Chapitre 1 - Programmation Orientée Objet

Séance 1 - Classe, objet, attribut, méthode

Introduction

Un **paradigme de programmation** pourrait se définir comme une philosophie dans la manière de programmer : c'est un parti-pris revendiqué dans la manière d'aborder le problème à résoudre. Une fois cette décision prise, des outils spécifiques au paradigme choisi sont utilisés.

Jusqu'à présent vous avez vu un seul paradigme de programmation : le **paradigme impératif** (ou la programmation impérative).
La programmation impérative repose sur des notions qui vous sont familières :

- la séquence d'instructions (les instructions d'un programme s'exécutent l'une après l'autre)
- l'affectation (on attribue une valeur à une variable, par exemple : a = 5)
- l'instruction conditionnelle (if/else/elif)
- la boucle (while et for)

La programmation impérative est loin d'être le seul paradigme de programmation (même si c'est sans doute le plus courant). Cette année nous allons étudier deux autres paradigmes : le **paradigme objet** et le **paradigme fonctionnel**.

La **Programmation Orientée Objet** (**POO**) est un **paradigme** de programmation informatique, c'est-à-dire une manière de programmer.

Il a été élaboré au début des années 1960.

Un **objet** représente un **concept**, une **entité** du monde physique, comme une voiture, une personne, etc

L'intérêt d'une approche par objets est que chaque objet peut être développé séparément, ce qui est très pratique pour travailler sur un projet en équipe.

Nous aborderons dans ce chapitre les notions essentielles de la POO à savoir la **classe**, les **objets**, les **attributs** et les **méthodes**.

Une classe est une famille d'objets équivalente à un nouveau type de données.

On connaît déjà par exemple les classes **`str`** *ou* **`list`** *et les nombreuses méthodes permettant de les manipuler en utilisant la notation pointée.*

```
s="nsi"
s.upper()
```

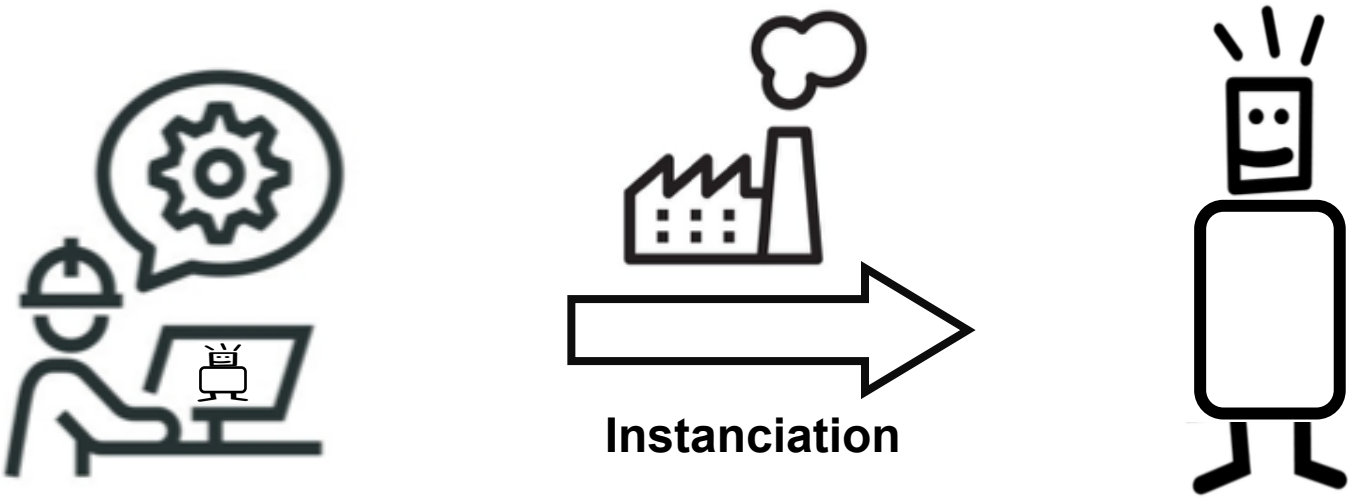
```
L=[10,20,30]
L.append(40)
```

Un objet ou une instance est un exemplaire particulier d'une classe.

Par exemple, **`s`** *est une instance de la classe* **`str`** *et* **`L`** *est une instance de la classe* **`list`**.

Classe et objet

Nous allons fabriquer des robots pour définir le concept objet.



Définition d'une classe Robot

```
class Robot :
```

Création d'un objet de type Robot

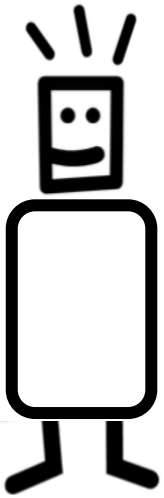
```
r1 = Robot()
```

On crée ainsi un **objet** de type Robot.

