Python : Orienté objets

Tables des matières

1. Introduction
2. Structures
3. Classes et objets
4. Encapsulation
5. Data model
6. Héritage
7. Polymorphisme
8. Classes abstraites
9. Membres statiques
10. Interfaces
11. Constructeurs

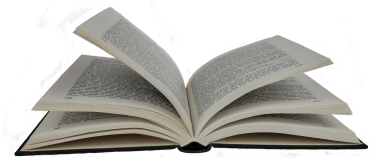
1. Introduction

- Définition de la programmation orientée objet
- Comparaison avec la programmation procédurale
- Principes fondamentaux
- Historique et évolution
- Importance de la POO

Définition de la programmation orientée objet

La POO offre une approche de programmation qui repose sur la modélisation des entités du monde réel sous forme d'objets, chacun possédant des caractéristiques **(attributs)** et des actions **(méthodes)** qui lui sont propres.

Cette approche permet de structurer le code de manière logique et intuitive, en favorisant la réutilisabilité, la modularité et la maintenance aisée du code. En permettant de représenter des entités complexes sous forme d'objets autonomes interagissant entre eux.



La POO permet à un objet de prendre pratiquement n'importe quelle forme, ce qui rend la modélisation des concepts du monde réel ou fictif plus flexible et puissante.

Comparaison avec la programmation procédurale

Contrairement à la programmation procédurale, qui divise le code en séquences d'instructions, la POO organise le code autour de données et de fonctions qui agissent sur ces données, rassemblées au sein d'objets.

Cette approche permet une meilleure modélisation des concepts du monde réel, en favorisant la compréhension du code et en réduisant les risques d'erreurs.

La POO encourage la réutilisation du code grâce à la création de bibliothèques d'objets réutilisables, ce qui accélère le processus de développement et améliore la qualité du code produit.

Procédurale :

```python
def creer_chat(nom, couleur):
    return {"nom": nom, "couleur": couleur}

def miauler(chat):
    return f"{chat['nom']} miaule."

# Création d'un chat et appel de la fonction miauler
mon_chat = creer_chat("Whiskers", "gris")
print(miauler(mon_chat))  # Affiche : Whiskers miaule.
```

POO :

```python
class Chat:
    def __init__(self, nom, couleur):
        self.nom = nom
        self.couleur = couleur

    def miauler(self):
        return f"{self.nom} miaule."

# Création d'un objet chat
mon_chat = Chat("Whiskers", "gris")
print(mon_chat.miauler())  # Affiche : Whiskers miaule.
```

Principes fondamentaux

Outre les concepts d'encapsulation, d'héritage et de polymorphisme qui seront vu dans les chapitre suivants, la POO repose sur le principe d'abstraction, qui consiste à isoler les aspects essentiels d'un objet tout en cachant les détails d'implémentation superflus.

Cette abstraction permet de simplifier la complexité du système en fournissant des interfaces claires et cohérentes pour interagir avec les objets.

En combinant ces principes, la POO offre un cadre flexible et extensible pour la conception de logiciels complexes, facilitant la maintenance et l'évolution du code au fil du temps.

Historiques et évolution de la POO

La POO trouve ses origines dans les travaux pionniers réalisés dans les années 1960 avec le langage de programmation Simula, qui a introduit les concepts de classes et d'objets.

Depuis lors, la POO a connu un développement rapide avec l'émergence de langages tels que Smalltalk, C++, Java et Python, qui ont étendu et popularisé les concepts de la POO. Aujourd'hui, celle-ci est largement utilisée dans le développement logiciel pour sa capacité à modéliser des systèmes complexes comme le montre le tableau suivant.

Année	Événement
1960	Simula, le premier langage de programmation à introduire les concepts de classes et d'objets, est développé.
Années 1970	Smalltalk émerge comme un langage de programmation influent popularisant les idées de la POO, y compris les classes, les objets et l'encapsulation.
Années 1980	C++, un langage de programmation hybride combinant la programmation procédurale avec des concepts de la POO comme l'héritage et le polymorphisme, est créé.

Historiques et évolution de la POO

Années 1990	Java est lancé, un langage de programmation orienté objet conçu pour être portable et sûr, contribuant à la popularisation de la POO.
Années 2000	Python gagne en popularité comme un langage de programmation polyvalent avec un support intégré pour la POO, offrant une syntaxe propre et une grande flexibilité.
Années 2010	L'importance de la POO dans l'ingénierie logicielle moderne est encore renforcée par l'essor des méthodologies de développement agile et des pratiques de développement centrées sur les objets.
Années 2020	La POO continue d'évoluer avec l'émergence de nouvelles tendances telles que la programmation orientée objet fonctionnelle et les architectures logicielles basées sur les microservices.
2024	Les principes de la POO demeurent fondamentaux dans la conception et le développement de logiciels, contribuant à la création de systèmes évolutifs, flexibles et maintenables.

Importance de la POO

Au-delà de ses avantages techniques, la POO favorise des pratiques de développement plus collaboratives et agiles, en encourageant la modularité, la réutilisabilité et la maintenabilité du code.

En permettant de représenter des entités du monde réel de manière naturelle, la POO facilite la communication entre les membres de l'équipe de développement et favorise une meilleure compréhension des exigences du projet.

Son impact s'étend également au-delà du développement logiciel, influençant des domaines tels que l'analyse des données, l'intelligence artificielle et l'ingénierie logicielle.

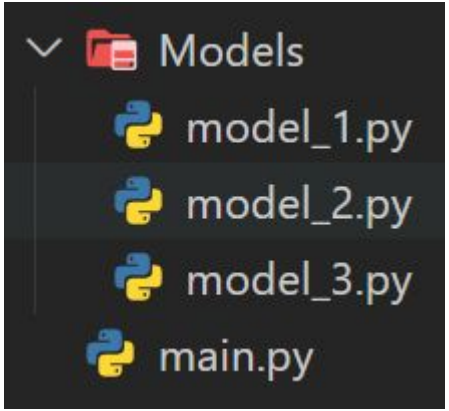
2. Structures

- Travailler avec plusieurs fichiers
- Logique d’imports
- Limite d’exécution
- Bonnes pratiques

Travailler avec plusieurs fichiers

Il est important de structurer le code sur plusieurs fichiers afin de séparer les différentes classes et fonctionnalités. Chaque fichier peut contenir des classes et des méthodes associées à des aspects spécifiques du projet.

Cette approche favorise la modularité et la réutilisabilité du code, en nous permettant de se concentrer sur des parties spécifiques de l'application.



Logique d’import

L'instruction **from Models.model_1 import model_1** est une façon de charger spécifiquement une classe ou une fonction appelée **model_1** depuis un fichier Python appelé **model_1.py** situé dans le répertoire **Models**.

Cette méthode d'importation permet d'accéder directement à la classe ou à la fonction sans avoir besoin de spécifier le nom du module lors de son utilisation. Ensuite, en utilisant **test = model_1()**, nous créons une instance de la classe **model_1**, prête à être utilisée dans notre programme.

