Module 17 of 20

Concurrency — threads, processes and asyncio

The GIL and the CPU-bound/I/O-bound decision; threads with Lock, Queue and Event and the race you must see once; concurrent.futures executors with ordered map; multiprocessing with pickling and the main guard; asyncio from coroutines and gather to TaskGroup, semaphores and queues; and the determinism rule for every concurrent program.

Python's concurrency story starts with one fact — the global interpreter lock lets one thread run bytecode at a time — and everything else follows from it. Threads do not speed up computation but do overlap waiting; processes get real parallelism at the price of pickling and start-up; asyncio makes waiting explicit and single-threaded, so ten thousand connections cost almost nothing. Choosing among the three is the first skill; writing programs whose output does not depend on the scheduler is the second, and it is the one this module enforces in every exercise.

The GIL and the three models classifies work as CPU-bound or I/O-bound and gives the decision table. Threads covers `Thread`, `join`, the race in `counter += 1`, `Lock` and `RLock`, `queue.Queue` with sentinels, and `Event`, `Semaphore` and `Barrier`. concurrent.futures covers executors, futures, `map` in submission order versus `as_completed`, and process pools with a main guard. multiprocessing goes lower: `Process`, `Pool` with `map` / `imap` / `starmap` and `chunksize`, pickling, start methods, `Queue` and shared state in outline. asyncio basics covers coroutines, `await`, `gather`, tasks, `sleep(0)` and the FIFO loop, timeouts and the blocking mistake; asyncio patterns adds `TaskGroup`, semaphores, queues, async iteration and bridging to threads and processes.

The exercises are whole programs whose output is deterministic by construction: per-thread result slots read after `join`, a locked counter, queue workers merged in order, `executor.map` over words and primes, `starmap` and a process `Queue`, `gather` with a finish log, a semaphore-bounded fan-out and an async producer-consumer under a `TaskGroup`. The checkpoint adds a threaded word counter, a queue pipeline keyed by index, and an asyncio job runner with a reproducible trace.

CONTENTS

1 The GIL and the three models — threads, processes, asyncio 14 min

2	Threads — Thread, Lock, Event, Queue and the race you must see once	15 min
3	concurrent.futures — executors, futures, map and as_completed	13 min
4	multiprocessing — Process, Pool, pickling, queues and shared state	13 min
5	asyncio basics — coroutines, await, tasks and gather	15 min
6	asyncio patterns — TaskGroup, queues, semaphores, async iteration and bridging	14 min
✓	Checkpoint — Concurrency	25 min

Exercises in this module compile on CPython 3.11. Reference solutions and the answer key follow each lesson.

LESSON 1 · 14 MIN

The GIL and the three models — threads, processes, asyncio

Python has three ways to do more than one thing at a time, and choosing among them starts from one fact about CPython: the *global interpreter lock* allows only one thread to execute Python bytecode at any moment. Threads therefore do not speed up CPU-bound Python code, but they do speed up code that *waits* — on the network, the disk, a subprocess — because a waiting thread releases the lock. Processes sidestep the lock with separate interpreters; asyncio gives waiting a single-threaded, explicit form. This lesson explains the GIL and its consequences, classifies work as CPU-bound or I/O-bound, gives the decision table for the three models, and states the rule that keeps concurrent programs testable and judgeable: collect results in a defined order, then print.

The GIL

CPython's memory management is reference counting, and updating a count from two threads at once would corrupt it; the GIL is the mutex that prevents that by serialising bytecode execution. A thread holds the lock while running Python code, and *releases it* when it blocks on I/O (a socket read, a file read, `time.sleep`) or when a C extension explicitly drops it (NumPy's numerical loops, `hashlib`, `zlib`). Every 5 ms (`sys.getswitchinterval()`) a running thread is asked to yield so others can run.

Consequences:

- Two threads doing pure-Python arithmetic take as long as one thread doing both — sometimes longer, from contention.
- Two threads each waiting on a network response overlap almost perfectly: while one waits, the other runs.
- Thread switches can happen between any two bytecodes, so `count += 1` from two threads is a *race* — the read and the write are separate operations (lesson 2).

Python 3.12 gave each sub-interpreter its own GIL and 3.13 ships an experimental free-threaded build (`--disable-gil`); both are reading for this track — on 3.11 the GIL is simply there.

CPU-bound versus I/O-bound

Work	Examples	Bottleneck	Model
CPU-bound	parsing, hashing in Python, image processing in pure Python, simulation	the processor	processes (`multiprocessing`, `ProcessPoolExecutor`), or a C extension that releases the GIL
I/O-bound, few tasks	reading a handful of files, a few HTTP calls, a subprocess	waiting	threads (`threading`, `ThreadPoolExecutor`)
I/O-bound, many tasks	thousands of connections, a chat server, a scraper with high concurrency	waiting, at scale	`asyncio`
Mixed	fetch then compute	both	asyncio or threads for the fetch, a process pool for the compute

The test is: if the program had infinitely fast I/O, would it still be slow? Then it is CPU-bound and threads will not help. `time.perf_counter` around the work versus around the waiting tells you which.

The three models

Threads share memory: one process, several threads, every object visible to all. Cheap to start, simple to reason about for a handful of workers, and dangerous exactly because everything is shared — the lock discipline of the next lesson is mandatory. Blocking calls just work.

Processes share nothing: each is a separate interpreter with its own GIL and memory. True parallelism for CPU work at the cost of start-up time (tens of milliseconds each), and of passing data by *pickling* it — arguments and results must be picklable, and copying a large object to a worker can cost more than the work. `concurrent.futures.ProcessPoolExecutor` hides most of the mechanics.

asyncio is one thread with an event loop: coroutines (`async def`) run until they `await` something that is not ready, hand control back to the loop, and are resumed when it is. No parallelism at all — one thing runs at a time — but tens of thousands of concurrent waits with

almost no memory, and switches happen only at `await`, which makes shared state safe *between* awaits. The cost: every blocking call in the program must be an `await`able one; a single `time.sleep(1)` freezes the whole loop.

The determinism rule

Concurrent programs are nondeterministic *in scheduling*: which thread runs first, which task finishes first, the order in which messages interleave. Output that depends on that order differs from run to run and cannot be tested or judged. The rule for every exercise in this module, and for well-designed programs generally:

Give each worker a slot for its result — an index into a list, a key in a dict, a return value collected by `map` in submission order.

Join every worker (or `gather` every task) before reading results.

Print the aggregated, ordered results *after* the join, from the main thread.

`executor.map` returns results in the order the inputs were submitted regardless of completion order; `asyncio.gather` does the same for coroutines; `as_completed` does not, and its results must be sorted before printing. A worker that prints directly interleaves with others and the interleaving is arbitrary.

What to say in an interview

The GIL serialises bytecode, so threads are for I/O and processes for CPU; asyncio is cooperative single-threaded concurrency for many I/O waits; a race is two threads touching one object between bytecodes; a lock makes a compound operation atomic; a deadlock is two locks acquired in different orders; the fix for shared state is to not share it — queues, immutable messages, results returned rather than written. Those six sentences are the whole of what most interviewers want, and the rest of this module is the practice behind them.

Pitfalls

Threads for a CPU-bound loop, then surprise that nothing got faster.

A blocking call (`requests.get`, `time.sleep`, a file read) inside an `async def`.

Printing from workers and expecting a stable order.

Reading results before the join.

Passing an unpicklable object (a lambda, an open file, a lock) to a process pool.

Assuming `x += 1` is atomic.

Key takeaways

The GIL lets one thread run Python bytecode at a time; it is released during I/O and in some C extensions.

CPU-bound work wants processes (or a GIL-releasing extension); I/O-bound work wants threads for a few tasks and asyncio for many.

Threads share memory and need locks; processes share nothing and pickle data; asyncio switches only at `await` and must never block.

Collect results into ordered slots, join everything, then print — the only way concurrent output is deterministic.

`x += 1` is not atomic; a race is a switch between its read and its write.

EXERCISE 1

Classify the work

Read lines `<name> <cpu_ms> <io_ms> <tasks>` until EOF. Let the I/O share be `io_ms / (cpu_ms + io_ms)`. A share of at least `0.75` is `io-bound` and recommends `threads` when `tasks` is at most 50, otherwise `asyncio`; a share of at most `0.25` is `cpu-bound` and recommends `processes`; anything else is `mixed` and recommends `threads+processes`. Print `<name>: <class> → <model>` per line, then `cpu-bound <a>, io-bound <b>, mixed <c>`.

