# POO en Python

## Classe Object
## Méthodes spéciales et surcharge des opérateurs
## Encapsulation

Partie 3/3

Anis SAIED

IPEIN

2025/26

# Introduction

**Rappel:**

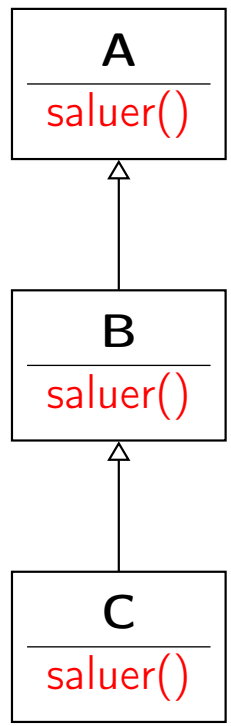
Définir une sous-classe: `class Enfant(Parent)`

`super()` fait référence à la classe parent la plus proche.

Le classe Enfant peut rédéfinir une méthode de la classe Parent.

**Exemple:**

```
class A:
    def saluer(self):
        print("Bonjour de A")
class B(A):
    def saluer(self):
        print("Bonjour de B")
        super().saluer()
class C(B):
    def saluer(self):
        print("Bonjour de C")
        super().saluer()
c = C()
c.saluer()
```



# Ordre de recherche des méthodes

• L'attribut **__mro__ (Method Resolution Order)** :
Affiche l'ordre dans lequel Python recherche les méthodes d'une classe lors de l'appel d'une méthode.

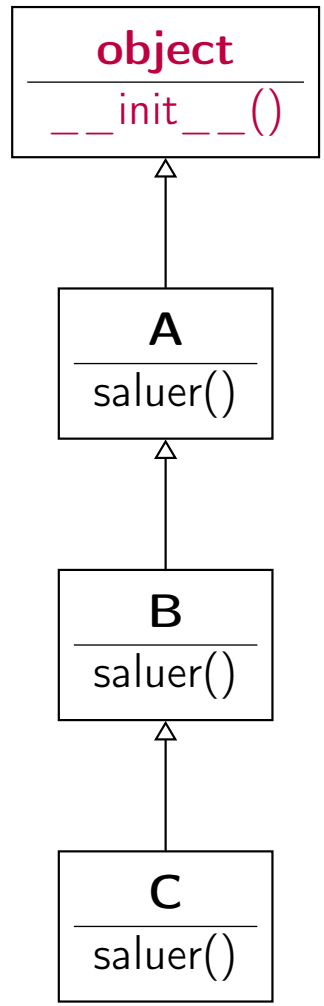
```
c = C()
c.saluer()
print(C.__mro__)
```

Affiche:

```
(<class 'C'>, <class 'B'>, <class 'A'>, <class 'object'>)
```

Python commence par chercher l'attribut/méthode dans la classe C, puis si elle n'existe pas, il le cherche dans la classe B, et ainsi de suite jsuqu'à arriver à la classe de base object.

→ Toute classe en Python hérite directement ou indirectement la classe `object`.



# La classe object

• L'attribut **__bases__** retourne un tuple contenant les classes de base directes d'une classe donnée.

```
print(B.__bases__)# (<class 'A'>,)
```

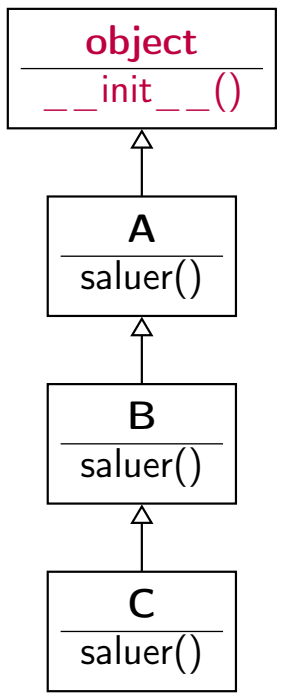
• Pour chaque nouvelle classe créée, Python lui affecte par défaut la classe object comme une classe de base.

• Les déclarations suivantes sont équivalentes:

```
class A:
```

```
class A():
```

```
class A(object):
```