Ce type d'importation sélective est couramment utilisé dans les projets pour simplifier l'accès à des fonctionnalités spécifiques et rendre le code plus lisible et modulaire.

```
1   from Models.model_1 import model_1
2   test = model_1()
```

Limite d'exécution

Lors de l'importation d'un fichier Python, il est souvent nécessaire de restreindre l'exécution du code contenu dans ce fichier.

Cela peut être réalisé en utilisant le nom du scope, accessible via la variable spéciale `__name__`. Lorsque le fichier est exécuté directement en tant que point d'entrée, la valeur de `__name__` est définie sur "**main**".

Ainsi, en vérifiant si `__name__` est égal à "**main**", nous pouvons conditionner l'exécution de certaines parties du code à cet état.

```python
def fonction_principale():
    print("Code exécuté uniquement si le fichier est le point d'entrée")

if __name__ == "__main__":
    fonction_principale()
```

Bonnes pratiques

Pour maintenir une structure de projet cohérente et efficace, il est recommandé d'adopter certaines bonnes pratiques :

- **Utiliser des noms descriptifs :** Choisissez des noms de classe qui reflètent clairement le rôle ou la fonction de la classe dans votre système. Les noms descriptifs facilitent la compréhension du code par d'autres développeurs.
- **Éviter les abréviations cryptiques :** Évitez les abréviations peu claires ou cryptiques dans les noms de classe. Optez plutôt pour des noms complets et explicites qui rendent le code plus compréhensible.
- **Respecter la convention de nommage PascalCase :** Suivez la convention de nommage **PascalCase** pour les noms de classe. Cela contribue à maintenir la cohérence dans le code Python et facilite la collaboration.
- **Un seul concept par classe :** Concevez vos classes de manière à ce qu'elles aient une seule responsabilité ou un seul concept. Cela favorise la modularité et la réutilisabilité du code.
- **Documenter les classes :** Utilisez des **docstrings** pour documenter vos classes, en fournissant des descriptions claires de leur fonctionnement, de leurs attributs et de leurs méthodes. Une documentation claire facilite l'utilisation et la maintenance du code.

3. Classe et objets

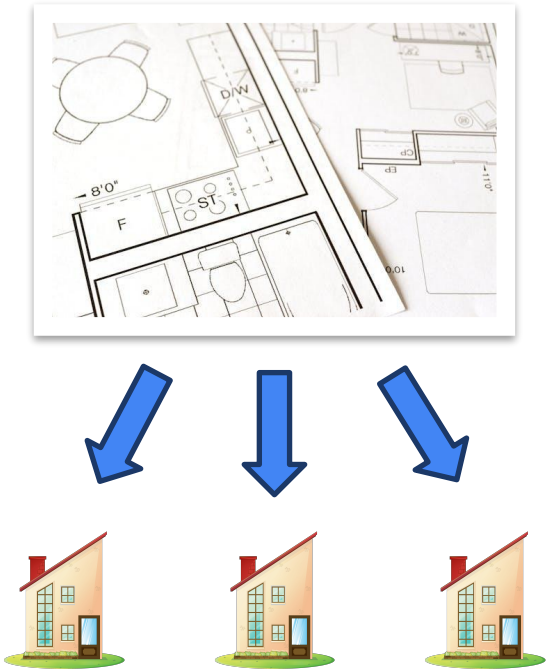
- Définition d’une classe
- Syntaxe de déclaration de classe
- Création et instanciation d'objets
- Attributs
- Méthodes

Définition d’une classe

Les classes en Python servent de modèles pour créer des objets ayant des **attributs** (états) et des **méthodes** (comportements) spécifiques.

Elles agissent comme des plans de construction ou des moules, définissant la structure et le comportement des objets qu'elles instancient.

Chaque objet créé à partir d'une classe est une instance de cette classe, permettant de créer autant d'objets que nécessaire avec les mêmes attributs et méthodes définis dans la classe.



Syntaxe de déclaration de classe

La syntaxe de déclaration de classe en Python est simple. Elle commence par le mot-clé **class**, suivi du nom de la classe que nous souhaiterons créer. Ce mot-clé est fondamental car il indique à Python que nous sommes sur le point de définir une nouvelle classe d'objets.

Dans cet exemple, nous avons utilisé le mot-clé class pour déclarer une classe nommée **Voiture**. Ceci crée une structure de classe vide, prête à être remplie avec des attributs et des méthodes spécifiques à la classe.

```
1 class Voiture:
2     pass
```

En utilisant la syntaxe de classe, nous pourrons définir des modèles de données et des fonctionnalités qui seront utilisés pour créer des objets de cette classe.

Création et instanciation des objets

L'étape après la déclaration d'une classe est la création et l'instanciation d'objets de cette classe. Pour créer un objet à partir d'une classe en Python, nous utiliserons simplement le nom de la classe suivi de parenthèses. Cette opération est appelée instanciation.

Dans cet exemple, nous créons un nouvel objet de la classe Voiture en utilisant la syntaxe **Voiture()**. Cela crée une instance de la classe Voiture, que nous stockons dans la variable **voiture1**. L'objet voiture1 est maintenant une instance de la classe Voiture et peut être utilisé pour accéder aux attributs et aux méthodes définis dans la classe.

```
1   # Création d'un objet de la classe Voiture
2   from Voiture import Voiture
3
4
5   voiture1 = Voiture()
```

Attributs

Les attributs représentent les caractéristiques ou les propriétés d'un objet.

Dans le contexte d'une **voiture**, les attributs pourraient inclure des informations telles que la **marque**, le **modèle**, la **couleur**, la **vitesse actuelle**, et le **nombre de portes**.

Chaque **attributs** joue un rôle essentiel dans la définition de l'état de l'objet et peut être consulté ou modifié par les méthodes de la classe associée ou par ces **attributs** en l'absence de **propriétés** qui les contrôlent.

```python
class Voiture:
    marque = None
    modele = "Corolla"
    couleur = None
    vitesse_actuelle = 0

# Création d'une instance de voiture
ma_voiture = Voiture()

# Affichage des attributs de la voiture
print("Marque :", ma_voiture.marque)
print("Modèle :", ma_voiture.modele)
print("Couleur :", ma_voiture.couleur)
print("Vitesse actuelle :", ma_voiture.vitesse_actuelle)
```

Méthodes

les **méthodes** jouent un rôle essentiel en permettant aux objets d'effectuer des actions ou d'interagir avec d'autres objets. Elles représentent les comportements ou les fonctionnalités associées à un objet particulier.

Celles-ci pourraient inclure des actions telles que l'accélération, le freinage ou le calcul de la consommation de carburant.

