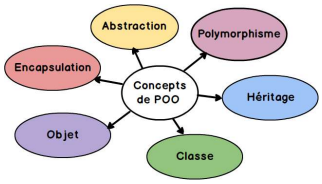
# La programmation orientée objet POO






# La programmation orientée objet - POO

## Le concept de la POO

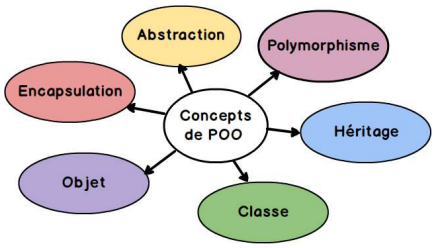
Avec Python tous les types qu'on utilise jusqu'à maintenant sont des **objets** déjà prédéfinit par Python, que ce soit les Entiers, Chaine de caractères, Listes, Tuples, Ensemble, Dictionnaires, …

Pour cette partie on va s'intéresser à créer nos propres **objets** en se basant sur **le paradigme de la POO**.

# La programmation orientée objet - POO

## Le concept de la POO

La **programmation orientée objet**, ou **POO**, est un **paradigme** de **programmation** qui permet de structurer les programmes de manière à ce que les **propriétés (attributs)** et les **comportements (méthodes)** soient regroupés dans des **objets** à part, et en respectant les concepts suivants.

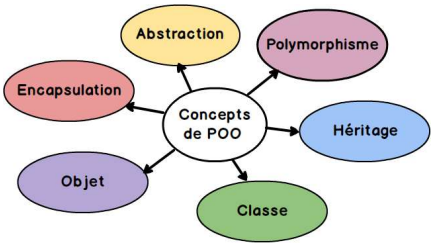





# La programmation orientée objet - POO

## Les principes de la POO

- **L'abstraction:** une vue simplifiée du système étudié, en focalisant sur une partie précise de ce système.
- **La classe:** Une classe est un prototype du système étudié, à partir duquel des **objets** sont créés. Il représente l'ensemble des **attributs** et **méthodes** communs à tous les objets d'un type.
- **L'objet:** une instanciation de la classe.
- **L'héritage:** est un moyen de former de nouvelles classes en utilisant des classes déjà définies.
- **L'encapsulation:** masque les détails d'implémentation d'une **classe** à d'autres **objets**.
- **Le polymorphisme:** est le processus d'utilisation d'un **attribut** ou d'une **méthode** de différentes manières pour différentes entrées de données.

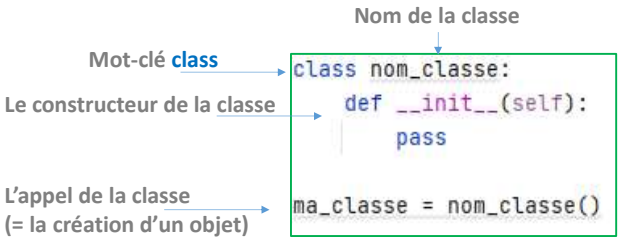


# La programmation orientée objet - POO

## Définition d'une classe - class

### Syntaxe et usage d'une classe

La syntaxe d'une classe se déclare à l'aide du mot clé **class** suivie du nom de la classe et puis le constructeur **__init__()**






# La programmation orientée objet - POO

## Les Classes

Pour créer une **classe** en Python, on utilise l'instruction :

```
1 class nom_classe :
```

On crée ensuite une **méthode** qui permet de **construire** les **objets**, appelé **constructeur** via l'instruction :

```
2     def __init__(self):
```

**Remarque**

- Le constructeur **__init__()** ne porte pas un **return,**
- Une classe peut avoir plusieurs **constructeurs**, mais seul le dernier qui est pris en considération.

On appelle une classe en utilisant son nom, et les paramètre de classe sont les paramètres de son constructeur [la méthode __init__()]. :

```
3 nom_classe()
```

# La programmation orientée objet - POO

## Les Classes

Exemple:

```
1  class voiture :
2      def __init__(self, marque, nbr_kilometre):
3          self.marque = marque
4          self.nbr_kilometre = nbr_kilometre
5
6
7  v1 = voiture("BW", 20500)
8  print("La marque de la voiture est: ", v1.marque)
9  print("Le nombre des kilométres de la voiture est: ", v1.nbr_kilometre)
```

Attributs

Résultat:

```
La marque de la voiture est:  BW
Le nombre des kilométres de la voiture est:  20500

Process finished with exit code 0
```



# La programmation orientée objet - POO

## Les méthodes

Une **méthode** est une **fonction** nommée au sein de la classe, permettant de définir des propriétés ou comportements des objets.