La classe **object** est

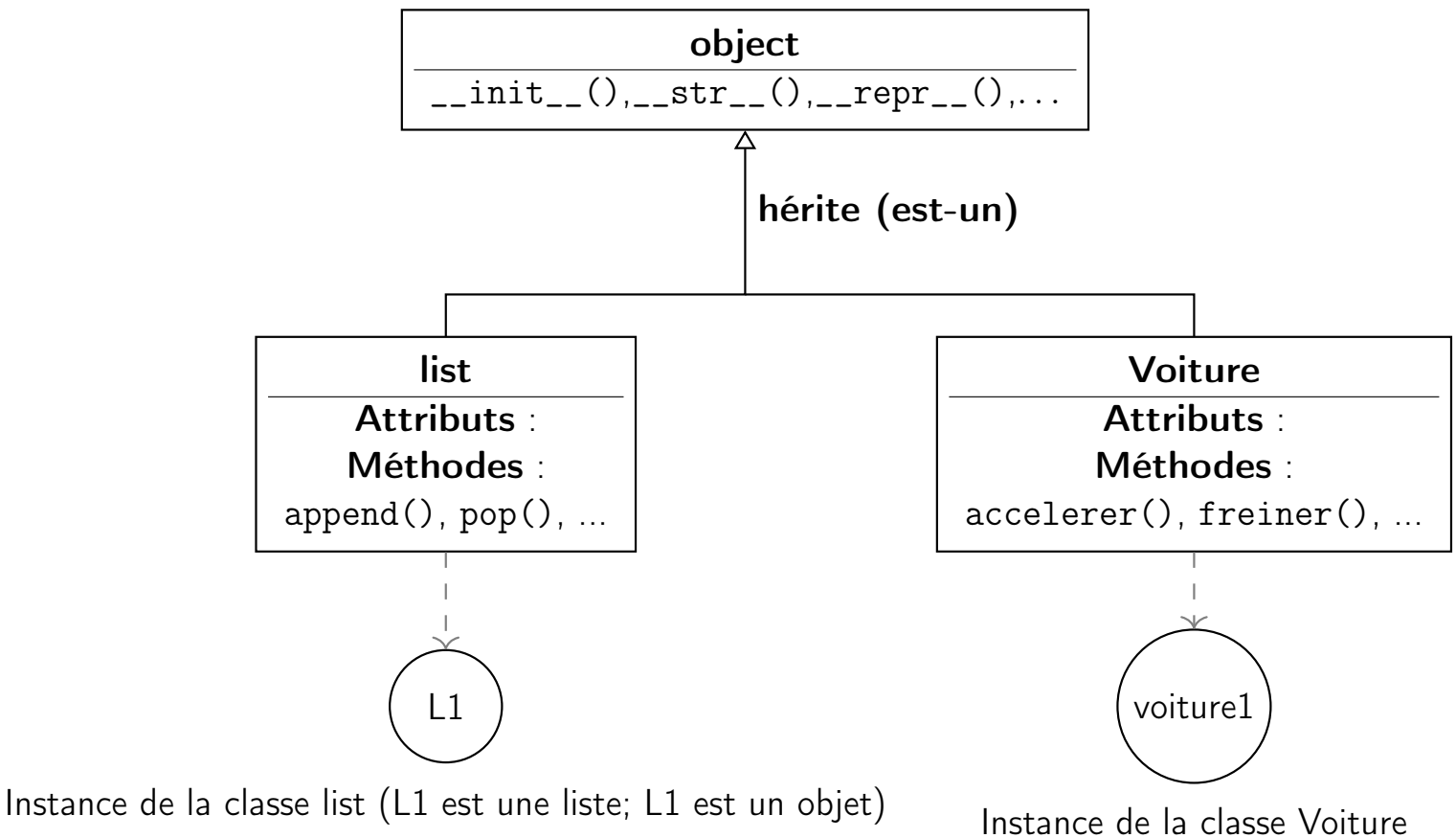
- La classe de base de toutes les classes en Python.
- Implicitement héritée par toutes les classes.
- Au sommet de la hiérarchie de l'héritage des classes.
- Définit des méthodes spéciales et attributs prédéfinis.

# Tout est object

## Relation entre la Classe `object` et les autres classes en Python

La classe **object** est la classe de base de toutes les classes en Python.

Les classes prédéfinies et les classes utilisateur héritent directement ou indirectement de la classe `object` → **Tout est Objet**




Instance de la classe list (L1 est une liste; L1 est un objet)

Instance de la classe Voiture

# Méthodes et attributs de la classe object

`dir(object)` retourne les méthodes et les attributs prédéfinis de la classe object.

```
>>> print(dir(object))
[..., '__class__', '__delattr__', '__dir__','__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__le__', '__lt__', '__ne__', '__new__', '__repr__',
 '__setattr__', '__str__', ...]
```

Exemples:

`__init__(self)` : Appelée lors de la création d'une instance de la classe pour initialiser ses attributs.

`__str__(self)` : Appelée lorsqu'un objet est converti en chaîne de caractères, par exemple avec `str()` ou `print()`, pour fournir une représentation lisible de l'objet.

`__repr__(self)` : Appelée par `repr()` pour obtenir une représentation détaillée et non ambiguë de l'objet, souvent utilisée pour le débogage.
Si vous ne définissez pas __str__() dans une classe, Python utilise __repr__() par défaut lorsque vous faites un print() de l'objet.

`__eq__(self, other)` : Appelée pour comparer deux objets via l'opérateur ==, en vérifiant si les instances sont égales en termes de contenu.