Elles encapsulent le comportement de l'objet, offrant ainsi une interface cohérente pour manipuler et interagir avec lui.

```python
class Voiture:
    marque = None
    modele = "Corolla"
    couleur = None
    vitesse_actuelle = 0

    def accelerer(self, acceleration):
        self.vitesse_actuelle += acceleration

    def freiner(self, deceleration):
        self.vitesse_actuelle -= deceleration

    def afficher_info(self):
        print("Marque :", self.marque if self.marque is not None else "Non définis !")
        print("Modèle :", self.modele)
        print("Couleur :", self.couleur if self.couleur is not None else "Non définis !")
        print("Vitesse actuelle :", self.vitesse_actuelle)

# Création d'une instance de voiture
ma_voiture = Voiture()

# Affichage des informations initiales de la voiture
print("Informations initiales de la voiture :")
ma_voiture.afficher_info()

# Accélération de la voiture
ma_voiture.accelerer(20)
print("\nNouvelle vitesse après accélération :")
ma_voiture.afficher_info()
```

Exercices

1. Créer une class Elephant contenant :

Caractéristiques :

- nom
- appétit (sur 100)
- satisfaction (sur 100)
- en_vie
- soigneur

Comportement :

- manger()

Exercices

2. Créer une class soigneur contenant :

Caractéristiques :

- nom
- date_naissance
- exprérience
- nombre_animaux_responsable

Comportement (si le soigneur est responsable de l'animal) :

- nourir(animal)
- entretenir(animal)

Exercices

3. Créer une class enclos contenant :

Caractéristiques :

- nom
- capiacite_max
- taille
- liste_animaux

Comportement :

- ajouter_animal(animal)
- enleve_animal(animal)
- aficher_animaux()
- Bonus : Imaginer un comportement permettant de simuler un passage de jour.

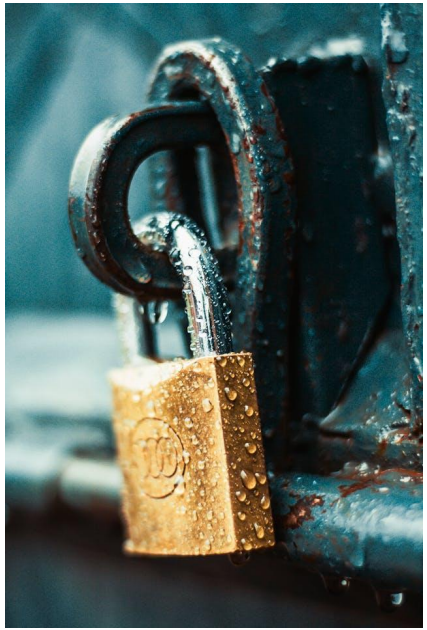
4. Encapsulation

- Objectif
- Déclaration avec @property
- Utilisations des méthodes getter, setter et deleter
- Propriétés en lecture seule
- Propriétés calculées
- Validation des valeurs

Objectif

Les **propriétés** offrent un moyen de contrôler l'accès aux attributs d'une classe. Elles nous permettent de définir des méthodes spéciales, telles que les **getters**, les **setters** et les **deleters**, qui seront automatiquement appelées lors de l'accès, de la modification ou de la suppression de ces attributs.

Grâce à l'utilisation de **propriétés**, il est possible de garantir l'encapsulation des données, ce qui signifie que les détails internes de la classe restent cachés à l'extérieur et ne peuvent être modifiés que de manière contrôlée via des méthodes spécifiques



Déclaration avec @property

La décoration @property nous permet de créer des attributs qui peuvent être accédés comme des attributs de classe, sans nécessiter l'utilisation de parenthèses.

Cette fonctionnalité permet de définir des propriétés directement dans la classe, facilitant ainsi l'accès et la gestion des données de manière transparente.

Par exemple, dans notre classe Voiture, nous pouvons définir une méthode infos décorée avec @property, qui permet d'accéder aux informations sur la voiture, telles que la marque, le modèle, la couleur et la vitesse actuelle, comme s'il s'agissait d'attributs de classe.

```python
class Voiture:
    marque = "Toyota"
    modele = "Corolla"
    couleur = "Rouge"
    vitesse_actuelle = 0

    @property
    def infos(self):
        return f"Marque: {self.marque}, ect..."

# Création d'une instance de la classe Voiture
ma_voiture = Voiture()

# Accès aux informations de la voiture
print("Informations de la voiture:")
print(ma_voiture.infos)
```

Utilisation des méthodes getter, setter et deleter

L'utilisation des méthodes **getter**, **setter** et **deleter** est une pratique essentielle en programmation orientée objet pour contrôler l'accès aux attributs d'une classe. Chaque méthode possède un rôle spécifique :

- **Getter (Accesseur) :** Cette méthode permet d'accéder à la valeur d'un attribut sans permettre de la modifier directement. Elle garantit un accès sécurisé et contrôlé aux données de la classe.

- **Setter (Mutateur) :** Le setter est utilisé pour modifier la valeur d'un attribut en appliquant des règles de validation ou de transformation. Il offre un moyen de modifier les données tout en maintenant l'intégrité de la classe.

- **Deleter (Suppresseur) :** Le deleter est responsable de la suppression d'un attribut de la classe. Il permet de libérer les ressources associées à l'attribut et de maintenir la cohérence de l'objet.

Utilisation des méthodes : getter

Un **getter** est une méthode utilisée pour accéder à la valeur d'un attribut d'une classe.

Celui-ci est implémenté à l'aide du décorateur **@property**, ce qui permet d'accéder à l'attribut comme s'il s'agissait d'un attribut de classe directement, sans avoir à utiliser de parenthèses pour appeler la méthode.

Ainsi, lorsque nous accédons à l'attribut marque de la classe Voiture à travers ma_voiture.marque, Python invoque automatiquement la méthode **getter** associée à cet attribut, permettant de récupérer sa valeur.

```python
class Voiture:
    _marque = "Toyota"

    # Utilisation de @property pour définir un getter pour l'attribut 'marque'
    @property
    def marque(self):
        """
        Méthode getter pour l'attribut '_marque'.
        Elle permet d'accéder à la valeur de l'attribut '_marque'.
        """
        return self._marque

    # Utilisation de @property.getter pour définir un getter pour l'attribut 'marque'
    @property.getter
    def marque(self):
        """
        Méthode getter pour l'attribut '_marque'.
        Elle permet également d'accéder à la valeur de l'attribut '_marque'.
        """
        return self._marque


# Création d'une instance de la classe Voiture
ma_voiture = Voiture()

# Accès à la marque (lecture seule)
print("Marque de la voiture :", ma_voiture.marque)
```

Utilisation des méthodes : setter

Un **setter** est une méthode utilisée pour modifier la valeur d'un attribut d'une classe.

Celui-ci est implémenté à l'aide du décorateur **@propriete.setter**. Lorsque nous attribuons une nouvelle valeur à un attribut à travers le setter, comme dans ma_voiture.marque = "Nissan", Python invoque automatiquement la méthode setter associée à cet attribut, permettant de mettre à jour sa valeur.