Exemple:

```
1  class voiture :
2      def __init__(self, marque, nbr_kilometre):
3          self.marque = marque
4          self.nbr_kilometre = nbr_kilometre
5
6      def marche(self, kilometre_parcourus):
7          return self.nbr_kilometre + kilometre_parcourus
8
9
10 v1 = voiture("BW", 20500)
11 print("La marque de la voiture est: ", v1.marque)
12 print("Le nombre des kilométres de la voiture est: ", v1.nbr_kilometre)
13 print("Le nombre des kilométres actuel de la voiture est:", v1.marche(200))
```

Attributs

Méthode

Résultat:

```
La marque de la voiture est:  BW
Le nombre des kilométres de la voiture est:  20500
Le nombre des kilométres actuel de la voiture est: 20700
```

# La programmation orientée objet - POO

## L'héritage de classe

Exemple:

L'héritage est un concept très utile. Cela permet de créer de nouvelles classes mais avec une base existante.

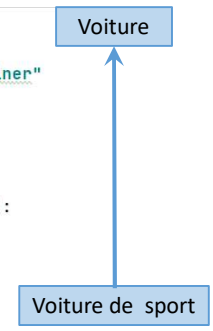
On a indiqué que **VoitureSport** a hérité de classe Voiture , elle récupère donc toutes ses **méthodes** et ses **attributs**

```python
class Voiture :
    def __init__(self):
        self.marque = "A déterminer"

    def get_marque(self):
        return self.marque

    def set_marque(self, marque):
        self.marque = marque


class VoitureSport(Voiture):
    pass
```






# La programmation orientée objet - POO

## L'héritage de classe

Exemple:

```python
class Voiture :
    def __init__(self):
        self.marque = "A déterminer"

    def get_marque(self):
        return self.marque

    def set_marque(self, marque):
        self.marque = marque

class VoitureSport(Voiture):
    pass

ma_voiture = VoitureSport()
print("La marque :", ma_voiture.get_marque())

ma_voiture.set_marque("Ferrari")
print("La nouvelle marque :", ma_voiture.get_marque())
```

Résultat:

```
La marque : A déterminer
La nouvelle marque : Ferrari

Process finished with exit code 0
```

# La programmation orientée objet - POO

## L'héritage multiple

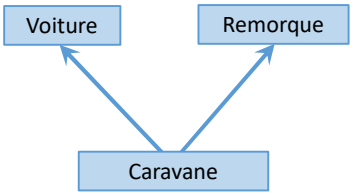
**L'héritage multiple** repose sur les **mêmes principes** que **l'héritage simple**. Cependant, au lieu de stipuler une seule classe mère, nous allons en indiquer **plusieurs**. Et, bien entendu, il faut initialiser chaque classe dont on hérite.

Exemple:

```python
class Voiture:
    def __init__(self):
        pass

class Remorque:
    def __init__(self):
        pass

class Caravane(Voiture,Remorque):
    def __init__(self):
        pass
```






# La programmation orientée objet - POO

## Polymorphisme / surcharge de méthodes

Comme nous l'avons vu si une **classe** hérite d'une autre classe, elle hérite les méthodes de la classe mère .

Exemple:

```python
class Voiture :
    def __init__(self):
        self.marque = "A déterminer"

    def get_marque(self):
        return self.marque

    def set_marque(self, marque):
        self.marque = marque

class VoitureSport(Voiture):
    def __init__(self):
        self.marque = "Ferrari"

ma_voiture = VoitureSport()
print("La marque :", ma_voiture.get_marque())
```

Résultat:

```
La marque : Ferrari

Process finished with exit code 0
```

Il est cependant possible de **redéfinir** la méthode de la classe parente en la redéfinissant. On parle alors de surcharger une méthode.

