre les objets d'un même type ou entre des objets de différents types.

Un objet peut représenter un concept, une entité du monde physique, ... : par exemple, un individu, une voiture, un nombre complexe, une matrice, etc.

Il s'agit donc de représenter ces objets et leurs relations. Par conséquent, l'étape de modélisation revêt une importance majeure et nécessaire pour la **Programmation Orientée Objet**. C'est elle qui permet de transcrire les éléments du réel sous forme virtuelle.

On peut par exemple se poser les questions suivantes:

- est-ce que tel ou tel concept a intérêt à être décrit par une classe,
- comment formaliser ce concept sous forme de classe,
- quelles interactions décrire ou interdire,
- etc.

2.2 : Notion de classe

La **classe** est un concept qui permet de créer des **types de données** complexes (au sens où elles sont composées de plusieurs données simples) ainsi que l'interaction entre ces nouveaux types de données.

Une classe est une abstraction permettant de créer une infinité d’objets.

Exemples : dans le cas des voitures ou des nombres complexes :

- la **classe** correspond à l'ensemble possible des voitures ou des nombres complexes,
- l'**objet** (ou **instance**) correspond a un exemple de voiture ou un nombre complexe particulier.

Commencons par créer ces deux classes, vides pour le moment :

In [15]:
```
class voiture:  # construction de la classe voiture
    pass  # mot clé tant que le classe est vide


class complexe:  # construction de la classe complexe
    pass
```

Une classe génère des objets faisant tous partie d’une **même famille** ou d'un **même type**.

D'un certain point de vue, le type `float` est une classe et `x = 2.0` est un objet (ou instance) de la classe `float`.

Comment déclarer/instancier un objet/une instance ?

In [16]:

```
ma_voiture = voiture()
x = complexe()
y = complexe()
z = float()  # en pratique on ne fait jamais ça
```

2.2.1 : Attributs et constructeur

Les attributs que vont posséder chacun des objets sont des **variables définies au sein de la classe**.

Il s'agit des **données** contenues par tous les objets de la classe. On parle de variables membres d’une classe.

Le constructeur est une **fonction fondamentale** qui initialise les attributs de la classe.

In [1]:

```python
class voiture:  # construction de la classe voiture
    def __init__(self):
        self.marque = "none"
        self.modèle = "none"
        self.type_essence = "none"
        self.plaque = "none"
        self.nb_roues = 0  # etc
```

- `marque`, `modèle`, `type_essence` et `plaque` sont des chaînes de caractères **attributs** de la **classe** **`voiture`**
- `nb_roues` est un attribut entier de la classe `voiture`

In [3]:

```
class complexe:  # construction de la classe complexe
    def __init__(self, x, y):
        self.reel = x
        self.imag = y

c1 = complexe(2.0, 3.0)
print(" partie réelle: ", c1.reel, "; partie imaginaire: ", c1.imag)
```

```
 partie réelle:  2.0 ; partie imaginaire:  3.0
```

`imag` et `reel` sont les deux attributs `float` de la classe : on voit qu'avec ces deux nombres on définit l'ensemble des complexes

Le constructeur qui initialise un objet s'écrit **toujours**: `__ init __ (self, args)`.
`self` correspond à l'objet à créer.

Le constructeur est une fonction comme une autre, mis à part le **self** en premier argument. D'autres arguments sont optionnels :

- `ma_voiture = voiture()` appelle le constructeur de `voiture` sans arguments,
- `c1 = complexe(2.0, 3.0)` appelle le constructeur avec les deux arguments `2.0` et `3.0`.

Pour accéder à la valeur d'un attribut d'un objet d'une classe, on écrit `objet.attribut`.

2.3.2 : Méthodes

Les **méthodes** sont des fonctions qui **appartiennent à la classe** et qui permettent d'agir sur les objets.

Contrairement a une fonction qui peut prendre n'importe quel argument, une **méthode s'applique forcément a un objet**, même si elle peut prendre des arguments supplémentaires.

Un exemple pour la classe complexe :

In [7]:

```
import numpy as np

class complexe:  # construction de la classe complexe
    def __init__(self, x, y):
        self.reel = x
        self.imag = y
    def module(self):
        return np.sqrt(self.reel * self.reel + self.imag * self.imag)
    def conjugué(self):
        c = complexe(self.reel, -self.imag)
        return c
    def print_c(self):
        print("partie réelle: ", self.reel, "; partie imaginaire: ", self.imag)

c1 = complexe(1.0, 1.0)
c1.print_c()
print("module =", c1.module())
print("conjugué : x = ", c1.conjugué().reel, "; y = ", c1.conjugué().imag)
```

```
partie réelle:  1.0 ; partie imaginaire:  1.0
module = 1.4142135623730951
conjugué : x =  1.0 ; y =  -1.0
```

Ici, on a deux méthodes:

- `module`, qui prend un objet et renvoie un double,
- `conjugué`, qui prend un objet et renvoie un autre objet.