Input: one job per line. **Output:** one line per job, then the summary.

```
parse 400 0 8
crawl 10 190 500
fetch 20 180 6
render 100 100 4
```

prints

```
parse: cpu-bound → processes
crawl: io-bound → asyncio
fetch: io-bound → threads
render: mixed → threads+processes
cpu-bound 1, io-bound 2, mixed 1
```

INPUT 1

```
parse 400 0 8
crawl 10 190 500
fetch 20 180 6
render 100 100 4
```

OUTPUT 1

```
parse: cpu-bound → processes
crawl: io-bound → asyncio
fetch: io-bound → threads
render: mixed → threads+processes
cpu-bound 1, io-bound 2, mixed 1
```

REFERENCE SOLUTION

```python
import sys


def classify(cpu_ms, io_ms, tasks):
    """Return (kind, model) for one job."""
    share = io_ms / (cpu_ms + io_ms)
    if share ⩾ 0.75:
        return "io-bound", ("threads" if tasks ⩽ 50 else "asyncio")
    if share ⩽ 0.25:
        return "cpu-bound", "processes"
    return "mixed", "threads+processes"


counts = {"cpu-bound": 0, "io-bound": 0, "mixed": 0}
for line in sys.stdin:
    parts = line.split()
    if len(parts) ≠ 4:
        continue
    name, cpu_ms, io_ms, tasks = parts[0], int(parts[1]), int(parts[2]), int(parts[3
    kind, model = classify(cpu_ms, io_ms, tasks)
    counts[kind] += 1
    print(f"{name}: {kind} → {model}")

print(f"cpu-bound {counts['cpu-bound']}, io-bound {counts['io-bound']}, mixed {count
```

EXERCISE 2

Slots after join

Read one line of integers. Start one thread per integer, named `worker-<i>`, whose target computes `sum(k*k for k in range(1, n + 1))` and stores `(threading.current_thread().name, value)` in `results[i]` — its own slot. Join every thread, then print `<name> n=<n> → <value>` in slot order and `total <sum of values>`. Nothing may be printed from inside a worker.

Input: one line of integers. **Output:** one line per slot, then the total.

```
3 5 10
```

prints

```
worker-0 n=3 → 14
worker-1 n=5 → 55
worker-2 n=10 → 385
total 454
```

INPUT 1

```
3 5 10
```

OUTPUT 1

```
worker-0 n=3 → 14
worker-1 n=5 → 55
worker-2 n=10 → 385
total 454
```

REFERENCE SOLUTION

```python
import threading

nums = [int(x) for x in input().split()]
results = [None] * len(nums)


def work(i, n):
    value = sum(k * k for k in range(1, n + 1))
    results[i] = (threading.current_thread().name, value)


threads = [threading.Thread(target=work, args=(i, n), name=f"worker-{i}") for i, n i
for t in threads:
    t.start()
for t in threads:
    t.join()

for n, (name, value) in zip(nums, results):
    print(f"{name} n={n} → {value}")
print("total", sum(value for _, value in results))
```

QUIZ

1. What does the GIL guarantee?

A. That threads never run

B. That only one thread executes Python bytecode at a time

C. That every operation is atomic

D. That processes share memory

2. A pure-Python loop takes 4 s. Split across 4 threads, roughly how long does it take?

A. 1 s

B. About 4 s, possibly more

C. 2 s

D. 0.5 s

3. Which model suits ten thousand concurrent network connections?

A. A thread per connection

B. A process per connection

C. asyncio

D. A single blocking loop

4. Why is `x += 1` from two threads a race?

A. Because integers are immutable

B. Because it is three bytecodes — load, add, store — and a switch can happen between them

C. Because threads cannot see globals

D. It is not; the GIL makes it atomic

5. What is the determinism rule for concurrent output?

A. Print from each worker as it finishes

B. Sleep long enough for workers to finish

C. Collect results into ordered slots, join everything, then print from the main thread

D. Use a single worker

ANSWER KEY · THE GIL AND THE THREE MODELS — THREADS, PROCESSES, ASYNCIO

1. B It serialises bytecode execution; it is released during blocking I/O and by some C extensions, and it does not make compound operations atomic.

2. B CPU-bound bytecode runs one thread at a time; contention can make it slower. Processes are the tool for CPU-bound work.

3. C Coroutines cost almost nothing per wait; ten thousand threads or processes would exhaust memory and scheduling.

4. B Two threads can both load the same value, both add, and both store — losing one increment.

5. C Scheduling order is arbitrary; output must depend only on the data.

LESSON 2 · 15 MIN

Threads — Thread, Lock, Event, Queue and the race you must see once

A thread is a second flow of control inside the same process, sharing every object with the first. `threading.Thread` starts one; `join` waits for it; a `Lock` serialises access to shared state; a `queue.Queue` moves work between threads safely; an `Event` signals. The lesson's centrepiece is the race condition: two threads incrementing one counter and losing updates — shown, explained at the bytecode level, and fixed — because until you have seen it you will not believe a one-line `+=` can be wrong. It ends with the recipes that keep threaded programs deterministic: per-thread result slots, one lock per shared structure, join before reading.

Starting and joining

```
import threading

def work(name, results, i):
    results[i] = f"{name} done"          # each thread writes its own slot

results = [None] * 3
threads = [threading.Thread(target=work, args=(f"t{i}", results, i)) for i in
for t in threads:
    t.start()
for t in threads:
    t.join()                              # wait for every thread before readi
print(results)                            # ['t0 done', 't1 done', 't2 done']
```

`Thread(target=fn, args=(...), kwargs={...}, name=..., daemon=False)`; `start()` runs `fn` in the new thread; `join(timeout=None)` blocks until it finishes. A *daemon* thread is killed when the main thread exits — right for a background heartbeat, wrong for anything that writes. `threading.current_thread().name` identifies the running thread. Subclassing `Thread` and overriding `run` is the older style; `target=` is clearer.

The race

```
counter = 0

def bump(n):
    global counter
    for _ in range(n):
        counter += 1

threads = [threading.Thread(target=bump, args=(100_000,)) for _ in range(4)]
for t in threads: t.start()
for t in threads: t.join()
print(counter)          # often less than 400000 — updates lost
```

`counter += 1` compiles to *load* `counter`, *add* 1, *store* `counter` — three bytecodes. The GIL can switch threads between any two of them: thread A loads 41, thread B loads 41, both store 42, and one increment is gone. (On 3.11 the switch interval makes the loss visible only with many iterations; on the judge the loss is real but its *size* varies, so an exercise must never print the racy value.) Any read-modify-write on shared state — `+=`, `d[k] = d[k] + 1`, `if x not in s: s.add(x)`, `xs.append` followed by `len(xs)` — is the same bug.

Locks

```
lock = threading.Lock()

def bump(n):
    global counter
    for _ in range(n):
        with lock:                          # acquire … release, even on except
            counter += 1
```

A `Lock` is held by one thread at a time; `with lock:` acquires it on entry and releases on exit. Inside the block the read-modify-write is *atomic with respect to other holders of the same lock*. Rules: one lock per shared structure, held for the shortest possible time, never while doing I/O or calling unknown code, and always acquired in the same order when a thread needs two (lock ordering is the cure for deadlock). `RLock` may be re-acquired by the thread that holds it — for recursive code — and `lock.acquire(timeout=1.0)` avoids waiting forever.

Some operations *are* atomic under the GIL — a single `list.append`, `dict[k] = v`, reading one attribute — but compound operations built from them are not, and relying on the

atomicity of individual operations is how programs become correct by accident and wrong after a refactor. Lock the compound operation.

Queue: the safe way to share

```
import queue

q = queue.Queue()

def producer(items):
    for item in items:
        q.put(item)
    q.put(None)                             # a sentinel: no more work

def consumer(results):
    while (item := q.get()) is not None:
        results.append(item * 2)
        q.task_done()
```

`queue.Queue` is thread-safe: `put` and `get` are locked internally, `get` blocks until an item is available (or `timeout=`), `maxsize` makes `put` block when full (back-pressure), and `join()` waits until every item has been `task_done()`. A queue turns "shared state with locks" into "messages between owners", which is the design that scales: each thread owns its data and communicates by passing values. `LifoQueue` and `PriorityQueue` are the stack and heap variants.

Event, Semaphore, Barrier

```
stop = threading.Event()
def poll():
    while not stop.is_set():
        check_once()
        stop.wait(0.5)                    # sleeps, but wakes early when set
stop.set()                                # from the main thread: tell pollers

limit = threading.Semaphore(3)            # at most three at once
with limit:
    call_api()

sync = threading.Barrier(4)               # four threads wait for each other,
sync.wait()
```

`Event` is a one-bit flag threads wait on; the graceful-shutdown signal. `Semaphore(n)` is a counting lock — a rate limiter for a shared resource. `Barrier(n)` synchronises a phase. `threading.local()` gives each thread its own attribute namespace for state that must not be shared (a per-thread connection).

Deterministic recipes

Per-thread result slots: `results[i] = …` with `i` fixed at creation; no shared counter to race on.

One aggregation after join: sum, sort, merge the slots in the main thread.

