Master 2 TNAH, Ecole nationale des chartes
8 octobre 2024

Introduction au langage de programmation Python

—

Séance 2 - Fonctions et modules

Marion Charpier

Programme du cours

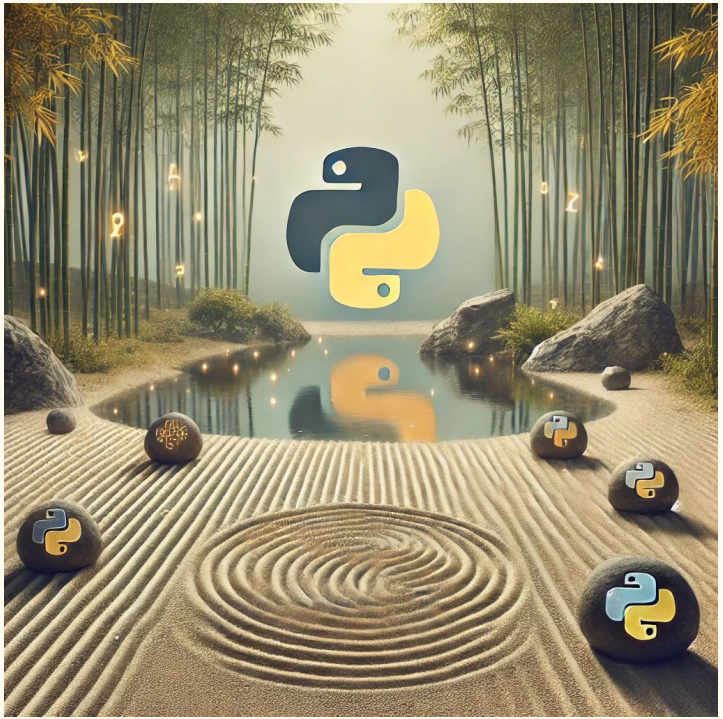
Séance	Date	Sujet
Séance 1	1er octobre 2024	Introduction à Python
Séance 2	**8 octobre 2024**	**Fonctions et Modules**
Séance 3	15 octobre 2024	Programmation orientée objet 1
Séance 4	22 octobre 2024	Programmation orientée objet 2
Séance 5	29 octobre 2024	Utilisation de l'API IIIF et manipulation de données 1
Séance 6	5 novembre 2024	Utilisation de l'API IIIF et manipulation de données 2

Plan de la séance 2

1) Révisions :
   a) Conditions : `if`, `elif`, `else`
      i) `Rappel indentation`
      ii) `Odre des conditions`
   b) Boucles : `for`, `while`
      i) `boucles for`
      ii) `break et continue`
      iii) `while`

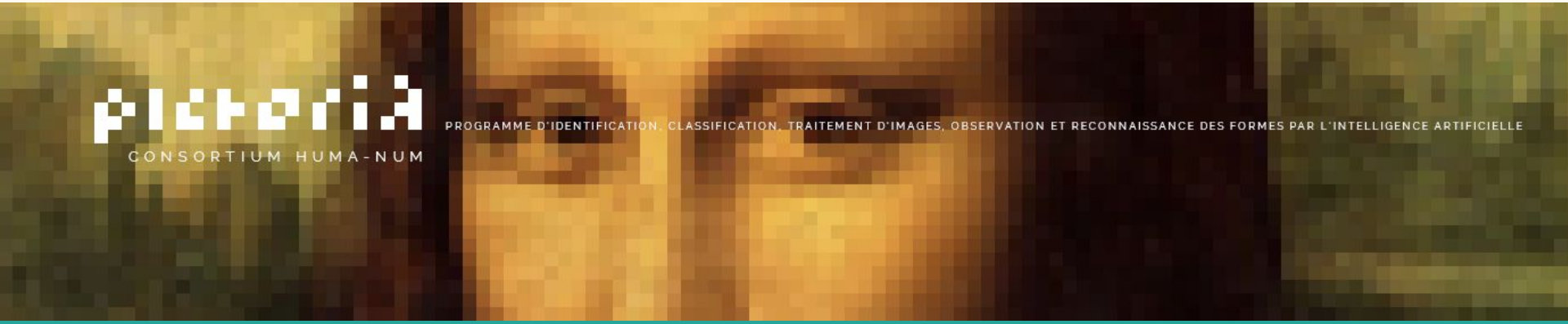
2) Les fonctions en Python
   a) Définition et utilisation
   b) Structure d'une fonction
   c) Concepts clés et bonnes pratiques pour les Fonctions
   d) Fonction anonyme (`lambda`)

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

- 17 octobre 2024, BnF DataLab : Séminaire-atelier pictorIA “L’IA pour l’archéologie et les arts visuels anciens”
- 19-20 décembre 2024 : Hackathon pictorIA x BnF DataLab autour des jeux de données patrimoniales visuelles et multimodales



Rappel des conditions `if`, `elif`, `else`

Syntaxe des conditions :

```
if condition:
    # instructions
elif autre_condition:
    # instructions
else:
    # instructions
```

L'**indentation** est **essentielle** en Python.

