



# Programmation Orientée Objet

Nassib ABDALLAH

# Plan du cours :

- **Introduction :**
  - Programme ? Famille de langages ?
  - Fonctions ? Programmation Orientée Objet ?
- **Classes & Objets.**
  - Constructeur.
  - Accès aux attributs / méthodes ?
- **Concepts de base de la POO**
  - Encapsulation
  - Héritage
  - Polymorphisme
- **Projet**

# Programmation Informatique :

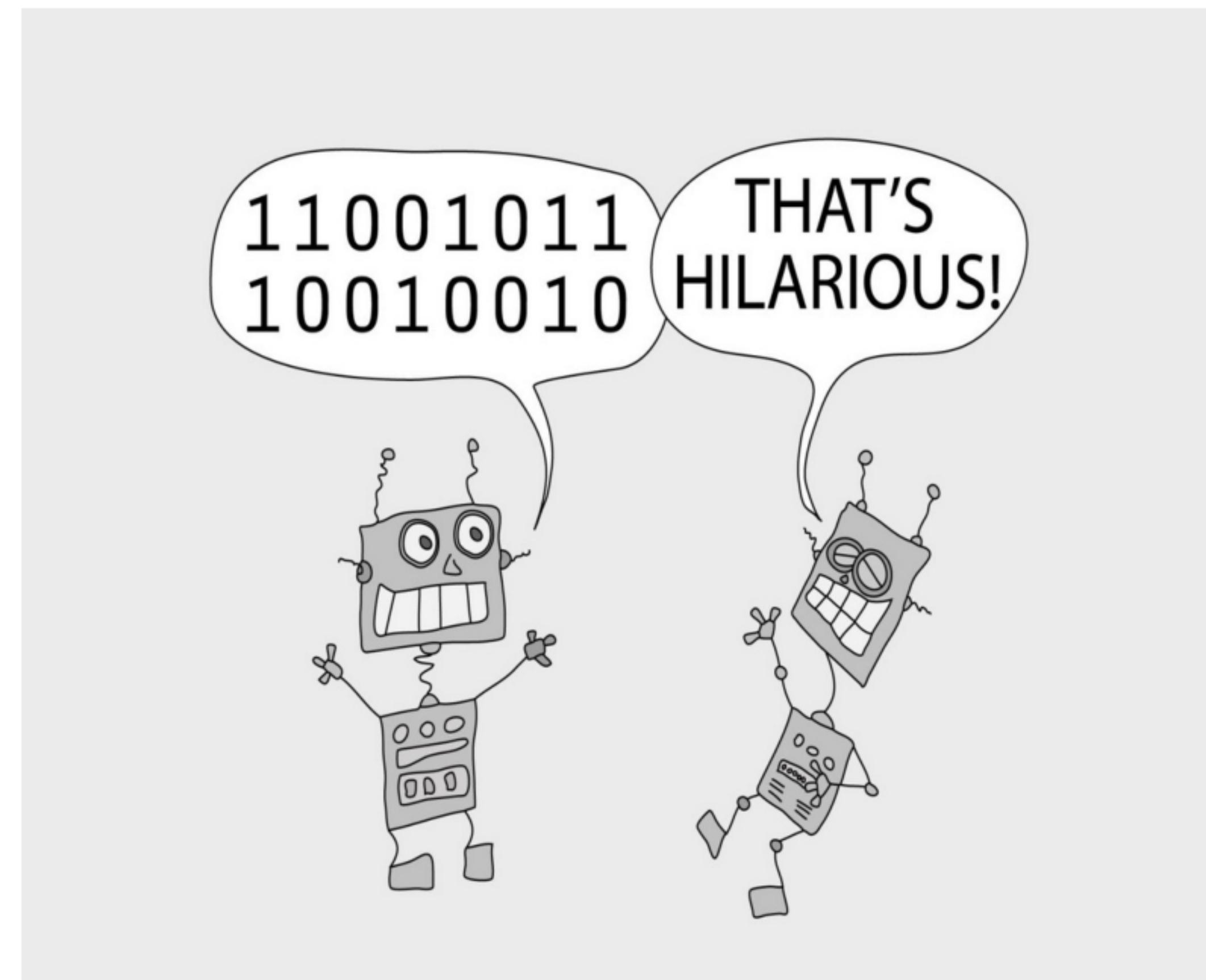
- **Algorithme :**
  - est un processus ou un ensemble de règles à suivre dans les calculs ou autres opérations permettant de résoudre un problème, en particulier par un ordinateur.
- **Programme informatique :**
  - est un ensemble d'instructions codé par un langage de programmation spécifique et qui peuvent être exécutées par un ordinateur pour effectuer une tâche spécifique.

# Langages de Programmation:

- **Langages de programmation:**
  - Les langages de programmation permettent aux programmeurs de développer des logiciels.
- **Familles de langages :**
  - Les langages machine
  - Les langages assembleurs
  - Les langages de haut niveau

# Langages machine

- **Il est composé d'instructions et de données à traiter codées en binaire.**
  - Comprend des 1 et des 0.
- **Le langage "natif" d'un ordinateur (processeur).**
- **Difficile à programmer - un 1 ou un 0 mal placé entraînera l'échec du programme.**
- **Le code machine est généré automatiquement.**
  - généralement par le compilateur d'un langage de programmation ou par l'intermédiaire d'un bytecode.



# Langages assembleurs

- **Les langages assembleurs sont un pas vers une programmation plus facile.**
- **Comment ?**
  - sont constitués d'un ensemble de commandes élémentaires liées à un processeur spécifique.

`MOV AX,1` – déplacer des données d'un endroit à un autre

- **Le langage d'assemblage est utilisé principalement pour la manipulation directe du hardware.**

**Note : Le code du langage assembleur doit être traduit en langage machine avant que l'ordinateur ne le traite.**

# Langages de haut niveau

- **Les langages de haut niveau représentent un pas de géant vers une programmation plus facile.**
- **La syntaxe est similaire à celle de l'anglais.**

**Exemple :**

```
if x>10 :  
    a=100
```

- **Interpréteur/Compilateur :**
  - Interpréteur - joue le rôle d'interface entre le code source et le processeur pendant l'exécution.
  - Compilateur - traduit l'ensemble du code source d'un projet logiciel en code machine avant son exécution.
- **Les langages haut niveau se divisent en deux groupes :**
  - Langage procédural.
  - Langages orientés objet (POO).

