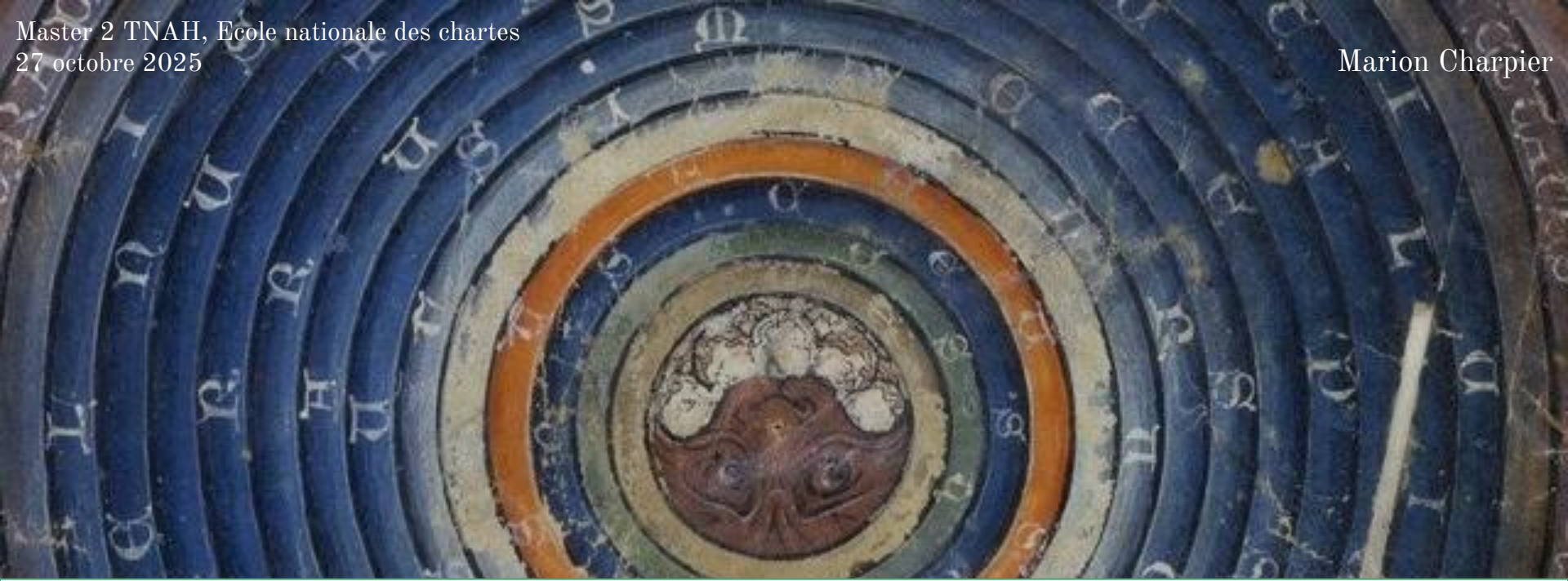
Master 2 TNAH, Ecole nationale des chartes
27 octobre 2025

Marion Charpier



Introduction au langage de programmation Python

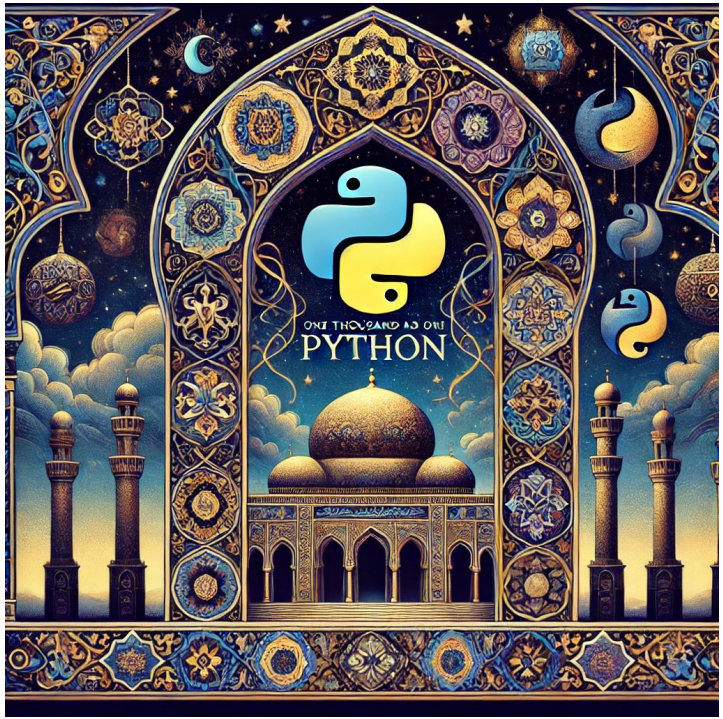
Séance 3 - Programmation orientée objet 1