On peut aussi écrire des méthodes **qui modifient un objet**.

Une méthode de classe appartient aussi à la classe, comme un attribut. On l'appelle donc également avec `objet.methode()`.

Un exemple pour la classe voiture :

In [1]:

```
class voiture:  # construction de la classe voiture
    def __init__(self):
        self.marque = "none"
        self.modèle = "none"
        self.type_essence = "none"
        self.plaque = "none"
        self.nb_roues = 0  # etc

    def allumer(self):
        self.est_allumée = True
        print("la voiture est allumée")

    def éteindre(self):
        self.est_allumée = False
        print("la voiture est éteinte")

ma_voiture = voiture()
ma_voiture.allumer()
ma_voiture.éteindre()
```

```
la voiture est allumée
la voiture est éteinte
```

Ici, dans les méthodes, on modifie un attribut qui n'as pas été introduit dans le **constructeur**.

Ce n'est pas grave. Si on initialise un attribut dans une méthode et non dans le constructeur, cet attribut **devient un attribut de la classe**.

2.3 : Notion d'objet et utilisation

On va maintenant regarder plus précisement la notion d'objet et l'utilisation des objets.

On dit souvent qu'un objet est composé :
d'un état, d'un comportement et d'une identité.

État: *Valeurs à un instant donné des attributs de l'objet*. Deux objets peuvent avoir le même état.

Comportement: *Description de la réaction d'un objet (et donc de son état) à une modification.* L'ensemble des méthodes de la classe (agissant sur l'objet) forment son comportement.

Identité: *Caractérisation unique d'un objet*. En Python, chaque objet obtient à a sa création un identifiant unique.

In [3]:

```
x = complexe(1.0, 2.5)
y = complexe(1.0, 2.5)
print("État de x:")
x.print_c()
print("État de y:")
y.print_c()

print("-" * 45)
print("Identité de x avant y = x : ", id(x))
print("Identité de y avant y = x : ", id(y))
y = x
print("Identité de x après y = x : ", id(x))
print("Identité de y après y = x : ", id(y))
```

```
État de x:
partie réelle:  1.0 ; partie imaginaire:  2.5
État de y:
partie réelle:  1.0 ; partie imaginaire:  2.5
---------------------------------------------
Identité de x avant y = x :  140519118530064
Identité de y avant y = x :  140519118530016
Identité de x après y = x :  140519118530064
Identité de y après y = x :  140519118530064
```

On voit sur l'exemple précédent que l'opérateur égal `=`, comme précédemment, fait une copie par référence ou adresse et non par valeur (copie profonde).

il faut donc faire attention : une modification de `x` après un `y = x` entraîne une modification de `y`. C'est le même processus que pour les listes et toutes les variables. On parlera plus tard des copies profondes.

La fonction `id()` est ce qu'on appelle un built-in. Il s'agit d'une fonction qui peut prendre en entrée tout objet.

2.4 : Cycle de vie

Tout objet possède un **cycle de vie**. Il débute par la construction de l'instance et se termine par sa destruction. Après sa construction et avant sa destruction, tout objet évoluera et verra son état être modifié par les **méthodes** de la classe et des **fonctions exterieures**.

On a déjà vu le constructeur qui permet d'initialiser l'objet. On l'appelle toujours `__ init __ (self, args)`.

Une fois l'objet en fin de vie, il sera détruit au moyen d'une méthode spéciale dédiée à cet effet : le destructeur.

En Python, la construction et l'appel de cette méthode peut être réalisée directement par le langage de programmation dans le but de gérer efficacement la mémoire, mais on peut aussi le faire manuellement. Notons que certains langages (notamment le `C++`, étudié en L3), demandent de créer et d'appeler explicitement un destructeur.

Cette opération lancée en toile de fond par un langage et visant à supprimer de la mémoire des objets non utilisés est appelée **ramasse-miettes** (*garbage collector* en anglais).

In [1]:

```
class voiture:  # construction de la classe voiture
    def __init__(self):
        self.marque = "none"
        self.modèle = "none"
        self.type_essence = "none"
        self.plaque = "none"
        self.nb_roues = 0  # etc

    def allumer(self):
        self.est_allumée = True
        print("la voiture est allumée")

    def éteindre(self):
        self.est_allumée = False
        print("la voiture est éteinte")

    def __del__(self):
        print("objet effacé", id(self))


ma_voiture = voiture()
print("id ma_voiture", id(ma_voiture))
ma_voiture = None
```

```
id ma_voiture 139733678284128
objet effacé 139733678284128
```

Dès qu'on écrit `ma_voiture = None`, la variable `ma_voiture` ne "pointe" plus vers l'objet. L'objet n'est donc plus **référencé par une seule variable**.