Le setter permet de contrôler l'affectation des valeurs aux attributs, ce qui permet de valider ou de transformer les données avant de les assigner.

```python
class Voiture:
    _marque = "Toyota"

    @property
    def marque(self):
        return self._marque

    @marque.setter
    def marque(self, nouvelle_marque):
        """
        Méthode setter pour l'attribut '_marque'.
        Elle permet de modifier la valeur de l'attribut '_marque'.
        """
        print("Modification de la marque en", nouvelle_marque)
        self._marque = nouvelle_marque


# Création d'une instance de la classe Voiture
ma_voiture = Voiture()

# Affichage de la marque initiale
print("Marque initiale de la voiture :", ma_voiture.marque)

# Modification de la marque à l'aide du setter
ma_voiture.marque = "Nissan"

# Affichage de la nouvelle marque après modification
print("Nouvelle marque de la voiture :", ma_voiture.marque)
```

Utilisation des méthodes : deleter

Le **deleter** est une fonctionnalité essentielle pour assurer la gestion complète des attributs d'une classe.

Il permet de définir un comportement spécifique lors de la suppression d'un attribut, offrant ainsi un contrôle supplémentaire sur la manipulation des données.

L'utilité du **deleter** réside dans sa capacité à garantir la cohérence et l'intégrité des données de la classe, en permettant par exemple de libérer des ressources associées à un attribut lors de sa suppression.

```python
class Voiture:
    _marque = "Toyota"

    @property
    def marque(self):
        return self._marque

    @marque.deleter
    def marque(self):
        """
        Méthode deleter pour l'attribut '_marque'.
        Elle permet de supprimer la valeur de l'attribut '_marque'.
        """
        print("Suppression de la marque")
        del self._marque


# Création d'une instance de la classe Voiture
ma_voiture = Voiture()

# Affichage de la marque initiale
print("Marque initiale de la voiture :", ma_voiture.marque)

# Suppression de la marque à l'aide du deleter
del ma_voiture.marque

# Affichage de la marque après suppression (devrait générer une erreur)
print("Marque de la voiture après suppression :", ma_voiture.marque)
```

Propriétés en lecture seule

Les propriétés en lecture seule sont des attributs dont la valeur ne peut être modifiée une fois initialisée. Elles sont définies en utilisant uniquement une méthode getter avec le décorateur @property, ce qui empêche toute modification directe de la valeur de la propriété.

Par exemple, dans une classe Voiture, nous pouvons définir une propriété en lecture seule pour l'identifiant de la voiture, garantissant ainsi que cet attribut ne peut être modifié après sa définition initiale ou via une méthode interne à la classe.

```python
class Voiture:
    _identifiant = "ABC123"  # Attribut défini dans la classe

    @property
    def identifiant(self):
        return self._identifiant

# Création d'une instance de la classe Voiture
ma_voiture = Voiture()

# Accès à la propriété en lecture seule
print("Identifiant de la voiture :", ma_voiture.identifiant)

# Tentative de modification de l'identifiant (ne fonctionnera pas)
ma_voiture.identifiant = "XYZ789"  # générera une erreur
```

Propriétés calculées

Les propriétés calculées offrent la possibilité de définir des attributs dont la valeur est déterminée dynamiquement à partir d'autres attributs de l'objet.

Contrairement aux propriétés simples dont la valeur est stockée directement, les propriétés calculées sont évaluées au moment de l'accès à la propriété.

Cela permet de définir des comportements personnalisés pour l'accès aux données, en permettant de réaliser des calculs ou des transformations sur les données existantes.

```python
class Voiture:
    _vitesse = 0
    _temps_ecoule = 0

    @property
    def vitesse(self):
        return self._vitesse

    @vitesse.setter
    def vitesse(self, nouvelle_vitesse):
        self._vitesse = nouvelle_vitesse

    @property
    def temps_ecoule(self):
        return self._temps_ecoule

    @temps_ecoule.setter
    def temps_ecoule(self, nouveau_temps):
        self._temps_ecoule = nouveau_temps

    @property
    def distance_parcourue(self):
        # Calcul de la distance parcourue en fonction de la vitesse et du temps écoulé
        return self.vitesse * self.temps_ecoule

# Création d'une instance de la classe Voiture
ma_voiture = Voiture()

# Attribution d'une vitesse et d'un temps écoulé
ma_voiture.vitesse = 80  # km/h
ma_voiture.temps_ecoule = 2  # heures

# Accès à la propriété calculée "distance_parcourue"
print("Distance parcourue par la voiture :", ma_voiture.distance_parcourue, "kilomètres")
```

Validation des valeurs

La validation des valeurs des propriétés est importante pour garantir l'intégrité des données. Les méthodes setter permettent de contrôler les valeurs avant de les attribuer à une propriété, assurant ainsi la conformité aux règles métier.

Dans notre exemple avec la classe Voiture, nous avons défini un validateur pour la propriété vitesse. Ce validateur garantis que la vitesse est positive. En cas de valeur invalide, une exception sera levée.

```python
class Voiture:
    def __init__(self):
        self._vitesse = 0

    @property
    def vitesse(self):
        return self._vitesse

    @vitesse.setter
    def vitesse(self, nouvelle_vitesse):
        if nouvelle_vitesse < 0:
            raise ValueError("La vitesse ne peut pas être négative.")
        self._vitesse = nouvelle_vitesse

ma_voiture = Voiture()

# Modification de la vitesse
try:
    ma_voiture.vitesse = 120  # km/h
    print("Vitesse de la voiture :", ma_voiture.vitesse)
except ValueError as e:
    print("Erreur :", e)
```

Exercices

1. Mettre en place des propriétés sur les différents attributs des classes.

Elephant :

- nom (lecture et écriture)
- rassasier (sur 100) (lecture seule)
- bonheur (sur 100) (lecture seule)
- en_vie (lecture seule)
- soigneur (lecture et écriture)

Soigneur :

- nom (lecture et écriture)
- date_naissance (lecture seule)
- exprérience (lecture seule)
- nombre_animaux_responsable (lecture seule)

Enclos :

- nom (lecture et écriture)
- capiacite_max (lecture et écriture)
- taille (lecture seule)
- liste_animaux (lecture seule)

Exercices

2. Mettre en place une propreté calculés qui va donner l'âge du soigneur sur base de sa date de naissance.

3. Réalisez des tests pour vérifier si votre programme continue à tourner correctement, le cas échéant appliquer les correctifs.

5. Data model

- Attributs spéciaux
- Méthodes spéciales
- opérateurs

Attributs spéciaux

Les attributs spéciaux (entourés de double underscore) sont accessibles depuis le type d'une classe ou l'instance d'une classe. Ceux-ci sont en lecture seule.

