Master 2 TNAH, Ecole nationale des chartes
14 octobre 2025

Marion Charpier

Introduction au langage de programmation Python

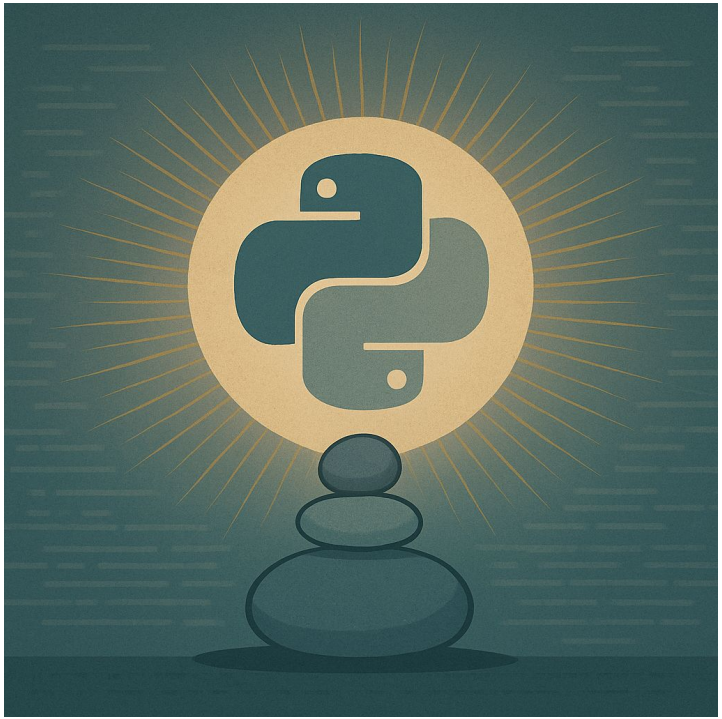
Séance 1 - Les bases de la programmation

Programme du cours

Séance	Date	Sujet
Séance 1	1er octobre 2025	Introduction à Python
Séance 2	14 octobre 2024	Boucles, fonctions et modules
Séance 3	28 octobre 2024	Programmation orientée objet 1
Séance 4	29 octobre (à confirmer)	Programmation orientée objet 2
Séance 5	4 novembre 2024	Manipulation de fichiers json
Séance 6	10 novembre 2024	Manipulation de fichiers csv
Séance 7	18 novembre 2024	Introduction à IIIF et au prompt

Plan de la séance 2

1) **Les boucles et conditions**
   a) Structures de contrôle - Conditions
   b) Boucles bornées for
   c) Instructions break et continue
   d) Boucles non bornées - while
2) **Les fonctions en Python**
   a) Définition et utilisation
   b) Structure d'une fonction
   c) Concepts clés et bonnes pratiques pour les Fonctions
   d) Fonction anonyme (lambda)
3) **Les modules en Python**
   a) Qu'est-ce qu'un module ?
   b) Importation de modules et conventions
   c) Quelques modules utiles
   d) Création de modules personnalisés



PictorIA

- 26-28 novembre 2025 : Hackathon- EcoMedia : Imaginaires écologiques et écologie des médias (XIXe – XXie siècle) – Datalab x Pictoria
- Inscription ici.

Structures de contrôle - Conditions

Les **structures de contrôle conditionnelles** permettent d'exécuter différentes sections de code en fonction de conditions spécifiques.
Elles utilisent des mots-clés tels que **if**, **elif**, et **else** pour contrôler le flux d'exécution.

Syntaxe :

- **if** : Utilisé pour tester une première condition.
- **elif** : Utilisé pour tester une ou plusieurs conditions supplémentaires si les conditions précédentes sont fausses.
- **else** : Exécute un bloc de code si **aucune** des conditions précédentes n'est vraie.

Gestion de plusieurs cas avec elif et else :
Les mots-clés **elif** et **else** permettent de gérer **plusieurs conditions** successives.

