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	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 1

1) **Présentation du langage Python**
2) **Installation de VSCODE**
3) **Base de la programmation**
   a) Variables et types de données
      i) Basiques : Integer, Float, String et f-string, Booléen
      ii) Listes
      iii) Tuples
      iv) Dictionnaires
   b) Opérateurs logiques et manipulation de variables
      i) Concaténation
      ii) Index
      iii) Tranche
      iv) Méthodes utiles : `type()`, `len()`, `split()`, `sort()`, `append()`, `keys()`, `values()`
   c) Structures de contrôle : conditions et boucles
      i) Conditions : `if`, `elif`, `else`
      ii) Boucles bornées : `for`
      iii) Boucles non bornées : `while` (avec `break` et `continue`)

Qu'est ce que Python ?

Création et Origine

- **Créé en 1991** par Guido van Rossum
- **Nom inspiré** par la troupe comique britannique **"Monty Python's Flying Circus"**
- Développé comme un successeur du langage ABC, avec **l'objectif de créer un langage simple**, lisible et puissant pour le prototypage et le développement rapide
- **Philosophie :** Mettre l'accent sur la clarté du code, la simplicité d'utilisation et une communauté ouverte

« ... In December 1989, I was looking for a **"hobby" programming project that would keep me occupied during the week around Christmas.** My office ... would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus). »

Qu’est ce que Python ? Un langage de programmation de Haut Niveau

Langage de Haut Niveau

Intérêt

- **Facilité d'écriture** : Plus simple de coder qu'avec un langage proche de la machine.
- **Rapidité de développement** : Programmes plus rapides à écrire et à déboguer.
- **Portabilité accrue** : Les programmes sont plus facilement transférables d'une machine à une autre grâce à un niveau d'abstraction élevé des particularités des processeurs.

Inconvénients

- **Moins performants** : Coût plus élevé en termes de ressources (temps d'exécution, mémoire).
- **Moins de contrôle direct** sur le matériel.

```python
print("Hello World")
```

Langages de Bas Niveau (ex. C)

Intérêt

- **Proches du langage machine :** instructions proches du code binaire exécuté par l’ordinateur
- **Contrôle précis du matériel :** gestion directe de la mémoire, des registres et du processeur
- **Très performants :** utilisés pour les systèmes d’exploitation, les pilotes ou les logiciels embarqués

Inconvénients

- **Mais complexes :** nécessitent une connaissance approfondie de l’architecture matérielle et une gestion manuelle des **ressources (mémoire, pointeurs, etc.)**

```c
#include <stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}
```

Le zen de python

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Installation de Python

Installation sur Linux Ubuntu

1. **Utilisation du gestionnaire de paquets**
   - Ouvrez le **Terminal**.
   - Mettez à jour la liste des paquets :
     sudo apt update
2. **Installation**
   - Python 3 et pip (gestionnaire de paquets Python) :
     sudo apt install python3 python3-pip
3. **Vérification de l'installation**
   - Tapez la commande suivante pour vérifier la version de Python installée :
     python3 --version

Installation sur macOS

1. **Téléchargement de Python**
   - Rendez-vous sur le site officiel de Python : python.org/downloads
   - Cliquez sur **"Download Python 3.12.6"** pour macOS.
2. **Installation**
   - Ouvrez le fichier **.pkg** téléchargé.
   - Suivez les instructions de l'installateur en laissant les options par défaut.
3. **Vérification de l'installation**
   - Ouvrez le **Terminal**.
   - Tapez la commande suivante pour vérifier la version de Python installée :
     python3 --version

Installation de Visual Studio Code (VSCode)

Installation de VS Code

Linux (Ubuntu/Debian)

- Télécharger **.deb (64-bit)** depuis code.visualstudio.com
- Installer via :
  sudo dpkg -i code_*.deb
  sudo apt-get install -f

macOS

- Télécharger le fichier **.zip** depuis code.visualstudio.com
- Glisser **VS Code.app** dans **Applications**
- **Ouvrir depuis Launchpad ou Spotlight**

```python
import torch
import whisper
import gradio as gr
import tempfile
import os
import json

def load_whisper_model(model_file=None, model_name=None):

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # 1st case : if a personnal model is loaded (.pt)
    if model_file is not None:
        try:
            model_path = model_file.name if hasattr(model_file, 'name') else model_file
            whisper_model = whisper.load_model(model_path, device=device)
            print(f'Model loaded from {model_path}')
            return whisper_model

        except Exception as e:
            print(f"An error occurred while loading the model: {e}")
            raise
```

Variables et types de données - Basiques

Définition

Les **variables** sont des conteneurs utilisés pour **stocker des données**.
Elles permettent d'**assigner** une valeur à un nom et de la **réutiliser** dans le code. Les variables facilitent la manipulation et le calcul des valeurs.