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 3

1) Révisions :
   a) Commentaire de code
   b) Fonctions
   c) Modules
2) Introduction aux classes et objets
   a) Qu'est ce que la POO?
   b) Les classes
   c) Les objets
3) Les méthodes et les attributs
   a) Les attributs d'instance
   b) Les attributs de classe
   c) La méthode "__init__" et le paramètre "self"



Commenter son code

Améliorer la lisibilité et la compréhension

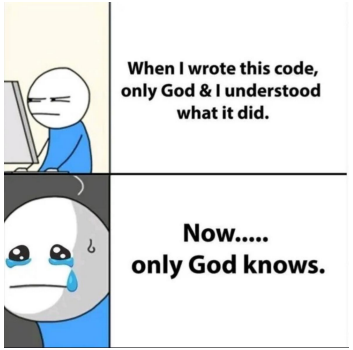
- **Clarté du code** : Les commentaires aident à expliquer le **but** et la **logique** derrière des segments de code, ce qui facilite la compréhension pour quiconque lit le code, y compris vous-même.
- **Complexité réduite** : Dans des projets complexes, comme l'analyse de données historiques, les commentaires peuvent simplifier la compréhension de processus compliqués ou d'algorithmes spécifiques.

Faciliter la maintenance et la collaboration

- **Maintenance simplifiée** : Les commentaires permettent de comprendre rapidement le fonctionnement du code, ce qui est essentiel lors de la mise à jour ou de la correction de bugs.
- **Collaboration efficace** : Dans un environnement de travail collaboratif, les commentaires aident les autres membres de l'équipe à comprendre votre logique et à contribuer plus facilement au projet.

Documentation pour soi-même et pour les autres

- **Mémoire personnelle** : Les commentaires servent de rappel sur les décisions prises et les raisons derrière certaines implémentations.
- **Transmission de connaissances** : Ils sont un moyen de partager des informations importantes avec les futurs développeurs ou chercheurs qui travailleront sur le code.



Commentaires globaux (docstrings)

Les docstrings sont des chaînes de documentation placées au début des modules, classes ou fonctions pour expliquer leur utilisation.

Documentation des modules, classes et fonctions

- **Modules :** Fournissent une vue d'ensemble du module et de son utilité dans le projet.
- **Classes :** Expliquent le rôle de la classe, ses attributs et ses méthodes principales.
- **Fonctions :** Décrivent le but de la fonction, ses paramètres, ses valeurs de retour et ses exceptions éventuelles.

Utilisation de triple guillemets `""" ... """`

Les docstrings sont encadrées par trois guillemets simples ou doubles, ce qui permet d'écrire des commentaires sur plusieurs lignes.

Conventions de documentation (PEP 257)

La **PEP 257** est un document officiel qui définit les conventions pour les docstrings en Python.

Principales recommandations :

- La première ligne doit être un résumé concis.
- Sauter une ligne entre le résumé et la description détaillée.
- Utiliser des verbes à l'infinitif pour les fonctions ("Calculer", "Extraire") et au présent pour les modules/classes ("Contient", "Gère").

```
def perspective_transformation(img_file):
        """
        This function applies blabla
        :param parameter_1:
        - Type: str
        - Description: Blabla

        :return:
        - Type: numpy.ndarray
        - Description: Blabla
"""
# My function starts here
```

Commentaires en ligne

Les commentaires en ligne sont utilisés pour annoter des parties spécifiques du code, généralement des lignes ou des blocs particuliers.

Utilisation du symbole

- En Python, le symbole # est utilisé pour indiquer un commentaire en ligne.
- Tout ce qui suit le # sur la même ligne est ignoré par l'interpréteur.