Pool the work: `ThreadPoolExecutor.map` (next lesson) gives results in submission order.

Lock the compound operations on anything genuinely shared, and print nothing from inside a thread.

Sleep is not synchronisation: `time.sleep` to "let the other thread finish" is a race with extra steps; `join`, `Event` and `Queue.join` are the tools.

Pitfalls

`x += 1` on shared state without a lock.

Two locks acquired in different orders by different threads — deadlock.

Holding a lock during I/O, serialising the whole program.

A daemon thread that writes files.

Printing from threads and reading the interleaving as meaningful.

Reading results before `join`.

Key takeaways

`Thread(target=…)`, `start()`, `join()`; daemon threads die with the process; join before reading results.

`counter += 1` is three bytecodes; two threads lose updates — the race — and `with lock:` makes the compound operation atomic.

One lock per shared structure, held briefly, acquired in a fixed order; `RLock` for re-entrancy.

`queue.Queue` passes work between threads safely with blocking `get` / `put`, sentinels and `join`; `Event`, `Semaphore`, `Barrier` signal, limit and synchronise.

Give each thread its own result slot, aggregate after the join, never print from a worker.

EXERCISE 1

A counter with a lock

Write `SafeCounter` with a `value` attribute, a `threading.Lock` and an `increment()` method that adds 1 under the lock. Read `T N`; start `T` threads that each call `increment()` `N` times; join them; print `expected <T*N>`, `counter <value>` and `lost <expected - value>` (which must be 0).

Input: `T N`. **Output:** three lines.

```
4 10000
```

prints

```
expected 40000
counter 40000
lost 0
```

INPUT 1

```
4 10000
```

OUTPUT 1

```
expected 40000
counter 40000
lost 0
```

REFERENCE SOLUTION

```
import threading


class SafeCounter:
    def __init__(self):
        self.value = 0
        self._lock = threading.Lock()

    def increment(self):
        with self._lock:
            self.value += 1


def bump(counter, n):
    for _ in range(n):
        counter.increment()


t_count, n = map(int, input().split())
counter = SafeCounter()
threads = [threading.Thread(target=bump, args=(counter, n)) for _ in range(t_count)]
for t in threads:
    t.start()
for t in threads:
    t.join()

expected = t_count * n
print("expected", expected)
print("counter", counter.value)
print("lost", expected - counter.value)
```

EXERCISE 2

Producer and consumers

Read `K` on the first line and integers on the second. A producer thread puts every integer on a `queue.Queue`, then `K` `None` sentinels. `K` consumer threads each loop on `get()` until they receive a sentinel, appending the square of each item to their own list `slots[i]` and calling `task_done()`. Join everything, merge the slots, sort, and print `processed <count>` and `squares <space-separated sorted squares>`.

Input: `K`, then the integers. **Output:** two lines.

```
2
3 1 4 1 5
```

prints

```
processed 5
squares 1 1 9 16 25
```

INPUT 1

```
2
3 1 4 1 5
```

OUTPUT 1

```
processed 5
squares 1 1 9 16 25
```

REFERENCE SOLUTION

```
import queue
import threading

k = int(input())
items = [int(x) for x in input().split()]
q = queue.Queue()
slots = [[] for _ in range(k)]


def producer():
    for item in items:
        q.put(item)
    for _ in range(k):
        q.put(None)


def consumer(i):
    while True:
        item = q.get()
        if item is None:
            q.task_done()
            break
        slots[i].append(item * item)
        q.task_done()


threads = [threading.Thread(target=producer)] + [threading.Thread(target=consumer, a
for t in threads:
    t.start()
for t in threads:
    t.join()

merged = sorted(x for slot in slots for x in slot)
print("processed", len(merged))
print("squares", " ".join(map(str, merged)))
```

QUIZ

1. What does `t.join()` do?

A. Starts the thread

B. Blocks the caller until the thread finishes

C. Kills the thread

D. Merges two threads

2. What happens to a daemon thread when the main thread exits?

A. It keeps running

B. It is killed abruptly

C. It is joined first

D. It raises

3. Two threads each need locks A and B. How is deadlock avoided?

A. Use more locks

B. Always acquire them in the same order

C. Use daemon threads

D. Sleep between acquisitions

4. Why is `queue.Queue` preferred over a shared list with a lock?

A. It is faster

B. It is thread-safe by construction, blocks on `get`, and turns shared state into messages between owners

C. It is unbounded

D. It preserves insertion order

5. Which is a correct way to tell polling threads to stop?

A. `time.sleep(1)`

B. Set a `threading.Event` they check and wait on

C. Delete the thread objects

D. Raise in the main thread

ANSWER KEY · THREADS — THREAD, LOCK, EVENT, QUEUE AND THE RACE YOU MUST SEE ONCE

1. B Join before reading anything a thread produced.

2. B Daemon threads are for background work that may be abandoned — never for anything that writes.

3. B With a fixed order, no thread can hold B while waiting for A held by a thread waiting for B.

4. B Each thread owns its data and communicates by passing values — the design that scales.

5. B `stop.set()` wakes `stop.wait()` early; sleeping is not synchronisation.

LESSON 3 · 13 MIN

concurrent.futures — executors, futures, map and as_completed

`concurrent.futures` is the interface most concurrent Python code should use: an *executor* runs callables on a pool of threads or processes, each call returns a *future* that will hold the result, and the same three methods — `submit`, `map`, `shutdown` — work for both pools. The result of a job is fetched with `future.result()`, which re-raises any exception the job raised, and `map` hands results back in submission order — the property that makes threaded output deterministic. This lesson covers the two executors, `submit` versus `map`, `as_completed` and `wait`, exceptions and timeouts, pool sizing, and when to switch from threads to processes.

The executor and the future

```
from concurrent.futures import ThreadPoolExecutor

def fetch(url):
    return len(download(url))

with ThreadPoolExecutor(max_workers=4) as pool:      # shutdown(wait=True) on
    future = pool.submit(fetch, "https://a")          # starts as soon as a wo
    ...
    size = future.result()                            # blocks until done; re-
```

`submit(fn, *args, **kwargs)` schedules one call and returns a `Future` immediately. `result(timeout=None)` waits for and returns the value; `exception()` returns the exception instead of raising; `done()`, `running()`, `cancel()` inspect and control it; `add_done_callback(fn)` runs `fn(future)` on completion. The `with` block waits for every submitted job before exiting, so results are always complete after it.

map: ordered results

```
with ThreadPoolExecutor(max_workers=4) as pool:
    sizes = list(pool.map(fetch, urls))               # results in the order o
```

`map(fn, *iterables, timeout=None, chunksize=1)` submits every element and yields results *in input order*, blocking on each in turn. It is the one-line replacement for "start N threads, join them, read the slots" and the tool to reach for first. Its limitation: an exception from any element is raised when its position is reached, and iteration stops there — for per-job error handling, use `submit` and inspect each future.

---

as_completed and wait

```
from concurrent.futures import as_completed, wait, FIRST_COMPLETED

futures = {pool.submit(fetch, url): url for url in urls}
for future in as_completed(futures):                  # yields futures as they
    url = futures[future]
    try:
        print(url, future.result())
    except Exception as e:
        print(url, "failed:", e)

done, pending = wait(futures, timeout=5, return_when=FIRST_COMPLETED)
```

`as_completed` is for showing progress or acting on results as they arrive; because its order is the completion order, anything it produces must be **sorted** before it is printed as an answer. `wait` blocks until all, or the first, or the first exception, with a timeout, and returns the two sets. Mapping futures back to their inputs with a dict, as above, is the standard idiom.

Processes

```
from concurrent.futures import ProcessPoolExecutor

def cpu_heavy(n):
    return sum(i * i for i in range(n))

if __name__ == "__main__":                            # required: workers impo
    with ProcessPoolExecutor() as pool:               # max_workers defaults t
        totals = list(pool.map(cpu_heavy, [10**6] * 4, chunksize=1))
```

Same interface, separate interpreters, real parallelism for CPU-bound functions. The constraints: the function must be importable by name (a module-level `def`, not a lambda or

a nested function), arguments and results must be picklable, the main module must guard its entry point (workers re-import it), and each task carries a fixed cost of pickling and process communication — batch small tasks with `chunksize` or they will be slower than a plain loop.

Exceptions and timeouts

A job that raises does not crash the pool; the exception is stored in its future and re-raised by `result()` (or returned by `exception()`). `result(timeout=2)` raises `TimeoutError` if the job is not done in time — the job itself keeps running; futures cannot interrupt a running callable, only `cancel()` one that has not started. Design long jobs to check a stop flag or an `Event` themselves.