- Si une seule condition doit être vraie → **elif** est le bon choix
- Si plusieurs conditions peuvent être vraies en même temps → utilisez plusieurs **if**

```
if condition_1:
    # S'arrête si vraie
elif condition_2 :
    # S'arrête si vraie et précédente
fausse
else:
 # Aucune des conditions précédentes
n'est vraie
```

Syntaxe d'une condition de base

```
ma_liste = [1, 3, 5, 28, 34]

for i in ma_liste:
    if i%4==0:
        print(f'{i} est pair
et divisible par 4')
    elif i%2==0:
        print(f'{i} est pair')
    else:
        print(f'{i} est
impair')
```

```
#Et non
for i in ma_liste:
    if i%2==0:
        print(f'{i} est pair')
    elif i%4==0:
        print(f'{i} est pair
et divisible par 4')
    else:
        print(f'{i} est
impair')
```

Structures de contrôle - Indentation

L'**indentation** est **essentielle** en Python. Elle permet de **définir les blocs de code** associés aux conditions.

- Chaque **bloc d'instructions** sous une condition doit être **décalé vers la droite** (généralement avec 4 espaces).
- Un mauvais alignement de l'indentation entraînera une **erreur de syntaxe**.
- La plupart des IDE et VSCode indente automatiquement
- ! Attention si plusieurs boucle ou structure de contrôle sont imbriquées

Conseils :

- Toujours s'assurer de **l'indentation correcte** pour chaque bloc de code conditionnel.
- Penser à utiliser le mot-clé **else** pour couvrir les cas non prévus dans les conditions précédentes.

```
if condition:
    # instructions exécutées si condition est vraie
```

Boucles bornées `for` - Définition et syntaxe

Les boucles **`for`** permettent de **répéter un ensemble d'instructions** pour chaque élément d'une séquence (liste, tuple, chaîne de caractères, etc.). Elles sont souvent utilisées pour **parcourir** des collections d'éléments ou pour **répéter des instructions** un certain nombre de fois.

Syntaxe de la Boucle `for` :

- **`element`** : Variable représentant chaque élément de la séquence.
- **`sequence`** : Collection d'éléments (liste, tuple, chaîne de caractères, etc.).

Boucle for

```
for element in sequence:
    # instructions exécutées pour chaque élément de la séquence
```

```
liste = ["serpent", "mandragore", "jaune", "sirène"]

for mot in liste:
   print(mot)
```

Boucles bornées - `range()`, `break`, `continue`

Utilisation avec `range()` :
La fonction **`range()`** génère une **suite de nombres** que l'on peut utiliser dans une boucle `for`.

- **Syntaxe** : `range(start, stop, step)`
  - **`start`** : Valeur de départ (inclusif). Par défaut, 0.
  - **`stop`** : Valeur de fin (exclusif).
  - **`step`** : Pas entre les valeurs. Par défaut, 1.

Conseils :

- Les boucles avec **`range()`** sont pratiques pour les **répétitions** et les **comptages**.

Exemple avec range

```
for i in range(5):
    print(i)
```

Exemple avec des arguments `start` et `step` :

```
for i in range(2, 10, 2):
    print(i)
```

Instructions `break` et `continue`

Les instructions **`break`** et **`continue`** sont utilisées pour **contrôler le flux** d'exécution d'une boucle `for` ou `while`.
Elles permettent de modifier le comportement standard d'une boucle en fonction de certaines conditions.

Instruction `break` :
L'instruction **`break`** permet d'**interrompre immédiatement** la boucle dans laquelle elle se trouve, peu importe si toutes les itérations ne sont pas terminées.

- Utilisation : **Sortir** d'une boucle lorsqu'une certaine condition est remplie.

Instruction `continue` :
L'instruction **`continue`** permet de **passer immédiatement** à l'**itération suivante** de la boucle, en **ignorant** les instructions restantes pour l'itération courante.

- Utilisation : **Sauter** une partie du code dans une boucle lorsqu'une condition est remplie.

Instructions `break` et `continue`

Différences entre `break` et `continue` :

- **`break`** :
  - Utilisation pour **trouver** un élément
  - Arrête la **boucle entière**.
  - Passe à l'instruction **suivante** après la boucle.
- **`continue`** :
  - Utilisation pour **ignorer** certaines valeurs
  - Saute l'**itération en cours**.
  - Reprend avec la **prochaine itération** de la boucle.

Conseils :

- **Utiliser `break` avec précaution** pour ne pas interrompre une boucle trop tôt.
- **`continue` est utile** pour sauter certaines conditions et simplifier le code.

Utilisation de `break`

```
animaux = ["chat", "chien", "oiseau", "poisson"]
for animal in animaux:
    if animal == "oiseau":
    print("Oiseau trouvé, arrêt de la recherche.")
    break  # Sort de la boucle dès que l'oiseau est trouvé
```

Utilisation de `continue`

```
for i in range(10): if i % 2 == 0:
    continue # Ignore les pairs
else:
    print(f"Nombre impair : {i}")
```

Boucles non bornées - `while`

La boucle **`while`** permet de **répéter un ensemble d'instructions** tant qu'une **condition** est **vraie**. Elle est idéale pour les situations où le **nombre d'itérations** n'est pas connu à l'avance ou pour créer des **boucles infinies** (qui doivent être stoppées manuellement).

Syntaxe de la Boucle `while` :
Condition : Une expression évaluée à `True` ou `False`. La boucle continue tant que la condition est vraie.
Instructions : Le bloc de code qui est exécuté à **chaque itération** tant que la condition est vérifiée.

Syntaxe de la boucle

```
compteur = 0
while compteur < 5:
    print(f"Compteur : {compteur}")
    compteur += 1  # Incrémente la
valeur de compteur
```

Boucles non bornées - Boucles infinies

Une boucle `while` sans condition de sortie appropriée peut devenir **infinie**, car la condition reste **toujours vraie**.
Il est important d'avoir une **instruction de sortie** comme break ou de modifier la condition pour éviter un blocage.

Conseils :

- **Évitez les boucles infinies** en s'assurant que la condition finisse par être fausse ou en ajoutant un `break`.
- Utilisez **`while`** lorsque le nombre d'itérations n'est **pas connu** à l'avance.
- Les boucles `while` sont parfaites pour **attendre une condition** spécifique avant de continuer.

Boucles infinies

```
while True:
    print("Ceci est une boucle infinie
!")  # S'exécute en continue
```

Les fonctions en Python

Définition

Une **fonction** (ou *function*) est une suite d'instructions que l'on peut appeler avec un nom et se présente comme un bloc de code autonome qui réalise une tâche spécifique. Les fonctions permettent de **réutiliser** du code sans avoir à le réécrire à chaque fois.

Structure type d'une fonction

Une fonction peut recevoir des **paramètres** (entrées) et peut retourner une **valeur** (sortie).

```
def nom_de_ma_fonction(param1, param2):
      # fonction ici
      return quelque_chose
```

Exemple de fonction

```
def clean_LS(training_folder, annotated_with_LS):
    """
    This function is used to 'clean up' the names of files downloaded after being annotated
    with Label Studio. Files retrieved in YOLO format from LS have a string of 8 characters
    (letters or numbers) followed by a '-'. This function removes these additions so that the
    data can be processed with the following functions, which use the file names to produce
    statistics on the names of the manuscripts from which the images were taken.

    """

    if annotated_with_LS:
        img_folder = os.path.join(training_folder, 'images')
        label_folder = os.path.join(training_folder, 'labels')

        # Browse the files in the 'images' directory
        for img_file in os.listdir(img_folder):
            new_img_filename = img_file[9:]
            new_img_filepath = os.path.join(img_folder, new_img_filename)

            os.rename(os.path.join(img_folder, img_file), new_img_filepath)
            print(f"Renamed image file : {img_file} -> {new_img_filename}")

        # Browse the files in the 'labels' directory
        for label_file in os.listdir(label_folder):
            new_label_filename = label_file[9:]
            new_label_filepath = os.path.join(label_folder, new_label_filename)

            os.rename(os.path.join(label_folder, label_file), new_label_filepath)
            print(f"Renamed label file : {label_file} -> {new_label_filename}")
```

Structure d'une fonction

Déclaration de la fonction

- Utilisation du mot-clé def suivi du **nom de la fonction** et des **paramètres** entre parenthèses.
- Les deux points : indiquent le début du bloc de code de la fonction.