Bonnes pratiques pour les commentaires en ligne

- **Pertinence** : Les commentaires doivent ajouter de la valeur et ne pas simplement répéter ce que fait le code.
- **Actualisation** : Maintenir les commentaires à jour avec le code pour éviter les incohérences.
- **Clarté** : Utiliser un langage simple et direct.

Commentaires en ligne : Bonnes pratiques

Quand et comment les utiliser

- **Explications supplémentaires** : Pour expliquer une logique complexe ou non évidente.
- **Marquages temporaires** : Pour noter des tâches à faire (`# TODO`) ou des questions (`# FIXME`).
- **Désactivation de code** : Pour commenter du code qui ne doit pas être exécuté temporairement.

Explication d'une logique complexe

```
# Open image and get dimensions to test the
transformation
img = cv2.imread(img_file)
rows, cols = img.shape[:2]
```

Marquage d'une tâche à faire

```
# TODO: Ajouter la gestion des exceptions
pour les données manquantes
def retrieve_detect_classes():
      pass
```

Désactivation de code

```
TP_img_height, TP_img_width =
cv2.imread(Path(img_file).with_suffix('').as_
posix() + '_PT' +
Path(img_file).suffix).shape[:2]
# print(f"Origal size: {img_height},
{img_width}\nNew size: {TP_img_height},
{TP_img_width}")
```

Rappel sur les fonctions

Importance des fonctions en programmation

- **Modularité** : Les fonctions permettent de décomposer un programme complexe en sous-parties plus simples et gérables. Cela facilite la compréhension et le développement du code.
- **Réutilisabilité** : Une fonction peut être appelée plusieurs fois avec des paramètres différents, évitant ainsi la duplication du code et réduisant les erreurs potentielles.
- **Clarté** : En isolant des tâches spécifiques dans des fonctions, le code devient plus lisible et plus facile à maintenir. Cela aide également à documenter le programme en montrant clairement ce que chaque partie du code est censée faire.
- **Maintenance simplifiée** : Les modifications apportées à une fonction sont automatiquement répercutées partout où la fonction est utilisée, ce qui facilite les mises à jour et la correction des bugs.
- **Collaboration** : Dans un contexte de travail en équipe, les fonctions permettent à plusieurs développeurs de travailler sur différentes parties du code sans conflit.

Structure d'une fonction en Python

Définition avec `def`

- En Python, une fonction est définie à l'aide du mot-clé `def` suivi du nom de la fonction et de parenthèses contenant éventuellement des paramètres.
- **Syntaxe de base** :

Paramètres et arguments

- **Paramètres** : Variables définies dans la déclaration de la fonction qui permettent de passer des informations à la fonction.
- **Arguments** : Valeurs réelles que vous fournissez lors de l'appel de la fonction.
- **Exemple avec paramètres** :

Valeurs de retour : différence entre `print` et `return`

- `print` affiche une valeur à l'écran.
- `return` renvoie une valeur à l'endroit où la fonction a été appelée.

Fonction anonyme (`lambda`)

Définition

Une **fonction lambda** est une **fonction anonyme** qui n'a pas de nom. Elle est utilisée pour définir des **fonctions courtes** en une seule ligne.

Syntaxe

```
lambda paramètres : expression
```

- `lambda` : Mot-clé utilisé pour définir la fonction.
- `paramètres` : Variables en entrée (peuvent être 0, 1 ou plusieurs).
- `expression` : Le calcul ou la transformation à effectuer (doit retourner un résultat).

Utilisation

- **Créer des fonctions simples** sans utiliser `def`.
- Passer une fonction comme **argument** à une autre fonction.
- Utilisation avec des fonctions de **haut niveau** telles que `map()`, `filter()`, et `sorted()`.

Syntaxe des fonctions `lambda` :

```
carré = lambda x : x ** 2
print(carré(5))  # Affiche 25
```

Utilisation avec `sorted()` :

```
etudiants = [("Alice", 15), ("Bob", 12),
("Charlie", 18)]

etudiants_tries = sorted(étudiants,
key=lambda x: x[1])

print(etudiants_tries)
# Affiche : [('Bob', 12), ('Alice', 15),
('Charlie', 18)]
```

Création de modules personnalisés

Un **module personnalisé** est un **fichier Python** (`.py`) contenant vos propres fonctions, classes et variables que vous souhaitez **réutiliser** dans d'autres scripts ou projets. Il permet de **modulariser** le code en regroupant des fonctionnalités similaires dans un fichier séparé.