Règles de nommage des variables

- **Commencer par une lettre ou un underscore (_)**.
- Peut contenir des **lettres, chiffres, et underscore** (`variable_1` est valide).
- **Pas d'espaces** ou de **caractères spéciaux** (`$`, `@`, etc.).
- **Ne doit pas être un mot réservé** (comme `for`, `if`, `while`).

Liste des mots réservés en python

and	as	assert	break	class	continue	def	del
elif	else	except	exec	finally	for	from	global
if	import	in	is	lambda	not	or	pass
print	raise	return	try	while	with	yield	

Variables et types de données - Basiques

Types de données de base

- **Entiers (`int`)** : Représentent des **nombres sans décimales**.
- **Flottants (`float`)** : Représentent des **nombres avec décimales**.
- **Chaînes de caractères (`str`)** : Représentent du **texte**.
- **Booléens (`bool`)** : Représentent une **valeur de vérité** : True ou False.

Sensibilité à la casse

Les variables en Python sont **sensibles à la casse**, ce qui signifie que `variable`, `Variable` et `VARIABLE` sont **trois variables différentes**.

Les **f-strings** permettent d'insérer des **variables** et des **expressions** directement dans des chaînes de caractères, facilitant le formatage.

- Utilisez un **f** avant les guillemets pour créer une f-string.
- Placez les variables ou expressions entre **accolades {}**.

Listes

Définition : Séquence ordonnées modifiables

Création de liste :
Une liste est toujours contenue dans des crochets [ ] et chaque élément est séparé par une virgule.

Accès aux éléments

1) **Par index**

Les éléments d'une liste possède tous un index qui permet de les extraire en fonction de leur place dans la liste

Méthodes utiles

a) append() **pour ajouter un élément**
b) sort() **pour trier la liste**

Créations de liste

```
animals = ["draco", "unicorn", "baslisk"]
```

Requête par index

```
Index:   |  0   |  1       |  2
Animals: | "draco" | "unicorn" | "basilisk"
```

```
animals[0] retourne "draco"
```

Ajouter des éléments dans une liste

```
animal.append("amphisbena")
```

Trier les éléments de la liste

```
animals.sort()
```

Attention!!!

sort() **ne fonctionne que si les éléments qui composent la listent sont de même type**

Tuples - Définition et création

Définition : Séquences ordonnées **immuables** (non modifiables après leur création).

Création de tuples : Un tuple est toujours contenue dans des **parenthèses** `()` et chaque élément est séparé par une **virgule**.

Création de tuples

```
snakes = ("draco", "basilisk",  "asp",
"amphisbena")
```

Index

`animals[0] retourne "draco"`

Tuples - Utilisation

Immuabilité :

- Impossible de modifier, ajouter ou supprimer des éléments après la création du tuple.
- Toute tentative de modification provoquera une **erreur**

Utilités des tuples :

- **Stockage de données constantes** : Idéal pour les valeurs qui ne doivent pas changer.
- **Clés de dictionnaire** : Les tuples peuvent être utilisés comme clés si leurs éléments sont immuables.
- **Unpacking** : Le déballage permet d'assigner les éléments d'un tuple à des variables distinctes.

Immutabilité

```
creatures[0] = "phoenix"
# Erreur: impossible de modifier un
tuple
```

Déballage

```
coordinates = (50.0, 100.0)
latitude, longitude = coordinates
print(f"Latitude : {latitude},
Longitude : {longitude}")
```

Dictionnaires - Définition et création

Définition : Les **dictionnaires** sont des **collections non ordonnées** qui associent des **clés uniques** à des **valeurs**.
Chaque **clé** permet de **retrouver rapidement** la valeur associée. Ils sont utilisés lorsque les données doivent être consultées par un identifiant unique.

Création de dictionnaires :
Un dictionnaire est créé en utilisant des **accolades {}**, où chaque **paire clé-valeur** est séparée par **deux points :**, et chaque paire est séparée par une **virgule ,**.

Accès aux valeurs :
Les valeurs d'un dictionnaire sont accédées par leurs **clés** uniques.

Création d'un dictionnaire

```
personne = {"nom": "Alice", "age": 25, "ville": "Paris"}
```

Accès par clé

```
Clé:   | "nom"   | "âge" | "ville"
Valeur:| "Alice" |  25   | "Paris"
```

Dictionnaires - méthodes utiles

Les dictionnaires sont des structures de données très utilisées en Python, car ils permettent d'associer des **clés** à des **valeurs**.
Pour les manipuler facilement, il existe des méthodes pratiques qui offrent un accès direct :

- aux **clés** du dictionnaire
  - keys() : Retourne une **liste de toutes les clés** du dictionnaire.
- aux **valeurs** qu'elles contiennent
  - values() : Retourne une **liste de toutes les valeurs** du dictionnaire.
- ou encore aux **paires clé-valeur** sous forme de tuples.
  - items() : Retourne une **liste de tuples** représentant chaque **paire clé-valeur**.

