Asyncio, Threads, & Multiprocessing — Concise Study Notes

This document collects explanations, corrected example code (with comments), and comparisons for: `async def`, `await`, `asyncio`, event loop, `aiohttp` examples, mixing threads with asyncio, multiprocessing with asyncio, daemon vs non-daemon threads, profiling, race conditions, and deadlocks.

Key concepts

`async def`

- Declares a **coroutine function** (a special function that returns a coroutine object when called).
- Coroutines can be *paused* and *resumed* by the event loop with `await`.
- Example: `async def foo():` — calling `foo()` does **not** run it immediately; it returns a coroutine.

`await`

- Used **inside** an `async def` function to pause execution until an awaitable (another coroutine, `asyncio.sleep`, or `Future`) completes.
- While awaiting, the event loop may run other coroutines.

`asyncio`

- Python's standard library for asynchronous I/O, cooperative multitasking, and event loops.
- Useful for many concurrent I/O-bound tasks (networking, file I/O with async libraries, timers).

Event loop

- The engine that schedules and runs coroutines and callbacks.
- You generally use `asyncio.run()` to start a high-level event loop and run a top-level coroutine.
- Common low-level functions: `asyncio.get_running_loop()`, `loop.create_task()`, `loop.run_until_complete()` (older style).

Example 1 — Single coroutine (brewing chai)

File: `1_async_one.py` (corrected & commented)

```python
import asyncio

async def brew_chai():
    # start the task
    print("Brewing chai ....")
    # yield control and sleep asynchronously for 2 seconds
    await asyncio.sleep(2)
    # resumes after sleep
    print("Chai is ready ....")

if __name__ == "__main__":
    # run the coroutine until complete using asyncio's event loop
    asyncio.run(brew_chai())
```

Behavior: `Brewing chai ....` prints immediately, ~2s later `Chai is ready ....` prints.

Example 2 — Concurrent coroutines with `gather`

File: `02_async_two.py` (corrected & commented)

```
import asyncio

async def brew(name):
    print(f"Brewing {name} ...")
    await asyncio.sleep(2)
    print(f"{name} is ready")

async def main():
    # Run three brew coroutines concurrently
    await asyncio.gather(
        brew("Masala chai"),
        brew("green chai"),
        brew("ginger chai"),
    )

if __name__ == "__main__":
    asyncio.run(main())
```

Behavior: All three start nearly together and finish ~2 seconds later (not 6 seconds).

Example 3 — `aiohttp` for asynchronous HTTP requests

File: `03_async_three.py` (corrected & commented)

```
import asyncio
import aiohttp

async def fetch_url(session, url):
    """Fetch a URL using a shared aiohttp session and print status."""
    async with session.get(url) as response:
        print(f"Fetched {url} with status {response.status}")

async def main():
    urls = ["https://httpbin.org/delay/2"] * 3
    # Reuse one ClientSession for connection pooling and efficiency
    async with aiohttp.ClientSession() as session:
        tasks = [fetch_url(session, url) for url in urls]
        await asyncio.gather(*tasks)

if __name__ == "__main__":
    asyncio.run(main())
```

Note: `aiohttp.ClientSession()` should be created inside an async context (`async with`) and reused for multiple requests.

Mixing threads with asyncio — when & why

- `asyncio` is good for **concurrency** of I/O-bound tasks using a single thread.
- **Threads** are useful when you need to run **blocking** code (CPU-bound or I/O code without async support) without blocking the event loop.
- Use `loop.run_in_executor()` to run blocking functions in a thread or process pool while staying inside `asyncio`.

Example 4 — run blocking function in a thread executor

File: `04_thread_async.py` (corrected & commented)

```
import asyncio
import time
from concurrent.futures import ThreadPoolExecutor

# A normal blocking function (not async)
def check_stock(item):
    print("Checking items in store...")
    time.sleep(3)  # blocking sleep
    return f"Item found: {item}"

async def main():
    # get the currently running event loop
    loop = asyncio.get_running_loop()

    # Create a ThreadPoolExecutor to run blocking functions
    with ThreadPoolExecutor() as pool:
        # schedule check_stock to run in a thread from the pool
        result = await loop.run_in_executor(pool, check_stock, "Masala Chai")

    print(result)
```

```
if __name__ == "__main__":
    asyncio.run(main())
```

Behavior: `check_stock` runs in a background thread (3s blocking), while the main event loop awaits; other async tasks could run concurrently while the thread works.

Async + multiprocessing (CPU-bound work)

- For CPU-bound tasks you want **process-level parallelism** (multiprocessing) because Python threads are limited by the GIL for CPU work.
- `ProcessPoolExecutor` can be used via `loop.run_in_executor()` to run CPU-bound functions in other processes.