# Méthodes Spéciales de la Classe object I

## Exemples d'utilisation

Les méthodes spéciales (magiques) de la classe `object` peuvent être appelées de manière spécifique pour personnaliser le comportement des objets.

Exemples:

```python
class Voiture(object):

    # La méthode __init__ initialise l'objet après sa création
    def __init__(self, marque, modele):
        print(f"Appel de __init__ pour {self}")
        self.marque = marque
        self.modele = modele

    # Définit la représentation en chaîne de l'objet
    def __str__(self): # toujours retourne une chaine
        return f"Voiture: {self.marque} {self.modele}"

    # représentation souvent utilisée pour le débogage
    def __repr__(self): # toujours retourne une chaine
        return f"Voiture(marque='{self.marque}', modele='{self.modele}')"
```

# Méthodes Spéciales de la Classe object II

## Exemples d'utilisation

```
16        # La méthode __eq__ compare deux objets pour l'égalité
17        def __eq__(self, other):
18            if not isinstance(other, Voiture):
19                raise TypeError("Comparaison des objets de types différents")
20            return (self.marque==other.marque and self.modele==other.modele)
21        # Vous pouvez également redéfinir d'autres méthodes spéciales
22        # comme __lt__ pour <, __gt__ pour >, etc.
23    # Exemples d'utilisation
24    voiture1 = Voiture("Toyota", "Corolla",)
25    voiture2 = Voiture("Toyota", "Corolla")
26    voiture3 = Voiture("Tesla", "Model 3")
27
28    str(voiture1)  # Appel à __str__
29    print(voiture1)  # Appel à __str__
30
31    print(repr(voiture3))  # Appel à __repr__
32    >>>voiture3  # Appel à __repr__
33
34    # Comparaison des objets
35    print(voiture1 == voiture2)  # Appel à __eq__
36    print(voiture1 < voiture3)  # Appel à __lt__
```

# Encapsulation en Python

## Principes et Avantages

**Encapsulation** :

Principe clé de la programmation orientée objet.

Regroupe les données et les méthodes dans une même classe.

Protège les données sensibles contre les modifications non souhaitées.

**Avantages :**

**Protection des Données :** Limite l'accès direct aux attributs.

**Validation des Données :** Permet de contrôler les modifications via des méthodes.

# Encapsulation en Python

## Exemple sans Encapsulation

```python
class Personne:
    def __init__(self, nom, age):
        # Attribut publiques:  accessibles et modifiables directement.
        self.name = name
        self.age = age

# Utilisation de la classe sans encapsulation
p = Personne("Ali", 30)
print(p.nom)  # Ali
print(p.age)   # 30

p.age = -5     # Modification directe de l'attribut
print(p.age)   # -5 (l'âge est maintenant invalide)
```

**Risques d'utilisation des attributs publiques:** Les données peuvent devenir invalides ou incohérentes sans validation, car il n'y a pas de contrôle sur les modifications des attributs.

Solution : marquer les attributs sensibles comme privés et controller la nouvelle valeur à chaque tentative de modification.

# Encapsulation en Python

## Attributs privés

**Mise en œuvre en Python** :
En Python, l'encapsulation est réalisée en définissant des attributs et des méthodes comme **privés** en les préfixant par un **double underscore __**
**Syntaxe Générale :**

Attributs privés : `self.__attribut`

Tout accès à un attribut privé doit se faire via les méthodes (publiques) suivantes:

```
def get_attribut(self):
def set_attribut(self, value):
```

On peut définir une propriété qui simplifie l'accès à l'attribut privé:
`nom_propriete = property(get_attribut,set_attribut)`

**Exemple d'tilisation :**
Soit une classe Etudiant a un attribut note déclaré comme attribut privé.
Soit e une instance de la classe Etudiant.

Déclarer un attributs privé : `self.__note = 0`

Accès en lecture: `e.get_note()`

Modification: `e.set_note(15)`

Accès en lecture via la propriété note: `e.note`

Modification via la propriété note: `e.note=15`

# Encapsulation en Python I

## Attributs privés: Exemple

```python
class Personne:
    def __init__(self, nom, age):
        self.nom = nom # Attribut publique
        self.__age = age # Attribut privé : protégé contre les accès directs.

    # Méthodes d'Accès: contrôlent l'accès et la modification des données.
    def get_age(self):
        return self.__age
    def set_age(self, age):
        if age > 0:  self.__age = age
        else:  print("Age doit être positif.")
    # Définir une propriété age
    age = property(get_age, set_age)

# Utilisation de la classe avec encapsulation
p = Personne("Ali", 30)
print(p.nom) # Ali
print(p.__age) # Erreur
print(p.get_age()) # 30
p.set_age(-5);  p.age = -10 # Affiche "Age doit être positif."
print(p.age) # 30 (l'âge n'a pas été modifié)
```

# Questions ?