

Double-click (or enter) to edit

✓ Khởi tạo thread trong python

✓ 2.1 Khởi tạo từ hàm

```
# b1 : import thư viện vào
import threading

# b2 : Định nghĩa hàm cần chạy song song
def print_task():
    print("Thread is running")

# b3 : Tạo đối tượng Thread
t = threading.Thread(target=print_task)

# Khởi tạo thread
t.start()
# chờ thread hoàn thành (optional)
t.join()
```

↻ Thread is running

✓ 📌 Giải thích:

Thành phần Ý nghĩa target=... Hàm bạn muốn thread thực thi .start() Bắt đầu chạy thread .join() Đợi thread chạy xong mới tiếp tục

Double-click (or enter) to edit

```
import threading
import time

def _counter(counter, thread_name):
    while counter:
        time.sleep(0.01)
        print(f"{thread_name}: {counter}")
        counter -= 1

counter = 5

t1 = threading.Thread(target=_counter, args=(counter, "khanh thread"))
t2 = threading.Thread(target=_counter, args=(counter, "An thread"))

t1.start()
t2.start()

t1.join()
t2.join()
```

↻ khanh thread: 5
An thread: 5
khanh thread: 4
An thread: 4
khanh thread: 3
An thread: 3
khanh thread: 2
An thread: 2
khanh thread: 1
An thread: 1

✓ 2.2 Khởi tạo kế thừa

Một cách khác để khởi tạo một thread đó là kế thừa lại Threading module. Kiểu kế thừa này khá phổ biến trong lập trình, chắc các bạn còn nhớ khi khởi tạo model trên pytorch chúng ta cũng kế thừa lại nn.Module chứ ? Khi đó chúng ta chỉ cần override lại các method cần điều chỉnh từ class cha.

🔴 Mô tả: Một cách khác để tạo thread là kế thừa lại class threading.Thread.

Phù hợp với lập trình hướng đối tượng (OOP).

Giống như khi bạn tạo model trong PyTorch kế thừa nn.Module, ở đây bạn kế thừa Thread.

```
import threading
import time

class MyThread(threading.Thread):
    def __init__(self, name):
        super().__init__() # Gọi constructor của lớp cha
        self.name = name

    def run(self): # Phương thức override (ghi đè)
        print(f"🔵 Bắt đầu thread: {self.name}")
        for i in range(1,7):
            print(f"{self.name} đếm: {i}")
            time.sleep(0.9)
        print(f"🟢 Kết thúc thread: {self.name}")

# Tạo 2 thread
t1 = MyThread("Thread 1")
t2 = MyThread("Thread 2")
t3 = MyThread("Thread 3")
# Bắt đầu chạy
t1.start()
t2.start()
t3.start()
# Chờ cả 2 thread chạy xong
t1.join()
t2.join()
t3.join()
print("🌈 Tất cả các thread đã hoàn tất.")
```

```
🔄 🔵 Bắt đầu thread: Thread 1
Thread 1 đếm: 1
🔵 Bắt đầu thread: Thread 2
Thread 2 đếm: 1
🔵 Bắt đầu thread: Thread 3
Thread 3 đếm: 1
Thread 1 đếm: 2
Thread 2 đếm: 2
Thread 3 đếm: 2
Thread 1 đếm: 3
Thread 2 đếm: 3
Thread 3 đếm: 3
Thread 1 đếm: 4
Thread 3 đếm: 4
Thread 2 đếm: 4
Thread 1 đếm: 5
Thread 3 đếm: 5
Thread 2 đếm: 5
Thread 1 đếm: 6
Thread 3 đếm: 6
Thread 2 đếm: 6
🟢 Kết thúc thread: Thread 1
🟢 Kết thúc thread: Thread 2
🟢 Kết thúc thread: Thread 3
🌈 Tất cả các thread đã hoàn tất.
```

✓ 2.3. Cơ chế Thread Lock

🔒 Thread Lock là gì?

