Bien sûr Je vais te préparer un petit **tutoriel clair et progressif sur le multi-threading en Python**, adapté aux cas pratiques comme le web scraping.

Tutoriel : Multi-threading en Python

1. Pourquoi utiliser le multi-threading ?

- Le **multi-threading** permet d'exécuter plusieurs tâches *en parallèle* dans un même processus.
- En Python, à cause du **GIL (Global Interpreter Lock)**, les threads ne permettent pas de paralléliser du calcul pur (CPU). Mais pour les tâches **I/O-bound** (réseau, lecture/écriture de fichiers), le multi-threading permet **d'attendre plusieurs opérations à la fois** → gros gain en scraping, téléchargements, API calls.

---

2. Exemple simple sans thread

On simule une tâche lente avec `time.sleep`.

```
import time

def task(n):
    print(f"Tâche {n} démarrée")
    time.sleep(2)  # simulation d'un appel réseau
    print(f"Tâche {n} terminée")

def main():
    start = time.time()
    for i in range(5):
        task(i)
    print("Durée totale :", time.time() - start)

if __name__ == "__main__":
    main()
```

 Ici, 5 tâches × 2s = environ **10 secondes**.

---

3. Multi-threading avec `Thread`

On peut lancer les tâches en **parallèle** avec `threading.Thread`.

```
import time
import threading

def task(n):
    print(f"Tâche {n} démarrée")
    time.sleep(2)
    print(f"Tâche {n} terminée")

def main():
```

```
    start = time.time()
    threads = []

    # lancer les threads
    for i in range(5):
        t = threading.Thread(target=task, args=(i,))
        t.start()
        threads.append(t)

    # attendre la fin
    for t in threads:
        t.join()

    print("Durée totale :", time.time() - start)

if __name__ == "__main__":
    main()
```

 Ici, les 5 tâches s'exécutent ensemble → durée totale ≈ **2 secondes**.

---

4. L'approche moderne : `concurrent.futures.ThreadPoolExecutor`

C'est la méthode recommandée pour du code clair et maintenable.

```
import time
from concurrent.futures import ThreadPoolExecutor, as_completed

def task(n):
    print(f"Tâche {n} démarrée")
    time.sleep(2)
    return f"Tâche {n} terminée"

def main():
    start = time.time()
    results = []

    # ThreadPoolExecutor gère les threads automatiquement
    with ThreadPoolExecutor(max_workers=5) as executor:
        futures = [executor.submit(task, i) for i in range(5)]
        for f in as_completed(futures):
            results.append(f.result())  # récupère le résultat d'une tâche

    for r in results:
        print(r)

    print("Durée totale :", time.time() - start)

if __name__ == "__main__":
    main()
```

 Même résultat (~2s) mais le code est plus lisible, plus sûr, et gère mieux les exceptions.

 Résumé :

- `threading.Thread` = bas niveau, plus verbeux.
- `ThreadPoolExecutor` = moderne, pratique, gère mieux les résultats et erreurs.
- Idéal pour les **tâches I/O-bound** comme le web scraping.

---

5. Exemple concret : Web scraping en multi-thread

Objectif

Mettre en place un scraper capable de télécharger plusieurs pages Wikipédia **en parallèle** grâce au **multi-threading**. L'exercice montre le gain de temps par rapport à une version séquentielle.

Étapes à suivre

1. **Préparer la liste d'URLs à scraper**
    - Exemple :

      ```
      URLS = [
      "https://fr.wikipedia.org/wiki/Graham_Chapman",
      "https://fr.wikipedia.org/wiki/John_Cleese",
      "https://fr.wikipedia.org/wiki/Terry_Gilliam",
      "https://fr.wikipedia.org/wiki/Eric_Idle",
      "https://fr.wikipedia.org/wiki/Terry_Jones",
      "https://fr.wikipedia.org/wiki/Michael_Palin"
      ]
      ```

    - Vous pouvez choisir d'autres pages, mais elles doivent être valides et publiques (format `https://fr.wikipedia.org/wiki/...`).
2. **Configurer un User-Agent explicite**
    - Rappel : toujours vous identifier poliment (ex. `MonScraper-Etudiant/1.0 (+email)`), et ne pas utiliser le User-Agent de Chrome.
    - C'est une bonne pratique et une obligation sur certains sites.
3. **Créer une fonction `fetch(url)`**
    - Cette fonction prend une URL en paramètre.
    - Elle envoie une requête HTTP avec la librairie `requests`.
    - Elle retourne quelques infos utiles :
        - l'URL
        - le code HTTP (200 si succès)
        - la taille de la page téléchargée (longueur du texte).
4. **Utiliser un `ThreadPoolExecutor`**
    - Importer depuis `concurrent.futures`.
    - Définir un **pool de threads** avec `max_workers` (par ex. 4).

- Soumettre toutes vos URLs au pool (`executor.submit`).
- Récupérer les résultats au fur et à mesure avec `as_completed` .

5. **Afficher les résultats**
   - Pour chaque URL, afficher :
     - le code HTTP,
     - la taille du texte HTML (en nombre de caractères).
   - Exemple attendu (forme libre) :

     ```
     https://fr.wikipedia.org/wiki/Monty_Python → 200, 180523 caractères
     ```

6. **Comparer le temps d’exécution**
   - Faites un test en **séquentiel** (boucle for classique) et en **multi-thread**.
   - Comparez les durées avec `time.time()` .
   - Observez la différence : avec 4–5 pages, le multi-thread doit être beaucoup plus rapide.

Conseils pour réussir

- Fixer un **timeout** dans `requests.get` (ex. 10s) pour éviter de bloquer un thread trop longtemps.
- Gérez les erreurs avec `try/except` pour qu’un échec sur une URL ne casse pas tout le programme.
- Ne mettez pas trop de `max_workers` : 4–8 suffisent largement pour cet exercice.
- Respectez la politesse : même si on fait du multi-threading, on ne veut pas surcharger le site (évitez 50 threads d’un coup).

---

6. Bonnes pratiques

- **Limiter `max_workers`** (4–10 est raisonnable pour du scraping).
- Toujours utiliser un **User-Agent clair**.
- **Respecter robots.txt** et insérer des **delays** si nécessaire.
- Gérer les erreurs (`try/except`) dans les tâches.
- Ne pas confondre avec **multi-processing** (`multiprocessing`) qui est pour le CPU-bound.