Sizing the pool

For threads doing I/O, more workers than cores is normal — 10 to 50 for network calls, bounded by what the remote side tolerates; the default is `min(32, cpu_count + 4)`. For processes doing CPU work, the CPU count (the default) is the ceiling that helps. Too many workers on a rate-limited API produces failures, not speed; a `Semaphore` inside the job or a smaller pool is the fix. Measure with `perf_counter` around the whole `with` block.

A pattern for deterministic output

```
def process(item):
    return item, expensive(item)                    # return the key with th

with ThreadPoolExecutor() as pool:
    results = dict(pool.map(process, items))        # or sorted(...) — order
for key in sorted(results):
    print(key, results[key])
```

Return the identity of the work with its result, collect everything inside the `with`, then print in an order defined by the data. This is the shape every judged exercise in this module takes.

Pitfalls

Printing inside the job.

Reading `as_completed` results as if they were ordered.

A lambda or nested function submitted to a process pool.

No `if __name__ == "__main__":` around a process pool on Windows and macOS (recursive process spawning).

`map` swallowing per-item errors into one raise at the failing position.

A pool of 100 threads against an API that allows 5 concurrent requests.

Key takeaways

`ThreadPoolExecutor` / `ProcessPoolExecutor` with the same interface: `submit` → `Future`, `map` → ordered results, `with` waits for everything.

`future.result()` blocks and re-raises; `exception()`, `done()`, `cancel()`, `add_done_callback` inspect and control.

`map` preserves input order — the deterministic tool; `as_completed` yields by completion and needs sorting; `wait` handles first/all/timeout.

Processes need importable, picklable work and a main guard; batch with `chunksize`.

Return the key with the result, collect inside the `with`, print in data order.

EXERCISE 1

Ordered map over words

Read one line of words. With a `ThreadPoolExecutor(max_workers=3)`, use `map` to compute for each word a tuple `(word, len(word), vowels)` where `vowels` counts the letters in `aeiou`. Print `<word> len=<n> vowels=<v>` in input order — `map` preserves it — then `most vowels <word>` (ties go to the earlier word).

Input: one line of words. **Output:** one line per word, then the winner.

```
queue thread future
```

prints

```
queue len=5 vowels=4
thread len=6 vowels=2
future len=6 vowels=3
most vowels queue
```

INPUT 1

```
queue thread future
```

OUTPUT 1

```
queue len=5 vowels=4
thread len=6 vowels=2
future len=6 vowels=3
most vowels queue
```

REFERENCE SOLUTION

```python
from concurrent.futures import ThreadPoolExecutor

words = input().split()


def describe(word):
    return word, len(word), sum(ch in "aeiou" for ch in word)


with ThreadPoolExecutor(max_workers=3) as pool:
    rows = list(pool.map(describe, words))

for word, length, vowels in rows:
    print(f"{word} len={length} vowels={vowels}")
best = max(rows, key=lambda row: row[2])
print("most vowels", best[0])
```

EXERCISE 2

Primes in a process pool

Write a module-level `prime_sum(n)` that returns the sum of the primes below `n` using a sieve. Read one line of integers; under `if __name__ == "__main__":` run them through a `ProcessPoolExecutor(max_workers=2)` with `map` and print `primes below <n> sum to <s>` in input order.

Input: one line of integers. **Output:** one line per integer.

```
10 20 100
```

prints

```
primes below 10 sum to 17
primes below 20 sum to 77
primes below 100 sum to 1060
```

INPUT 1

```
10 20 100
```

OUTPUT 1

```
primes below 10 sum to 17
primes below 20 sum to 77
primes below 100 sum to 1060
```

REFERENCE SOLUTION

```python
from concurrent.futures import ProcessPoolExecutor


def prime_sum(n):
    if n < 3:
        return 0
    sieve = [True] * n
    sieve[0] = sieve[1] = False
    for p in range(2, int(n ** 0.5) + 1):
        if sieve[p]:
            for multiple in range(p * p, n, p):
                sieve[multiple] = False
    return sum(i for i, is_prime in enumerate(sieve) if is_prime)


if __name__ == "__main__":
    nums = [int(x) for x in input().split()]
    with ProcessPoolExecutor(max_workers=2) as pool:
        for n, total in zip(nums, pool.map(prime_sum, nums)):
            print(f"primes below {n} sum to {total}")
```

QUIZ

1. What does `executor.map(fn, items)` return?

A. Results in completion order

B. Results in the order of `items`, blocking on each in turn

C. A list of futures

D. Nothing until `shutdown`

2. What does `future.result()` do if the job raised?

A. Returns `None`

B. Re-raises the exception in the caller

C. Returns the exception

D. Cancels the pool

3. What does `as_completed(futures)` yield?

A. Futures in submission order

B. Futures as they finish — arbitrary order

C. Results in submission order

D. Only successful futures

4. Why must a `ProcessPoolExecutor` worker be a module-level function?

A. For speed

B. Because it is pickled by name and imported in the worker process

C. Because lambdas are slow

D. It need not be

5. Can `future.cancel()` stop a job that is already running?

A. Yes, immediately

B. No — only a job that has not started; a running job must check a flag itself

C. Yes, after its next I/O

D. Only in a process pool

ANSWER KEY · CONCURRENT.FUTURES — EXECUTORS, FUTURES, MAP AND AS_COMPLETED

1. B That ordering is what makes `map` the deterministic tool.

2. B `exception()` returns it instead; the pool itself is unaffected by a failing job.

3. B Sort before printing anything derived from it as an answer.

4. B Lambdas and nested functions cannot be pickled by reference.

5. B Futures cannot interrupt a running callable.

LESSON 4 · 13 MIN

multiprocessing — Process, Pool, pickling, queues and shared state

`multiprocessing` is the lower-level module under `ProcessPoolExecutor`: it starts interpreter processes, runs functions in them, and moves data between them. It exists because processes are how CPU-bound Python gets parallel, and its constraints — everything crosses the boundary by pickling, the main module is re-imported by workers, start-up costs tens of milliseconds — are the constraints of that design. This lesson covers `Process` and `Pool`, the `map` / `imap` / `starmap` family, what can and cannot be pickled, the start methods and the main guard, `Queue` / `Pipe` for messages, `Value` / `Array` / `Manager` for shared state in outline, and the cost model that decides when a pool pays.

Process

```
from multiprocessing import Process, Queue

def work(n, out):
    out.put((n, sum(i * i for i in range(n))))

if __name__ == "__main__":
    out = Queue()
    procs = [Process(target=work, args=(n, out)) for n in (10**5, 2 * 10**5)]
    for p in procs: p.start()
    results = sorted(out.get() for _ in procs)     # collect, then order by tl
    for p in procs: p.join()
    print(results)
```

Same shape as `Thread` — `target`, `args`, `start`, `join` — but the function runs in a new interpreter with its own memory. `Process` has `pid`, `exitcode`, `is_alive()`, `terminate()`. Results come back through a `Queue` (or a return value via a pool), never through a shared variable: a global the child modifies is the child's copy.

Pool

```python
from multiprocessing import Pool

def square(n):
    return n * n

def power(base, exp):
    return base ** exp

if __name__ == "__main__":
    with Pool(processes=4) as pool:
        pool.map(square, range(10))                          # ordered list; b
        pool.map(square, range(10**6), chunksize=1000)       # batch small tas
        list(pool.imap(square, range(10)))                   # lazy, ordered
        list(pool.imap_unordered(square, range(10)))         # lazy, completio
        pool.starmap(power, [(2, 3), (3, 2)])                # unpack argument
        pool.apply_async(square, (7,)).get(timeout=5)        # one job, a resu
```

`Pool.map` is `ProcessPoolExecutor.map` with more knobs; `imap` streams results; `starmap` takes argument tuples. `chunksize` is the lever that matters: each task costs a pickle round trip and a queue hop, so a million tiny tasks need batching into thousands. The `with` block terminates the workers on exit; `pool.close(); pool.join()` is the explicit form that waits for them.

Pickling

Everything that crosses to a worker — the function, its arguments, its result — is serialised with `pickle`. That works for built-in types, dataclasses, plain classes defined at module level, and functions defined at module level; it fails for lambdas, nested functions, bound methods of unpicklable objects, open files, locks, database connections and generators. The error is `PicklingError` or `AttributeError: Can't pickle local object`. The fixes: define the worker function at module level; pass data, not resources (open the file *inside* the worker); use `functools.partial` on a module-level function instead of a lambda; and for large arrays, prefer shared memory (`multiprocessing.shared_memory`, NumPy) or have the worker load its own data rather than shipping it.

Start methods and the main guard

A worker process must obtain the code to run. With `fork` (Linux default) it inherits a copy of the parent's memory; with `spawn` (macOS and Windows default) it starts a fresh interpreter and *imports the main module* to find the function. That import re-executes top-level code, so a script that creates a pool at top level spawns workers that create pools that spawn workers. Hence the rule: everything that starts processes lives under `if __name__ == "__main__":`. `multiprocessing.set_start_method("spawn")` chooses explicitly; `fork` is faster but unsafe with threads already running.