Là cơ chế khóa luồng dùng để tránh xung đột dữ liệu (race condition) khi nhiều thread cùng truy cập và chỉnh sửa một biến dùng chung.

```
import threading

lock = threading.Lock()

def critical_section():
    lock.acquire() # 🚦 Chiếm quyền truy cập
    try:
        # thao tác dữ liệu dùng chung
        pass
    finally:
        lock.release() # 🚦 Trả lại quyền truy cập

# ✅ Hoặc dùng with để viết gọn và an toàn hơn:
with lock:
    # thao tác an toàn với dữ liệu dùng chung
    pass

# → Python sẽ tự động gọi lock.acquire() khi bắt đầu và lock.release() khi kết thúc khối lệnh, kể cả khi có lỗi xảy ra.
```

```
import threading
import time

# Tạo đối tượng lock dùng chung
threadLock = threading.Lock()

class MyThread(threading.Thread):
    def __init__(self, name):
        super().__init__()
        self.name = name

    def run(self):
        # Dùng lock để đảm bảo chỉ một thread được in ra tại một thời điểm
        with threadLock:
            print(f"👉 Bắt đầu thread: {self.name}")
            for i in range(1, 6):
                print(f"{self.name} đếm: {i}")
                time.sleep(0.3)
            print(f"👎 Kết thúc thread: {self.name}")

t1 = MyThread("Thread 1")
t2 = MyThread("Thread 2")

t1.start()
t2.start()

t1.join()
t2.join()

print("🎉 Tất cả các thread đã hoàn tất.")
```

```
🔄 👉 Bắt đầu thread: Thread 1
Thread 1 đếm: 1
Thread 1 đếm: 2
Thread 1 đếm: 3
Thread 1 đếm: 4
Thread 1 đếm: 5
👎 Kết thúc thread: Thread 1
👉 Bắt đầu thread: Thread 2
Thread 2 đếm: 1
Thread 2 đếm: 2
Thread 2 đếm: 3
Thread 2 đếm: 4
Thread 2 đếm: 5
👎 Kết thúc thread: Thread 2
🎉 Tất cả các thread đã hoàn tất.
```

✓ Khởi tạo process trong python

✓ 4 CÁCH KHỞI TẠO PROCESS TRONG PYTHON

STT	Cách khởi tạo	Mô tả ngắn	Dùng khi...
1	Truyền hàm vào <code>multiprocessing.Process</code>	Tạo process bằng cách truyền hàm (<code>target</code>)	Dễ học, dùng cho xử lý đơn giản
2	Kế thừa class <code>multiprocessing.Process</code>	Tạo class riêng và override phương thức <code>run()</code>	Tổ chức tốt hơn, dễ mở rộng (OOP)
3	Dùng <code>multiprocessing.Pool</code>	Quản lý tiến trình theo kiểu "bể"	Khi có nhiều task giống nhau (batch jobs)
4	Dùng <code>concurrent.futures.ProcessPoolExecutor</code>	API hiện đại, dễ viết, dễ đọc, dễ quản lý	Viết gọn, tương thích tốt với <code>async</code> /code mới

1. Dùng `multiprocessing.Process` với hàm

```
from multiprocessing import Process
```

```
def work():
    print("Hello")
```

```
p = Process(target=work)
p.start()
p.join()
```

→ Hello

=> ✓ Đơn giản, dễ hiểu ! Chưa theo OOP

✓ 2. Kế thừa class `Process`

```
from multiprocessing import Process
# kế thừa (inherit) class để chạy một tiến trình mới (process) trong Python
class MyProcess(Process):
    def run(self):
        print("Hello from subclass")
if __name__ == "__main__":
    p = MyProcess()
    p.start()
    p.join()
```

→ Hello from subclass

if **name** == "**main**": dùng để: Đảm bảo chỉ thực thi đoạn code khi chạy trực tiếp (python file.py)

Không bị gọi lại khi tạo process mới

Cũng áp dụng tương tự cho: Thread, Tkinter, Flask, etc.