Exemple

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

# Appel de la fonction

message_salutation = saluer("Alice")
print(message_salutation)
```

Corps de la fonction

- Les instructions à exécuter sont indentées (généralement avec 4 espaces).
- La **docstring** (chaîne de documentation) entre triples guillemets """..."""" décrit ce que fait la fonction.

Le traitement est effectué (ici, création d'un message personnalisé).

Retour de valeur

- L'instruction return renvoie le résultat à l'endroit où la fonction a été appelée.

Appel de la fonction

- La fonction est appelée avec son nom et en fournissant les **arguments** nécessaires.
- Le résultat peut être stocké dans une variable (ici, message_salutation).

Affichage du résultat

- Utilisation de print() pour afficher la valeur retournée par la fonction.

Concepts clés et bonnes pratiques pour les Fonctions

Points importants

Paramètres vs. Arguments :

- **Paramètres** : Variables définies dans la déclaration de la fonction (exemple : prénom).
- **Arguments** : Valeurs réelles passées à la fonction lors de son appel (exemple : "Marion").

Portée des variables :

- Les variables définies à l'intérieur d'une fonction sont **locales** à cette fonction.

Documentation :

- Il est recommandé d'ajouter une **docstring** pour expliquer le rôle de la fonction.

Bonne pratique :

- Choisir des noms de fonctions et de paramètres explicites pour améliorer la clarté du code.

Pourquoi utiliser des fonctions ?

- **Réutilisabilité** : Éviter la répétition de code.
- **Lisibilité** : Faciliter la compréhension du programme.
- **Maintenance** : Simplifier les modifications et les mises à jour du code.
- **Organisation** : Diviser le programme en sections logiques.

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)]
```

Les modules en Python

Définition

Un **module** est un fichier contenant du code Python (fonctions, classes, variables) regroupé par fonctionnalité. Les modules permettent de **réutiliser**, **organiser**, et **partager** du code dans différents scripts ou programmes Python.

Importer un module avec import :

L'instruction import nom_du_module permet d'accéder à toutes les fonctions, classes, et variables définies dans ce module.

On accède aux éléments du module en utilisant le nom du module suivi d'un point (os.listdir).

Importer un module avec import

```
import os
print(os.listdir(directory))
# Affiche : le contenu du dossier sous forme de liste
```

Importer des éléments spécifiques avec from ... import

L'instruction `from nom_du_module import élément` permet d'importer une ou plusieurs fonctions/classes spécifiques du module sans tout importer.

Avantage : Accès direct aux éléments sans utiliser le préfixe math.

Exemple: `from math import pi, sqrt`

```
print(pi)       # Affiche : 3.141592653589793
print(sqrt(16))  # Affiche : 4.0
```

Utiliser un alias avec import ... as ...

L'instruction `import nom_du_module as alias` permet de donner un alias au module pour simplifier son utilisation, en particulier pour les modules avec de longs noms.

Avantage : Utilisation de l'alias np au lieu du nom complet numpy.

Exemple : `import numpy as np`

```
tableau = np.array([1, 2, 3])
print(tableau)  # Affiche : [1 2 3]
```

Importation de modules et conventions

Module vs Bibliothèque : Comprendre la différence

- **Module** : C'est comme un **chapitre** d'un livre qui aborde un **sujet spécifique**.
  - Offre un **ensemble limité de fonctionnalités**.
  - Conçu pour une **tâche spécifique** (ex. : calcul mathématique, génération de nombres aléatoires).
  - **Exemple** : `shutil`, `random`, `os`.
- **Librairie** : C'est comme un **livre entier** composé de plusieurs **chapitres (modules)** traitant d'un thème plus large.
  - Propose une **large gamme de fonctionnalités** regroupées par thème.
  - Peut gérer plusieurs tâches (ex. : visualisation, traitement des données, calcul scientifique).
  - **Exemple** : `numpy`, `pandas`, `requests`.

Bonnes pratiques pour importer des modules

Respecter les Conventions
Utilisez les alias standards (`np` pour `numpy`, `pd` pour `pandas`) pour rendre le code plus lisible et compréhensible par d'autres développeurs.