Queue and Pipe

`multiprocessing.Queue` is the cross-process version of `queue.Queue`: `put` / `get` with blocking and timeouts, pickling underneath. `Pipe()` returns two connected `Connection` objects for two-party messaging (`send` / `recv`). Both are for streaming work and results between long-running processes; a pool's `map` is simpler when the work is a batch.

Shared state, in outline

`Value("i", 0)` and `Array("d", 10)` allocate typed memory shared between processes, with a `.get_lock()` for compound updates; `Manager()` starts a server process holding proxies for a dict or list that several processes may modify (slow, but general). Both are last resorts: the design that scales is no shared state — each worker computes from its inputs and returns a result, and the parent aggregates.

The cost model

Starting a process costs 10–100 ms; pickling costs time proportional to the data's size; a result must be pickled back. A pool pays when each task is at least milliseconds of CPU work and its data is small relative to the work, or when the data is loaded inside the worker. It does not pay for a hundred tiny tasks, for tasks that ship large arguments, or for I/O-bound work (threads or asyncio are cheaper). On the judge two cores exist, so a pool of two can genuinely halve a CPU-bound computation — and every result must still be collected and ordered before printing.

Pitfalls

No main guard: recursive spawning on macOS/Windows.

A lambda or nested function as the worker.

Expecting a global modified in a worker to change in the parent.

Shipping a large object to every task instead of loading it in the worker.

A million tiny tasks without `chunksize`.

`imap_unordered` output printed in arrival order.

Key takeaways

`Process(target=…)` mirrors `Thread` but in a separate interpreter; results return through a `Queue` or a pool, never shared variables.

`Pool.map` / `imap` / `starmap` run batches; `chunksize` amortises the per-task pickling cost; `imap_unordered` must be sorted.

Workers and their arguments are pickled: module-level functions, plain data, no lambdas or resources.

Workers re-import the main module under `spawn`, so process creation lives under the main guard.

Shared memory and managers exist; returning results and aggregating in the parent is the design to prefer.

EXERCISE 1

starmap and the unordered check

Read lines `base exp mod` until EOF. Write a module-level `pow_mod(base, exp, mod)` returning `pow(base, exp, mod)` and a module-level `indexed(job)` that takes `(i, base, exp, mod)` and returns `(i, pow_mod(base, exp, mod))`. Under the main guard, open `Pool(2)`, compute the results with `starmap` and print `<base>^<exp> mod <mod> = <r>` per job in input order; then run the indexed jobs through `imap_unordered`, sort the pairs by index, and print `unordered matches <True/False>` comparing them with the ordered results.

Input: one job per line. **Output:** one line per job, then the comparison.

```
2 10 1000
3 4 5
7 0 13
```

prints

```
2^10 mod 1000 = 24
3^4 mod 5 = 1
7^0 mod 13 = 1
unordered matches True
```

INPUT 1

```
2 10 1000
3 4 5
7 0 13
```

OUTPUT 1

```
2^10 mod 1000 = 24
3^4 mod 5 = 1
7^0 mod 13 = 1
unordered matches True
```

REFERENCE SOLUTION

```
import sys
from multiprocessing import Pool


def pow_mod(base, exp, mod):
    return pow(base, exp, mod)


def indexed(job):
    i, base, exp, mod = job
    return i, pow_mod(base, exp, mod)


if __name__ == "__main__":
    jobs = [tuple(int(x) for x in line.split()) for line in sys.stdin if line.strip(
    with Pool(2) as pool:
        ordered = pool.starmap(pow_mod, jobs)
        unordered = sorted(pool.imap_unordered(indexed, [(i, *job) for i, job in enu
    for (base, exp, mod), result in zip(jobs, ordered):
        print(f"{base}^{exp} mod {mod} = {result}")
    print("unordered matches", [r for _, r in unordered] == ordered)
```

EXERCISE 2

Two processes and a queue

Read one line of integers and split it into two parts: the first `(n + 1) // 2` numbers and the rest. Start one `multiprocessing.Process` per part running a module-level `part_sum(index, chunk, q)` that puts `(index, sum(chunk))` on a `multiprocessing.Queue`. In the parent, collect both results

from the queue, join the processes, sort by index, and print `part <i> sum <s>` for each and `total <sum>`.

Input: one line of integers. **Output:** three lines.

```
1 2 3 4 5
```

prints

```
part 0 sum 6
part 1 sum 9
total 15
```

INPUT 1

```
1 2 3 4 5
```

OUTPUT 1

```
part 0 sum 6
part 1 sum 9
total 15
```

REFERENCE SOLUTION

```python
from multiprocessing import Process, Queue


def part_sum(index, chunk, q):
    q.put((index, sum(chunk)))


if __name__ == "__main__":
    nums = [int(x) for x in input().split()]
    half = (len(nums) + 1) // 2
    parts = [nums[:half], nums[half:]]
    q = Queue()
    procs = [Process(target=part_sum, args=(i, chunk, q)) for i, chunk in enumerate(
    for p in procs:
        p.start()
    results = sorted(q.get() for _ in procs)
    for p in procs:
        p.join()
    for index, total in results:
        print(f"part {index} sum {total}")
    print("total", sum(total for _, total in results))
```

QUIZ

1. A global list is appended to inside a `Process` target. What does the parent see afterwards?

A. The appended item

B. Its own unchanged list — the child modified a copy in its own memory

C. An error

D. A proxy

2. Why does `pool.map(f, range(10**6))` benefit from `chunksize`?

A. It does not

B. Each task costs a pickle round trip, so batching thousands per task amortises the overhead

C. It uses less memory

D. It preserves order

3. Which of these can be sent to a pool worker?

A. A lambda

B. An open file

C. A module-level function and a list of ints

D. A `threading.Lock`

4. Under the `spawn` start method, what does a worker do first?

A. Forks the parent

B. Imports the main module to find the target function

C. Reads stdin

D. Shares the parent's memory

5. Which is the recommended design for shared state between processes?

A. `Manager().dict()` everywhere

B. `Value` with a lock

C. No shared state: workers return results and the parent aggregates

D. Global variables

ANSWER KEY · MULTIPROCESSING — PROCESS, POOL, PICKLING, QUEUES AND SHARED STATE

1. B Processes share nothing; results travel back through a queue, a pipe or a pool's return value.

2. B A million tiny tasks without chunking can be slower than a plain loop.

3. C Everything crossing the boundary is pickled; lambdas, files and locks are not picklable.

4. B That import re-executes top-level code — hence the `if __name__ == "__main__":` guard around process creation.

5. C Shared memory and managers exist as last resorts; returning results scales and stays testable.

LESSON 5 · 15 MIN

asyncio basics — coroutines, await, tasks and gather

`asyncio` runs many waiting operations on one thread. An `async def` function is a *coroutine*: calling it builds an object that does nothing until the event loop drives it, and inside it `await` hands control back to the loop until the awaited thing is ready. The loop runs whichever coroutine can make progress, so ten downloads overlap on one thread — with no locks needed, because a switch happens only at `await`. This lesson covers the vocabulary (coroutine, task, event loop, awaitable), `asyncio.run`, `await` versus creating tasks, `gather` for ordered fan-out, `sleep` as the model of a wait, and the mistakes — calling without awaiting, blocking the loop — that every beginner makes once.

Coroutines and the loop

```
import asyncio

async def fetch(name, delay):
    await asyncio.sleep(delay)          # yield to the loop for `delay` second
    return f"{name} ready"

async def main():
    result = await fetch("a", 0.1)      # runs fetch to completion, then conti
    print(result)

asyncio.run(main())                     # creates the loop, runs main, closes
```

`fetch("a", 0.1)` does not run anything — it returns a coroutine object. `await` runs it and produces its return value; while it is suspended in `asyncio.sleep`, the loop may run other coroutines. `asyncio.run(coro)` is the entry point: exactly one per program, at the top level, never inside a running loop. Everything that is awaited must be an *awaitable*: a coroutine, a `Task`, a `Future`, or an object with `__await__`.

Sequential versus concurrent

```
async def main():
    a = await fetch("a", 0.2)           # 0.2 s
    b = await fetch("b", 0.2)           # then another 0.2 s — sequential, 0.4

async def main():
    a, b = await asyncio.gather(fetch("a", 0.2), fetch("b", 0.2))   # both at
```

`await` one thing at a time is sequential — the loop has nothing else to run. Concurrency comes from starting several before waiting: `gather(*awaitables)` schedules them all and returns their results *as a list in argument order*, whatever order they finished in. That ordering is what makes `gather` the deterministic tool, like `executor.map`.