# Petit rappel sur les fonctions...

- **Qu'est-ce qu'une fonction en Python ?**
  - ☰ En Python, une fonction est un groupe d'instructions qui accomplit une tâche spécifique.
  - ☰ Une fonction est un bloc de code qui ne fonctionne que lorsqu'il est appelé.
- **Pourquoi ?**
  - ☰ Les fonctions aident à diviser notre programme en morceaux plus petits. Au fur et à mesure que notre programme s'agrandit, les fonctions le rendent plus organisé et plus facile à gérer.
  - ☰ Elles évitent les répétitions et rendent le code réutilisable.

# Comment définir une fonction

## Syntaxe de la fonction :

1. Le mot-clé **def** : le début de l'en-tête de la fonction.
2. Un **nom de fonction** pour identifier la fonction de manière unique.
3. Les **paramètres** (arguments) par lesquels nous passons des valeurs à une fonction. Ils sont facultatifs.
4. **Deux points** ( :) pour marquer la fin de l'en-tête de la fonction.
5. Une ou plusieurs **instructions** python valides qui composent le corps de la fonction. Les instructions doivent avoir le même niveau d'indentation.
6. Une instruction de retour est optionnelle. Le mot-clé **return** est utilisé pour quitter une fonction et pour retourner une valeur de la fonction.

```
def funct_nom(params):  
    """docstring"""  
    statement(s)
```

# Fonction vs Procédure

## Sans *return* :

```
def greet(name) :  
    """
```

Cette fonction salue la personne qui est passée en tant que un paramètre

```
    """  
  
    print("Bonjour, " + name + "!")
```

- **Console :**

- **>>> greet('Paul')**
- **>>> v = greet('Paul')**
- **>>> print(v)**

- **Sortie :**

- **Bonjour, Paul!**
- **None**

## Avec *return* :

```
def greet(name) :  
    """
```

Cette fonction salue la personne qui est passée en tant que un paramètre

```
    """  
  
    return "Bonjour, " + name + "!"
```

- **Console :**

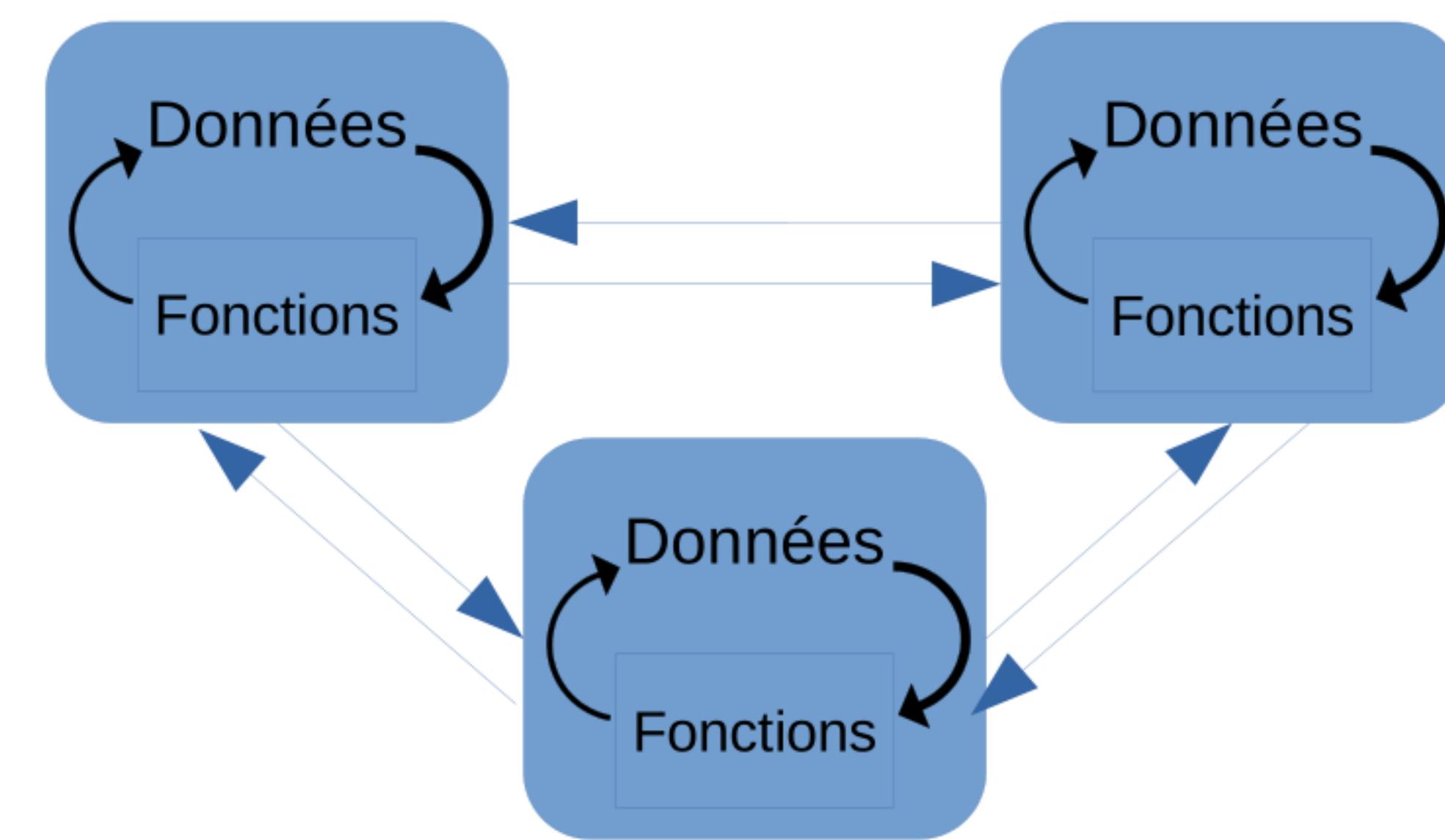
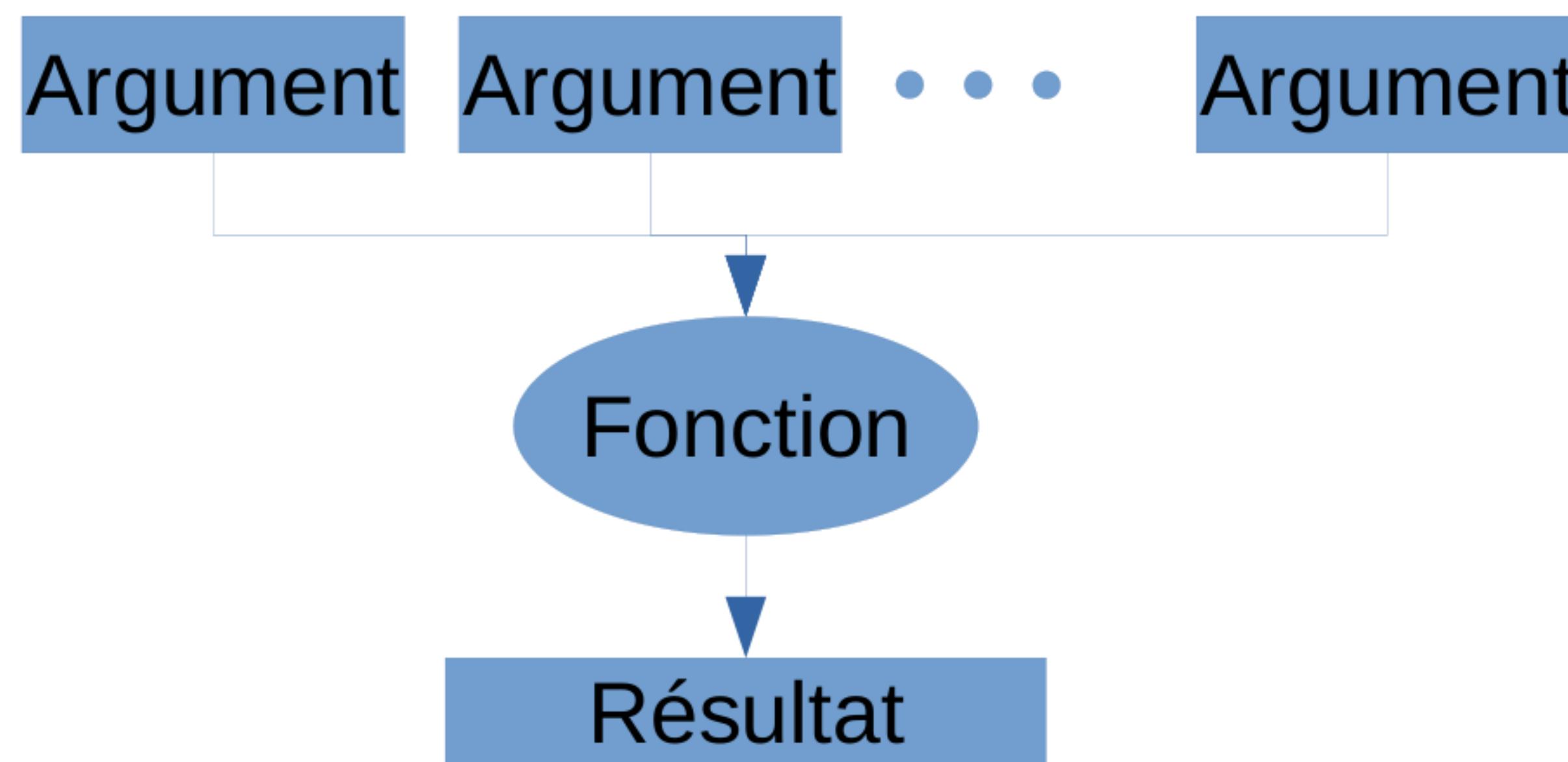
- **>>> greet('Paul')**
- **>>> v = greet('Paul')**
- **>>> print(v)**

