

# La Programmation Orientée Objet en Python

Découvrez les concepts fondamentaux de la POO et apprenez à structurer vos programmes Python de manière professionnelle et maintenable.

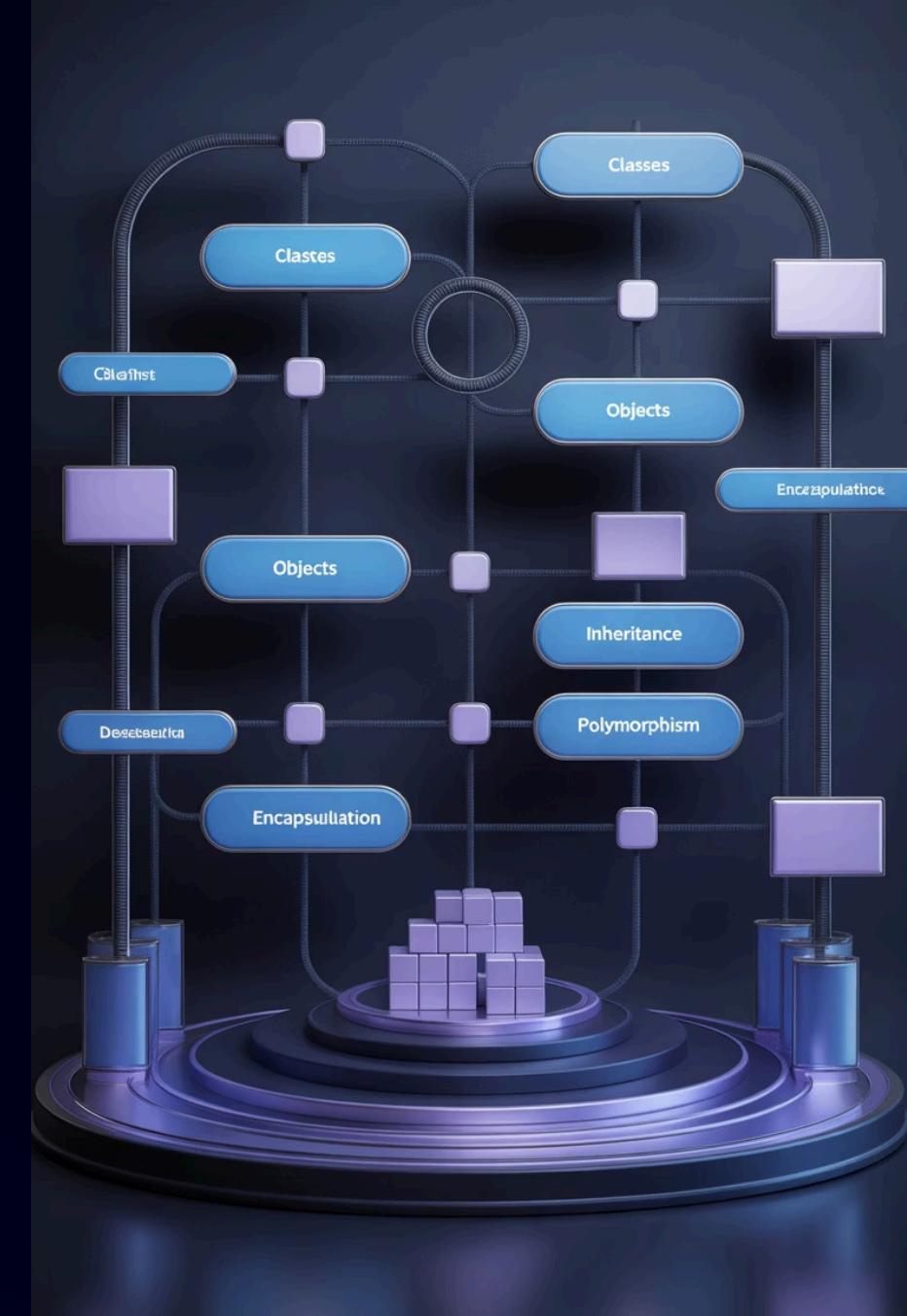
# Qu'est-ce que la Programmation Orientée Objet ?

La Programmation Orientée Objet (POO) est un **paradigme de programmation** qui structure le code autour d'objets plutôt que de fonctions et de logique pure. Un objet combine des données (appelées **attributs** ou **propriétés**) et des comportements (appelés **méthodes**).