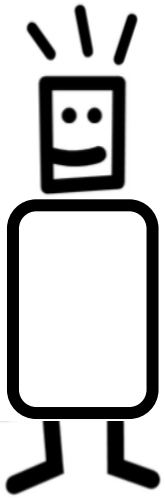
En POO, on dit que l'on crée une **instance** de la **classe** Robot.

Une phrase emblématique de la POO consiste à dire qu'un objet est une instance de classe.

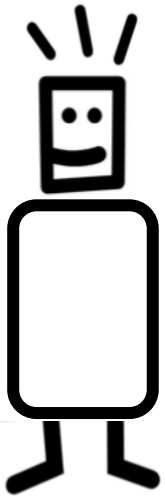
```
class Robot :
```



```
r1 = Robot()
```



```
r2 = Robot()
```



```
r3 = Robot()
```

Nous avons ici 3 **instances** de la classe Robot (3 objets).

On fait bien ici la distinction entre **classe** et **objet**.

Ici nous avons une seule classe Robot**, et trois objets de type** Robot.

In [1]:

```
1  class Robot :
2      pass
3
4  r1=Robot()
5  r2=Robot()
6  print(r1)
7  print(r2)
8  print(type(r1))
9  print(type(r2))
```

Par convention en Python, le nom identifiant une classe débute par une majuscule.

Pour l'instant, la classe est déclarée vide, d'où l'utilisation du mot-clé `pass`

```
<__main__.Robot object at 0x10a5646d8>
<__main__.Robot object at 0x10a564710>
<class '__main__.Robot'>
<class '__main__.Robot'>
```

Le message renvoyé par Python indique que `r1` *contient une référence à une instance de la classe* `Robot` *, qui est définie elle-même au niveau principal du programme. Elle est située dans un emplacement bien déterminé de la mémoire vive, dont l'adresse apparaît ici en notation hexadécimale.*

A l'intérieur de la classe, on définit les caractéristiques et les comportements communs aux objets de la classe.

Les caractéristiques ou propriétés s'appellent les **attributs** et les comportements s'appellent les **méthodes**.

On peut définir un **objet** comme une ***capsule*** contenant des attributs et des méthodes :

objet = attributs + méthodes

On peut représenter les classes via des diagrammes **UML** (*Unified Modeling Language*).

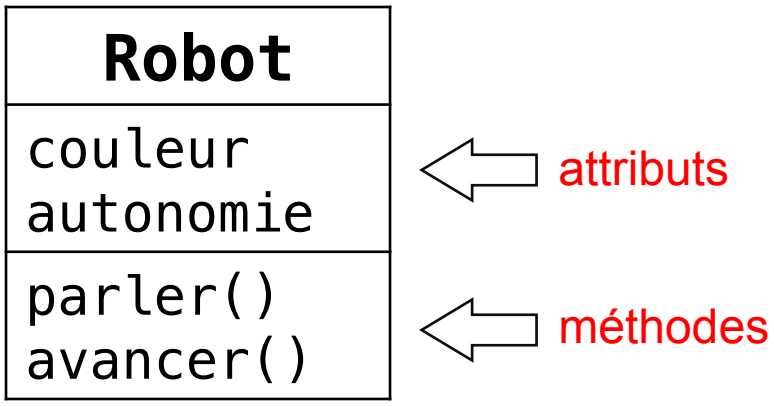
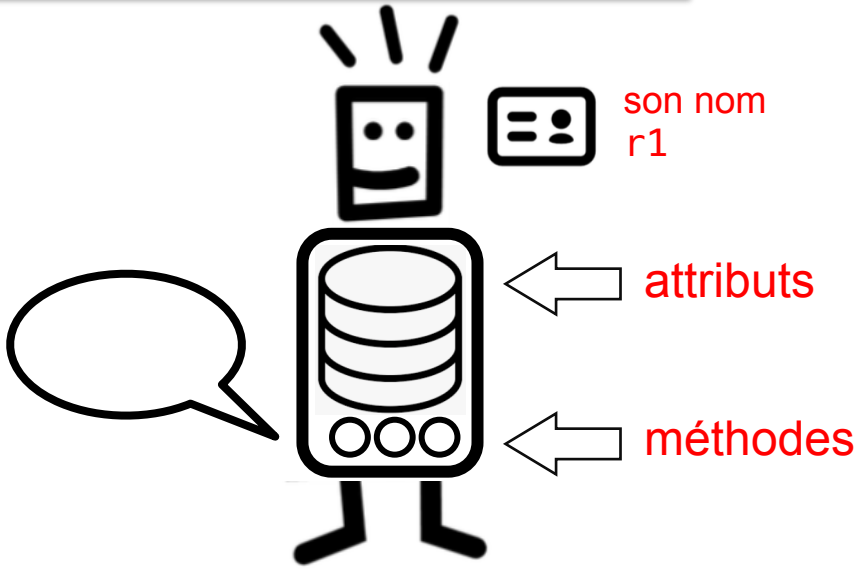