Méthodes

```
personne.keys()  # retourne ["nom", "âge", "ville"]

personne.values()  # retourne ["Alice", 25, "Paris"]

personne.items()  # retourne [("nom", "Alice"), ("age", 25), ("ville", "Paris")]
```

Sets (Ensembles) - Définition et création

Définition : Les **sets** (ensembles) sont des collections non ordonnées d'éléments **uniques** et **immutables** (les éléments eux-mêmes doivent être immuables). Ils sont utilisés pour stocker plusieurs éléments sans duplication.

Création de Sets :

Un set est créé en utilisant des **accolades `{}`** ou la fonction **`set()`**.

Création d'un set

```
creatures = {"draco", "basilisk",  "asp", "amphisbena"}
```

Création d'un set vide

```
empty_set = set()  # Utiliser set() car {} crée un dictionnaire vide
```

Sets (Ensembles) - Caractéristique et méthodes

Caractéristiques des sets

- **Non ordonnés** : Les sets n'ont pas d'**index**, donc l'accès par position n'est pas possible.
- **Éléments uniques** : Les éléments d'un set sont **uniques**, ce qui signifie qu'un même élément ne peut pas apparaître plusieurs fois dans le set.
- **Mutabilité du set** : Le set lui-même est **modifiable** (ajout ou suppression d'éléments), mais **les éléments du set** doivent être **immutables** (ex: chaînes de caractères, nombres).

Méthodes utiles

- `add(element)` : Ajouter un élément au set.
- `remove(element)` : Supprimer un élément (provoque une erreur si l'élément n'existe pas).
- `discard(element)` : Supprimer un élément sans erreur si l'élément n'existe pas.
- `clear()` : Supprimer toutes les données du set

```
creatures.add("griffin")  # Ajoute "griffin" au set

creatures.remove("unicorn")  # Supprime "unicorn" du set

creatures.discard("phoenix")  # Ne génère pas d'erreur si "phoenix" n'est pas dans le set
```

Opérateurs logiques

Les **opérateurs logiques** sont utilisés pour **comparer des expressions** et **retourner des valeurs de vérité** (`True` ou `False`) en fonction des conditions spécifiées.
Ils permettent de **combiner** ou de **négocier** des conditions logiques, facilitant ainsi la prise de décision dans les structures de contrôle (`if`, `while`, etc.).

Opérateurs Logiques :

- **ET (`and`)** :Vérifie si toutes les conditions sont vraies.
- **OU (`or`)** : Vérifie si au moins une des conditions est vraie.
- **NON (`not`)** : Inverse la valeur de vérité d'une condition.

Opérations logiques

```
a = 5
b = 10

if a and b:
# retourne True
print(f"{a} et {b} sont positifs)

if a or b:
    # retourne True
if not a:
    # retourne True
```

Opérateurs de comparaison

Les **opérateurs de comparaison** sont utilisés pour **comparer des expressions** et **retourner des valeurs de vérité** (`True` ou `False`) en fonction des conditions spécifiées.

- **Égal à (`==`)** : Vérifie si deux valeurs sont égales.
- **Différent de (`!=`)** : Vérifie si deux valeurs sont différentes.
- **Supérieur à (`>`)** : Vérifie si une valeur est strictement supérieure à une autre.
- **Inférieur à (`<`)** : Vérifie si une valeur est strictement inférieure à une autre.
- **Supérieur ou Égal à (`>=`)** : Vérifie si une valeur est supérieure ou égale à une autre.
- **Inférieur ou Égal à (`<=`)** : Vérifie si une valeur est inférieure ou égale à une autre.

Opérateurs de comparaison

```
5 == 5  # retourne True
5 != 10  # retourne True
10 > 5  # retourne True
5 < 10  # retourne True
10 >= 5  # retourne True
5 <= 5  # retourne True
```

Manipulation de variables

La **manipulation de variables** en programmation consiste à **créer, modifier, combiner, et utiliser** des variables pour **stocker et manipuler des données** tout au long de l'exécution d'un programme. Les variables permettent de **mémoriser** des valeurs (numériques, textuelles, booléennes, etc.) et de **réaliser des calculs**, **opérations logiques** ou encore des **manipulations de chaînes**.

Manipulation de variables

- **Concaténation de chaînes** : Permet de combiner plusieurs chaînes en une seule en utilisant l'opérateur +.
- **Indexation** : Permet d'accéder à un caractère spécifique dans une chaîne en utilisant un index.
- **Tranches** (Slicing) : Permet d'extraire une sous-chaîne en utilisant des indices de début et de fin.