- **Sortie :**

- **Bonjour, Paul!**
- **Bonjour, Paul!**

# Différence entre la programmation structurelle et la POO ?

- **Différence entre la programmation structurelle et la POO ?**
  - Programmation fonctionnelle : les arguments d'une fonction peuvent provenir de résultats d'autres fonctions.
  - Programmation orientée objet : les objets contiennent leurs propres fonctions (méthodes) et décrivent les entrées/sorties des programmes.



# Qu'est-ce que la programmation orientée objet en Python.

## • Qu'est-ce que la POO ?

- La programmation orientée objet (POO) est un concept dont les éléments clés sont les **objets** et les **classes**.
- Contrairement au procédural, l'accent de la POO n'est pas mis sur la structure, mais sur la modélisation des données.

Idée clé de la POO :

*La programmation orientée objet permet de modéliser le code sous forme d'objets ayant des propriétés et des méthodes qui interagissent entre eux.*

# Qu'est-ce que la programmation orientée objet en Python.

## • Qu'est-ce que la POO ?

- La programmation orientée objet (POO) est un concept dont les éléments clés sont les **objets** et les **classes**.
- Contrairement au procédural, l'accent de la POO n'est pas mis sur la structure, mais sur la modélisation des données.

Idée clé de la POO :

*La programmation orientée objet permet de modéliser le code sous forme d'objets ayant des propriétés et des méthodes qui interagissent entre eux.*

# Définition d'une classe

- **Une classe est un type de données spécial qui définit comment construire un certain type d'objet.**
  - "modèle" pour créer des objets.
- **La classe contient un ensemble de caractéristiques structurelles (attributs) et un ensemble de caractéristiques comportementales (méthodes).**
- **Programmer en approche objet consiste principalement à créer (ou réutiliser) des classes et les lier entre elles.**

Voiture
PrixHT
Modèle
nb_porte
Construire(val)
prixTTC()

# Exemple de la classe Voiture en Python

```
class Voiture:  
    # Constructeur  
    def __init__(self,m,p,n=5):  
        self.modele = m  
        # définition et initialisation  
        # de trois attributs  
        self.prixHT = p  
        self.quantite = n  
    # définition de trois méthodes  
    def ajouter(self,q):  
        self.quantite = self.quantite + q  
    def retirer(self,q):  
        self.quantite = self.quantite - q
```