Étapes pour créer un module personnalisé

1. **Créer un fichier `.py` avec le nom du module**

   - Exemple : Créez un fichier `mon_module.py`.

2. **Définir vos fonctions, classes ou variables dans ce fichier**

```
# mon_module.py

def saluer(nom):
    """Fonction qui affiche un message de salutation."""
    return f"Bonjour, {nom} !"

def addition(a, b):
    """Retourne la somme de deux nombres."""
    return a + b

PI = 3.14159
```

3. **Enregistrer le fichier** `mon_module.py` dans le même répertoire que votre script principal ou dans un dossier accessible par Python.

Importer et utiliser son module

Importer le module personnalisé dans un autre script

```
   # script_principal.py
   import mon_module

   print(mon_module.saluer("Alice"))  #
Affiche : Bonjour, Alice !
   print(mon_module.addition(5, 3))   #
Affiche : 8
   print(mon_module.PI)               #
Affiche : 3.14159
```

Utiliser `from ... import ...` pour importer des éléments spécifiques

```
   from mon_module import saluer, PI

   print(saluer("Bob"))   # Affiche :
Bonjour, Bob !
   print(PI) # Affiche : 3.14159
```

Organisation des modules

- **Modules simples :**
  - Un fichier `.py` unique.
- **Packages :**
  - Un dossier contenant plusieurs modules `.py` et un fichier `__init__.py`.
  - Permet de structurer les modules en sous-modules.

Exemple de structure de package

```
mon_package/
     __init__.py
     module1.py
     module2.py
```

Synthèse sur les modules

Organisation du code

- **Structuration** : Les modules permettent de structurer le code en le divisant en morceaux logiques. Cela rend le code plus facile à lire et à comprendre.
- **Gestion** : Dans des projets complexes, il est plus facile de gérer plusieurs petits fichiers que de travailler avec un seul grand fichier.
- **Isolation** : Les modules permettent d'isoler des fonctionnalités spécifiques, réduisant ainsi les risques d'interférences entre différentes parties du code.

Réutilisabilité

- **Réemploi du code** : Une fois qu'un module est écrit et testé, il peut être réutilisé dans plusieurs programmes sans avoir à réécrire le code.
- **Partage** : Les modules peuvent être partagés entre différents projets ou même avec d'autres développeurs, facilitant la collaboration.
- **Maintenance** : Les corrections de bugs ou les améliorations apportées à un module sont automatiquement disponibles partout où le module est utilisé.

Différence entre module et package

- **Module** : Un module est un seul fichier Python (.py) qui contient du code (fonctions, classes, variables).
- **Package** : Un package est un ensemble de modules regroupés dans un répertoire. Un package est identifié par la présence d'un fichier __init__.py dans le répertoire.
- **Hiérarchie** : Les packages permettent de créer une hiérarchie de modules, facilitant l'organisation de projets complexes.

Exemple :

```
mon_projet/
├── __init__.py
├── analyse/
│   ├── __init__.py
│   ├── statistique.py
│   └── texte.py
└── visualisation/
        ├── __init__.py
        └── graphique.py
```

Le fichier `__init__.py`

Qu'est-ce que le fichier `__init__.py` ?

- **Définition** : `__init__.py` est un fichier spécial en Python qui indique à l'interpréteur qu'un répertoire doit être traité comme un **package**.
- **Emplacement** : Il se trouve à la racine du package et éventuellement dans les sous-packages.

Rôle et importance du fichier `__init__.py`

- **Identification de package** : Sans ce fichier, Python ne reconnaît pas le répertoire comme un package, surtout dans les versions antérieures à Python 3.3.
- **Initialisation du package** : Le fichier peut être utilisé pour exécuter du code d'initialisation lors de l'importation du package.
- **Contrôle des importations** : Il permet de spécifier quels modules ou objets sont accessibles lors de l'importation du package.