Ce paradigme permet de modéliser des concepts du monde réel de manière intuitive. Par exemple, un chien possède des caractéristiques (nom, race, âge) et des comportements (aboyer, courir, manger).

## Avantages de la POO

- **Réutilisabilité** : créez des classes réutilisables dans différents projets
- **Maintenabilité** : code mieux organisé et plus facile à modifier
- **Modularité** : décomposition en unités logiques indépendantes
- **Abstraction** : masquez la complexité interne



# Créer une Classe : Définition et Structure

Une **classe** est un modèle ou un plan qui définit la structure et le comportement d'objets. En Python, on utilise le mot-clé `class` pour créer une classe. Par convention PEP 8, **les noms de classes commencent toujours par une majuscule**.

01

## Déclarer la classe

Utilisez le mot-clé `class` suivi du nom de la classe et de parenthèses

02

## Définir le constructeur

La méthode `__init__()` initialise les propriétés de l'objet

03

## Ajouter des méthodes

Créez des fonctions avec `self` comme premier paramètre

```
class Dog():
    """Classe représentant un chien avec ses caractéristiques"""

    def __init__(self, name, breed, age):
        self.name = name
        self.breed = breed
        self.age = age
        self.is_alive = True
```

Dans cet exemple, nous créons une classe `Dog` avec un constructeur qui accepte trois paramètres : `name`, `breed` et `age`. Le paramètre `self` fait référence à l'instance en cours de création.

# Le Constructeur `__init__()` et `self`

Le **constructeur** est une méthode spéciale qui s'exécute automatiquement lors de la création d'un objet. En Python, cette méthode s'appelle `__init__()` (avec deux underscores de chaque côté).



## Le paramètre `self`

`self` représente l'instance en cours. Il permet d'accéder aux attributs et méthodes de l'objet depuis l'intérieur de la classe.



## Initialisation des attributs

Dans `__init__()`, on définit les propriétés d'instance avec `self.nom_attribut = valeur`



## Paramètres du constructeur

Les paramètres passés au constructeur permettent de personnaliser chaque instance créée

```
class Dog():
    def __init__(self, name, breed, age):
        # Création des propriétés d'instance
        self.name = name
        self.breed = breed
        self.age = age
        # On peut aussi initialiser des valeurs par défaut
        self.is_vaccinated = False

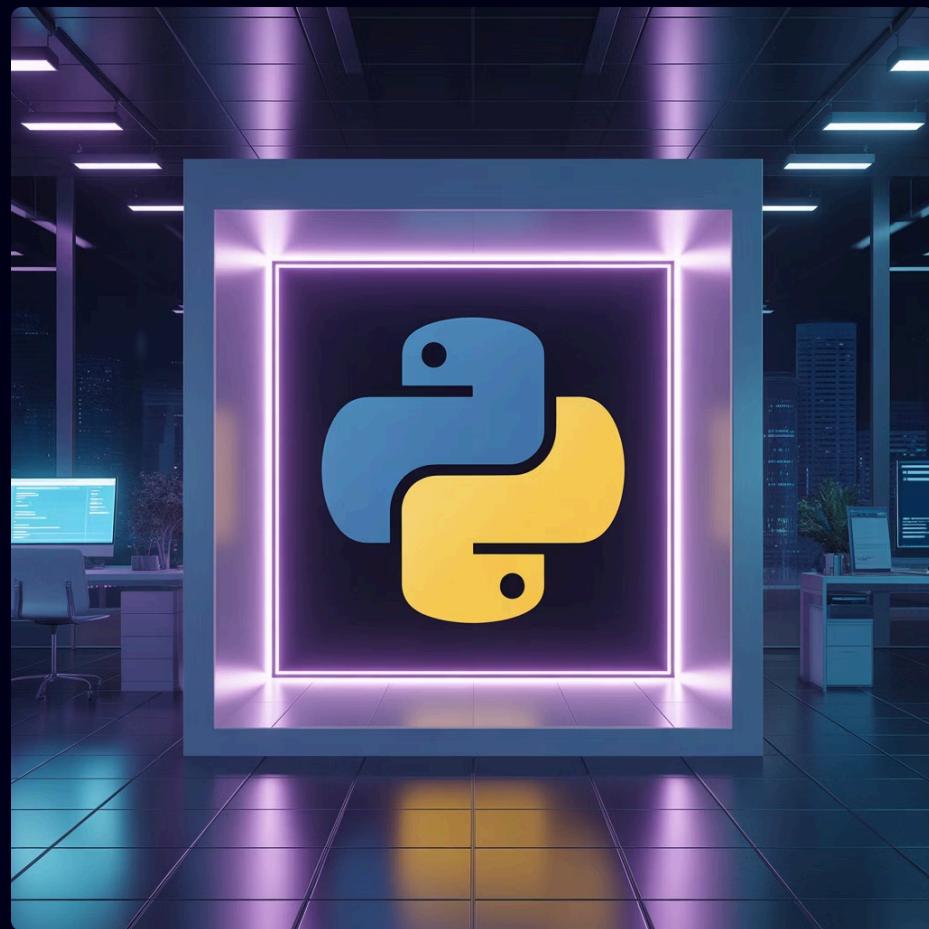
    # Création d'une instance
    mon_chien = Dog("Bernie", "Labrador", 7)
    print(mon_chien.name) # Affiche : Bernie
    print(mon_chien.age) # Affiche : 7
```