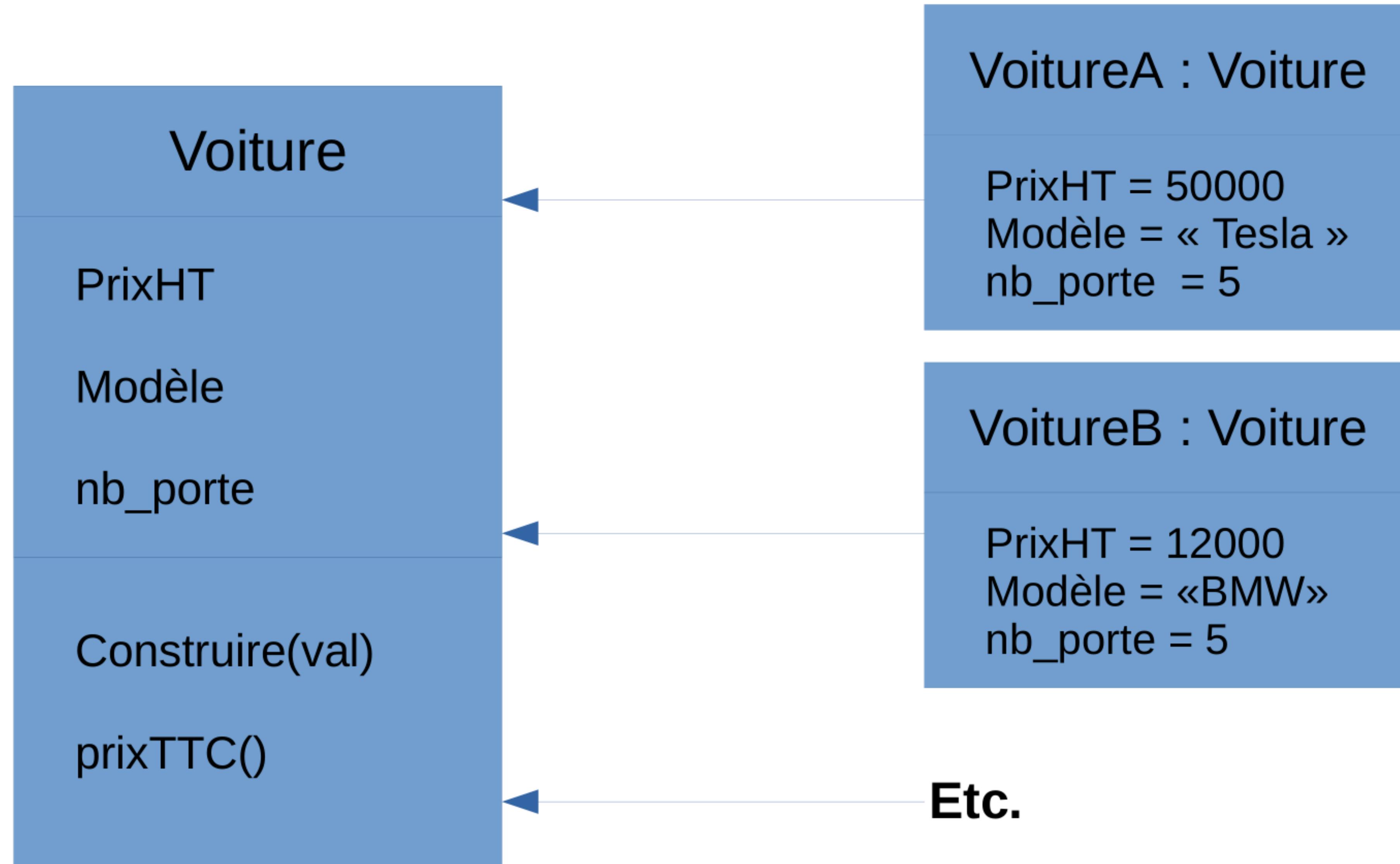
- On constate que les attributs et les méthodes sont tous situés à l'intérieur de la classe (cf indentation)
- On constate également que les attributs sont introduits par le biais d'une méthode spécifique `__init__` que l'on appelle un constructeur et que l'on précisera plus tard.

# Définition d'un objet (1/2)

- **Qu'est-ce qu'un objet ?**

- A partir d'une classe on peut instancier des objets caractérisés par **les valeurs de leurs attributs.**
- Par exemple, s'il n'existe qu'une classe Voiture, il existe des centaines d'objets «Voiture». Tesla, Renault, Peugeot, BMW.
- En réalité, **une classe permet de décrire un ensemble d'objets** qui partagent une structure (ensemble d'attributs) et un comportement (ensemble de méthodes) communs.

# Exemple :



# Définition d'un objet (2/2)

- **Un objet a un état, un comportement et une identité.**
- **L'état d'un objet englobe tous les attributs de l'objet (structure), ainsi que les valeurs associées à ces propriétés.**

VoitureC : Voiture

PrixHT = 18000

Modèle = «Renault»

NB : généralement, les attributs n'évoluent pas au cours du programme, au contraire des valeurs dont la vocation est de changer pendant le programme.

# Comportement d'un objet.

- Le comportement d'un objet est la façon dont il agit et réagit en terme de circulation de messages (ce comportement est caractérisé par les méthodes rattachées à chaque objet).
  - Un message n'est donc rien d'autre qu'un destinataire (un objet), une méthode (que le destinataire doit exécuter donc doit posséder) et les paramètres associés à cette méthode.

**La différence entre un envoi de message et un appel de fonction classique est la présence systématique d'un destinataire.**

# Identité d'un objet

- L'identité est la propriété d'un objet qui le distingue de tous les autres objets, indépendamment de son état (un objet est unique)

VoitureA : Voiture

PrixHT = 50000

Modèle = « Tesla »

VoitureB : Voiture

PrixHT = 12000

Modèle = «BMW»

Ci-dessus deux objets différents

# Opérations sur les objets

- **On distingue généralement quatre opérations de base sur les objets:**
  - Modification : altère l'état d'un objet (écriture).
  - Sélection : accède à l'état d'un objet (lecture).
  - Création : crée un objet.
  - Destruction : détruit un objet.
- **Exemple : le comportement de VoitureA peut être « Construire(3) » qui modifie son état (son attribut nb\_porte), ou peut être «prixTTC()» qui accède à son état (son attribut prixHT)-**

# Retour sur la classe Voiture en Python

```
class Voiture:  
    def __init__(self,m,p,n=5):  
        self.modele = m  
        self.prixHT = p  
        self.quantite = n  
  
    def ajouter(self,q):  
        self.quantite = self.quantite + q  
  
    def retirer(self,q):  
        self.quantite = self.quantite - q
```

# Création d'objet.

**Par l'intermédiaire du constructeur ( la méthode `__init__` appelée par le biais du nom de la classe.**

```
nom_objet = nom_classe()
```

```
### Créer plusieurs objets Voiture
```

```
BMW = Voiture('BMW',12000, 5)  
Renault = Voiture('Renault',15000,5)  
Tesla = Voiture('Tesla',40000,5)  
>>> BMW
```

```
Out : <class '__main__.Voiture'>
```

La variable BMW contient la référence désignant l'emplacement mémoire de l'objet créé (qu'on appelle aussi instance) de la classe Voiture.