Tasks

```
async def main():
    task = asyncio.create_task(fetch("a", 0.2))     # scheduled now; runs at t
    other = await do_something_else()
    result = await task                              # collect it later
```

`create_task(coro)` wraps a coroutine in a `Task` and schedules it; it starts running the first time the current coroutine awaits anything. A task is a `Future`: `done()`, `result()`, `cancel()`, `add_done_callback`. Keep a reference to every task you create (in a list or a set) — the loop holds tasks weakly, and a task with no reference may be garbage-collected mid-flight. `gather` creates tasks for the coroutines it is given, so the common pattern needs no explicit `create_task` at all.

---

sleep(0) and the order of interleaving

```
async def worker(name, log):
    log.append(f"{name} start")
    await asyncio.sleep(0)              # yield once: let the others reach thi
    log.append(f"{name} end")

async def main():
    log = []
    await asyncio.gather(worker("a", log), worker("b", log))
    print(log)                          # ['a start', 'b start', 'a end', 'b e
```

`asyncio.sleep(0)` yields exactly once. The loop's scheduling is FIFO and deterministic: tasks run in the order they were created, each until its next `await`. That makes small asyncio traces reproducible — the sequence above is the same every run — unlike thread interleavings, which depend on the operating system. For judged exercises this is the model: control the order with `await` points, collect with `gather`, print after.

The loop in one paragraph

The event loop is a queue of callbacks and a selector. `call_soon` appends a callback to the ready queue; each iteration runs every callback that was ready at the start, in order, then asks the operating system which sockets and timers are ready and queues their callbacks. A task is one callback that runs its coroutine until the next `await`; the awaited future, when it completes, calls `call_soon` for the task again. Nothing pre-empts anything: a coroutine keeps the loop until it awaits, which is why shared state is safe between awaits — and why a long computation starves every other task.

Timeouts and cancellation

```
try:
    result = await asyncio.wait_for(fetch("a", 5), timeout=1.0)
except asyncio.TimeoutError:
    result = "timed out"

task.cancel()                              # raises CancelledError inside the tas
```

`wait_for` cancels the awaitable when the timeout expires. Cancellation is cooperative: `CancelledError` is thrown into the coroutine at its current `await`, and the coroutine may clean up in `finally` — but should not swallow the exception. 3.11 also has `asyncio.timeout()` as a context manager (`async with asyncio.timeout(1.0):`).

Blocking the loop

`time.sleep(1)`, `requests.get(...)`, a CPU loop of a second, `input()` — anything that does not `await` — freezes every coroutine for its duration, because the loop is one thread. The replacements: `await asyncio.sleep`, an async HTTP client (`aiohttp`, `httpx`), and for

something with no async version, `await asyncio.to_thread(blocking_fn, *args)`, which runs the call on a thread pool and awaits its result. A CPU-bound stage belongs in a `ProcessPoolExecutor` via `loop.run_in_executor`.

Pitfalls

Calling a coroutine without `await` — nothing runs, and Python warns "coroutine … was never awaited".

`await` in a plain `def` — a syntax error; the caller must be `async def` too, all the way up to `asyncio.run`.

A blocking call in a coroutine.

Awaiting sequentially when the work is independent.

Not keeping references to created tasks.

`asyncio.run` inside a running loop (notebooks) — use `await` directly there.

Key takeaways

`async def` builds a coroutine; `await` runs it and yields to the loop while it waits; `asyncio.run` drives the top-level one.

Sequential `await`s do not overlap; `gather(...)` runs awaitables concurrently and returns results in argument order.

`create_task` schedules a coroutine to run at the next `await`; keep a reference; a task is a future.

Scheduling is FIFO and reproducible: `sleep(0)` yields once, so small traces are deterministic.

Never block the loop; `to_thread` and `run_in_executor` bridge to blocking and CPU-bound code; `wait_for` / `timeout` bound a wait, cancellation is cooperative.

EXERCISE 1

gather with a finish log

Read lines `<name> <steps>` until EOF. Write `async def job(name, steps, log)` that awaits `asyncio.sleep(0)` `steps` times, appends `name` to `log`, and returns `f"{name}={steps * 10}"`. In `main`, run every job with `asyncio.gather`, print the results joined by `", "` (they come back in

argument order), then `finished <log joined by spaces>` — the completion order, which the FIFO loop makes reproducible.

Input: one job per line. **Output:** two lines.

```
a 3
b 1
c 2
```

prints

```
a=30, b=10, c=20
finished b c a
```

INPUT 1

```
a 3
b 1
c 2
```

OUTPUT 1

```
a=30, b=10, c=20
finished b c a
```

REFERENCE SOLUTION

```python
import asyncio
import sys


async def job(name, steps, log):
    for _ in range(steps):
        await asyncio.sleep(0)
    log.append(name)
    return f"{name}={steps * 10}"


async def main(jobs):
    log = []
    results = await asyncio.gather(*(job(name, steps, log) for name, steps in jobs))
    print(", ".join(results))
    print("finished", " ".join(log))


jobs = [(parts[0], int(parts[1])) for parts in (line.split() for line in sys.stdin)
asyncio.run(main(jobs))
```

EXERCISE 2

Sequential versus concurrent

Read lines `<name> <delay_ms>` until EOF. Write `async def fetch(name, ms)` that awaits `asyncio.sleep(ms / 1000)` and returns `f"{name} ready"`. Run all fetches concurrently with `gather` and print each result on its own line in input order; then print `sequential <sum of delays> ms` and `concurrent <largest delay> ms` — the two wall-clock totals the two strategies would take.

Input: one fetch per line. **Output:** the results, then two summary lines.

```
db 30
api 10
cache 5
```

prints

```
db ready
api ready
cache ready
sequential 45 ms
concurrent 30 ms
```

INPUT 1

```
db 30
api 10
cache 5
```

OUTPUT 1

```
db ready
api ready
cache ready
sequential 45 ms
concurrent 30 ms
```

REFERENCE SOLUTION

```
import asyncio
import sys


async def fetch(name, ms):
    await asyncio.sleep(ms / 1000)
    return f"{name} ready"


async def main(jobs):
    results = await asyncio.gather(*(fetch(name, ms) for name, ms in jobs))
    for line in results:
        print(line)
    delays = [ms for _, ms in jobs]
    print(f"sequential {sum(delays)} ms")
    print(f"concurrent {max(delays)} ms")


jobs = [(parts[0], int(parts[1])) for parts in (line.split() for line in sys.stdin)
asyncio.run(main(jobs))
```

QUIZ

1. What does calling `fetch()` return when `fetch` is `async def`?

A. Its result

B. A coroutine object that has not run

C. A thread

D. `None`

2. What does `await asyncio.gather(a(), b())` return?

A. A set of results

B. A list of results in argument order

C. The first result

D. Results in completion order

3. What does `time.sleep(1)` inside a coroutine do?

A. Yields to the loop for a second

B. Freezes the whole event loop for a second

C. Raises

D. Runs in a thread

4. When does a task created with `create_task` start running?

A. Immediately, in parallel

B. At the next `await` in the current coroutine

C. When `result()` is called

D. When the loop closes

5. What does `asyncio.sleep(0)` do?

A. Nothing

B. Yields to the loop exactly once

C. Sleeps for one tick of the clock

D. Cancels the task

ANSWER KEY · ASYNCIO BASICS — COROUTINES, AWAIT, TASKS AND GATHER

1. B Nothing runs until it is awaited or wrapped in a task; Python warns if it never is.

2. B Argument order, whatever finished first — the deterministic fan-out tool.

3. B Only `await`able waits yield; use `await asyncio.sleep(1)` or `to_thread` for blocking calls.

4. B One thread: the new task runs when the current one yields control.

5. B It is the way to let other ready tasks run — and, with FIFO scheduling, to write reproducible traces.

LESSON 6 · 14 MIN

asyncio patterns — TaskGroup, queues, semaphores, async iteration and bridging

The basics give you coroutines and `gather`; real programs need the shapes built on them: a group of tasks that fails together, a bounded number of concurrent requests, a producer feeding consumers through a queue, an `async for` over a stream, an `async with` around a connection, and a way to call the synchronous world without freezing the loop. This lesson covers `TaskGroup` (new in 3.11), `Semaphore` for rate limiting, `asyncio.Queue` workers, `as_completed` and its ordering caveat, async iterators, generators and context managers, `to_thread` and `run_in_executor`, and the guidance on when asyncio is the wrong tool.

TaskGroup

```
async def main():
    async with asyncio.TaskGroup() as tg:              # 3.11+
        t1 = tg.create_task(fetch("a", 0.2))
        t2 = tg.create_task(fetch("b", 0.1))
    print(t1.result(), t2.result())                    # both done when the bl
```

A `TaskGroup` owns the tasks created inside it: the `async with` block waits for all of them, and if one raises, the others are *cancelled* and the exceptions are raised together as an `ExceptionGroup` (`except*` catches by type). This is structured concurrency — no task outlives its block, no exception is lost — and it is the replacement for `gather` when tasks should fail together. `gather(..., return_exceptions=True)` is the other choice: run everything, collect exceptions as values.