Utilisations courantes du fichier `__init__.py`

Rendre le package importable

- **Package vide** : Un fichier `__init__.py` vide est suffisant pour indiquer que le répertoire est un package.

Bonnes pratiques avec `__init__.py`

- **Clarté et simplicité** : Gardez le fichier __init__.py aussi simple que possible pour faciliter la maintenance.
- **Éviter le code complexe** : Ne placez pas de logique complexe ou de traitements lourds dans `__init__.py` pour ne pas ralentir l'importation du package.

Initialiser des variables ou des configurations

- **Code d'initialisation** : Vous pouvez inclure du code pour initialiser des variables globales ou des configurations spécifiques au package.

```
# __init__.py
from .module1 import fonction1
from .module2 import classe1
```

Définir `__all__` pour contrôler les importations

- **Contrôle des symboles exportés** : En définissant la liste __all__, vous spécifiez les modules ou objets qui seront importés lors d'un `from package import *`.

```
# __init__.py
__all__ = ['module1', 'module2']
```

Importation de modules

L'importation de modules permet d'accéder aux fonctions, classes et variables définies dans d'autres fichiers Python.

Importation standard

- **Syntaxe** : `import module`
- **Description** : Cette syntaxe importe le module entier. Pour accéder aux éléments du module, vous devez utiliser la notation pointée `module.element`
- **Avantages** :
  - Évite les conflits de noms, car les éléments du module sont accessibles via le nom du module.
  - Rend le code plus clair en indiquant explicitement de quel module provient une fonction.

Importation spécifique

- **Syntaxe** : `from module import element`
- **Description** : Cette syntaxe importe un ou plusieurs éléments spécifiques d'un module, ce qui vous permet de les utiliser directement sans préfixe.

Exemple :

```
# Importation d'une fonction spécifique
from mathématiques import addition

resultat = addition(5, 3)
print(resultat)  # Affiche 8
```

Importation de modules multiple

- **Syntaxe** : `from module import element1, element2, element3`
- **Avantages** :
  - Simplifie le code en évitant de répéter le nom du module.
  - Peut améliorer la lisibilité si vous utilisez fréquemment certaines fonctions spécifiques.

Attention :

- Risque de conflit de noms si différentes fonctions du même nom sont importées de modules différents.
- Il est important de s'assurer que les noms des fonctions importées sont uniques dans le contexte du script.

Utilisation d'alias

- **Syntaxe** :
  - Pour le module entier : `import module as alias`
  - Pour un élément spécifique : `from module import element as alias`
- **Description** : Les alias permettent de renommer un module ou un élément importé pour simplifier son utilisation ou éviter des conflits de noms.
- **Avantages** :
  - Réduit la longueur des noms de modules ou de fonctions, rendant le code plus concis.
  - Aide à éviter les conflits de noms en renommant les éléments importés.

Utilisation courante :

- Certains modules standard ou bibliothèques populaires ont des alias conventionnels.
  - **NumPy** : `import numpy as np`
  - **Pandas** : `import pandas as pd`
  - **Matplotlib** : `import matplotlib.pyplot as plt`

La Programmation Orientée Objet

La **programmation orientée objet (POO)** est un paradigme de programmation où le code est structuré autour d'**objets**. Un **objet** est une entité qui combine des **données** (appelées **attributs**) et des **actions** (appelées **méthodes**) qui peuvent être effectuées sur ces données.

Intérêt de la POO

La POO est essentielle pour plusieurs raisons, notamment pour la modélisation, l'organisation et la réutilisation du code.

- **Attributs** : Ce sont les informations ou caractéristiques que l'objet possède. Par exemple, pour un objet de type "Chien", les attributs pourraient être "nom", "race" et "âge". Ils définissent l'état de l'objet.
- **Méthodes** : Ce sont des fonctions associées à l'objet, qui décrivent les actions que cet objet peut réaliser. Pour l'objet "Chien", une méthode pourrait être "aboyer" ou "courir". Elles permettent d'interagir avec ou de modifier les attributs de l'objet.

Avantages de la programmation orientée objet

- **Modularité** : Le code est organisé en petites unités (classes) qui peuvent être développées et testées indépendamment.
- **Réutilisation du code** : Les classes peuvent être réutilisées dans différents programmes, ce qui économise du temps et des ressources.
- **Facilité de maintenance** : Les modifications apportées à une classe se répercutent sur toutes ses instances, simplifiant les mises à jour.