**Contrairement aux fonctions, lorsqu'une classe est appelée, elle crée un objet au lieu d'exécuter du code.**

# Constructeur. Commençons à construire !

**Le constructeur Python est une fonction spéciale qui est appelée lorsque nous créons l'instance de la classe.**

**La tâche des constructeurs consiste à initialiser (attribuer des valeurs) aux attributs de la classe lorsqu'un objet de la classe est créé.**

**Le nom de cette fonction est toujours `_init_` et elle peut avoir un nombre quelconque de paramètres. Le premier paramètre est toujours l'objet lui-même, généralement nommé comme variable `self`.**

**En Python, les méthodes, comme `_init_()`, avec deux tirets bas appartiennent à un groupe spécial de méthodes appelées méthodes de surcharge ou méthodes magiques.**

Nous n'avons pas besoin d'appeler ces méthodes explicitement, elles sont appelées automatiquement lorsqu'un objet participe à une action.

# Syntaxe de la méthode `__init__`

- **la syntaxe de la fonction `__init__` est la suivante :**

```
def __init__(self, params)
```

- Le mot-clé **def** est utilisé pour le définir car il s'agit d'une fonction.
- Le premier argument fait référence à l'objet courant.
  - Il lie l'instance à la méthode `init()`.
  - Il est généralement nommé "**self**" pour suivre le nom de l'objet.
- Les arguments de la méthode `init()` sont facultatifs. Nous pouvons définir un constructeur avec un nombre quelconque d'arguments.

# Autoréférence (le mot réservé *self*)

- On peut accéder de l'intérieur d'une méthode à son objet d'appel par le mot réservé *self* qui doit toujours apparaître comme premier paramètre de la méthode.
- Exemple :

```
def Ajouter(self, x) :  
    self.quantite= self.quantite + x
```

Il s'agit ici de l'attribut **quantite** de l'objet d'appel (par exemple l'objet *Tesla*) auquel on soustrait la valeur x donnée en argument (par exemple 50).

# **Différent type de constructeur d'objets.**

## **Différent scénario pour un constructeur d'objets.**

1. Classe sans constructeur.
2. Constructeur simple sans arguments.
3. Constructeur de classe avec des arguments.

# Classe sans constructeur.

- Nous pouvons créer une classe sans aucune définition de constructeur.
- Dans ce cas, le constructeur de la superclasse est appelé pour initialiser l'instance de la classe.
  - La classe "**object**" est la base de toutes les classes en Python.

```
### un personnage de jeu de classe vide.  
## Le nom de la classe doit commencer par une lettre majuscule !  
class GameCharacter :  
    pass  
  
## IronMan est un objet de la classe GameCharacter  
IronMan = GameCharacter()  
>>> IronMan  
Out : <class '__main__.GameCharacter'>
```

# Exemple de la class *Data1* avec un constructeur sans argument.

- Nous pouvons créer un constructeur sans aucun argument.

- C'est utile à des buts de collecte de données, comme le comptage des instances de la classe.

```
class Data1:  
    count = 0  
  
    def __init__(self):      # constructeur  
        print('Data1 Constructor')  
        Data1.count += 1  
  
d1 = Data1()  
d2 = Data1()  
print("Data1 Object Count =", Data1.count)  
  
Output :  
Data1 Constructor  
Data1 Constructor  
Data1 Object Count = 2
```

# Constructeur de classe avec des arguments.

- En général, le constructeur prend des arguments.

- Ces arguments sont généralement utilisés pour initialiser les variables d'instance.

```
class GameCharacter :  
    # le constructeur de la classe.  
    # un attribut : nom  
    def __init__(self, nom) :  
        self.nom = nom  
  
    # personnages principaux du jeu avec des  
    # attributs uniques  
thor = GameCharacter(name = 'thor')  
loki = GameCharacter(name = "loki")  
  
print("GameChar's Name : ", thor.name)  
print("GameChar's Name : ", loki.name)  
  
Output :  
GameChar's Name : thor  
GameChar's Name : loki
```

# Accès aux attributs et méthodes.

- **Par l'intermédiaire de l'opérateur « . » associé au nom de l'objet :**

```
BMW.prixHT = 22000 # modifier le prix de l'objet BMW
```

```
BMW.getPrixTTC() # pour appeler la fonction getPrixTTC sur  
l'objet BMW.
```

# Revenons à l'exemple de la classe Voiture en Python

```
class Voiture:  
    def __init__(self,m,p,n=5):  
        self.modele = m  
        self.prixHT = p  
        self.quantite = n  
  
    def ajouter(self,q):  
        self.quantite = self.quantite + q  
  
    def retirer(self,q):  
        self.quantite = self.quantite - q
```

Console :

```
>>> BMW = Voiture('BMW',  
12000.99 , 5)  
  
>>> print(BMW.quantite)  
## OUT : 5  
>>> BMW.retirer(1)  
>>> print(BMW.quantite)  
## OUT : 4  
>>> BMW.ajouter(5)  
>>> print(BMW.quantite)  
## OUT 9
```