Bounding concurrency

```python
sem = asyncio.Semaphore(5)                          # at most five in flight

async def fetch_limited(url):
    async with sem:
        return await fetch(url)

results = await asyncio.gather(*(fetch_limited(u) for u in urls))
```

A thousand coroutines are free to *create*; a thousand simultaneous connections are not free for the server. `Semaphore(n)` is the standard limiter, and `gather` still returns the results in order. `asyncio.Lock`, `Event` and `Condition` mirror their threading namesakes; they exist for coordination between coroutines and are needed less often, because no switch can occur between two plain statements.

Producer and consumers with a queue

```python
async def producer(q, items):
    for item in items:
        await q.put(item)
    await q.put(None)

async def consumer(q, out):
    while (item := await q.get()) is not None:
        out.append(item * 2)
        q.task_done()

async def main():
    q = asyncio.Queue(maxsize=10)                   # maxsize: back-pressure
    out = []
    await asyncio.gather(producer(q, range(5)), consumer(q, out))
    print(out)                                      # [0, 2, 4, 6, 8]
```

`asyncio.Queue` is the coroutine version of `queue.Queue`: `await q.put`, `await q.get`, `task_done`, `await q.join()`. With several consumers, each needs a sentinel (or the parent cancels the workers after `q.join()`), and each consumer's results should go into its own slot and be merged in order afterwards.

---

as_completed

```
for coro in asyncio.as_completed([fetch("a", 0.3), fetch("b", 0.1)]):
    result = await coro                                  # "b ready" first
```

For progress reporting; not for ordered answers. Sort before printing, or use `gather`.

Async iteration and context managers

```
async def ticker(n):                                     # an async generator
    for i in range(n):
        await asyncio.sleep(0)
        yield i

async def main():
    async for i in ticker(3):                            # await between items
        print(i)
    squares = [i * i async for i in ticker(3)]           # async comprehension

    async with aiohttp.ClientSession() as session:       # __aenter__/__aexit__
        ...
```

`async for` drives an object with `__aiter__` / `__anext__` (an async generator is the easy way to write one); `async with` drives `__aenter__` / `__aexit__`. Both exist because the protocol methods need to `await` — a database cursor fetching rows, a connection being opened and closed. `contextlib.asynccontextmanager` writes an async context manager from an async generator with one `yield`, exactly like its sync twin.

Bridging to blocking and CPU-bound code

```
data = await asyncio.to_thread(read_big_file, path)                # blocking

loop = asyncio.get_running_loop()
total = await loop.run_in_executor(process_pool, cpu_heavy, n)     # CPU → a
```

`to_thread` (3.9+) is the everyday bridge for a library with no async version; `run_in_executor` accepts a specific executor, including a process pool for CPU work. Both return awaitables, so the loop keeps serving other coroutines while the work runs elsewhere. The reverse direction —

calling into a running loop from a thread — is `asyncio.run_coroutine_threadsafe(coro, loop)` or `loop.call_soon_threadsafe`.

Choosing asyncio

Reach for asyncio when the program is mostly waiting on many things at once and the libraries involved are async-native — web servers and clients, websockets, chat, scrapers, anything with thousands of connections. Prefer threads when the count is small and the libraries are blocking (a handful of `requests` calls, a database driver with no async version): `ThreadPoolExecutor.map` is three lines and needs no rewrite. Prefer processes for CPU. Mixing is normal: an async web handler that hands a CPU stage to a process pool with `run_in_executor` is the standard shape.

Pitfalls

A `TaskGroup` task's exception silently cancelling siblings that were expected to finish — use `gather(return_exceptions=True)` when independence matters.

Unbounded fan-out against a rate-limited service.

Consumers that never get a sentinel and wait forever.

Printing from `as_completed` and expecting input order.

`for` instead of `async for` on an async generator (`TypeError: 'async_generator' object is not iterable`).

A blocking library call inside a coroutine because "it's fast".

Key takeaways

`TaskGroup` gives structured concurrency: the block waits for its tasks, one failure cancels the rest, errors arrive as an `ExceptionGroup`.

`Semaphore(n)` bounds concurrency while `gather` keeps result order; `Lock` / `Event` / `Condition` mirror threading.

`asyncio.Queue` with sentinels connects producers and consumers; `maxsize` gives back-pressure.

`async for` / `async with` drive awaitable protocols; async generators and `asynccontextmanager` write them.

`to_thread` and `run_in_executor` keep the loop alive while blocking or CPU work runs elsewhere; asyncio is for many async-native waits, threads for a few blocking ones, processes for CPU.

EXERCISE 1

Bounded fan-out

Read `limit` on the first line and job ids on the second. Create `asyncio.Semaphore(limit)` and a coroutine `run(job_id)` that, inside `async with sem:`, increments a shared `in_flight` counter, records the peak, awaits `asyncio.sleep(0)` twice, decrements the counter and returns `job_id * job_id`. Gather every job and print `results <space-separated squares in input order>` and `peak <highest in_flight seen>`.

Input: `limit`, then the ids. **Output:** two lines.

```
2
1 2 3 4 5
```

prints

```
results 1 4 9 16 25
peak 2
```

INPUT 1

```
2
1 2 3 4 5
```

OUTPUT 1

```
results 1 4 9 16 25
peak 2
```

REFERENCE SOLUTION

```
import asyncio

limit = int(input())
ids = [int(x) for x in input().split()]
state = {"in_flight": 0, "peak": 0}


async def main():
    sem = asyncio.Semaphore(limit)

    async def run(job_id):
        async with sem:
            state["in_flight"] += 1
            state["peak"] = max(state["peak"], state["in_flight"])
            await asyncio.sleep(0)
            await asyncio.sleep(0)
            state["in_flight"] -= 1
            return job_id * job_id

    results = await asyncio.gather(*(run(job_id) for job_id in ids))
    print("results", " ".join(map(str, results)))
    print("peak", state["peak"])


asyncio.run(main())
```

EXERCISE 2

Async producer and consumers

Read `C` on the first line and words on the second. With an `asyncio.Queue(maxsize=2)`, a producer puts every word then `C` `None` sentinels; consumer `i` takes items until a sentinel and appends `word.upper()` to its own list `slots[i]`. Run the producer and the consumers under an `asyncio.TaskGroup`, then merge the slots, sort, and print `consumers <C>`, `processed <n>` and `sorted <space-separated words>`.

Input: `C`, then the words. **Output:** three lines.

```
2
gil loop task
```

prints

```
consumers 2
processed 3
sorted GIL LOOP TASK
```

INPUT 1

```
2
gil loop task
```

OUTPUT 1

```
consumers 2
processed 3
sorted GIL LOOP TASK
```

REFERENCE SOLUTION

```python
import asyncio

c = int(input())
words = input().split()
slots = [[] for _ in range(c)]


async def producer(q):
    for word in words:
        await q.put(word)
    for _ in range(c):
        await q.put(None)


async def consumer(q, i):
    while True:
        word = await q.get()
        if word is None:
            break
        slots[i].append(word.upper())


async def main():
    q = asyncio.Queue(maxsize=2)
    async with asyncio.TaskGroup() as tg:
        tg.create_task(producer(q))
        for i in range(c):
            tg.create_task(consumer(q, i))


asyncio.run(main())
merged = sorted(w for slot in slots for w in slot)
print("consumers", c)
print("processed", len(merged))
print("sorted", " ".join(merged))
```

QUIZ

1. What happens in a `TaskGroup` when one task raises?

A. The others continue and the error is lost

B. The others are cancelled and the errors are raised together as an `ExceptionGroup`

C. The loop stops

D. Only that task is retried

2. What is `asyncio.Semaphore(5)` for?

A. Five event loops

B. At most five coroutines inside its `async with` block at once

C. Five threads

D. A five-second timeout

3. Why does `for x in async_gen():` fail?

A. Async generators are empty

B. An async generator needs `async for`, because fetching each item may await

C. It works

D. Generators cannot be async

4. How do you call a blocking function from a coroutine without freezing the loop?

A. Just call it

B. `await asyncio.to_thread(fn, *args)`

C. `time.sleep` first

D. Wrap it in `async def`

5. Which workload is the strongest case for asyncio over threads?

A. Three blocking database calls

B. Thousands of concurrent connections with async-native libraries

C. A CPU-bound image filter

D. Reading one file

ANSWER KEY · ASYNCIO PATTERNS — TASKGROUP, QUEUES, SEMAPHORES, ASYNC ITERATION AND BRIDGING

1. B Structured concurrency: nothing outlives the block and nothing is lost; use `gather(return_exceptions=True)` when tasks are independent.