L'objet est donc inaccessible : dans ce cas, le destructeur est automatiquement appelé, et l'objet est détruit.

A la fin du programme, tous les objets sont détruits (ici, à la fin du notebook).

2.5 : Affichage d’un objet

In [23]:

```
x = complexe(2.0, 3.0)
print(x)
x.print_c()
```

```
<__main__.complexe object at 0x7f361087e0d0>
partie réelle:  2.0 ; partie imaginaire:  3.0
```

On voit que le `print` classique ne marche pas sur l'objet : il affiche simplement la classe d'appartenance et l'adresse mémoire de l'objet, mais pas ses attributs.

Il faudrait utiliser la méthode `print_c` pour afficher un complexe : ce n'est pas très pratique.

In [2]:

```
class complexe:  # construction de la classe complexe
    def __init__(self, x, y):
        self.reel = x
        self.imag = y

    def __str__(self):
        return "partie réelle: " + str(self.reel) + " ; partie imaginaire: " + str(self.i


c = complexe(2.0, 3.0)
print(c)
```

```
partie réelle: 2.0 ; partie imaginaire: 3.0
```

On a ici modifié, pour la classe `complexe`, la définition de la fonction d'entrée-sortie `__str__`. Cette fonction doit renvoyer une *chaîne de caractères* : dans notre cas, on la forme avec l'opérateur `+`, qui permet de concaténer plusieurs chaînes de caractères, et en appliquant la fonction `str` (une fonction de base de `Python`) aux parties réelle et imaginaire pour les convertir de `float` en chaîne de caractères.

Cette fonction est ensuite appelée par la fonction `print`.

On observe que les fonctions fondamentales de l'objet, comme le constructeur et l'affichage, sont toujours précédées et suivies de `__`.

2.6 : Encapsulation

L'**encapsulation** est un des principes fondateurs de la programmation orientée objet.

Elle discrimine deux types d'éléments (attributs ou méthodes) qui seront :

- soit **publics**, accessibles depuis le corps du programme (en-dehors de la classe),
- soit **privés**, inaccessibles depuis le corps du programme : il faudra faire appel à des méthodes spécifiques.

Remarque : Pour le moment, nous n'avons introduit que des éléments publics.

L'encapsulation permet :

- de contrôler l'accès qu'aura l'utilisateur aux variables privées,
- de masquer l'implémentation pour faciliter la vie à l'utilisateur,
- de faire facilement évoluer l'implémentation de la classe : les éléments privés peuvent être modifiés sans toucher aux éléments publics.

2.6.1 : L'encapsulation en `Python`

On fait un premier exemple en reprenant la classe `voiture`.

On va créer un attribut privé déterminant si la voiture est allumée ou non. On crée aussi deux méthodes, `allumer` et `éteindre`, qui permettent de contrôler cet attribut privé.

In [13]:

```
class voiture:
    def __init__(self):
        self.__est_allumée = False
    def allumer(self):
        self.__est_allumée = True
        print("La voiture est allumée")
    def éteindre(self):
        self.__est_allumée = False
        print("La voiture est éteinte")

ma_voiture = voiture()
ma_voiture.allumer()
print(ma_voiture._voiture__est_allumée)
print(ma_voiture.__est_allumée)
```

```
La voiture est allumée
True
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-13-820fa77ebfab> in <module>
     12 ma_voiture.allumer()
     13 print(ma_voiture._voiture__est_allumée)
---> 14 print(ma_voiture.__est_allumée)

AttributeError: 'voiture' object has no attribute '__est_allumée'
```

- Le nom d'un attribut privé en `Python` commencera toujours par `__` .
- Contrairement aux fonctions fondamentales de l'objet (`__init__` , `__str__` , etc.), les attributs privés sont simplement *précédés* par `__` .
- On ne peut pas accéder directement à la valeur de `ma_voiture.__est_allumée` : le programme s'arrête, et le message d'erreur précise la classe `voiture` n'a pas d'attribut `__est_allumée` .

- Pour accéder ou modifier un attribut privé, il faut donc nécessairement faire appel à des méthodes de la classe : on parle d'**accesseur** (*getter* en anglais) et de **mutateur** (*setter* en anglais).
- On peut de même rendre ces méthodes privées en les préfixant par `__`.

2.6.2 : Accesseurs et mutateurs

- Par convention, pour un attribut privé `__x`, on appelle `get_x` son accesseur et `set_x` son mutateur.
- L'accesseur, `get_x`, ne prend jamais d'argument, et doit simplement renvoyer la valeur de `x`.
- Le mutateur, `set_x`, prend (au moins) un argument, modifie la valeur de `x` en fonction de cet argument, et ne renvoie rien.