Organisation et réutilisation du code

En structurant le code autour de classes et d'objets, il devient plus facile de :

- **Gérer la complexité** : En découpant le programme en morceaux logiques.
- **Collaborer** : Plusieurs développeurs peuvent travailler sur différentes classes sans entrer en conflit.
- **Éviter la redondance** : Réduire la duplication de code en réutilisant des classes existantes.

Les classes

Qu'est-ce qu'une classe ?

Une **classe** est un modèle ou un plan de construction pour créer des objets en programmation orientée objet. Elle définit un ensemble d'attributs (données) et de méthodes (fonctions) qui caractérisent un objet spécifique. En d'autres termes, une classe décrit **ce qu'est** un objet et **ce qu'il peut faire**.

Syntaxe

La déclaration d'une classe en Python suit une syntaxe simple

- `class` : Mot-clé pour définir une classe.
- `NomDeLaClasse` : Le nom que vous donnez à votre classe.
- `:` : Deux-points indiquant le début du bloc de code de la classe.

Corps de la classe : Indenté sous la déclaration, il contient les attributs et les méthodes de la classe.

Exemple de syntaxe simple :

```
class Personnage :
    pass
```

Conventions de nommage (CamelCase)

En Python, il est courant de suivre certaines conventions pour nommer les classes :

- **CamelCase** : Chaque mot commence par une majuscule et il n'y a pas de séparateurs (comme des underscores). Par exemple : `PersonnageHistorique`, `DocumentAncien`, `AnalyseStatistique`.

Pourquoi utiliser CamelCase ?

- **Lisibilité** : Facilite la lecture du code en distinguant clairement les classes des autres objets.
- **Conformité** : Respecter les conventions facilite la collaboration avec d'autres développeurs et la compréhension du code.

Les objets

Qu'est-ce qu'un objet (instance de classe) ?

Un **objet** est une **instance concrète** d'une **classe**. En programmation orientée objet, la classe sert de modèle ou de plan pour créer des objets, tandis que l'objet est une réalisation spécifique de ce modèle.

- **Objet** : Entité autonome en mémoire qui possède des **attributs** (données) et des **méthodes** (comportements) définis par sa classe.
- **Instance** : Terme utilisé pour décrire un objet créé à partir d'une classe. On dit que l'objet est une instance de cette classe.

La relation entre une classe et un objet est fondamentale en POO :

- **Classe** : Décrit **ce que** l'objet est, en définissant ses attributs et ses méthodes. Elle est abstraite et ne consomme pas de ressources mémoire significatives.
- **Objet** : Est une **réalisation concrète** de la classe. Il a des valeurs spécifiques pour ses attributs et peut exécuter les méthodes définies par la classe. Chaque objet est indépendant et possède sa propre existence en mémoire.

Processus d'instanciation

1. **Déclaration de la classe** : On définit le modèle avec ses attributs et ses méthodes.
2. **Création de l'objet** : On instancie la classe pour créer un objet réel en mémoire.
3. **Utilisation de l'objet** : On peut accéder à ses attributs et appeler ses méthodes.

Exemple :

```
class Personnage:
      pass
Zelda = Personnage()
print(Zelda)
```

Les attributs de classe

Qu'est-ce qu'un attribut de classe ?

En programmation orientée objet, un **attribut de classe** est une variable définie directement dans la classe. Les attributs de classe sont **partagés par toutes les instances** de la classe. Cela signifie que si vous modifiez un attribut de classe, la modification sera reflétée dans toutes les instances, sauf si une instance a un attribut d'instance du même nom.

- **Attribut de classe** : Variable partagée par toutes les instances de la classe.
- **Portée** : Accessible via la classe elle-même ou via ses instances.
- **Utilisation typique** : Pour des valeurs communes à toutes les instances ou pour des constantes liées à la classe.