Nom	Accessible sur	Objectif
__name__	classe	Renvoie le nom de la classe.
__bases__	classe	Renvoie un tuple des classes parentes de la classe.
__class__	objet	Permet de connaître à partir de quelle classe l'objet a été créé.
__dict__	objet	Renvoie un dictionnaire des méthodes et attributs disponibles.
__doc__	classe et objet	Renvoie la "docstring" de la classe interrogée.

Attributs spéciaux : démonstration

```python
class MaClasse:
    """Ceci est la documentation de MaClasse."""

    def methode(self):
        pass

# Accéder à l'attribut __name__
print("Nom de la classe:", MaClasse.__name__)

# Accéder à l'attribut __bases__
print("Classes parentes:", MaClasse.__bases__)

# Créer un objet de la classe MaClasse
objet = MaClasse()

# Accéder à l'attribut __class__
print("Classe de l'objet:", objet.__class__)

# Accéder à l'attribut __dict__
print("Méthodes et attributs disponibles:", objet.__dict__)

# Accéder à l'attribut __doc__
print("Documentation de la classe:", MaClasse.__doc__)
```

Méthodes spéciales

Les méthodes spéciales sont des fonctions prédéfinies avec des noms spéciaux entourés de double undescore également permettent de définir le comportement d'une classe. Celle-ci peuvent être redéfinies par la classe.

Nom	Rôle
__init__	Permet l'initialisation des champs corrects lors de la création de l'objet.
__repr__	Envoie la représentation "officielle" en chaîne de caractères d'un objet.
__str__	Envoie la représentation "informelle" en chaîne de caractères d'un objet.
__len__	Renvoie la longueur de l'objet.
__getitem__	Permet d'accéder à un élément de l'objet en utilisant la notation d'indexation (par exemple, objet[key]).

Méthodes spéciales : démonstration

```python
class Voiture:
    _marque = "Honda"
    _model = "Civic"
    _annee = 2006

    def __repr__(self):
        return f"voiture({self._marque}, {self._model}, {self._annee})"

    def __str__(self):
        return f"{self._annee} {self._marque} {self._model}"

    def __len__(self):
        # Définir la longueur de la voiture
        return len(str(self))

    def __getitem__(self, key):
        # Permet d'accéder à un attribut spécifique de la voiture
        if key == 'marque':
            return self._marque
        elif key == 'model':
            return self._model
        elif key == 'annee':
            return self._annee
        else:
            raise KeyError(f"Attribut '{key}' non valide pour la voiture")
```

```python
# Création d'une voiture
voiture = Voiture()

# Affichage de la représentation officielle et informelle de la voiture
print(repr(voiture))  # Affiche la représentation officielle
print(str(voiture))   # Affiche la représentation informelle

# Obtention de la longueur de la voiture
print(len(voiture))

# Accès aux attributs de la voiture en utilisant la notation d'indexation
print(voiture['marque'])
print(voiture['model'])
print(voiture['annee'])
```

Opérateurs spéciaux

Il sera possible également de manipuler les opérateurs afin de redéfinir le comportement lors de la comparaison.

Méthode spéciale	Rôle	code classique
__eq__	Renvoie True si les objets sont égaux, False sinon.	obj1 == obj2
__lt__	Renvoie True si l'objet actuel est strictement inférieur à l'autre objet, False sinon.	obj1 < obj2
__le__	Renvoie True si l'objet actuel est inférieur ou égal à l'autre objet, False sinon.	obj1 <= obj2
__gt__	Renvoie True si l'objet actuel est strictement supérieur à l'autre objet, False sinon.	obj1 > obj2
__ge__	Renvoie True si l'objet actuel est supérieur ou égal à l'autre objet, False sinon.	obj1 >= obj2

Opérateurs spéciaux : démonstration

```python
class Car:
    _marque = ""
    _model = ""
    _annee = 0

    def __eq__(self, other):  # Opérateur ==
        return self._marque == other._marque and self._model == other._model and self._annee == other._annee

    def __ne__(self, other):  # Opérateur !=
        return not self == other

    def __lt__(self, other):  # Opérateur <
        if self._annee != other._annee:
            return self._annee < other._annee
        elif self._marque != other._marque:
            return self._marque < other._marque
        else:
            return self._model < other._model

    def __le__(self, other):  # Opérateur <=
        return self < other or self == other

    def __gt__(self, other):  # Opérateur >
        return not (self < other or self == other)

    def __ge__(self, other):  # Opérateur >=
        return not (self < other)
```

```python
# Création de quelques voitures
car1 = Car()
car1._marque = "Toyota"
car1._model = "Camry"
car1._annee = 2020

car2 = Car()
car2._marque = "Honda"
car2._model = "Accord"
car2._annee = 2019

car3 = Car()
car3._marque = "Toyota"
car3._model = "Corolla"
car3._annee = 2020

# Comparaison des voitures
print(car1 == car2)  # False
print(car1 != car2)  # True
print(car1 < car2)   # True
print(car1 > car2)   # False
print(car1 <= car3)  # True
print(car2 >= car3)  # False
```

6. Heritage

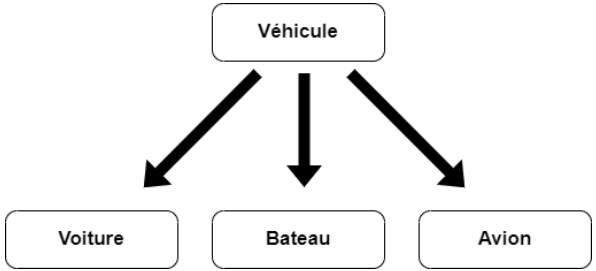
- Définition
- Héritage simple
- La méthode << Super() >>
- Héritage multiple
- Redéfinition de méthodes
- Trucs et astuces

Définition de l’héritage

L'héritage est un concept fondamental en programmation orientée objet, offrant la possibilité aux classes de partager des attributs et des méthodes avec d'autres classes, appelées superclasses.

Cette relation parent-enfant permet aux sous-classes d'hériter et d'étendre les fonctionnalités de la superclasse, favorisant ainsi la réutilisabilité du code et la modularité de la conception logicielle.

En héritant des caractéristiques de la superclasse, les sous-classes peuvent bénéficier d'une implémentation prédéfinie, tout en ayant la flexibilité d'ajouter leurs propres fonctionnalités spécifiques.



Heritage simple

Pour mettre en place l'héritage, nous devons suivre une structure claire et méthodique. Tout d'abord, nous définissons la classe parente, qui contient les attributs et les méthodes communs à plusieurs classes.

Ensuite, nous déclarons la classe enfant en spécifiant entre parenthèses le nom de la classe parente, indiquant ainsi l'héritage. Dans la sous-classe, nous pouvons personnaliser le comportement en ajoutant de nouveaux attributs ou en définissant des nouvelles méthodes.

```python
class Vehicule:
    marque = "honda"
    model = "Civic"
    def afficher_info(self):
        print("Marque :", self.marque)
        print("Modèle :", self.model)


class Voiture(Vehicule):
    couleur = None


# Création d'une instance de la classe Voiture
ma_voiture = Voiture()
ma_voiture.marque = "Toyota"
ma_voiture.model = "Corolla"
ma_voiture.couleur = "rouge"

# Affichage des informations de la voiture
print("Informations de la voiture :")
ma_voiture.afficher_info()
```

La méthode << Super() >>

La méthode super() permet d'accéder aux méthodes et aux attributs de la classe parente à partir de la classe enfant.

