Python POO

m2iformation.fr

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Programmation Orientée Objet en Python

Les bases de la POO

Plan du cours

1. Introduction - Pourquoi la POO ?
2. Les classes et les objets
3. Le constructeur `__init__` et `self`
4. Les méthodes membres
5. L'héritage simple
6. L'héritage multiple

Introduction - Pourquoi la POO ?

Approche sans POO (problématique)

```
# Pour 3 chiens, cela devient vite ingérable !
chien1_nom = "Rex"
chien1_race = "Berger Allemand"
chien1_age = 3

chien2_nom = "Bella"
chien2_race = "Golden Retriever"
chien2_age = 5

chien3_nom = "Max"
chien3_race = "Labrador"
chien3_age = 2
```

Problèmes identifiés

- Code répétitif et difficile à maintenir
- Pas de lien clair entre les données et les actions
- Risque d'erreurs en manipulant les variables

La POO apporte des solutions

- Organisation du code autour d'objets du monde réel
- Regroupement des données et des comportements
- Réutilisabilité et maintenabilité du code

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les classes et les objets

Définition analogique

Une classe = Un moule à gâteau (le plan, la recette)

Un objet = Un gâteau fait avec ce moule (une instance concrète)

```
class Chien:
    """Classe représentant un chien"""
    pass  # Pour l'instant, notre classe est vide
```

Conventions importantes :

- Nom de classe en **PascalCase** (première lettre en majuscule)
- **Docstring** pour décrire la classe
- **Indentation** obligatoire pour le contenu

Création d'objets (instances)

```
class Chien:
    pass

# Création de 3 objets différents à partir de la même classe
rex = Chien()
bella = Chien()
max_chien = Chien()

print(type(rex))  # <class '__main__.Chien'>
print(rex == bella)  # False - ce sont deux objets différents
```

Points clés

- Une classe peut créer un nombre **illimité** d'objets
- Chaque objet est **unique** en mémoire
- La classe définit la **structure**, l'objet contient les **données**

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Le constructeur `__init__` et `self`

Le constructeur `__init__`

```python
class Chien:
    """Classe représentant un chien avec ses caractéristiques"""

    def __init__(self, nom, race, age):
        """Constructeur de la classe Chien"""
        self.nom = nom      # Attribut d'instance
        self.race = race
        self.age = age
```

`self` représente l'**instance elle-même** (l'objet en cours de création/manipulation)

Création d'objets avec données

```
# Création d'objets avec des données spécifiques
rex = Chien("Rex", "Berger Allemand", 3)
bella = Chien("Bella", "Golden Retriever", 5)

# Accès aux attributs
print(rex.nom)    # "Rex"
print(bella.race) # "Golden Retriever"
```

- `__init__` s'exécute **automatiquement** à la création de l'objet
- `self` est automatiquement passé par Python (pas besoin de l'écrire lors de l'appel)
- Les attributs (`self.nom`, etc.) sont **propres à chaque instance**

Visualisation de `self`

```
# Ce qui se passe en coulisses :
# rex = Chien("Rex", "Berger Allemand", 3)
#
# Python fait automatiquement :
# 1. Crée un nouvel objet vide
# 2. Appelle __init__(rex, "Rex", "Berger Allemand", 3)
# 3. Retourne l'objet initialisé
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les méthodes membres

Définition

Les **méthodes** sont des fonctions définies à l'intérieur d'une classe qui définissent les **comportements** des objets.

```python
class Chien:
    def __init__(self, nom, race, age):
        self.nom = nom
        self.race = race
        self.age = age

    def aboyer(self):
        """Le chien aboie"""
        return f"{self.nom} fait : Woof! Woof!"

    def se_presenter(self):
        """Le chien se présente"""
        return f"Je m'appelle {self.nom}, je suis un {self.race} de {self.age} ans."

    def vieillir(self):
        """Le chien vieillit d'un an"""
        self.age += 1
        return f"{self.nom} a maintenant {self.age} ans."
```

Utilisation des méthodes

```
rex = Chien("Rex", "Berger Allemand", 3)

print(rex.aboyer())
# "Rex fait : Woof! Woof!"

print(rex.se_presenter())
# "Je m'appelle Rex, je suis un Berger Allemand de 3 ans."

print(rex.vieillir())
# "Rex a maintenant 4 ans."