Exemple

```
texte = "Bonjour" + " " + "le monde"
# retourne "Bonjour le monde"
texte[0:7]  # retourne 'Bonjour'
```

Manipulation de variables - Fonctions utiles

Lorsque l'on débute avec Python, certaines fonctions intégrées (appelées **built-in functions**) reviennent très souvent et facilitent énormément la manipulation des données.

Elles permettent par exemple :

- de **connaître le type** d'une variable,
  - `type(variable)`
- de **mesurer la taille** d'une séquence,
  - `len(sequence)`
- ou encore de **transformer une chaîne de caractères** en liste.
  - `split(sep)`

Fonctions utiles

```
type(texte)  # retourne <class 'str'>
len(texte)  # retourne 13
mots = texte.split(" ")  # retourne
['Bonjour', 'le', 'monde']
```

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
```

Notions clés à retenir

Types de Variables

- Entiers (`int`), Flottants (`float`), Chaînes de Caractères (`str`), et Booléens (`bool`).
- Comprendre comment **déclarer**, **affecter** et **manipuler** les variables.

Structures de Contrôle

- **Conditions (`if`, `elif`, `else`)** : Permettent de prendre des décisions en fonction de certaines conditions.
- **Boucles (`for`, `while`)** : Facilitent la répétition d'instructions en parcourant des séquences ou en fonction de conditions.

Opérateurs Logiques et de Comparaison

- Utilisation de `and`, `or`, `not` pour combiner ou négocier des conditions.
- Comparer des valeurs avec `==`, `!=`, `<`, `>`, `<=`, `>=`.

Manipulation de Données

- **Listes**, **tuples**, **dictionnaires**, et **sets** pour organiser et gérer les données.
- Utilisation de méthodes spécifiques (`append()`, `remove()`, `keys()`, etc.) pour manipuler chaque type de collection.

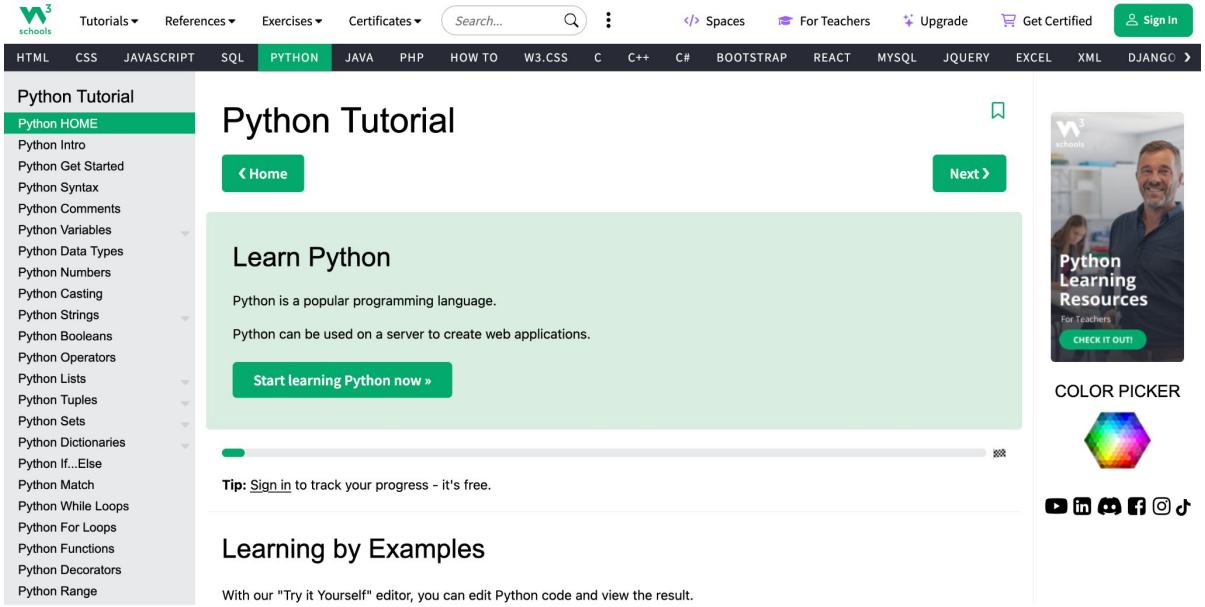
Pour aller plus loin...

Pratiquer

Exercez-vous à écrire des très programmes simples qui utilisent les concepts appris.

Entraînez vous

https://www.w3schools.com/python/
-> ‘Python For Loops’



Save the date!

26 – 28 novembre 2024 - Hackathon PictorIA au DataLab (BnF)

- Événement incontournable autour de l'IA et des données.
- Travail collaboratif, innovation et créativité.
- Ouvert à tous les participants.

⚠️ **Participation obligatoire** (sauf pour les étudiants déjà inscrits à la **PSL Week**).

pictoria
CONSORTIUM HUMA-NUM