Cela est particulièrement utile lorsque nous redéfinissons des méthodes dans la sous-classe tout en souhaitant appeler la méthode équivalente de la classe parente.

super() permet donc d'appeler la méthode parente de manière dynamique et flexible, sans avoir à spécifier explicitement le nom de la classe parente.

```
 9  class Voiture(Vehicule):
10      couleur = "Rouge"
11
12      def afficher_info(self):
13          super().afficher_info()
14          print("Couleur :", self._couleur)
```

Heritage multiple

En Python, contrairement à d'autre langages, il est possible d'appliquer l'héritage multiple. Ce qui signifie qu'une classe peut hériter des attributs et des méthodes de plusieurs autres classes.

Cela permet à une classe enfant d'avoir accès aux fonctionnalités de plusieurs classes parentes.

```python
class Vehicule:
    marque = None

    def afficher_marque(self):
        print("Marque du véhicule :", self.marque)


class Modele:
    model = None

    def afficher_modele(self):
        print("Modèle du véhicule :", self.model)


class Voiture(Vehicule, Modele):

    def afficher_info(self):
        self.afficher_marque()
        self.afficher_modele()


# Création d'une instance de la classe Voiture
ma_voiture = Voiture()
ma_voiture.marque = "Toyota"
ma_voiture.model = "Corolla"

# Affichage des informations de la voiture
ma_voiture.afficher_info()
```

Redéfinition de méthodes

La redéfinition de méthode permet de redéfinir une méthode existante dans la classe parente dans une classe enfant, en fournissant une implémentation spécifique à cette classe enfant.

Cela permet à la classe enfant de modifier ou d'étendre le comportement de la méthode héritée de la classe parente sans modifier la classe parente elle-même.

Cette technique est utile pour personnaliser le comportement des méthodes en fonction des besoins spécifiques de chaque classe enfant.

```python
class Vehicule:
    def demarrer(self):
        print("Le véhicule démarre.")

    def arreter(self):
        print("Le véhicule s'arrête.")


class Voiture(Vehicule):
    def demarrer(self):
        print("La voiture démarre en appuyant sur l'accélérateur.")

    # La méthode arreter() n'est pas redéfinie, donc elle est héritée de la classe parente


# Création d'une instance de la classe Voiture
ma_voiture = Voiture()

# Appel des méthodes surchargées
ma_voiture.demarrer()  # Affiche: La voiture démarre en appuyant sur l'accélérateur.
ma_voiture.arreter()   # Affiche: Le véhicule s'arrête.
```

Trucs et astuces : issubclass()

La fonction **issubclass()** permet de vérifier si une classe est une sous-classe d'une autre classe. Elle prend deux arguments : la classe enfant et la classe parente, et renvoie **True** si la première est une sous-classe de la seconde, sinon False.

Cette fonction est utile pour vérifier les relations d'héritage entre les classes et peut être utilisée dans divers contextes, comme la gestion des exceptions ou la définition de comportements spécifiques basés sur la hiérarchie des classes.

```python
class Vehicule:
    pass

class Voiture(Vehicule):
    pass

class Moto(Vehicule):
    pass

# Vérification si Voiture est une sous-classe de Vehicule
print(issubclass(Voiture, Vehicule))  # Output: True

# Vérification si Moto est une sous-classe de Vehicule
print(issubclass(Moto, Vehicule))     # Output: True

# Vérification si Vehicule est une sous-classe de Moto
print(issubclass(Vehicule, Moto))     # Output: False
```

Exercices

1. Créez de la classe Girafe qui reprendra les mêmes éléments que la classe Éléphant mais qui implémentera maintenant en plus les éléments suivants :

Caractéristiques :

- longueur_cou (lecture & écriture)

Comportement :

- manger_feuilles()
- boire_eau()

Exercices

2. Adaptez la classe Éléphant afin qu'elle contienne à présent les éléments suivants :

Caractéristiques :

- longueur_defense (lecture & ecriture)

Comportement :

- prendre_bain_de_boue()
- aspirer_eau()

Exercices

3. Mettez en place le concept d'héritage en créant la classe Animal qui reprendra les éléments communs aux classes Girafe et Éléphant.

4. Définissez une méthode observer_environnements() au sein de cette classe Animal et redéfinissez-la ensuite dans les enfants pour adapter son implémentation.

5. Réalisez des tests en implémentant les différentes fonctionnalités ajoutées et vérifiez leurs bon fonctionnement.

7. Polymorphisme

Le polymorphisme en Python permet à des objets de différentes classes d'être traités de manière uniforme s'ils partagent une interface commune.

Cela signifie que des méthodes portant le même nom peuvent agir différemment selon le type de l'objet sur lequel elles sont appelées.

```python
class Voiture:
    def description(self):
        return "Je suis une voiture standard."

class VoitureSport(Voiture):
    def description(self):
        return "Je suis une voiture de sport rapide."

class VoitureElectrique(Voiture):
    def description(self):
        return "Je suis une voiture électrique économe en carburant."

# Fonction générique prenant une voiture en argument
def afficher_description(voiture):
    print(voiture.description())

# Création d'instances de différentes voitures
voiture_standard = Voiture()
voiture_sport = VoitureSport()
voiture_electrique = VoitureElectrique()

# Appel de la fonction avec différentes instances de voitures
afficher_description(voiture_standard)
afficher_description(voiture_sport)
afficher_description(voiture_electrique)
```

Exercices

1. Mettre en place une méthode comportement_hasard() dans les classes girafe et éléphant qui permettront de choisir au hasard un des comportement propre à chaque animal.

2. Réalisez des tests afin d'appeler la méthode l'instance sur différents objets.

8. Classes abstraites

- Objectif
- Module abc
- Décorateurs abstraits
- Implémentations

Objectif

Les classes abstraites servent de modèles pour d'autres classes. Elles peuvent contenir des méthodes et des propriétés abstraites, qui sont des éléments déclarés mais non implémentés.

Ces classes ne peuvent pas être instanciées directement, mais sont conçues pour être sous-classées afin d'implémenter les méthodes et les propriétés abstraites nécessaires.

Les méthodes abstraites définissent des comportements que les sous-classes doivent fournir, tandis que les propriétés abstraites spécifient les attributs que les sous-classes doivent posséder.

module abc

Le module **abc**, abréviation de "Abstract Base Classes", est un module intégré à Python qui fournit des fonctionnalités pour la définition de classes abstraites et l'implémentation de l'héritage d'interface.

```python
from abc import ABC

class Vehicule(ABC):
    def mouvement(self):
        pass
```

décorateurs abstraits

Les décorateurs abstraits du module abc stipulé dans le slide précédent sont utilisés pour définir des classes abstraites ainsi que des méthodes et des propriétés abstraites de manière explicite.