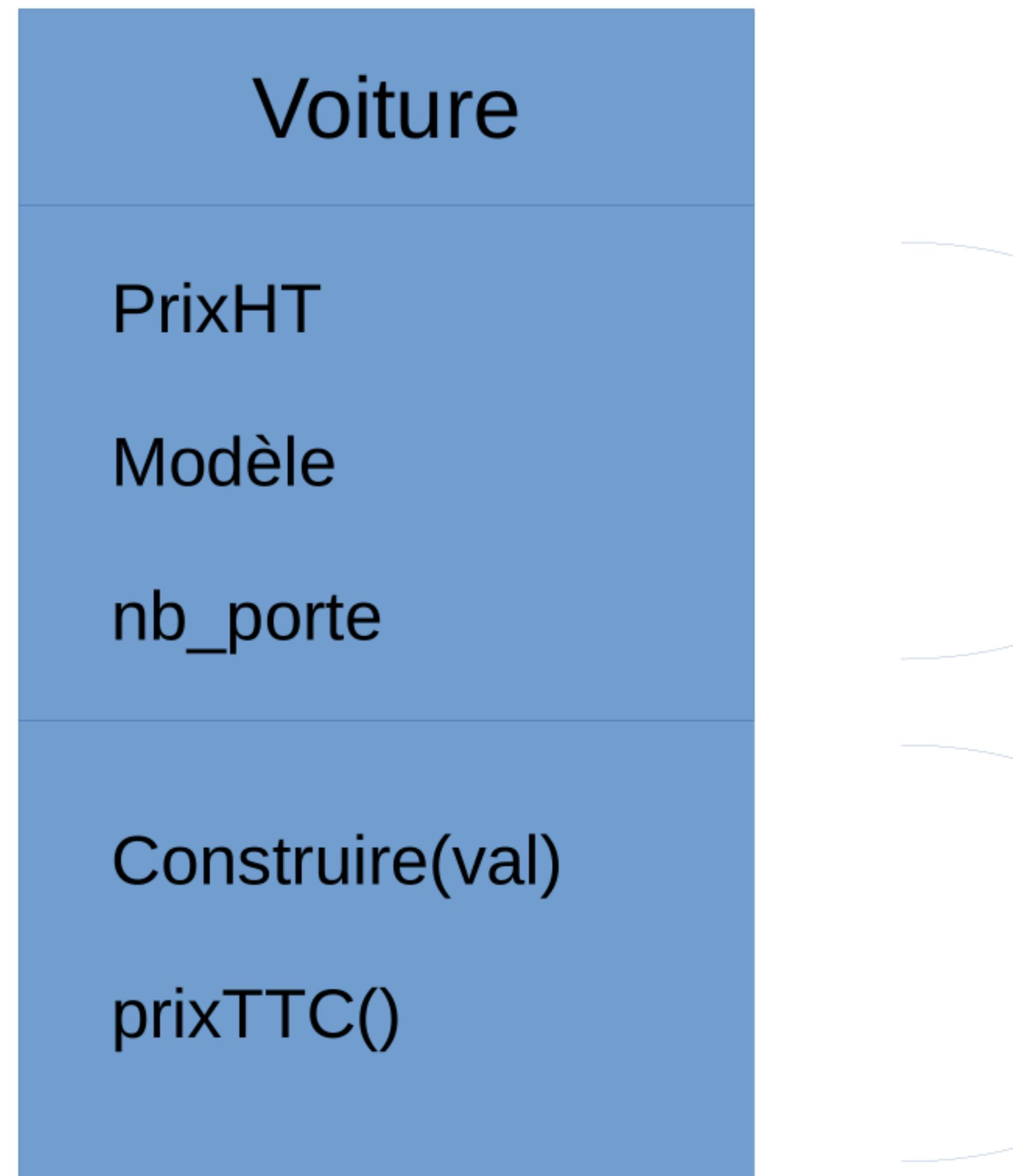
# Encapsulation

- En approche objet, en général les attributs ne sont pas accessibles (private) de l'extérieur de la classe à laquelle ils appartiennent; seules les méthodes principales sont accessibles (public) offrant ainsi une interface claire et documentée pour les utilisateurs de cette classe.
- Les données restent ainsi « cachées » à l'intérieur de la classe et l'utilisateur se focalise sur les fonctionnalités de la classe (par exemple il peut utiliser une fonction de tri sans savoir si elle est implémentée par un tableau, une liste ou un arbre) : ce principe s'appelle l'encapsulation des données.

# Précisions sur les utilisateurs (ou clients) d'une classe.

- **On distingue généralement la notion de programmeur d'une classe (celui qui la définit en terme d'attributs et de méthodes) de la notion d'utilisateur (ou client) d'une classe (celui qui s'en sert en terme de fonctionnalités à appliquer sur un objet).**
- **Le programmeur d'une classe doit connaître la structure des données de la classe pour implémenter ses algorithmes (par le biais de méthodes).**
- **L'utilisateur d'une classe (ou client) ne s'intéresse qu'à son interface et n'a rien à faire de la structure des données associée ...**

# Exemple



**Données encapsulées** inaccessibles aux utilisateurs de la classe.

**Interface claire et documentée** accessible aux utilisateurs de la classe.

# Commentaires par rapport à Python

- **Contrairement à d'autres langages, python ne contrôle pas les droits d'accès.**
  - Un attribut ne peut pas être déclaré private...
- **La philosophie de Python est de faire confiance au programmeur plutôt que de lui imposer des contraintes ...**
- **Néanmoins dans nos programmes nous nous fixerons comme règle de toujours passer par l'interface (i.e les méthodes) pour accéder aux données ou les modifier de l'extérieur de la classe.**

# Méthodes de contrôle de l'accès.

- **Python propose plusieurs méthodes pour limiter l'accès aux variables et aux méthodes dans le programme :**
  - Utilisation d'un seul tiret bas.
  - Utilisation du double tiret bas.
  - Utilisation des méthodes accès et de modification pour accéder aux variables privées.

# Utilisation d'un souligné « \_ »

- Une règle de programmation Python courante pour identifier une variable privée consiste à la préfixer par un souligné ‘\_’.

```
class Person:  
  
    def __init__(self, name, age=0):  
        self.name = name  
        self._age = age  
  
    def display(self):  
        print(self.name)  
        print(self._age)
```

```
person = Person('Dev', 30)  
  
#accessing using class method  
person.display()  
  
#accessing directly from outside  
print(person.name)  
print(person._age)
```

Output :

Dev  
30

Dev  
30

# Utilisation de deux soulignés « \_\_ »

- L'utilisation de deux soulignés comme préfixe d'un nom d'attribut (ou de méthode) permet de définir cet attribut **private** et donc non accessible par les utilisateurs de la classe.

```
class Person:  
    def __init__(self, name, age=0):  
        self.name = name  
        self.__age = age  
  
    def display(self):  
        print(self.name)  
        print(self.__age)
```

```
person = Person('Dev', 30)  
#accessing using class method  
person.display()  
#accessing directly from outside  
print('Trying to access  
variables from outside the class  
'')  
print(person.name)  
print(person.__age)◀
```

**Output :**  
Dev  
30  
Trying to access variables from outside  
the class  
Dev  
Traceback (most recent call last):  
 File "<ipython-input-20-  
8e693167d041>", line 17, in <module>  
 print(person.\_\_age)  
AttributeError: 'Person' object has no  
attribute '\_\_age'

# Méthode d'accès et de modification

**Pour accéder aux variables privées et les modifier, il est recommandé d'utiliser les méthodes accesseurs (getter) et mutateurs (setter)**