# La programmation orientée objet - POO

## Polymorphisme / surcharge de méthodes

Exemple:

```python
class Voiture :
    def __init__(self):
        self.marque = "A déterminer"

    def get_marque(self):
        return "Il faut déterminer la marque"

    def set_marque(self, marque):
        self.marque = marque

class VoitureSport(Voiture):
    def __init__(self, marque, roues):
        self.marque = marque
        self.roues = roues

    def get_marque(self):
        return self.marque

    def get_roues(self):
        return self.roues

ma_voiture = VoitureSport("Ferrari", 4)
print("La marque :", ma_voiture.get_marque())
print("Le nombre des roues :", ma_voiture.get_roues())
```

Résultat:

```
La marque : Ferrari
Le nombre des roues : 4

Process finished with exit code 0
```



# La programmation orientée objet - POO

## Polymorphisme / surcharge de méthodes

Exemple:

```python
class Voiture :
    def __init__(self, marque):
        self.marque = marque

    def get_marque(self):
        return self.marque

    def set_marque(self, marque):
        self.marque = marque


class VoitureSport(Voiture):
    def __init__(self, marque, vitesse):
        super().__init__(marque)
        self.vitesse = vitesse

    def get_marqueSport(self):
        return super().get_marque() + " Sport"

    def get_vitesse(self):
        return super().get_marque()

    def set_vitesse(self, vitesse):
        self.vitesse = vitesse

ma_voiture = VoitureSport('BMW', 230)
print("La marque est : ", ma_voiture.get_marque())
print("La marque est : ", ma_voiture.get_marqueSport())
print("La vitesse est : ", ma_voiture.get_vitesse())
```

La fonction super() accède aux méthodes héritées surchargées dans une classe. La fonction super() est utilisée dans la classe fille avec héritage simple et multiple pour accéder à la fonction de la classe parent ou superclasse suivante.

Résultat:

```
La marque est :  BMW
La marque est :  BMW Sport
La vitesse est :  BMW
```

# La programmation orientée objet - POO

## L'encapsulation

L'encapsulation est un concept très utile. Il permet en particulier d'éviter une modification par erreur des données d'un objet. En effet, il n'est alors pas possible d'agir directement sur les données d'un objet ; il est nécessaire de passer par ses méthodes qui jouent le rôle d'interface obligatoire.

Attributes / Methods
1) Public
2) _Protected
3) __Private

| | Intérieur de la même classe | Extérieur de la classe | Depuis l'héritage de la classe |
|---|---|---|---|
| **Public** | Oui | Oui | Oui |
| **_Protected** | Oui | Non | Oui |
| **__Private** | Oui | Non | Non |



# La programmation orientée objet - POO

## L'encapsulation

Exemple:

```
1  class Voiture:
2      def __init__(self, nbr_roues):
3          self.__roues = nbr_roues
4
5  ma_voiture = Voiture(4)
6  print("Le nombre des Roues de la voiture est :", ma_voiture.__roues)
```

Résultat:

```
    print("Le nombre des Roues de la voiture est :", ma_voiture.__roues)
AttributeError: 'Voiture' object has no attribute '__roues'

Process finished with exit code 1
```

# La programmation orientée objet - POO

## L'encapsulation et les méthodes : Getter et Setter

Quelque soit le langage, pour la **programmation orientée objet** il est de préférable de passer par des **méthodes** pour récupérer ou changer les valeurs des **attributs**.

Avec l'encapsulation, on exige le passer par les méthodes de type **getter** (ou **accesseur** en francais) et **setter** ( **mutateurs** ) pour récupérer et changer la valeur d'un attribut.

=> Cela permet de garder une cohérence pour le programmeur, Il permet en particulier d'éviter une modification par erreur des données d'un objet. En effet, il n'est alors pas possible d'agir directement sur les données d'un objet ; il est nécessaire de passer par les getter et les setter qui jouent le rôle d'interface obligatoire.



# La programmation orientée objet - POO

## Les méthodes : Getter et Setter

Exemple:

```python
class Voiture(object):
    def __init__(self, nbr_roues):
        self.__roues = nbr_roues

    def get_roues(self):
        return self.__roues

    def set_roues(self, roues):
        self.__roues = roues

ma_voiture = Voiture(4)
print("Le nombre des Roues de la voiture est :", ma_voiture.get_roues())
ma_voiture.set_roues(6)
print("Le nombre des Roues de la voiture est devenu :", ma_voiture.get_roues())
```

Résultat:

```
Le nombre des Roues de la voiture est : 4
Le nombre des Roues de la voiture est devenu : 6
```