print(rex.age)
# 4
```

Avantages des méthodes

- Le code est **logiquement organisé** (les actions du chien sont avec ses données)
- **Réutilisable** : tous les chiens peuvent aboyer
- **Modifiable** : facile de changer le comportement d'aboiement

Méthodes avec paramètres

```
class Chien:
    def __init__(self, nom, race, age):
        self.nom = nom
        self.race = race
        self.age = age

    def jouer_avec(self, autre_chien):
        """Le chien joue avec un autre chien"""
        return f"{self.nom} joue avec {autre_chien.nom} !"
```

Utilisation avec paramètres

```
rex = Chien("Rex", "Berger Allemand", 3)
bella = Chien("Bella", "Golden Retriever", 5)

print(rex.jouer_avec(bella))
# "Rex joue avec Bella !"
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

L'héritage simple

Concept de l'héritage

L'**héritage** permet de créer une nouvelle classe basée sur une classe existante, en **réutilisant** et en **étendant** ses fonctionnalités.

Analogie : C'est comme un enfant qui hérite des caractéristiques de ses parents mais peut aussi avoir ses propres spécificités.

Classe parente : Animal

```python
class Animal:
    """Classe parente représentant un animal générique"""

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def manger(self):
        return f"{self.nom} mange."

    def dormir(self):
        return f"{self.nom} dort."
```

Classe enfant : Chien

```python
class Chien(Animal):
    """Classe enfant héritant d'Animal"""

    def __init__(self, nom, age, race):
        # Appel du constructeur parent
        super().__init__(nom, age)
        self.race = race

    def aboyer(self):
        """Méthode spécifique aux chiens"""
        return f"{self.nom} fait : Woof!"
```

Classe enfant : Chat

```
class Chat(Animal):
    """Classe enfant héritant d'Animal"""

    def __init__(self, nom, age, couleur):
        super().__init__(nom, age)
        self.couleur = couleur

    def miauler(self):
        """Méthode spécifique aux chats"""
        return f"{self.nom} fait : Miaou!"
```

Utilisation de l'héritage

```
rex = Chien("Rex", 3, "Berger Allemand")
felix = Chat("Felix", 2, "Noir")

# Méthodes héritées
print(rex.manger())    # "Rex mange."
print(felix.dormir())  # "Felix dort."

# Méthodes spécifiques
print(rex.aboyer())    # "Rex fait : Woof!"
print(felix.miauler()) # "Felix fait : Miaou!"
```

Points clés de l'héritage

- `super()` permet d'appeler les méthodes de la classe parente
- Les classes enfants **héritent** de tous les attributs et méthodes du parent
- On peut **ajouter** de nouveaux attributs et méthodes spécifiques

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

L'héritage multiple

Concept

Une classe peut hériter de **plusieurs** classes parentes.

Attention : À utiliser avec précaution, peut rendre le code complexe.

```
class Nageur:
    """Capacité de nager"""

    def nager(self):
        return f"{self.nom} nage dans l'eau."

class Coureur:
    """Capacité de courir"""

    def courir(self):
        return f"{self.nom} court rapidement."
```

Classe avec héritage multiple

```
class Chien(Animal, Nageur, Coureur):
    """Un chien peut hériter de plusieurs capacités"""

    def __init__(self, nom, age, race):
        Animal.__init__(self, nom, age)
        self.race = race

    def aboyer(self):
        return f"{self.nom} fait : Woof!"
```

Utilisation de l'héritage multiple

```
rex = Chien("Rex", 3, "Labrador")

print(rex.manger())   # Hérité d'Animal
print(rex.nager())    # Hérité de Nageur
print(rex.courir())   # Hérité de Coureur
print(rex.aboyer())   # Propre à Chien
```

Ordre de résolution

Python cherche les méthodes de **gauche à droite** dans la liste des classes parentes.

```
class Chien(Animal, Nageur, Coureur):
    pass
```

Ordre de recherche : Chien → Animal → Nageur → Coureur

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Récapitulatif des concepts

Concepts clés abordés

- **Classe** : Le moule, le plan de construction
- **Objet** : Une instance concrète de la classe
- `__init__` : Le constructeur qui initialise les objets
- `self` : Référence à l'instance elle-même
- **Attributs** : Les données de l'objet
- **Méthodes** : Les comportements de l'objet
- **Héritage** : Réutilisation et extension de classes existantes

Avantages de la POO

1. **Organisation** : Code structuré autour d'entités logiques
2. **Réutilisabilité** : Une classe peut être utilisée partout
3. **Maintenabilité** : Modifications localisées dans les classes
4. **Modularité** : Chaque classe est indépendante
5. **Abstraction** : Cache la complexité interne

