Concurrency Exercises — Applying the Models

Why these exercises exist

Concurrency concepts are best understood by applying them to realistic workloads. These exercises demonstrate why Python provides multiple concurrency models and when each should be used.

Exercise 1 — I/O-Bound API Aggregation (Threads)

```
from concurrent.futures import ThreadPoolExecutor
import time

def fetch_api(i):
    time.sleep(1)
    return f"data-{i}"

with ThreadPoolExecutor(max_workers=5) as executor:
    results = list(executor.map(fetch_api, range(5)))
```

This exercise shows how threads overlap waiting time for I/O-bound tasks, significantly reducing wall-clock execution time.

Exercise 2 — Event-Driven I/O (async / await)

```
import asyncio

async def fetch(i):
    await asyncio.sleep(1)
    return f"data-{i}"

async def main():
    results = await asyncio.gather(*(fetch(i) for i in range(5)))
    print(results)

asyncio.run(main())
```

This exercise demonstrates scalable concurrency using a single-threaded event loop and cooperative multitasking.

Exercise 3 — CPU-Bound Computation (Processes)

```
from concurrent.futures import ProcessPoolExecutor

def compute(n):
    total = 0
    for i in range(n):
        total += i * i
    return total

with ProcessPoolExecutor() as executor:
    results = list(executor.map(compute, [10_000_000] * 4))
```

This exercise shows how processes bypass the GIL and enable true parallelism for CPU-bound workloads.

Key Insight

Correctly classifying workloads and choosing the appropriate concurrency model leads to simpler, faster, and more reliable systems.