=> ✓ Rõ ràng, mở rộng tốt (OOP style) ! Dài hơn chút

✓ 3. Dùng `multiprocessing.Pool`

```
from multiprocessing import Pool
```

```
def square(x):
    return x * x
# Tạo một Pool gồm 4 process (số lượng tiến trình chạy song song)
# Dùng pool.map để gọi hàm square cho từng phần tử trong danh sách [1, 2, 3, 4]
# Mỗi phần tử sẽ được xử lý song song trên các process khác nhau
with Pool(4) as pool:
    print(pool.map(square, [1, 2, 3, 4]))
```

→ [1, 4, 9, 16]

=> ✓ Dễ dùng với nhiều task giống nhau ! Không linh hoạt như Process

```
# 4. Dùng concurrent.futures.ProcessPoolExecutor
from concurrent.futures import ProcessPoolExecutor

def work(x):
    return x * x

with ProcessPoolExecutor() as executor:
    result = executor.map(work, [1, 2, 3])
    print(list(result))
```

→ [1, 4, 9]

=> ✓ Gọn, hiện đại, dễ quản lý ✓ Kết hợp tốt với async hoặc các hàm lặp

✓ Process Pool vs Thread Pool

♦ Khác nhau:		
Điểm khác	Process Pool	Thread Pool
Loại	Đa tiến trình	Đa luồng
Bộ nhớ	Tách biệt	Chia sẻ
Bị GIL ảnh hưởng	✗ Không	✓ Có
Dùng cho	Tác vụ tính toán (CPU)	Tác vụ chờ I/O
Ví dụ	Xử lý ảnh, số liệu lớn	Gọi API, đọc file, truy vấn DB

✓ Khi nào dùng? 📁 Process Pool → Xử lý nặng, tốn CPU

🌐 Thread Pool → Gọi API, đọc file, tải web...

```
# Ví dụ 1 - Dùng Process Pool (Xử lý CPU nặng)
from concurrent.futures import ProcessPoolExecutor
import time

def heavy_task(n):
    total = 0
    for i in range(10**6):
        total += i * n
    return total

if __name__ == "__main__":
    with ProcessPoolExecutor() as executor:
        results = executor.map(heavy_task, [1, 2, 3, 4])
        print(list(results))

→ [499999500000, 999999000000, 1499998500000, 1999998000000]
```

✓ Dùng ProcessPoolExecutor để chạy 4 tác vụ tính toán nặng cùng lúc. → Phù hợp với CPU-bound (tính toán lớn).


```
# Ví dụ 2 - Dùng Thread Pool (Gọi API giả lập)
from concurrent.futures import ThreadPoolExecutor
import time
```

```
def fake_api_call(name):
    print(f"{name} bắt đầu")
    time.sleep(2) # Giả lập chờ phản hồi từ API
    print(f"{name} xong")
    return f"{name} done"

with ThreadPoolExecutor() as executor:
    results = executor.map(fake_api_call, ["API 1", "API 2", "API 3"])
    print(list(results))
```

```
API 1 bắt đầu
API 2 bắt đầu
API 3 bắt đầu
API 1 xong
API 2 xong
API 3 xong
['API 1 done', 'API 2 done', 'API 3 done']
```

✓ Bài tập áp dụng

 Bài tập 1 – Thread: Trạm thu phí thông minh (Smart Toll Booth) Mô tả: Một trạm thu phí có 3 làn đường (Lane A, Lane B, Lane C), mỗi làn có một nhân viên thu phí xử lý xe đi qua. Mỗi xe mất 1–3 giây để xử lý. Hãy mô phỏng việc xử lý xe bằng các thread, sao cho các làn xe hoạt động song song.

Yêu cầu:

Sử dụng `threading.Thread`

Mỗi thread mô phỏng một làn xe xử lý từ 5–7 xe

In rõ: xe nào đi qua làn nào, mất bao nhiêu thời gian

```
import threading
import time
import random

# Lớp mô phỏng một làn thu phí
class TollLane(threading.Thread):
    def __init__(self, lane_name):
        super().__init__()
        self.lane_name = lane_name
        self.num_cars = random.randint(5, 7)

    def run(self):
        for i in range(1, self.num_cars + 1):
            processing_time = random.randint(1, 3)
            print(f"🚗 Xe {i} đang qua {self.lane_name}... (xử lý {processing_time} giây)")
            time.sleep(processing_time)
            print(f"✅ Xe {i} đã qua {self.lane_name} (sau {processing_time} giây)")

        print(f"⌘ {self.lane_name} đã xử lý xong {self.num_cars} xe.\n")

if __name__ == "__main__":
    # Tạo 3 thread đại diện cho 3 làn xe
    lane_a = TollLane("Làn A")
    lane_b = TollLane("Làn B")
    lane_c = TollLane("Làn C")

    # Bắt đầu chạy song song
    lane_a.start()
    lane_b.start()
    lane_c.start()

    # Chờ các thread hoàn tất
    lane_a.join()
    lane_b.join()
    lane_c.join()

    print("🎉 Tất cả các xe đã được xử lý xong.")
```

```

🔄 Xe 1 đang qua Làn A... (xử lý 1 giây)
🚗 Xe 1 đang qua Làn B... (xử lý 3 giây)
🚗 Xe 1 đang qua Làn C... (xử lý 2 giây)
✅ Xe 1 đã qua Làn A (sau 1 giây)
🚗 Xe 2 đang qua Làn A... (xử lý 2 giây)
✅ Xe 1 đã qua Làn C (sau 2 giây)
🚗 Xe 2 đang qua Làn C... (xử lý 3 giây)
✅ Xe 1 đã qua Làn B (sau 3 giây)
🚗 Xe 2 đang qua Làn B... (xử lý 1 giây)
✅ Xe 2 đã qua Làn A (sau 2 giây)
🚗 Xe 3 đang qua Làn A... (xử lý 2 giây)
✅ Xe 2 đã qua Làn B (sau 1 giây)
🚗 Xe 3 đang qua Làn B... (xử lý 3 giây)
✅ Xe 3 đã qua Làn A (sau 2 giây)
🚗 Xe 4 đang qua Làn A... (xử lý 2 giây)
✅ Xe 2 đã qua Làn C (sau 3 giây)
🚗 Xe 3 đang qua Làn C... (xử lý 3 giây)
✅ Xe 3 đã qua Làn B (sau 3 giây)
🚗 Xe 4 đang qua Làn B... (xử lý 1 giây)
✅ Xe 4 đã qua Làn A (sau 2 giây)
🚗 Xe 5 đang qua Làn A... (xử lý 2 giây)
✅ Xe 4 đã qua Làn B (sau 1 giây) ✅ Xe 3 đã qua Làn C (sau 3 giây)
🚗 Xe 4 đang qua Làn C... (xử lý 2 giây)

🚗 Xe 5 đang qua Làn B... (xử lý 2 giây)
✅ Xe 5 đã qua Làn A (sau 2 giây)
🚗 Xe 6 đang qua Làn A... (xử lý 3 giây)
✅ Xe 4 đã qua Làn C (sau 2 giây) ✅ Xe 5 đã qua Làn B (sau 2 giây)
⌘ Làn B đã xử lý xong 5 xe.

🚗 Xe 5 đang qua Làn C... (xử lý 2 giây)
✅ Xe 6 đã qua Làn A (sau 3 giây)
⌘ Làn A đã xử lý xong 6 xe.

✅ Xe 5 đã qua Làn C (sau 2 giây)
🚗 Xe 6 đang qua Làn C... (xử lý 2 giây)
✅ Xe 6 đã qua Làn C (sau 2 giây)
⌘ Làn C đã xử lý xong 6 xe.

🎉 Tất cả các xe đã được xử lý xong.

```

📌 Bài tập 1 – Quán cà phê đa luồng Mô tả: Có 3 nhân viên pha chế (Barista A, B, C). Mỗi người phục vụ một đơn hàng trong 1–2 giây. Mỗi người xử lý 5 đơn hàng. In ra ai đang pha cà phê nào và mất bao lâu.

Yêu cầu:

Dùng threading.Thread

Mỗi thread đại diện cho 1 nhân viên

In log như: Barista A đang pha cà phê #2 (2s) → Đã xong cà phê #2

📌 Bài tập 2 – Tính tổng song song Mô tả: Cho một list lớn gồm 10 triệu số nguyên. Hãy chia thành 4 phần và tính tổng từng phần bằng 4 process riêng biệt.

Yêu cầu:

Dùng multiprocessing.Process hoặc Pool

Mỗi process tính tổng 1 đoạn

In tổng từng phần và tổng cuối cùng

📌 Bài tập 3 – Tải file song song Mô tả: Giả lập 3 file cần được "tải xuống", mỗi file mất 2–4 giây. Mỗi tiến trình sẽ tải một file.

Yêu cầu:

Dùng concurrent.futures.ProcessPoolExecutor hoặc ThreadPoolExecutor

Random thời gian tải từng file

In thứ tự hoàn tất của từng file (có thể không đúng thứ tự ban đầu)

Double-click (or enter) to edit