Pour aller plus loin

- Encapsulation (attributs privés avec `_` et `__`)
- Polymorphisme
- Classes abstraites
- Méthodes spéciales (`__str__`, `__repr__`, `__eq__`, etc.)
- Composition vs héritage

Ressources complémentaires

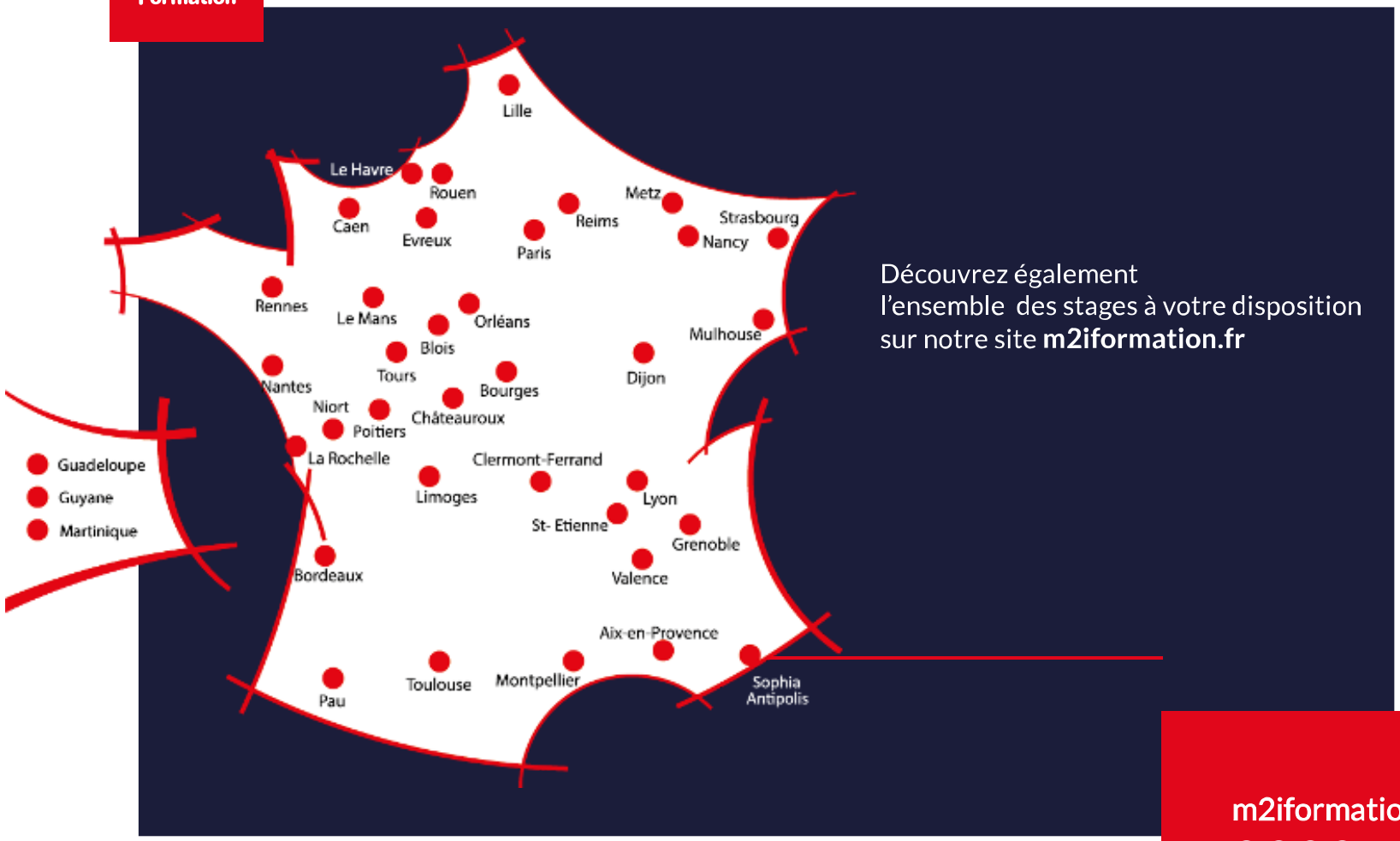
Documentation officielle Python :

- https://docs.python.org/fr/3/tutorial/classes.html

Exercices en ligne :

- https://www.practicepython.org/
- https://www.codingame.com/

--



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr