



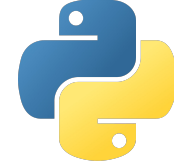
Introduction au langage de programmation Python

Séance 4 - Programmation orientée objet 2



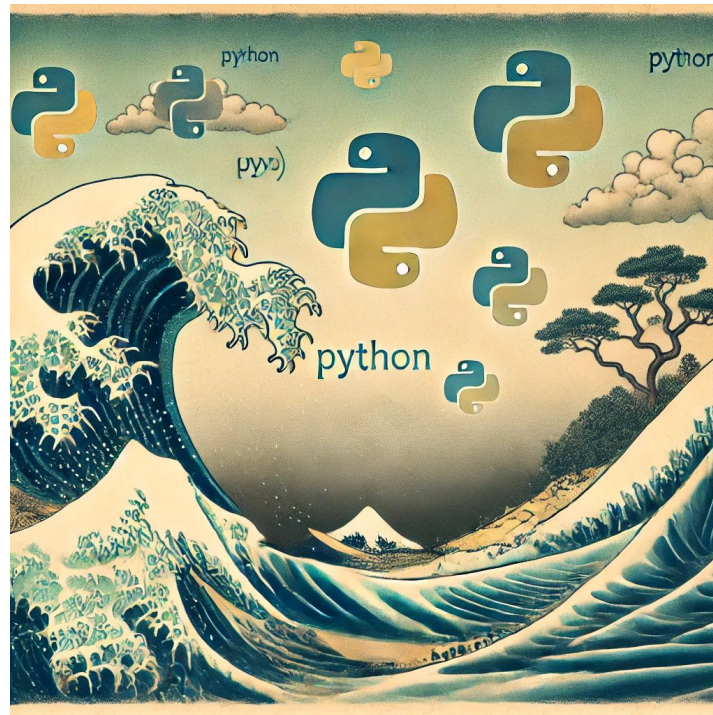
Programme du cours

| Séance | Date | Sujet |
|----------|------------------|----------------------------------|
| Séance 1 | 1er octobre 2025 | Introduction à Python |
| Séance 2 | 14 octobre 2025 | Boucles, fonctions et modules |
| Séance 3 | 28 octobre 2025 | Programmation orientée objet 1 |
| Séance 4 | 29 octobre 2025 | Programmation orientée objet 2 |
| Séance 5 | 4 novembre 2025 | Manipulation de fichiers json |
| Séance 6 | 10 novembre 2025 | Manipulation de fichiers csv |
| Séance 7 | 18 novembre 2025 | Introduction à IIIF et au prompt |



Plan de la séance 4

- 1) Révisions
 - a) Le typage des paramètres
 - b) Introduction aux classes et objets
 - c) Les méthodes et les attributs
- 2) Héritage
 - a) Qu'est-ce que l'héritage en POO ?
 - b) Pourquoi utiliser l'héritage ?
 - c) Comment fonctionne l'héritage ?
 - d) Avantages de l'héritage
- 3) Les décorateurs
 - a) Qu'est-ce qu'un décorateur en Python ?
 - b) Pourquoi utiliser des décorateurs ?
 - c) Comment fonctionnent les décorateurs ?
 - d) Gérer les fonctions avec `*args` et `**kwargs`



Typage des paramètres en python : Dynamique mais flexible

Python étant un langage **dynamiquement typé**, vous **ne pouvez pas forcer un type strictement**. Cependant, il existe plusieurs techniques pour **indiquer** ou **vérifier** les types des paramètres :

- **Le typage avec annotations**

- **Annotations de type** sont utilisées pour **indiquer** les types des paramètres et des retours de fonction.
- **Non contraignant** : Python **ne vérifie pas** le type à l'exécution. C'est simplement une aide à la documentation et pour les outils de vérification statique.

Exemple typage avec annotations :

```
def get_labels(labels_file: str) -> str:  
    # Logique de la fonction  
    return labels_file
```

- `labels_file: str` indique que le paramètre devrait être une chaîne de caractères.
- `-> str` indique que la fonction doit retourner une chaîne.

Typage des paramètres en python : Comment contraindre ?

Vérification dynamique

- Utilisez `isinstance()` pour **vérifier manuellement** le type et forcer un comportement en cas de type invalide.

Bibliothèques de validation

- Pour des **projets plus complexes**, des bibliothèques comme `Pydantic` peuvent être utilisées pour **valider les types automatiquement**.

Exemple de vérification dynamique :

```
def get_labels(labels_file: str) -> str:
    # Vérifier si le format est le bon,
    sinon lever une erreur
    if not isinstance(labels_file, str):
        raise TypeError("labels_file doit
être une chaîne de caractères")
    return labels_file
```

A RETENIR

- Les annotations de type **indiquent** les types mais ne les forcent pas.
- Pour forcer les types, utilisez **des vérifications manuelles** ou des **bibliothèques** comme `Pydantic` pour valider automatiquement les types.

Gérer les exceptions

Pourquoi gérer les exceptions ?

- **Exceptions** : Erreurs qui surviennent pendant l'exécution d'un programme, perturbant son fonctionnement normal.
- **Gestion des exceptions** : Permet de contrôler le flux du programme en cas d'erreurs inattendues.
- **Objectif** : Éviter les arrêts brutaux du programme et fournir des messages d'erreur clairs.

Lever une exception avec **raise**

- **Utilisation de **raise**** : Permet de déclencher manuellement une exception.
- **Syntaxe** : `raise NomDeLErreur("Message d'erreur")`
- **Exemple**

```
def division(a, b):  
    if b == 0:  
        raise ZeroDivisionError("La division par  
zéro n'est pas autorisée.")  
    return a / b
```

Gérer les exceptions avec **try** et **except**

- **Bloc **try**** : Contient le code qui pourrait provoquer une exception.
- **Bloc **except**** : Contient le code à exécuter si une exception survient.
- **Syntaxe** :

```
try:  
    # Code susceptible de provoquer  
    une exception  
except NomDeLErreur:  
    # Code à exécuter en cas  
    d'exception
```

Vous pouvez utiliser les exceptions natives ou créer vos propres exceptions:

- **Définir une classe qui hérite de **Exception**** :

```
class MonErreur(Exception):  
    pass
```
- **Lever l'exception personnalisée** :

```
raise MonErreur("Votre message  
d'erreur")
```

La Programmation orientée objet : Rappel

Les éléments clés d'une classe

Une classe est un modèle ou plan qui définit les caractéristiques et comportements communs à un ensemble d'objets.

Attributs

- **Description** : Variables qui représentent les **données** ou l'**état** des objets créés à partir de la classe.
- **Types d'attributs** :
 - **Attributs d'instance** : Propres à chaque objet (instance) de la classe.
 - **Attributs de classe** : Partagés par toutes les instances de la classe.

Schéma de la structure basique d'une classe en Python

```
class MaClasse:
    # Attributs de classe (optionnels)
    attribut_de_classe = valeur_par_defaut

    def __init__(self, param1, param2):
        # Initialisation des attributs
d'instance
    self.attribut1 = param1
    self.attribut2 = param2

    # Méthode d'instance
    def ma_methode(self, parametre):
        # Actions effectuées par la méthode
        pass
```


La Programmation orientée objet : Rappel

Méthodes

- **Description** : Fonctions définies à l'intérieur de la classe qui décrivent les **comportements** ou les **actions** que les objets peuvent effectuer.
- **Caractéristiques** :
 - Opèrent généralement sur les attributs de l'objet.
 - Le premier paramètre est généralement **self**, qui fait référence à l'instance courante.

Instance (Objet)

- **Description** : Représentation concrète de la classe créée grâce au processus d'**instanciation**.
- **Instanciation** : Création d'un nouvel objet à partir de la classe en appelant la classe comme une fonction.

Schéma de la structure basique d'une classe en Python

```
class MaClasse:
    # Attributs de classe (optionnels)
    attribut_de_classe = valeur_par_defaut

    def __init__(self, param1, param2):
        # Initialisation des attributs
d'instance
    self.attribut1 = param1
    self.attribut2 = param2

    # Méthode d'instance
    def ma_methode(self, parametre):
        # Actions effectuées par la méthode
        pass
```


P00 - Appeler une méthode et utiliser return

Une **méthode** est une fonction définie à l'intérieur d'une **classe**. Pour appeler une méthode **depuis une autre méthode** de la même classe :

→ on utilise `self.nom_de_la_methode()`.

`self` représente **l'instance actuelle** de la classe (l'objet sur lequel on agit). Une méthode peut donc :

- exécuter d'autres méthodes de la classe,
- ou combiner leurs résultats.

Exemple :

`class Exemple:`

```
    def dire_bonjour(self):  
        print("Bonjour !")
```

```
    def saluer_utilisateur(self):  
        print("Je vais dire bonjour :)")  
        self.dire_bonjour() # appel de la méthode depuis la  
même classe
```

Utilisation de `print()` ou `return`

`print()` affiche une information à l'écran (effet visuel uniquement).

`return` renvoie une **valeur** au code appelant, qui peut ensuite être :

- stockée dans une variable,
- utilisée dans un calcul,
- ou affichée plus tard.

Si une méthode ne contient pas de `return`, elle renvoie automatiquement `None`.

Bonne pratique :

- utiliser `print()` pour montrer,
- utiliser `return` pour transmettre des données.

Différence entre attributs de classe et attributs d'instance

| Caractéristique | Attributs de Classe | Attributs d'Instance |
|-----------------|--|---|
| Définition | Définis dans la classe, en dehors des méthodes | Définis dans le constructeur <code>__init__</code> avec <code>self</code> |
| Portée | Partagés par toutes les instances | Propres à chaque instance |
| Accès | <code>Classe.attribut</code> ou <code>instance.attribut</code> | <code>instance.attribut</code> |
| Modification | Affecte toutes les instances (si modifié via la classe) | N'affecte que l'instance concernée |
| Utilisation | Pour des valeurs communes ou des constantes | Pour des valeurs spécifiques à chaque instance |

L'héritage en Programmation Orientée Objet

Définition

L'héritage est un mécanisme en POO qui permet à une classe (appelée **classe enfant** ou **sous-classe**) d'**hériter** des attributs et méthodes d'une autre classe (appelée **classe parente** ou **super-classe**).

Exemple :

```
class ClasseEnfant(ClasseParent):  
  
    pass
```

Intérêts

Réutilisation du code :

- Évite la duplication de code en permettant aux classes enfants d'utiliser les attributs et méthodes définis dans les classes parentes.
- Facilite la maintenance du code ; les modifications apportées à la classe parente se répercutent sur les classes enfants.

Organisation hiérarchique des classes :

- Permet de modéliser des relations du type "est un" ("is a" en anglais) entre les classes.
- Exemple : Un **Philosophe** est un **PersonnageHistorique**.

Extensibilité du code :

- Facilite l'ajout de nouvelles fonctionnalités en créant de nouvelles classes enfants qui héritent des classes existantes.
- Permet d'ajouter ou de modifier des comportements spécifiques dans les classes enfants sans affecter les classes parentes.

Construire une classe d'héritage avec `super()`

Le mot-clé `super()`

- **Utilité de `super()` :**
 - Permet d'appeler les méthodes de la **classe parente** depuis la classe enfant.
 - Facilite l'initialisation correcte des **attributs hérités**.

Rôle :

- Initialise les attributs définis dans la classe parente.
- Doit être appelé **avant** d'initialiser les attributs spécifiques de la classe enfant.

Syntaxe :

```
super().__init__(param1, param2, ...)
```

- Appelle le constructeur (`__init__`) de la classe parente.
- **Ne pas inclure `self`** dans les arguments de `super().__init__()`.

Construire une classe d'héritage avec `super()`

- Structure d'une classe enfant avec `super()`

```
class ClasseEnfant(ClasseParente):  
    def __init__(self, attP1, attP2, attE1,  
attE2):  
    super().__init__(attP1, attP2)  
    self.attE1 = attE1  
    self.attE2 = attE2
```

Explication

`ClasseEnfant(ClasseParente) :`

- Indique que `ClasseEnfant` hérite de `ClasseParente`.

Méthode `__init__` de la classe enfant :

- **Paramètres :**
 - `attP1`, `attP2` : Attributs hérités de la classe parente.
 - `attE1`, `attE2` : Attributs spécifiques à la classe enfant.
- **Appel à `super().__init__()` :**
 - Initialise les attributs de la classe parente avec `attP1` et `attP2`.
- **Initialisation des attributs de la classe enfant :**
 - `self.attE1 = attE1`
 - `self.attE2 = attE2`

Appeler les méthodes de la classe parente

Utilisation de `super().methode_parente()` :

- Permet d'appeler une méthode de la classe parente depuis la classe enfant.

Pourquoi utiliser `super()` ?

1. **Réutiliser le comportement de la classe parente :**
 - Profiter des fonctionnalités déjà définies pour ne pas réécrire le code existant.
2. **Étendre ou modifier le comportement :**
 - Ajouter du code supplémentaire ou personnaliser le comportement pour répondre aux besoins spécifiques de la classe enfant.

Exemple

```
class ClassePersonne:
    def __init__(self, nom, prenom):
        self.nom = nom
        self.prenom = prenom

    def sePresenter(self):
        return f'Je suis {self.prenom} {self.nom}'

class JeMePresente(ClassePersonne):
    def __init__(self, nom, prenom, age):
        super().__init__(nom, prenom)
        self.age = age

    def sePresenter(self):
        super().sePresenter() # Appelle la
méthode de la classe parente
        return f"Je m'appelle {self.prenom} {self.nom} et j'ai {self.age} ans" # Code
supplémentaire spécifique à la classe enfant
```

Les décorateurs

Qu'est-ce qu'un décorateur ?