Dans l'exemple suivant, on s'intéresse au cas du compteur kilométrique d'une voiture : l'utilisateur (le conducteur de la voiture) peut connaître le nombre de kilomètres parcourus par la voiture, mais la seule modification possible est une augmentation liée au fait de rouler.

In [9]:

```
class voiture:
    def __init__(self):
        self.__km = 0
    def get_km(self):
        return self.__km
    def __set_km(self, valeur):
        self.__km = valeur
    def roule(self, distance):
        distance_totale = self.__km + distance
        self.__set_km(distance_totale)
    def compteur(self):
        print("Le compteur affiche", self.get_km(), "km.")

ma_voiture = voiture()
ma_voiture.compteur()
ma_voiture.roule(10)
ma_voiture.roule(82.5)
ma_voiture.compteur()
```

```
Le compteur affiche 0 km.
Le compteur affiche 92.5 km.
```

Remarques :

- Le nombre de kilomètres parcouru par la voiture, `__km`, est une variable privée.
- Pour afficher le nombre de kilomètres parcourus, on utilise la fonction `compteur`, qui appelle l'accesseur `get_km`.
- On peut aussi appeler directement l'accesseur `get_km`.
- Le mutateur `__set_km` est une méthode privée : on ne peut pas l'appeler en-dehors de la classe.
- La seule façon de modifier `__km` est donc d'utiliser la méthode publique `roule`, qui va incrémenter le nombre total de kilomètres parcourus par la voiture d'une valeur donnée en entrée.

Sans surprise, essayer d'appeler directement `__set_km` (par exemple pour faire diminuer le compteur de kilomètres) résulte en une erreur.

In [10]:
```
ma_voiture.compteur()
ma_voiture.__set_km(0)
```

```
Le compteur affiche 92.5 km.
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-10-d0143e51432d> in <module>
      1 ma_voiture.compteur()
----> 2 ma_voiture.__set_km(0)

AttributeError: 'voiture' object has no attribute '__set_km'
```

2.6.3 : Limites de l'encapsulation en `Python`

Contrairement à d'autres langages (notamment le `C++`, étudié en L3), les éléments privés en `Python` ne sont pas *véritablement* privés.

En effet, on peut tout de même y accéder en préfixant le nom de la classe précédé d'un `_` : par exemple, on peut accéder à l'attribut privé `__km` de la classe `voiture` avec la syntaxe `_voiture__km`.

In [3]:

```
ma_voiture.compteur()
ma_voiture._voiture__km = 50
ma_voiture.compteur()
ma_voiture._voiture__set_km(0)
ma_voiture.compteur()
```

```
Le compteur affiche 92.5 km.
Le compteur affiche 50 km.
Le compteur affiche 0 km.
```

- Ceci nécessite tout de même d'avoir accès au code source de la classe pour connaître les noms des éléments privés.
- En pratique, malgré ces limites, on pourra tout de même considérer que les éléments préfixés par `__` sont privés.

2.7 : TD

2.7.1 : Exercice 1 - Classes

On veut créer une classe pour manipuler un **lecteur audio**.

Cette classe va contenir les attributs

- `morceaux` (liste des morceaux),
- `en_lecture` (permet de savoir si le lecteur est en marche ou non),
- `position` (position dans la liste des morceaux).

1) Quel serait le type de chaque attribut ? (deux réponses possibles pour `position`)

2) Décrire une méthode qui ajoute un morceau à la liste, et une autre qui renvoie le nombre total de morceaux.

3) Créer une méthode `lire_pause()` qui modifie l'état de lecture du player. Lors du premier appel, le player est en lecture. Au deuxième, il est en pause. Au suivant, il repasse en lecture et ainsi de suite.

4) Créer une méthode `stopper()` qui arrête la lecture : la position de lecture est réinitialisée et le player est en mode pause.

5) Créer une méthode `avancer()` qui passe au morceau suivant si c'est possible.

2.7.2 : Exercice 2 - Polynômes

On souhaite écrire une classe pour manipuler des **polynômes réels**.

1) Quels seraient les attributs de la classe polynôme ? Justifier.

2) Écrire les mutateurs et accesseurs pour le coefficient `i` du polynôme.

5) Écrire une méthode qui évalue un polynôme en un point $x \in \mathbb{R}$ donné.

3) Écrire une méthode qui convertit un polynôme de degré p en un polynôme de degré $q > p$, qui aura la même valeur quand il sera évalué en un point $x \in \mathbb{R}$.

4) Écrire une méthode qui additionne deux polynômes (plus difficile).