Ces décorateurs permettent de marquer une classe comme étant abstraite et de déclarer des méthodes et des propriétés comme abstraites, sans fournir d'implémentation.

Cela permet de créer une structure cohérente pour des classes destinées à être sous-classées, en spécifiant les comportements attendus sans les implémenter directement.

```python
from abc import ABC, abstractmethod, abstractproperty

class Vehicule(ABC):
    @abstractproperty
    def vitesse_max(self):
        pass

    @abstractmethod
    def mouvement(self):
        pass
```

Implémentations

Lorsqu'une classe abstraite définit des méthodes abstraites et des propriétés abstraites, toute sous-classe doit fournir une implémentation de ces membres pour être considérée comme complète.

En implémentant ces méthodes et propriétés abstraites, les sous-classes donnent vie aux fonctionnalités et caractéristiques définies dans la classe abstraite, en les adaptant à leur propre contexte et en fournissant une logique spécifique à leur domaine d'application.

```python
from abc import ABC, abstractmethod, abstractproperty

class Vehicule(ABC):
    @abstractproperty
    def vitesse_max(self):
        pass

    @abstractmethod
    def mouvement(self):
        pass

class Voiture(Vehicule):
    _vitesse_actuelle = 0

    @property
    def vitesse_actuelle(self):
        return self._vitesse_actuelle

    @property
    def vitesse_max(self):
        return 200

    def mouvement(self):
        return "La voiture roule"

ma_voiture = Voiture()

print("Vitesse actuelle :", ma_voiture.vitesse_actuelle)
print("Vitesse maximale :", ma_voiture.vitesse_max)

print(ma_voiture.mouvement())
```

Exercices

1. Modifier la classe Animal en classe abstraite.

2. Changer l'implémentation des méthodes observer_environnement() et ramasser_objet() en méthodes abstraites, avec toutes les implications que cela comporte.

3. Implémenter une nouvelle méthode abstraite probabilite_deces() dans la classe Animal et définir ce qui manque dans le programme pour que celui-ci continue à fonctionner.

4. Réaliser des tests pour vérifier le bon fonctionnement des nouveaux comportements.

9. Membres statiques

- Objectif
- Définition

Objectif

Les classes statiques sont des entités au sein d'une classe qui sont partagées par toutes les instances de cette classe, plutôt que par chaque instance individuellement.

Elles peuvent contenir des attributs ou des méthodes qui ne dépendent pas de l'état spécifique de chaque instance, mais sont plutôt liés à la classe dans son ensemble.

Ces membres peuvent être accessibles sans avoir besoin d'instancier la classe, ce qui les rend utiles pour les fonctions utilitaires, les constantes partagées, ou pour encapsuler des fonctionnalités qui ne dépendent pas de l'état de l'objet.

⚠️ **Contrairement à d'autres langages (telle que le C# et le Java), le python permet d'accéder aux membres statiques depuis une instance de la classe !**

Définition

Par exemple, la classe Voiture définit deux méthodes statiques, verifier_immatriculation et verifier_annee_fabrication, qui sont utilisées pour valider respectivement une immatriculation et une année de fabrication de voiture.

Ces méthodes peuvent être appelées directement à partir de la classe sans nécessiter d'instanciation d'objet. Contrairement aux méthodes d'instance, celles-ci ne possèdent pas l'argument « *self* ».

```python
class Voiture:
    @staticmethod
    def verifier_immatriculation(immatriculation):
        return immatriculation.isalnum() and len(immatriculation) == 7

    @staticmethod
    def verifier_annee_fabrication(annee):
        return 1900 <= annee <= 2024

immatriculation_valide = Voiture.verifier_immatriculation("AB123CD")
annee_valide = Voiture.verifier_annee_fabrication(2010)

print("Immatriculation valide :", immatriculation_valide)
print("Année de fabrication valide :", annee_valide)
```

Définition

Bien que nous n'ayons pas encore explicitement étudié les constructeurs, nous avons déjà appliqué le concept d'attributs statiques dans les chapitres précédents.

L'attribut **nombre_total_voitures** de la classe Voiture agit comme un compteur global.

Cet attribut est automatiquement mis à jour pour refléter le nombre total de voitures créées jusqu'à présent.

Les constructeurs seront abordés dans un chapitre ultérieur.

```python
class Voiture:
    nombre_total_voitures = 0

    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele
        Voiture.nombre_total_voitures += 1

    @staticmethod
    def afficher_nombre_total_voitures():
        print("Nombre total de voitures :", Voiture.nombre_total_voitures)

voiture1 = Voiture("Toyota", "Camry")
voiture2 = Voiture("Honda", "Civic")
voiture3 = Voiture("Ford", "Fusion")

Voiture.afficher_nombre_total_voitures()
```

Exercices

1. Définir une classe statique Outils qui comportera les méthodes suivantes :

- Clear_console() => nettoie la console
- Pause(seconde) => simule un temps de pause

2. Adaptez l'appel dans le programme.

10. Interfaces

- Objectif
- Duck Typing
- Définition
- Trucs et astuces

Objectif

Les interfaces en programmation orientée objet servent à définir un contrat entre les classes, spécifiant les méthodes qu'elles doivent implémenter.

L'objectif principal des interfaces est de garantir une certaine cohérence dans l'utilisation des classes en fournissant une structure commune pour les fonctionnalités partagées.

Elles permettent également de favoriser la modularité du code en nous permettant de travailler avec des abstractions plutôt que des implémentations concrètes.

Duck typing

Le duck typing, concept de typage en programmation, il tire son nom du test du canard : si un objet ressemble à un certain type par ses caractéristiques et son comportement, alors il est considéré comme ce type, indépendamment de son héritage formel.

En Python, cela signifie qu'un objet est considéré comme appartenant à un type si celui-ci implémente les méthodes spécifiques à ce type, sans nécessiter de déclaration explicite d'appartenance à ce type.

Par exemple, un objet est considéré comme une collection s'il implémente les méthodes spécifiques telles que **len**, **contains**, et **iter**, même s'il n'est pas explicitement déclaré comme une instance de la classe Collection.

Définition

la définition d'interfaces explicites n'est pas possible en python. Cependant, nous pouvons les simuler en utilisant des mécanismes tels que les classes abstraites, l'héritage multiple et le polymorphisme.

Ces approches permettent de définir un ensemble de méthodes que les classes doivent implémenter pour être considérées comme conformes à une interface donnée.