- Un **décorateur** est une fonction qui prend en entrée une autre fonction et retourne une nouvelle fonction avec un comportement modifié ou étendu.
- Il permet de modifier dynamiquement le comportement d'une fonction **sans changer son code source**.

Syntaxe avec @ :

```
def mon_decorateur(fonction):  
    def nouvelle_fonction():  
        print("Avant l'exécution de la  
fonction")  
        resultat = fonction()  
        print("Après l'exécution de la  
fonction")  
        return resultat  
    return nouvelle_fonction
```

```
@mon_decorateur  
def dire_bonjour():  
    print("Bonjour !")
```


Pourquoi utiliser des décorateurs ?

Ajouter des fonctionnalités sans modifier le code

source :

- Permet d'enrichir une fonction existante en y ajoutant des fonctionnalités supplémentaires.
- Idéal pour **respecter le principe Open/Closed** : les entités logicielles doivent être ouvertes à l'extension mais fermées à la modification.

Séparation des préoccupations :

- Permettent de **séparer le code métier** du code technique (par exemple, la journalisation, la gestion des exceptions, la vérification des permissions).
- Améliorent la **lisibilité** et la **clarté** du code.

Réutilisation du code :

1. Les décorateurs favorisent la **réutilisation** en permettant d'appliquer la même fonctionnalité à plusieurs fonctions sans dupliquer le code.
2. Facilitent la **maintenance** et l'**évolution** du code en centralisant les modifications.

Comment fonctionnent les décorateurs ?

Fonctionnement

- **Le décorateur enveloppe la fonction originale :**
 - Il peut ajouter du **code avant et après** l'exécution de la fonction cible.
- **Définition du décorateur :**
 - Un décorateur est une **fonction** qui accepte une fonction en argument et retourne une nouvelle fonction avec un comportement modifié.

Application du décorateur

- **Utilisation du symbole @ :**
 - Le décorateur est appliqué à une fonction cible en utilisant `@nom_du_décorateur` juste au-dessus de la définition de la fonction.

Exécution

- Lorsque `ma_fonction()` est appelée, c'est en réalité `fonction_modifiee()` qui est exécutée, avec le code supplémentaire.
- Le **code supplémentaire** autour de la fonction originale est exécuté en premier, puis la fonction d'origine est appelée.

Exemple

```
def mon_decorateur(fonction):  
    def nouvelle_fonction(*args, **kwargs):  
        # Code avant  
        resultat = fonction(*args, **kwargs)  
        # Code après  
        return resultat  
    return nouvelle_fonction
```

Gérer les fonctions avec `*args` et `**kwargs`

Qu'est ce que `*args` et `**kwargs` ?

`*args` :

- `*args` permet de passer un nombre variable d'arguments **positionnels** à une fonction.
- Tous les arguments supplémentaires sont capturés dans un **tuple**
-

`**kwargs` :

- `**kwargs` permet de passer un nombre variable d'arguments **nommés** (sous forme de paires clé-valeur).
- Ces arguments sont capturés dans un **dictionnaire**.

Utilisation dans les décorateurs :

- Assure la compatibilité avec toutes les fonctions, quelles que soient leurs signatures.

Exemple :

- Utilisation de `*args` :

```
def somme(*args):  
    return sum(args)
```

```
print(somme(1, 2, 3)) # Résultat : 6  
print(somme(4, 5))   # Résultat : 9
```

- Utilisation de `**kwargs` :

```
def afficher_infos(**kwargs):  
    for cle, valeur in kwargs.items():  
        print(f"{cle}: {valeur}")
```

```
afficher_infos(nom="Alice", age=25,  
ville="Paris")
```

Notions clés à retenir

Héritage

- Permet à une classe enfant de **hériter** des attributs et méthodes d'une classe parente.
- Établit une **hiérarchie** entre les classes pour une meilleure organisation du code.

Utilisation de **super()**

- Appelle les méthodes de la classe parente.
- Assure une **initialisation correcte** des attributs hérités.

Décorateurs

- Fonction qui **modifie ou étend** le comportement d'une autre fonction sans en changer le code source.

Pourquoi les Utiliser ?

- **Ajouter des fonctionnalités** de manière modulaire.
- Favoriser la **réutilisation du code** et la **séparation des préoccupations**.

args** et *kwargs**

- **Flexibilité des fonctions**
 - ***args** : accepte un nombre variable d'arguments positionnels.
 - ****kwargs** : accepte un nombre variable d'arguments nommés.

Pourquoi les utiliser ?

- **Rendre les fonctions et méthodes polyvalentes**, capables de gérer différents types et nombres d'arguments.
- **Assurer la compatibilité** des décorateurs avec toutes les fonctions, quelles que soient leurs signatures.