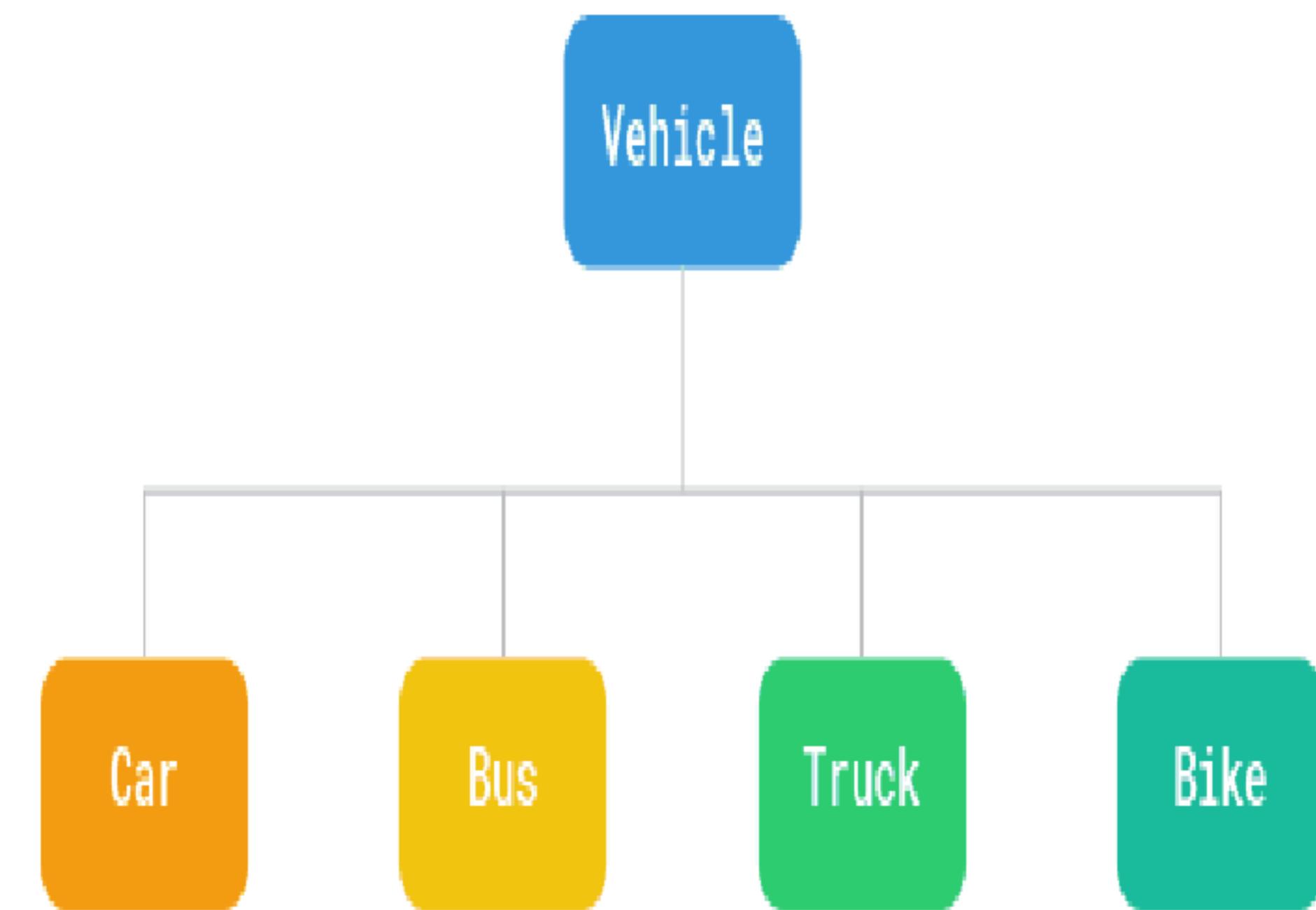
```
class Person:  
    def __init__(self, name, age=0):  
        self.name = name  
        self.__age = age  
    def display(self):  
        print(self.name)  
        print(self.__age)  
    def getAge(self):  
        print(self.__age)  
    def setAge(self, age):  
        self.__age = age
```

```
person = Person('Dev', 30)  
#accessing using class method  
person.display()  
#changing age using setter  
person.setAge(35)  
person.getAge()
```

Output :  
Dev  
30  
35

# Héritage (1/2)

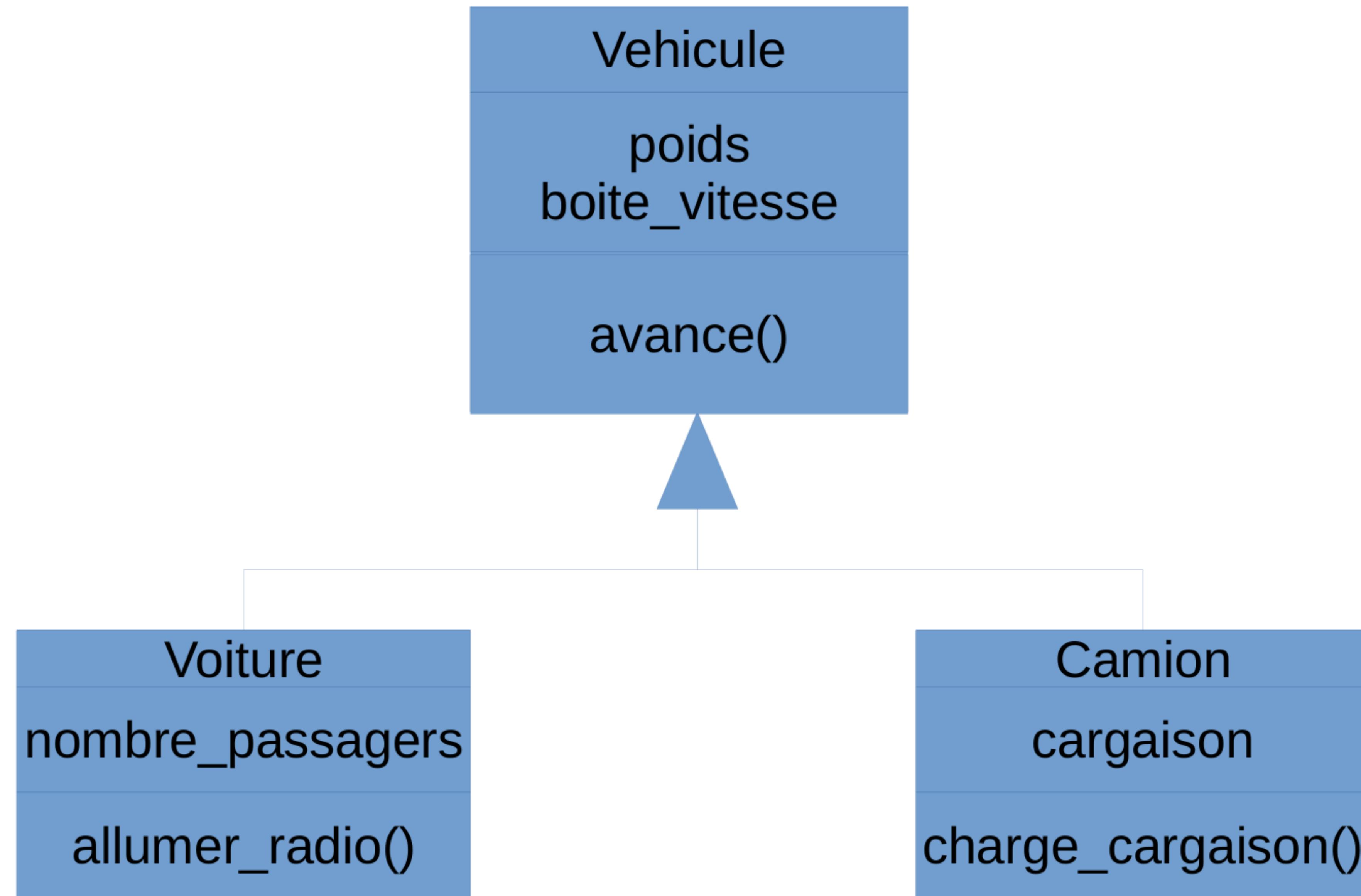
- L'héritage est considéré comme l'un des aspects les plus importants de la POO car il permet la réutilisation des modèles, ce qui rend le code plus fiable et assure la flexibilité et la cohérence du code...
- Dans le monde de la programmation orientée objet (POO), l'héritage fait référence au mécanisme de la capacité d'une classe à dériver ou à étendre les propriétés d'une autre classe.