# Les Méthodes d'Instance

Une **méthode** est une fonction définie à l'intérieur d'une classe qui décrit un comportement de l'objet. Toute méthode d'instance doit avoir `self` comme premier paramètre pour pouvoir accéder aux attributs et autres méthodes de l'instance.

## Caractéristiques des méthodes

- Premier paramètre obligatoire : `self`
- Accès aux attributs via `self.attribut`
- Peuvent appeler d'autres méthodes
- Peuvent retourner des valeurs



```
class Dog():
    def __init__(self, name, breed, age):
        self.name = name
        self.breed = breed
        self.age = age

    def aboyer(self, text):
        """Méthode pour faire aboyer le chien"""
        print(f"{self.name} says: {text}")

    def vieillir(self):
        """Augmente l'âge du chien"""
        self.age += 1
        print(f"{self.name} a maintenant {self.age} ans")

# Utilisation
mon_chien = Dog("Bernie", "Labrador", 7)
mon_chien.aboyer("Woof!")
mon_chien.vieillir()
```

Dans cet exemple, `aboyer()` utilise l'attribut `self.name` pour personnaliser le message, tandis que `vieillir()` modifie l'attribut `self.age`.

# Les Getters : Contrôler l'Accès aux Propriétés

Les **getters** (accesseurs) permettent de créer des propriétés calculées ou de contrôler la façon dont on accède aux données d'un objet. Python offre deux approches : la méthode `__getattr__()` et le décorateur `@property`.

</>



## Méthode `__getattr__()`

Intercepte les accès à des attributs inexistant pour créer des propriétés virtuelles dynamiques

## Décorateur `@property`

Syntaxe moderne et élégante pour créer des propriétés en lecture seule ou calculées

```
class Dog():
    def __init__(self, name, breed, age):
        self.name = name
        self.breed = breed
        self.age = age

    def __getattr__(self, item):
        """Getter dynamique avec __getattr__"""
        if item == "age_str":
            return f"{self.age} ans"
        elif item == "is_adult":
            return self.age >= 1
        raise AttributeError(item)

    @property
    def color(self):
        """Getter moderne avec @property"""
        return f"{self.name} is Blue"

    @property
    def double_age(self):
        """Propriété calculée"""
        return self.age * 2

# Utilisation
mon_chien = Dog("Bernie", "Labrador", 7)
print(mon_chien.age_str) # Affiche : 7 ans
print(mon_chien.is_adult) # Affiche : True
print(mon_chien.color) # Affiche : Bernie is Blue
print(mon_chien.double_age) # Affiche : 14
```

**Attention :** `__getattr__()` n'est appelé que si l'attribut n'existe pas déjà. Pour intercepter tous les accès, utilisez `__getattribute__()`.

# Quiz : Testez vos connaissances (1/2)

## Question difficile : Analyse de code

Observez le code suivant :

```
class Animal():
    def __init__(self, species, weight):
        self.species = species
        self.weight = weight

    def __getattr__(self, item):
        if item == "weight_category":
            if self.weight < 5:
                return "Petit"
            elif self.weight < 20:
                return "Moyen"
            else:
                return "Grand"
        raise AttributeError(item)

    @property
    def info(self):
        return f"{self.species} ({self.weight} kg)"

chat = Animal("Chat", 4)
elephant = Animal("Eléphant", 5000)
```

### Question A

Que retourne chat.weight\_category ?