Définition

La première étape consiste à créer une interface en définissant une classe abstraite avec les méthodes souhaitées, sans fournir leur implémentation. Cela crée un contrat que les classes devront respecter.

```python
from abc import ABC, abstractmethod

class Vehicule(ABC):
    @abstractmethod
    def accelerer(self, acceleration):
        pass

    @abstractmethod
    def freiner(self, deceleration):
        pass
```

Définition

Ensuite, nous mettons en place l'héritage entre cette interface et la classe concrète que nous voulons adapter à l'interface. Cela lie notre classe à celle-ci, garantissant que notre classe implémente toutes les méthodes requises.

Enfin, nous implémentons les méthodes de l'interface dans notre classe concrète. Cela signifie que nous fournissons une implémentation spécifique à chaque méthode abstraite définie dans l'interface, adaptée au comportement attendu de notre classe.

```python
from abc import ABC, abstractmethod

class Vehicule(ABC):
    @abstractmethod
    def accelerer(self, acceleration):
        pass

    @abstractmethod
    def freiner(self, deceleration):
        pass

class Voiture(Vehicule):
    def accelerer(self, acceleration):
        print(f"La voiture accélère de {acceleration} km/h")

    def freiner(self, deceleration):
        print(f"La voiture freine de {deceleration} km/h")
```

Trucs et atuces

Le polymorphisme permet d'utiliser la méthode isinstance() pour vérifier si un objet est une instance d'une interface spécifique. En passant l'objet à tester et l'interface comme arguments, cette méthode retourne True si l'objet implémente l'interface, sinon False.

```python
from abc import ABC, abstractmethod

class Interface(ABC):
    @abstractmethod
    def methode_interface(self):
        pass

class MaClasse(Interface):
    def methode_interface(self):
        print("Méthode de l'interface implémentée")

objet = MaClasse()

if isinstance(objet, Interface):
    print("L'objet implémente l'interface.")
else:
    print("L'objet n'implémente pas l'interface.")
```

Exercices

1. Mettez en place une interface IAnimal qui devra reprendre les comportements suivants :

- manger()
- observer_environnement()
- faire_une_sieste()
- probabilite_deces()
- diminuer_rassasier()
- diminuer_bonheur()

2. Implémentez l'interface au niveau des classes Éléphant et Girafe.

3. Agissez en conséquence pour que le programme continue à fonctionner.

11. Constructeurs

- Objectif
- Syntaxe
- Paramètres
- Constructeur par défaut
- Appel de constructeurs de superclasses
- Trucs & astuces

Objectif

Les constructeurs, fondamentaux en programmation orientée objet, sont des méthodes spéciales qui initialisent les objets dès leur création.

Ils sont essentiels pour configurer correctement les instances de classe, définissant ainsi l'état initial des attributs et préparant l'objet à une utilisation ultérieure.

En garantissant une initialisation adéquate, les constructeurs assurent que les nouveaux objets sont prêts à être utilisés, en leur attribuant des valeurs par défaut, en exécutant des vérifications ou en configurant d'autres paramètres nécessaires.

Syntaxe

En Python, les constructeurs sont définis par une méthode spéciale nommée __init__. Cette méthode est appelée automatiquement à chaque fois qu'une nouvelle instance de classe est créée.

La syntaxe pour déclarer un constructeur est donc simplement de définir une méthode __init__ à l'intérieur de la classe. Cette méthode prend généralement comme premier paramètre une référence à l'objet lui-même, conventionnellement nommée self.

```
1   class Voiture:
2       def __init__(self):
3           pass
```

Paramètres

Les constructeurs peuvent accepter des paramètres, qui sont des valeurs fournies lors de l'instanciation de la classe et utilisées pour personnaliser l'initialisation de l'objet.

Ces paramètres sont passés à la méthode __init__ lors de son appel et peuvent être utilisés à l'intérieur du constructeur pour initialiser les attributs de l'objet avec des valeurs spécifiques.

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

voiture1 = Voiture("Toyota", "Corolla", 2020)
voiture2 = Voiture("Honda", "Civic", 2018)
voiture3 = Voiture("Ford", "Mustang", 2019)
```

Constructeur par défault

Si aucun constructeur n'est explicitement défini dans une classe, Python fournit un constructeur par défaut qui ne fait rien.

Cependant, dès qu'un constructeur est défini, le constructeur par défaut n'est plus automatiquement utilisé, à moins qu'il ne soit explicitement appelé dans le constructeur défini.

```python
#  avec un constructeur par défaut
class Exemple:
    # Définition du constructeur par défaut
    def __init__(self):
        print("Constructeur par défaut appelé")

# avec un constructeur personnalisé
class Exemple2:
    # Définition du constructeur personnalisé
    def __init__(self, nom):
        self.nom = nom
        print(f"Constructeur personnalisé appelé avec le nom '{self.nom}'")

# Création d'une instance de la classe Exemple (constructeur par défaut appelé)
exemple = Exemple()

# Création d'une instance de la classe Exemple2 avec un nom spécifique
exemple = Exemple2("Objet_A")

# Affichage de la valeur de l'attribut 'nom' de l'instance créée
print("Nom de l'objet :", exemple.nom)
```

Appel de constructeurs de superclasses

Lorsque des classes héritent d'autres classes, il est souvent nécessaire d'appeler le constructeur de la classe parente pour initialiser les attributs hérités.

Cela se fait en appelant explicitement le constructeur de la superclasse à l'intérieur du constructeur de la sous-classe à l'aide de la fonction super().

```python
class Vehicule:
    def __init__(self, marque):
        self.marque = marque
        print("Constructeur de la classe Vehicule appelé")

class Voiture(Vehicule):
    def __init__(self, marque, modele):
        super().__init__(marque)
        self.modele = modele
        print("Constructeur de la classe Voiture appelé")

voiture = Voiture("Toyota", "Corolla")

print("Marque :", voiture.marque)
print("Modèle :", voiture.modele)
```

Trucs et astuces

Lors de la définition d'un constructeur dans une classe, il est possible de définir des valeurs par défaut pour certains paramètres.

Cela signifie que si aucun argument n'est fourni lors de la création d'une instance de cette classe, les paramètres avec des valeurs par défaut seront utilisés.

```python
class Voiture:
    def __init__(self, marque, modele="Modèle par défaut"):
        self.marque = marque
        self.modele = modele
        print("Constructeur avec marque et modèle appelé")

voiture1 = Voiture("Toyota", "Corolla")

voiture2 = Voiture("Honda")

print("Voiture 1 :", voiture1.marque, voiture1.modele)
print("Voiture 2 :", voiture2.marque, voiture2.modele)
```

Exercices

1. Mettre en place un constructeur reprenant les différents attributs de la classe animal.

2. Faire hériter le constructeur sur les classes enfants concernées en ajoutant les éléments propres à chacune d'entre elles en prenant les valeurs par défauts nécessaires.

3. Changer l'encapsulation des propriétés de l'ensemble des classes afin qu'elles soient en lectures seules.

4. Définir et appliquer ce qu'il manque pour que le programme continue de tourner correctement.

Merci pour votre attention.