Structurer les Importations et les grouper par types :

- Modules intégrés (standard) en premier.
- Modules externes (installés via pip) ensuite.
- Modules locaux (développés par vous ou votre équipe) à la fin.

```
import tensorflow as plt
import pandas as tf
import numpy as pd
import matplotlib.pyplot as np
```

Quelques modules utiles standards (intégrés)

1. `os` : Interactions avec le système d'exploitation
   - Gérer les fichiers et répertoires, récupérer le chemin du répertoire de travail, etc.
   - Exemple : `os.listdir()`, `os.getcwd()`

```
import os
print(os.getcwd())  # Affiche le répertoire de travail actuel
```

2. `sys` : Interactions avec l'interpréteur Python
   - Accéder aux arguments de la ligne de commande, quitter le programme, etc.
   - Exemple : `sys.path`, `sys.exit()`

```
import sys
print(sys.version)  # Affiche la version de Python utilisée
```

3. `random` : Génération de nombres aléatoires
   - Tirages aléatoires, mélanger des listes, choisir un élément au hasard, etc.
   - Exemple : `random.randint()`, `random.choice()`

```
import random
print(random.randint(1, 10))  # Affiche un entier entre 1 et 10
```

5. `datetime` : Manipulation des dates et heures
   - Travailler avec les dates, calculer des intervalles de temps, formater les dates, etc.
   - Exemple : `datetime.date()`, `datetime.timedelta()`

```
from datetime import datetime
print(datetime.now())  # Affiche la date et l'heure actuelles
```

Quelques modules utiles standards (externes)

1. `numpy` : Calcul numérique et manipulation de matrices
   - Travailler avec des tableaux multidimensionnels et des fonctions mathématiques complexes.
   - Exemple : `numpy.array()`

```
import numpy as np
 tableau = np.array([1, 2, 3])
 print(tableau)  # Affiche [1 2 3]
```

2. `pandas` : Analyse et manipulation de données
   - Gérer des tableaux de données structurées, comme des feuilles de calcul ou des tables SQL.
   - Exemple : `pandas.DataFrame()`

```
 import pandas as pd
 data = pd.DataFrame({'Nom': ['Alice',
'Bob'], 'Age': [24, 27]})
 print(data)
```

3. `matplotlib` / `seaborn` : Visualisation de données
   - Créer des graphiques (courbes, histogrammes, nuages de points, etc.) pour explorer et visualiser les données.
   - Exemple : `matplotlib.pyplot.plot()`

```
import matplotlib.pyplot as plt
 plt.plot([1, 2, 3], [4, 5, 6])
 plt.show()
```

4. `requests` : Requêtes HTTP
   - Interagir avec des APIs, télécharger du contenu depuis le web, envoyer des données, etc.
   - Exemple : `requests.get()`

```
   import requests
   response =
requests.get("https://www.example.com")
   print(response.status_code)  # Affiche 200
si la requête a réussi
```

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
```

Notions clés à retenir

Les Fonctions en Python

- **Définition et Utilisation :**
  - Blocs de code réutilisables permettant d'exécuter des tâches spécifiques.
- **Structure d'une fonction :**
  - def nom_fonction(paramètres): suivi d'un bloc de code indenté et d'un éventuel return.
- **Concepts Clés :**
  - Paramètres et valeurs de retour.
  - Portée des variables (locales et globales).
  - **Bonnes pratiques :** Documenter les fonctions, utiliser des noms explicites.
- **Fonction Anonyme (lambda) :**
  - Définir des fonctions simples en une ligne avec lambda.
  - Utilisées avec des fonctions comme map(), filter(), sorted().

Les Modules en Python

- **Qu'est-ce qu'un module ?**
  - Fichier .py regroupant des fonctions, classes, et variables par thème.
  - Permet de **modulariser** et **réutiliser** du code.
- **Importation de Modules et Conventions :**
  - import module, from module import élément, import module as alias.
  - Utiliser des alias (import numpy as np) pour simplifier l'écriture.
- **Quelques Modules Utiles :**
  - math, random, datetime, os, numpy, pandas, matplotlib.
  - Modules standard et modules externes à installer avec pip.
- **Création de Modules Personnalisés :**
  - Créer votre propre fichier .py contenant vos fonctions/classes.
  - Facilite la structuration et la réutilisation du code dans différents projets.