### Question B

Que se passe-t-il si on écrit  
elephant.info = "Nouvelle valeur" ?

### Question C

Pourquoi \_\_getattr\_\_() se termine-t-il par raise AttributeError(item) ?



# Quiz : Corrigé et explications (2/2)

## Réponse A

`chat.weight_category` retourne "Petit" car le poids du chat (4 kg) est inférieur à 5. La méthode `__getattr__()` intercepte cet accès et calcule dynamiquement la catégorie.

## Réponse B

Une **erreur AttributeError** sera levée ! Une propriété créée avec `@property` sans setter associé est **en lecture seule**. On ne peut pas lui assigner de valeur directement.

## Réponse C

Le `raise AttributeError(item)` est essentiel pour **maintenir le comportement standard** de Python. Si l'attribut demandé n'est pas géré par notre getter, on doit lever une exception pour signaler qu'il n'existe pas.

- Point important :** Sans le `raise AttributeError`, accéder à un attribut inexistant retournerait `None` au lieu de lever une erreur, ce qui masquerait les bugs potentiels dans votre code.

# Les Setters : Contrôler la Modification des Propriétés

Les **setters** (mutateurs) permettent de contrôler et valider les modifications apportées aux attributs d'un objet. Comme pour les getters, Python propose deux approches : `__setattr__()` et le décorateur `@property_name.setter`.

## Méthode `__setattr__()`

Cette méthode intercepte **toutes** les affectations d'attributs. Elle est puissante mais nécessite de la prudence.

```
class Dog():
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __setattr__(self, key, value):
        if key == "human_age":
            # Conversion âge humain -> âge chien
            self.age = int(value / 7)
        else:
            # Appel du setter parent
            super().__setattr__(key, value)
```

- **Important :** N'oubliez pas `super().__setattr__()` sinon aucun attribut normal ne pourra être défini !

## Décorateur `@property.setter`

Syntaxe moderne pour créer un setter associé à une propriété spécifique.

```
class Dog():
    def __init__(self, name, age):
        self.name = name
        self.age = age

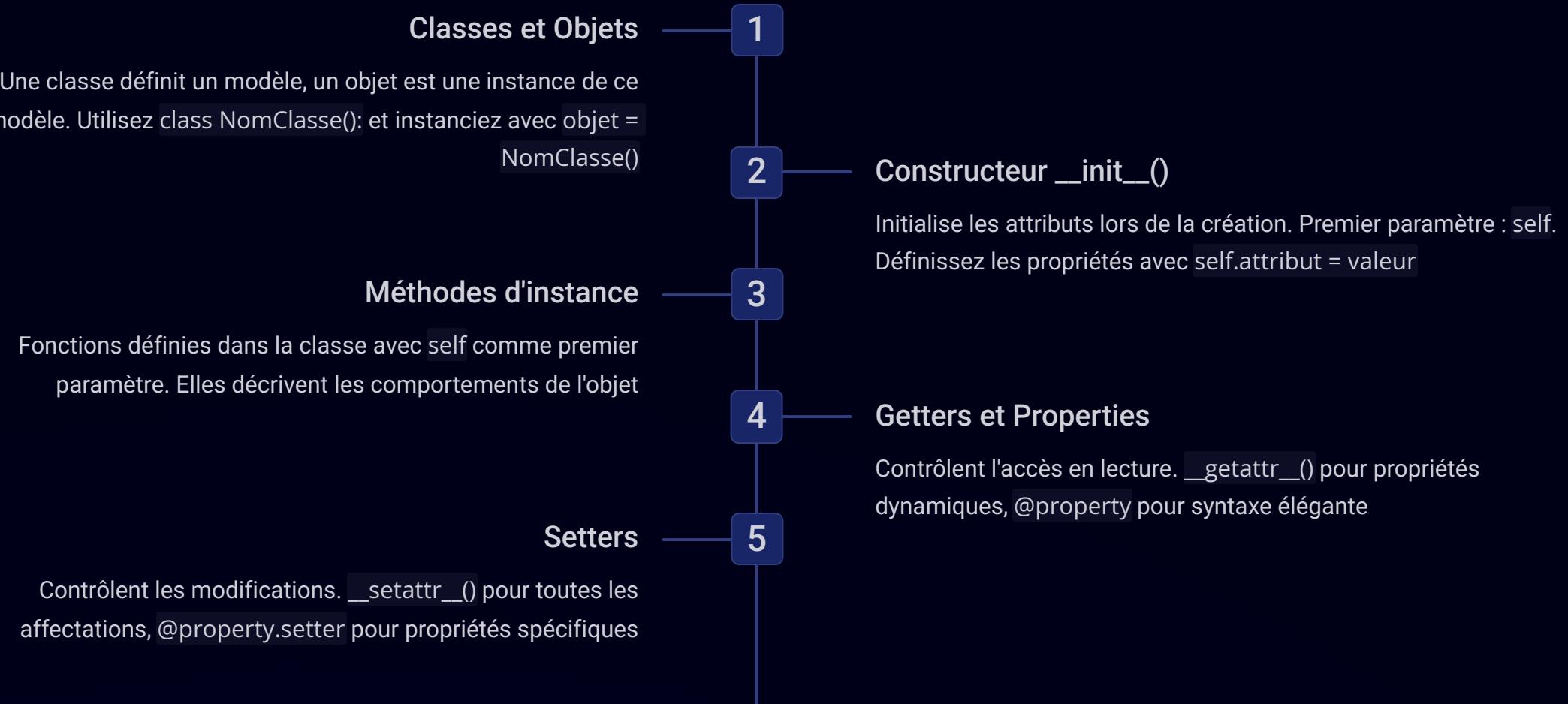
    @property
    def double_age(self):
        return self.age * 2

    @double_age.setter
    def double_age(self, value):
        self.age = int(value / 2)

# Utilisation
chien = Dog("Max", 4)
chien.double_age = 10
print(chien.age) # Affiche : 5
```

