Multi-Threading in Python (with Annotations)

1. Introduction

- Multithreading allows programs to perform tasks concurrently.
- Python provides the `threading` module and `concurrent.futures.ThreadPoolExecutor`.
- Threads are useful for I/O-bound tasks but limited for CPU-bound tasks due to the GIL.
- Using annotations (type hints and decorators) helps clarify threading-related code.

---

2. Importing Modules

```python
import threading
import time
from typing import Any, Callable
```

---

3. Creating and Running a Thread

```python
def worker() -> None:   # function returns nothing
    print("Thread is running")

t: threading.Thread = threading.Thread(target=worker)
t.start()
t.join()
```

---

4. Naming Threads

```python
def task() -> None:
    print(f"Running in thread: {threading.current_thread().name}")

t1: threading.Thread = threading.Thread(target=task, name="Worker-1")
t1.start()
t1.join()
```

---

5. Using `threading.current_thread()`

```python
print("Main thread:", threading.current_thread().name)
```

---

6. Daemon Threads

```
def background() -> None:
    while True:
        print("Background task running...")
        time.sleep(2)

t: threading.Thread = threading.Thread(target=background, daemon=True)
t.start()
time.sleep(5)
```

---

7. Extending the `Thread` Class

```
class MyThread(threading.Thread):
    def run(self) -> None:  # overriding with annotation
        print(f"Custom thread running: {self.name}")

t: MyThread = MyThread()
t.start()
t.join()
```

---

8. Locking (Thread Synchronization)

```
lock: threading.Lock = threading.Lock()
counter: int = 0

def increment() -> None:
    global counter
    for _ in range(100000):
        with lock:
            counter += 1

t1: threading.Thread = threading.Thread(target=increment)
t2: threading.Thread = threading.Thread(target=increment)

t1.start(); t2.start()
t1.join(); t2.join()
print("Final counter:", counter)
```

---

9. Condition Variables

```
condition: threading.Condition = threading.Condition()
data_ready: bool = False

def producer() -> None:
    global data_ready
    with condition:
        print("Producing data...")
        data_ready = True
        condition.notify()

def consumer() -> None:
    with condition:
        condition.wait()
        if data_ready:
            print("Consuming data...")

threading.Thread(target=consumer).start()
threading.Thread(target=producer).start()
```

---

10. ThreadPoolExecutor (Annotated)

```
from concurrent.futures import ThreadPoolExecutor
from typing import List

def square(n: int) -> int:
    return n * n

with ThreadPoolExecutor(max_workers=3) as executor:
    results: List[int] = list(executor.map(square, [1, 2, 3, 4, 5]))
    print(results)
```

---

11. Using Decorators with Threads

We can use decorators to simplify thread launching.

```
from functools import wraps

def run_in_thread(func: Callable[..., Any]) -> Callable[..., threading.Thread]:
    @wraps(func)
    def wrapper(*args, **kwargs) -> threading.Thread:
        t = threading.Thread(target=func, args=args, kwargs=kwargs)
```

```
        t.start()
        return t
    return wrapper

@run_in_thread
def delayed_task(name: str, delay: int) -> None:
    time.sleep(delay)
    print(f"Task {name} completed after {delay}s")

# Runs asynchronously
delayed_task("A", 2)
delayed_task("B", 3)
```

---

12. Commonly Used Annotations in Multi-Threading

Annotation / Type Hint	Use Case
`-> None`	Methods returning nothing (e.g., thread worker functions)
`Callable[..., Any]`	General function type used for thread targets
`threading.Thread`	Explicit type for threads
`threading.Lock`	Explicit type for lock objects
`threading.Condition`	For signaling between threads
`List[Type]`	Type hint for storing multiple thread results
`@wraps(func)`	Preserves function metadata when writing thread decorators

---

13. Best Practices

- Annotate thread worker functions with `-> None`.
- Use `Callable` when writing decorators for threading.
- Type-hint synchronization objects (`Lock`, `Condition`, etc.).
- Prefer `ThreadPoolExecutor` with annotated results for clarity.

---

25 Exercises on Multi-Threading (with Annotations)

Beginner

1. Write a thread function with `-> None` annotation.
2. Start two annotated threads that print messages.
3. Annotate a thread function with a parameter (`name: str`).
4. Create a custom thread subclass with annotated `run()` method.
5. Demonstrate a daemon thread with annotation.

Intermediate

6. Use a global `counter: int` with annotated increment function.
7. Write a thread-safe increment function using `lock: threading.Lock`.
8. Annotate producer-consumer functions with `condition: threading.Condition`.
9. Create multiple threads with proper type hints.
10. Implement a countdown timer function with annotations.

Synchronization

11. Annotate a function that uses `Semaphore`.
12. Annotate an event-driven thread function.
13. Create a barrier synchronization example with annotations.
14. Use `Queue[int]` for inter-thread communication.
15. Annotate a function that consumes items from a queue.

Advanced

16. Use `ThreadPoolExecutor` with `List[int]` as result type.
17. Write a decorator `@run_in_thread` with annotations.
18. Annotate a function that downloads a file in a thread.
19. Annotate a function that logs messages from multiple threads.
20. Annotate a `BankAccount` withdrawal function running across threads.

Challenge

21. Create a thread pool where results are annotated as `List[str]`.
22. Implement a retry decorator with `Callable[..., Any]` type hints.
23. Write a thread function that takes a `Callable` argument.
24. Annotate a multi-threaded prime number checker.
25. Build a mini web scraper using `ThreadPoolExecutor` with proper annotations.