- Elle permet de **définir les blocs de code** associés aux conditions.
- Un mauvais alignement de l'indentation entraînera une **erreur de syntaxe**.

Les conditions fonctionnent avec **:**

- **Opérateurs de comparaison :** `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Opérateurs logiques :** `and`, `or`, `not`

Attention à l'ordre des conditions

Importance de l'ordre des conditions :

- Les structures `if...elif...else` sont évaluées séquentiellement.
- **La première condition vraie est exécutée, et les suivantes sont ignorées pour cette itération.**

Conséquence :

- Si une condition générale est placée avant une condition spécifique, la condition spécifique risque de ne jamais être évaluée.

Bonne pratique :

- **Placez les conditions les plus spécifiques en premier**, suivies des conditions plus générales.

Rappel sur les boucles bornées - `for`

Syntaxe de la Boucle `for` :

```
for element in sequence:
    # instructions
```

Rappel

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

Rappel de l'utilisation des boucles avec `range()` :

```
for i in range(100):
    print(i)
#range signifie ici génère moi les nombres de 0 à 100 (mais sans le 10 donc plutôt 0 à 99)

for i in range(3,8):
    print(i)
#range signifie ici génère moi les nombres de 3 à 7

for i in range(0,21,2):
    print(i)
#range signifie ici génère moi les nombres de 0 à 20 avec un incrément de 2 (de 2 en 2)
```

Rappel de l'utilisation de break et de continue :

- `break` :
  a. **But** : Sort immédiatement de la boucle en cours.
  b. **Quand l'utiliser** : Lorsque vous souhaitez arrêter complètement la boucle en fonction d'une condition.
  c. **Exemple** :

```
for i in range(10):
    print(i)
    if i >= 5:
        break
```

- `continue` :
  a. **But** : Passe à l'itération suivante de la boucle, en sautant le reste du code pour l'itération en cours.
  b. **Quand l'utiliser** : Lorsque vous souhaitez ignorer certaines itérations sans arrêter la boucle.
  c. **Exemple** :

```
texte = "Humanités Numériques"
for lettre in texte:
    if lettre == 'é':
        continue
    print(lettre, end='')
sortie : “Humanits Numriques”
```

Boucles non bornées - `while`

Syntaxe de la Boucle `while` :

```
while condition:
        # instructions
```

While fonctionne tant qu'une condition est valable (booléen)

Points clés :

- **Flexibilité :** La boucle `while` est adaptée lorsque le nombre d'itérations n'est pas connu à l'avance.
- **Conditions dynamiques :** Permet d'arrêter le processus en fonction de conditions qui peuvent évoluer pendant l'exécution.
- **Contrôle :** Offre un contrôle précis sur le flux du programme, essentiel pour gérer des tâches complexes ou dépendantes de facteurs externes.

Exemples

```
i = 0
while i <= 6: #Renvoie un booléen: tant que True continue
    print("Dans la boucle", i) #Affiche l'itération
    i+= 1 #Modifie la variable de base et relance la boucle
print("Fin") # Affiche Fin à lorsque i > 6
```

Exemple : Supprimer les éléments vides ou non valides d'une liste de métadonnées.

```
métadonnées = ["Titre", "", "Auteur", None, "Date de
publication", "", ""]

while "" in métadonnées or None in métadonnées:
       if "" in métadonnées:
       métadonnées.remove("")
       if None in métadonnées:
       métadonnées.remove(None)

print("Métadonnées nettoyées :", métadonnées)
```

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.

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.

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)
```

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 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.

- **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.

Un module s'appelle en **début de code.**

- **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 ...**

```
from math import pi, sqrt
print(pi)       # Affiche : 3.141592653589793
print(sqrt(16))  # Affiche : 4.0
```

- **Utiliser un alias avec import ... as ...**

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

Importation de modules et conventions

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

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 l'importation de 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.

```
# Importations standard
import os
import sys

# Importations externes
import numpy as np
import pandas as pd

# Importations locales
import mon_module
```

Quelques modules utiles

Modules 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
```

4. `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
```

Modules Externes (à installer via `pip`)

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

Créer son module

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.