Les setters sont particulièrement utiles pour valider les données (vérifier qu'un âge est positif, qu'un email est valide, etc.) ou pour effectuer des conversions automatiques.

# Synthèse : Maîtriser la POO en Python



**Prochaines étapes :** Explorez l'héritage, le polymorphisme et les méthodes de classe pour approfondir votre maîtrise de la POO. Ces concepts fondamentaux vous serviront dans tous vos projets Python, que vous choisissez la spécialisation SISR ou SLAM !

# Définitions clés en POO Python



## Variables d'instance

Attributs spécifiques à chaque objet créé, définis avec `self.nom_variable` dans `__init__()`. Elles stockent l'état unique de chaque instance.

```
class Dog:  
    def __init__(self, name, breed):  
        self.name = name # Variable  
        d'instance  
        self.breed = breed # Variable  
        d'instance  
  
    dog1 = Dog("Buddy", "Golden  
Retriever")  
    dog2 = Dog("Lucy", "Beagle")  
    print(dog1.name) # Affiche : Buddy
```



## Méthodes de classe

Méthodes qui appartiennent à la classe elle-même (pas aux instances), utilisent `@classmethod` et `cls` comme premier paramètre. Elles peuvent agir sur des attributs de classe ou servir de constructeurs alternatifs.

```
class Dog:  
    species = "Canis familiaris" #  
    Variable de classe  
  
    @classmethod  
    def create_from_string(cls, dog_str):  
        name, breed = dog_str.split("-")  
        return cls(name, breed)  
  
    my_dog =  
    Dog.create_from_string("Max-Poodle")  
    print(my_dog.name) # Affiche : Max
```



## L'instanciation

Processus de création d'un objet à partir d'une classe avec `nom_objet = NomClasse(paramètres)`. Chaque instantiation crée une nouvelle instance unique de la classe.

```
class Dog:  
    def __init__(self, name, breed):  
        self.name = name  
        self.breed = breed  
  
    # Instanciation de deux objets Dog  
    dog1 = Dog("Rex", "Berger Allemand")  
    dog2 = Dog("Milo", "Labrador")  
  
    print(dog1.breed) # Affiche : Berger  
    Allemand  
    print(dog2.name) # Affiche : Milo
```

# Méthodes d'Instance vs. Méthodes de Classe : Comprendre les Différences

Pour maîtriser pleinement la POO en Python, il est crucial de distinguer les méthodes qui opèrent sur des instances spécifiques (méthodes d'instance) de celles qui agissent au niveau de la classe elle-même (méthodes de classe).