Diagramme UML
de la classe Robot




```
r1 = Robot()
```

*Nous représenterons par la suite une **classe avec son diagramme** UML*
*et **ses objets avec nos robots**.*

Si l'on joue avec des cartes :

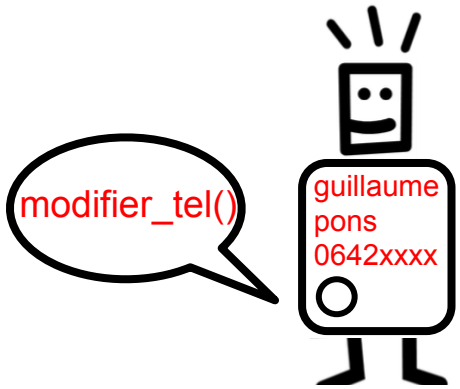
Carte
couleur valeur


```
c1=Carte("coeur",7)
```

```
c2=Carte("pique",10)
```

… avec des personnes :

Personne
prenom nom tel
modifier_tel()




```
p1=Personne("guillaume","pons","0642xxxxxx")
p1.modifier_tel("0843xxxxxx")
```

Création d'un attribut

```
1  class Robot :
2      couleur = "rouge"
3
4  r1=Robot()
5  # syntaxe pour accéder à l'attribut couleur : r1.couleur
6  print(r1.couleur)
7
8  r2=Robot()
9  print(r2.couleur)
```

```
rouge
rouge
```

*Les attributs définis ainsi sont appelés **attributs de classe** car ils sont liés à la classe, par opposition aux **attributs d'instance** dont la vie est liée à l'instance à laquelle ils sont rattachés. Les attributs de classe sont automatiquement reportés dans les instances de cette classe et deviennent des attributs d'instance.*

Tous les objets créés seront rouges… ce n'est pas très pratique… nous y reviendrons dans un instant avec une méthode spéciale d'initialisation.

Création d'une méthode

Une méthode est une fonction définie dans une classe, ayant comme premier argument (ou paramètre) une instance de cette classe.

L'utilisation du mot clé **def** se fait comme lorsqu'on définit une fonction normale Python.

Il faut indiquer son **nom** (par exemple parler) puis indiquer les arguments entre parenthèses.

Son premier argument est toujours self qui désigne l'objet sur lequel s'appliquera la méthode.

```
1  class Robot :
2      def parler(self):
3          return "Bonjour!"
4
5  r1=Robot()
6  r1.parler()
```

```
'Bonjour!'
```

On provoque l'appel d'une méthode en indiquant le nom de la variable qui fait référence à cet objet, la notation pointée, le nom de la méthode et les parenthèses. *On dit que la méthode est "appelable". Lors de l'appel de la méthode, le paramètre self n'est pas utilisé et la valeur qu'il prend est la référence à l'objet.*

Le corps de la méthode peut faire référence à self.

Par exemple, la méthode changer_couleur possède ici 2 paramètres :

self et une nouvelle couleur new.

Cette méthode va modifier l'attribut couleur de l'objet self.

```
class Robot :
    couleur = "rouge"
    def changer_couleur(self,new):
        self.couleur = new

r1=Robot()
print(r1.couleur)
r1.changer_couleur("jaune")
print(r1.couleur)
```

```
rouge
jaune
```

Lors de l'appel de la méthode, le paramètre self n'est pas utilisé, la valeur qu'il prend est la référence à l'objet. Il y a donc toujours un paramètre de moins que lors de la définition de la méthode.

On connaît déjà par exemple les classes **`str`** *ou* **`list`** *et les nombreuses méthodes permettant de les manipuler en utilisant la notation pointée.*

```
s="nsi"
s.upper()
```

```
L=[10,20,30]
L.append(40)
```

Notion de constructeur

Lors de la création d'un objet, nous pouvons utiliser une méthode spéciale appelée **constructeur**. Un constructeur n'est rien d'autre qu'une **méthode**, sans valeur de retour, qui porte un nom imposé par le langage Python : **__init__()**.
Cette méthode sera appelée lors de la **création** de l'objet.
Lors de l'instanciation d'un objet, la structure de base de l'objet est créée "nue" en mémoire, et la méthode __init__ est automatiquement appelée pour initialiser l'objet.
C'est dans cette méthode spéciale que sont créés les attributs d'instance avec leur valeur initiale.
Son paramètre self représente l'objet qui va être créé.
Ci-dessous le corps de __init__ indique qu'à chaque fois que l'on créé un objet, il faut lui associer 2 attributs d'instance qui lui sont propres, par exemple couleur et autonomie (en heures).

```
1  class Robot:
2      def __init__(self):
3          self.couleur = "rouge"
4          self.autonomie = 10
5
6  r1=Robot()
7  print(r1.couleur)
8  print(r1.autonomie)
```

```
rouge
10
```

Instanciation d'un objet avec ses attributs

Avec le constructeur que nous avons écrit, tous les objets de la classe sont initialisés à la création avec les mêmes valeurs. Nos robots sont tous rouges. Ceci n'est pas très pratique.