Exemple :

```
class Personnage:
        fonction = "princesse"
        royaume = "Hyrule"
        pouvoir = "prêtresse"
Zelda = Personnage()
print(Zelda.fonction, Zelda.royaume, Zelda.pouvoir)
print(f"Zelda est la {Zelda.fonction} du royaume
d'{Zelda.royaume}, où elle est {Zelda.pouvoir}")
```

Caractéristiques des Attributs de Classe

- **Partage des Valeurs** :
  - Les attributs de classe sont communs à toutes les instances.
  - Toute modification de ces attributs via la classe affectera toutes les instances.
- **Accès** :
  - Peut être accédé via la classe : `Personnage.fonction`
  - Peut être accédé via une instance : `Zelda.fonction`
- **Modification** :
  - Si vous modifiez l'attribut de classe via la classe, toutes les instances refléteront ce changement.
  - Si vous modifiez l'attribut via une instance, vous créez un **attribut d'instance** du même nom pour cette instance uniquement.

Les attributs d'instance

Qu'est-ce qu'un attribut d'instance ?

- **Définition** : Un **attribut d'instance** est une variable qui appartient à une **instance spécifique** d'une classe. Chaque objet créé (instance) possède sa propre copie de ces attributs, permettant à chaque objet d'avoir des valeurs uniques et indépendantes des autres instances.
- **Création** : Les attributs d'instance sont généralement définis dans la méthode spéciale `__init__` (le constructeur) en utilisant le mot-clé `self`.
- **Accès** : On y accède via l'objet (instance) lui-même, par exemple `objet.attribut`.

Exemple :

```
class Personnage:
     def __init__(self, fonction, royaume,
pouvoir):
          self.fonction = fonction
          self.royaume = royaume
          self.pouvoir = pouvoir
```

Avantages des attributs d'instance

- **Personnalisation** : Permettent à chaque objet d'avoir des données spécifiques.
- **Isolation des Données** : Les modifications des attributs d'une instance n'affectent pas les autres instances.
- **Flexibilité** : Facilité de créer de nouveaux objets avec des caractéristiques différentes sans modifier la classe.

La méthode `__init__` et le paramètre `self`

Qu'est-ce que la méthode `__init__` ?

- **Méthode Spéciale** : `__init__` est une méthode spéciale en Python appelée **constructeur**.
- **Initialisation des Instances** : Elle est automatiquement appelée lors de la création d'une nouvelle instance d'une classe.
- **Rôle Principal** : Initialiser les **attributs d'instance** avec des valeurs spécifiques fournies lors de l'instanciation.

Syntaxe générale :

```
class MaClasse :
def __init__(self, paramètre1, paramètre2,
...): self.attribut1 = paramètre1
self.attribut2 = paramètre2 ...
```

Le paramètre `self`

- **Référence à l'Instance** : `self` fait référence à l'instance **actuelle** de la classe.
- **Accès aux Attributs et Méthodes** : Permet d'accéder aux attributs et méthodes de l'objet à l'intérieur de la classe.
- **Convention de Nom** : Bien que le nom `self` soit une convention, il est fortement recommandé de le respecter pour la lisibilité du code.

Pourquoi `self` est-il nécessaire ?

- **Distinction des Contextes** : Il différencie les **attributs d'instance** des variables locales ou des attributs de classe.
- **Accès aux Données de l'Instance** : Sans `self`, il serait impossible de modifier l'état de l'instance depuis ses méthodes.

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 __init__ avec self
Portée	Partagés par toutes les instances	Propres à chaque instance
Accès	`Classe.attribut` ou `instance.attribut`	`instance.attribut`
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

Notions clés à retenir

Les classes

- Définissent des **modèles** pour créer des objets.
- Agissent comme des plans ou des prototypes pour les objets.

Les objets

- Sont des **instances de classes**.
- Possèdent des **attributs** (données) et des **méthodes** (comportements) spécifiques.

Attributs

- **Attributs de classe** :
  - **Partagés par toutes les instances** de la classe.
  - Définis directement dans la classe.
- **Attributs d'instance** :
  - **Propres à chaque objet**.
  - Initialisés dans la méthode `__init__` avec `self`.

La méthode `__init__`

- Initialise les **attributs d'instance** lors de la création d'un objet.
- Est appelée automatiquement lors de l'instanciation.

Le paramètre `self`

- Fait référence à l'**instance elle-même**.
- Utilisé pour accéder aux **attributs** et **méthodes** de l'objet à l'intérieur de la classe.