## Héritage (2/2)

- **Une sous classe bénéficie alors de toutes les caractéristiques de la classe dont elle dépend (on dit qu'elle hérite de ses ancêtres).**
- **De plus, la sous-classe ajoute ses caractéristiques propres car elle est plus spécifique que ses ancêtres.**

# Exemple d'héritage



# Exemple d'héritage de Véhicule en Python

```
class Vehicule:  
    def __init__(self,p,b):  
        self.poids = p  
        self.boite_vitesse = b  
  
    def avance(self):  
        print("la voiture avance...")
```

```
class Voiture(Vehicule):  
    def __init__(self,p,b,n):  
        Vehicule.__init__(self,p,b)  
        self.nombre_passager = n  
  
    def allumer_radio(self):  
        print("la radio est allumée...")
```

```
class Camion(Vehicule):  
    def __init__(self,p,b,c):  
        Vehicule.__init__(self,p,b)  
        self.cargaison = c  
  
    def charge_cargaison(self):  
        print("on charge la cargaison...")
```

# Exemple de sous\_classe en Python

```
# définir la nom de la sous-classe
# la classe mère est donnée entre parenthèses
class Voiture(Vehicule):

    # définir le constructeur de la sous_classe
    def __init__(self,p,b,n):
        # appel au constructeur de la classe mère (avec self)
        Vehicule.__init__(self,p,b)
        # définition et initialisation de l'attribut spécifique
        self.nombre_passager = n

    def allumer_radio(self):
        <code pour allumer la radio> # méthode spécifique
```

# Commentaires

- **Le constructeur de la sous-classe doit prendre en charge l'intégralité de la construction de l'objet.**
- **Ainsi, le constructeur de Voiture doit :**
  - Appeler le constructeur de Vehicule pour initialiser les attributs poids et boite\_vitesse.
  - Initialiser l'attribut nombre\_passagers.

# Polymorphisme

- **Le polymorphisme signifie avoir des formes différentes.**
  - **En POO, le polymorphisme fait référence à la capacité d'une fonction portant le même nom à avoir des fonctionnalités différentes.**
- Étonnamment, mais nous avons déjà vu le polymorphisme en action (par exemple la méthode `__init__()`).**
- **Le phénomène de ré-implémentation d'une fonction dans la classe dérivée est connu sous le nom de "méthode de surcharge" (Method Overriding).**

# Exemple de polymorphisme.

```
class Animal:  
    def type(self):  
        print("Various types of  
animals")  
  
    def age(self):  
        print("Age of the animal.")  
  
class Rabbit(Animal):  
    def age(self):  
        print("Age of rabbit.")  
  
class Horse(Animal):  
    def age(self):  
        print("Age of horse.")
```

Console :

```
>>> obj_animal = Animal()  
>>> obj_rabbit = Rabbit()  
>>> obj_horse = Horse()  
  
>>> obj_animal.type()  
>>> obj_animal.age()  
  
>>> obj_rabbit.type()  
>>> obj_rabbit.age()  
  
>>> obj_horse.type()  
>>> obj_horse.age()
```

Output :

```
Various types of animals  
Age of the animal.  
  
Various types of animals  
Age of rabbit.  
  
Various types of animals  
Age of horse.
```

# Projet

- **Il s'agit d'un projet noté qui vise à vous faire utiliser les connaissances que vous avez acquises pour pratiquer et maîtriser les fonctionnalités python dont vous aurez besoin pour le suite de votre parcours.**
- **Ce projet comporte quatre parties principales :**
  - } la première consiste à refaire les exemples qu'on a vu en cours et commenter les résultats.
  - } La deuxième partie consiste à coder un jeu.
  - } La troisième partie consiste à utiliser l'héritage, l'encapsulation et le polymorphisme.
  - } Enfin, la quatrième partie consiste à combiner les trois parties ( importer les modules qui correspondent à chaque partie) et appeler les fonctions dans un seul fichier principal qui permettra à l'utilisateur d'utiliser votre programme.

**Il est obligatoire de commenter votre code et d'expliquer en détail ce que vous faites dans chacune de vos fonctions ou scripts !**

- Anaconda :  
<https://www.anaconda.com/products/individual#Downloads>
- Lien du cours+projet :  
<https://github.com/nassib/m1-2020-2021>
- E-mail:
  - } [Nassib.abdallah@univ-angers.fr](mailto:Nassib.abdallah@univ-angers.fr)

# À vos marques, prêt, partez !