Comme __init__ est une méthode comme une autre, il suffit de lui ajouter des paramètres permettant de transmettre les valeurs que l'on souhaite pour les attributs dont l'initialisation doit varier.

```
1  class Robot:
2      def __init__(self,couleur,auto):
3          self.couleur = couleur
4          self.autonomie = auto
5      def parler(self):
6          return "Bonjour!"
7      def changer_couleur(self,new):
8          self.couleur = new
9
10 r1=Robot("jaune",10)
11 r2=Robot("rouge",20)
```

Dans cet exemple, le paramètre couleur et l'attribut couleur ont le même nom. Ceci ne pose pas de problème car ces variables ne sont pas dans le même espace de noms. Les paramètres sont des variables locales, comme c'est habituellement le cas pour une fonction. Les attributs de l'objet sont eux dans l'espace de noms de l'instance. Les attributs se distinguent facilement car ils ont self devant.

La méthode __init__() peut aussi comporter des paramètres avec pour chacun des **valeurs par défaut**. Nous obtenons ainsi une classe plus perfectionnée.

Sans préciser de paramètres, les objets créés auront par défaut une couleur rouge et une autonomie de 10 heures.

Préciser un (ou des) paramètre(s) permettra alors de choisir d'autres options.

```
1  class Robot :
2      def __init__(self, couleur="rouge", autonomie=10):
3          self.couleur = couleur
4          self.autonomie = autonomie
5
6  r1=Robot()
7  print(r1.couleur, r1.autonomie)
8  r2=Robot("jaune")
9  print(r2.couleur, r2.autonomie)
10 r3=Robot("vert",20)
11 print(r3.couleur, r3.autonomie)
12 r4=Robot(autonomie=5)
13 print(r4.couleur, r4.autonomie)
```

```
rouge 10
jaune 10
vert 20
rouge 5
```

Méthode de classe ou méthode statique

```
class Robot:
    nb_robots = 0
    def __init__(self,couleur,auto):
        self.couleur = couleur
        self.autonomie = auto
        Robot.nb_robots += 1
    def parler(self):
        return "Bonjour!"
    def changer_couleur(self,new):
        self.couleur = new

    @classmethod
    def get_nb_robots(cls):
        return Robot.nb_robots

print(Robot.get_nb_robots())
r1=Robot("jaune",10)
print(Robot.get_nb_robots())
r2=Robot("rouge",20)
print(Robot.get_nb_robots())
r3=Robot("vert",15)
print(Robot.get_nb_robots())
```

```
0
1
2
3
```

```
class Robot:
    nb_robots = 0
    def __init__(self,couleur,auto):
        self.couleur = couleur
        self.autonomie = auto
        Robot.nb_robots += 1
    def parler(self):
        return "Bonjour!"
    def changer_couleur(self,new):
        self.couleur = new

    @staticmethod
    def get_nb_robots():
        return Robot.nb_robots

print(Robot.get_nb_robots())
r1=Robot("jaune",10)
print(Robot.get_nb_robots())
r2=Robot("rouge",20)
print(Robot.get_nb_robots())
r3=Robot("vert",15)
print(Robot.get_nb_robots())
```

```
0
1
2
3
```

Méthode spéciale __str__

Vous avez sûrement déjà pu constater que, quand on instancie des objets issus de nos propres classes, si on essaye de les afficher directement dans l'interpréteur ou grâce à print, on obtient quelque chose du style :

```
<__main__.Robot at 0x10a13a320>
```

On a certes les informations utiles, mais pas forcément celles que l'on veut.
Il existe une méthode spéciale __str__, spécialement utilisée pour afficher l'objet avec print.
La méthode __str__ est également appelée si vous désirez convertir votre objet en chaîne avec le constructeur str.

```
1  class Robot:
2      def __init__(self,couleur,auto):
3          self.couleur = couleur
4          self.autonomie = auto
5      def __str__(self):
6          return f"Robot de couleur {self.couleur} \
7  avec une autonomie de {self.autonomie} heures"
8
9  r1=Robot("jaune",10)
10 print(r1)
11 r1
```

```
Robot de couleur jaune avec une autonomie de 10 heures
```

```
<__main__.Robot at 0x112a81a90>
```

POO_Seance1_Exercices.ipynb