File: `process_async.py` (corrected & commented)

```
import asyncio
from concurrent.futures import ProcessPoolExecutor

# CPU-bound function (pure Python example)
def encrypt(data):
    # pretend to do CPU-heavy work (reverse string shown for demo)
    return f"lock {data[::-1]}"

async def main():
    loop = asyncio.get_running_loop()
    # run encrypt in a separate process to avoid GIL limitations
    with ProcessPoolExecutor() as pool:
        result = await loop.run_in_executor(pool, encrypt,
"credit_card_1234")
        print(result)

if __name__ == "__main__":
    asyncio.run(main())
```

Background thread + asyncio example

File: `bgworker.py` (corrected & commented)

```
import asyncio
import threading
import time

# A background thread that logs system health every second
def background_threading():
    while True:
        time.sleep(1)
        print("Logging the system health")

async def fetch_order():
    # simulate async I/O
    await asyncio.sleep(3)
    print("order fetched")

if __name__ == "__main__":
    # Start the background thread as daemon so it won't block program exit
    threading.Thread(target=background_threading, daemon=True).start()
    asyncio.run(fetch_order())
```

Behavior: While the main coroutine sleeps for 3s, the background thread prints every second. Because it is a daemon thread, the program can exit once `fetch_order()` completes.

Daemon vs non-daemon threads

- **Daemon thread:** background worker that does not prevent process exit. When only daemon threads remain, the Python program exits.
- **Non-daemon thread:** the process waits for these threads to finish before exiting.

Corrected demonstration: `daemon.py` (daemon = True)

```
import threading
import time

def monitor_tea_temperature():
    while True:
        print("Monitoring tea temperature")
        time.sleep(2)

# daemon=True means this thread won't keep the process alive
t = threading.Thread(target=monitor_tea_temperature, daemon=True)
t.start()

print("Main program done")
# process will exit immediately after printing the line; the daemon thread
will be terminated
```

Non-daemon demonstration (rename file to `non_daemon.py` and set daemon=False or omit it):

```
import threading
import time

def monitor_tea_temperature():
    while True:
        print("Monitoring tea temperature")
        time.sleep(2)

# By default daemon is False, so this thread will keep the program alive
t = threading.Thread(target=monitor_tea_temperature, daemon=False)
t.start()

print("Main program done")
# program will keep running; to stop it you must terminate the thread or the process
```

Tip: Non-daemon threads should be joined or signaled to exit cleanly (use an `Event` object to request shutdown).

Debugging and profiling

Profiling

- Use the built-in profiler to find slow spots:

  ```
  python -m cProfile -s time your_script.py
  ```

- Consider `pyinstrument`, `yappi`, or `line_profiler` for richer views.

Race conditions and deadlocks

- **Race condition:** when multiple threads/processes access and modify shared state without proper synchronization, producing incorrect results.
- **Deadlock:** two or more threads/processes wait for locks held by each other and none can proceed.

Example of a deadlock-prone program (corrected for readability):

```
import threading
import time

lock_a = threading.Lock()
lock_b = threading.Lock()

def task_1():
    with lock_a:
        print("Task 1 acquired lock_a")
        time.sleep(0.1)
        with lock_b:
            print("Task 1 acquired lock_b")

def task_2():
    with lock_b:
```

```
        print("Task 2 acquired lock_b")
        time.sleep(0.1)
        with lock_a:
            print("Task 2 acquired lock_a")

# Start two threads; ordering can cause deadlock
t1 = threading.Thread(target=task_1)
t2 = threading.Thread(target=task_2)

t1.start()
t2.start()

# join to wait (in real deadlock this will hang)
t1.join()
t2.join()
```

Why deadlocks happen: if `task_1` holds `lock_a` and waits for `lock_b` while `task_2` holds `lock_b` and waits for `lock_a`, neither can proceed.

How to avoid deadlocks: - Always acquire locks in a consistent global order. - Use timeouts when acquiring locks (`lock.acquire(timeout=...)`) and handle failure. - Use higher-level concurrency primitives (Queues, `concurrent.futures`, `asyncio` synchronization primitives).

Quick comparisons & when to use what

Scenario	Recommended approach
Many concurrent network requests (I/O-bound)	`asyncio` + async libraries (e.g. `aiohttp`)
Blocking library you cannot change (e.g. legacy API)	Run it in a thread via `loop.run_in_executor(ThreadPoolExecutor)`
CPU-bound heavy computation	Use `multiprocessing` / `ProcessPoolExecutor`
Background periodic work that shouldn't block exit	Daemon thread or `asyncio.create_task()` + graceful shutdown

---

Practical tips for notes / studying

- Remember `asyncio` **is single-threaded by default**; concurrency comes from non-blocking I/O and switching between suspended coroutines.
- Use `asyncio.gather()` to run coroutines concurrently, `asyncio.create_task()` to start a task and continue.
- For mixed workloads, combine: `asyncio` for I/O, `ThreadPoolExecutor` for blocking calls, and `ProcessPoolExecutor` for CPU-heavy tasks.
- Always clean up `aiohttp.ClientSession()` (use `async with`), and shutdown executors properly.

If you want, I can also: - Provide short quiz questions for self-testing. - Add diagrams showing timeline differences (threads vs asyncio vs processes). - Produce a printable PDF of these notes.