	
<b>Méthodes d'instance</b> <ul style="list-style-type: none"><li>• Premier paramètre : <code>self</code></li><li>• Travaillent avec les données spécifiques à chaque objet</li><li>• Appelées sur une instance : <code>objet.methode()</code></li><li>• Accèdent aux attributs d'instance via <code>self.attribut</code></li></ul>	<b>Méthodes de classe</b> <ul style="list-style-type: none"><li>• Premier paramètre : <code>cls</code></li><li>• Décorateur <code>@classmethod</code></li><li>• Travaillent avec la classe elle-même, pas les instances</li><li>• Appelées sur la classe : <code>Classe.methode()</code> ou <code>objet.methode()</code></li><li>• Accèdent aux attributs de classe via <code>cls.attribut</code></li></ul>

## Exemples Pratiques avec la Classe Dog

Voyons comment ces deux types de méthodes se manifestent dans le code avec une classe Dog.

```
class Dog:  
    species = "Canis familiaris" # Attribut de classe  
  
    def __init__(self, name, age):  
        self.name = name # Attribut d'instance  
        self.age = age # Attribut d'instance  
  
    # Méthode d'instance  
    def aboyer(self):  
        return f"{self.name} aboie : Woof woof!"  
  
    # Méthode de classe  
    @classmethod  
    def get_species(cls):  
        return cls.species  
  
    # Méthode de classe (constructeur alternatif)  
    @classmethod  
    def create_puppy(cls, name):  
        # Crée un chiot (âge 0)  
        return cls(name, 0)  
  
    # Utilisation  
mon_chien = Dog("Buddy", 3)  
print(mon_chien.aboyer()) # Appel d'une méthode d'instance  
  
print(Dog.get_species()) # Appel d'une méthode de classe via la classe  
print(mon_chien.get_species()) # Appel d'une méthode de classe via l'instance  
  
mon_chiot = Dog.create_puppy("Lucky") # Appel du constructeur alternatif  
print(f"Mon chiot s'appelle {mon_chiot.name} et a {mon_chiot.age} an.")
```

Comme vous pouvez le voir, les méthodes d'instance comme `aboyer()` utilisent `self` pour accéder aux données spécifiques à "Buddy", tandis que les méthodes de classe comme `get_species()` et `create_puppy()` utilisent `cls` pour interagir avec la classe `Dog` elle-même.

# Méthodes de Classe : Puissance et Polyvalence en POO Python

Les méthodes de classe offrent une approche différente des méthodes d'instance. Elles opèrent sur la classe elle-même plutôt que sur une instance spécifique, apportant des avantages distincts pour la conception de votre code.

## Avantages des méthodes de classe



### Constructeurs alternatifs

Créer des instances de différentes manières, en fournissant des logiques de construction personnalisées.



### Accès aux données partagées

Travailler avec des attributs ou des états communs à toutes les instances de la classe.



### Logique métier au niveau classe

Effectuer des opérations qui concernent la classe entière ou son comportement global.



### Pas besoin d'instance

Peuvent être appelées directement sur la classe sans avoir à créer un objet au préalable.



### Factory methods

Créer des objets avec des configurations spécifiques ou à partir de sources de données externes.

## Cas d'usage pratiques

- Compteur d'instances créées
- Crédit d'objets à partir de formats différents (JSON, CSV, etc.)
- Validation de données avant création d'instance
- Configuration globale de la classe

## Exemples concrets avec la classe Dog

### Compter le nombre de chiens créés

```
class Dog:  
    total_dogs = 0 # Variable de classe  
  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
        Dog.total_dogs += 1 # Incrémente le compteur à chaque instance  
  
    @classmethod  
    def get_total_dogs(cls):  
        return cls.total_dogs  
  
dog1 = Dog("Buddy", 3)  
dog2 = Dog("Lucy", 5)  
print(f"Nombre total de chiens: {Dog.get_total_dogs()}") # Affiche: Nombre total de chiens: 2
```

### Créer un chien à partir d'une chaîne formatée

```
class Dog:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    @classmethod  
    def from_string(cls, dog_data):  
        name, age_str = dog_data.split(',')  
        return cls(name.strip(), int(age_str.strip()))  
  
dog_from_str = Dog.from_string("Max, 4")  
print(f"Chien créé: {dog_from_str.name}, {dog_from_str.age} ans") # Affiche: Chien créé: Max, 4 ans
```

### Obtenir des informations sur l'espèce

```
class Dog:  
    species = "Canis familiaris" # Variable de classe  
  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    @classmethod  
    def get_species_info(cls):  
        return f"Tous les chiens appartiennent à l'espèce: {cls.species}"  
  
print(Dog.get_species_info()) # Affiche: Tous les chiens appartiennent à l'espèce: Canis familiaris
```