2. B The standard limiter for fan-out against a rate-limited service.

3. B `__anext__` is a coroutine; `async for` awaits it.

4. B It runs on a worker thread and the loop keeps serving other coroutines; `run_in_executor` accepts a process pool for CPU work.

5. B Few blocking calls are simpler with a thread pool; CPU work wants processes.

CHECKPOINT · 25 MIN

Checkpoint — Concurrency

This checkpoint covers the whole module: the GIL and the three models, threads with `Lock`, `Queue` and `Event`, the race and its fix, `concurrent.futures` executors with ordered `map` and unordered `as_completed`, `multiprocessing` with pickling and the main guard, and asyncio from coroutines and `gather` through `TaskGroup`, semaphores, queues and bridging to blocking code — plus the determinism rule that every judged program obeys.

How it works. Twelve questions and three programs. You need 70% on the questions and every program accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- Why do threads not speed up a CPU-bound loop in CPython, and what does speed it up?
- What are the three bytecodes behind `counter += 1`, and where can the switch happen?
- What is the difference between `executor.map` and `as_completed` in the order of results?
- Why must code that starts processes live under `if __name__ == "__main__":`?
- What happens when a coroutine is called without `await`?
- What does `asyncio.gather` return, and in what order?
- What is the one way to make a threaded program's output deterministic?

The three programs are a threaded word counter whose workers fill per-thread slots and are merged after `join`, a pipeline of workers reading a `Queue` with sentinels whose per-worker results are merged in order, and an asyncio job runner with a semaphore that records a deterministic trace through `sleep(0)` yields and reports results with `gather`.

EXERCISE 1

Threaded word counter

Read `T` on the first line, then text lines until EOF. Thread `i` handles `lines[i::T]`: it lowercases each line, splits on whitespace, strips `.,;:!?` from each word, and counts the words into its own `collections.Counter` in `slots[i]`. Join the threads, merge the counters, and print `total <number of words>` followed by the top three words as `<word> <count>` ordered by count descending then word ascending (fewer if there are fewer distinct words).

Input: `T`, then lines of text. **Output:** the total, then up to three lines.

```
2
the cat sat on the mat
the dog sat
a cat
```

prints

```
total 11
the 3
cat 2
sat 2
```

INPUT 1

```
2
the cat sat on the mat
the dog sat
a cat
```

OUTPUT 1

```
total 11
the 3
cat 2
sat 2
```

REFERENCE SOLUTION

```
import sys
import threading
from collections import Counter

PUNCT = ".,;:!?"

t_count = int(sys.stdin.readline())
lines = [line.rstrip() for line in sys.stdin]
slots = [Counter() for _ in range(t_count)]


def count_lines(i):
    for line in lines[i::t_count]:
        for word in line.lower().split():
            word = word.strip(PUNCT)
            if word:
                slots[i][word] += 1


threads = [threading.Thread(target=count_lines, args=(i,)) for i in range(t_count)]
for t in threads:
    t.start()
for t in threads:
    t.join()

merged = Counter()
for slot in slots:
    merged.update(slot)
print("total", sum(merged.values()))
for word, count in sorted(merged.items(), key=lambda item: (-item[1], item[0]))[:3]:
    print(word, count)
```

EXERCISE 2

Queue pipeline by index

Read `W` on the first line, then text lines until EOF. Put `(index, line)` jobs on a `queue.Queue` followed by `W` sentinels. Each of `W` worker threads takes jobs and, for each, computes `"<index>: <words> words, <chars> chars, longest <word>"` where `words` is the whitespace word count, `chars` is `len(line)` and `longest` is the first longest word; it appends `(index, text)` to a shared results list under a `Lock`. Join the workers, sort the results by index, and print each text.

Input: `W`, then lines. **Output:** one line per input line, in input order.

```
2
the quick brown fox
jumps over
the lazy dog
```

prints

```
0: 4 words, 19 chars, longest quick
1: 2 words, 10 chars, longest jumps
2: 3 words, 12 chars, longest lazy
```

INPUT 1

```
2
the quick brown fox
jumps over
the lazy dog
```

OUTPUT 1

```
0: 4 words, 19 chars, longest quick
1: 2 words, 10 chars, longest jumps
2: 3 words, 12 chars, longest lazy
```

REFERENCE SOLUTION

```
import queue
import sys
import threading

w_count = int(sys.stdin.readline())
lines = [line.rstrip() for line in sys.stdin if line.strip()]
q = queue.Queue()
results = []
lock = threading.Lock()


def describe(index, line):
    words = line.split()
    longest = max(words, key=len)
    return f"{index}: {len(words)} words, {len(line)} chars, longest {longest}"


def worker():
    while True:
        job = q.get()
        if job is None:
            break
        index, line = job
        text = describe(index, line)
        with lock:
            results.append((index, text))


for index, line in enumerate(lines):
    q.put((index, line))
for _ in range(w_count):
    q.put(None)

threads = [threading.Thread(target=worker) for _ in range(w_count)]
for t in threads:
    t.start()
for t in threads:
    t.join()

for _, text in sorted(results):
    print(text)
```

EXERCISE 3

Async job runner with a trace

Read `limit` on the first line, then lines `<name> <steps>` until EOF. Write `async def run(name, steps)` that, inside `async with sem:` for a `Semaphore(limit)`, appends `start <name>` to a shared trace list, awaits `asyncio.sleep(0)` `steps` times, appends `done <name>`, and returns `(name, sum(range(1, steps + 1)))`. Gather every job in input order; print each trace line, then `<name> → <value>` for each result in gather order.

Input: `limit`, then one job per line. **Output:** the trace, then the results.

```
2
a 2
b 1
c 1
```

prints

```
start a
start b
done b
done a
start c
done c
a → 3
b → 1
c → 1
```

INPUT 1

```
2
a 2
b 1
c 1
```

OUTPUT 1

```
start a
start b
done b
done a
start c
done c
a → 3
b → 1
c → 1
```

REFERENCE SOLUTION

```
import asyncio
import sys

limit = int(sys.stdin.readline())
jobs = [(parts[0], int(parts[1])) for parts in (line.split() for line in sys.stdin)
trace = []


async def main():
    sem = asyncio.Semaphore(limit)

    async def run(name, steps):
        async with sem:
            trace.append(f"start {name}")
            for _ in range(steps):
                await asyncio.sleep(0)
            trace.append(f"done {name}")
            return name, sum(range(1, steps + 1))

    results = await asyncio.gather(*(run(name, steps) for name, steps in jobs))
    for line in trace:
        print(line)
    for name, value in results:
        print(f"{name} → {value}")


asyncio.run(main())
```

CHECKPOINT QUESTIONS

1. Which work gets faster with threads in CPython?

A. A pure-Python loop

B. Work that waits on I/O

C. Integer arithmetic

D. List sorting

2. What is the fix for `counter += 1` shared between threads?

A. Make it a global

B. Hold a `Lock` around the compound operation

C. Use `time.sleep`

D. Use a daemon thread

3. How does a consumer thread know a `Queue` will deliver no more work?

A. It checks `empty()`

B. It receives a sentinel such as `None`

C. The queue closes itself

D. It times out

4. `ThreadPoolExecutor.map` versus `as_completed`: which preserves input order?

A. Both

B. `map`

C. `as_completed`

D. Neither

5. Why must process-pool code sit under `if __name__ == "__main__":`?

A. Style only

B. Workers import the main module; unguarded top-level code would start pools recursively

C. For pickling

D. For the GIL

6. What crosses the boundary between a parent and a pool worker?

A. Shared memory by default

B. Pickled functions, arguments and results

C. References

D. Nothing

7. Which of these is an awaitable?

A. A plain function

B. A coroutine object, a `Task` or a `Future`

C. A thread

D. An integer

8. How many `asyncio.run` calls should a program make?

A. One per coroutine

B. One, at the top level

C. One per task

D. None

9. Why should you keep a reference to a task from `create_task`?

A. For speed

B. The loop holds tasks weakly; an unreferenced task may be garbage-collected mid-flight

C. To cancel it

D. To print it

10. What is `asyncio.Queue(maxsize=10)` giving you?

A. Ten consumers

B. Back-pressure: `put` waits when ten items are pending

C. Ten producers

D. Ten seconds of timeout

11. `await asyncio.wait_for(coro, timeout=1)` times out. What happens to `coro`?

A. It keeps running

B. It is cancelled and `TimeoutError` is raised

C. It returns `None`

D. It is retried

12. A concurrent program prints from each worker as it finishes. What is wrong?

A. Nothing

B. The interleaving depends on the scheduler, so the output is not reproducible

C. It is too slow

D. Workers cannot print

ANSWER KEY · CHECKPOINT — CONCURRENCY

1. B The GIL is released while a thread blocks; it is held while bytecode runs.

2. B The lock makes load-add-store atomic with respect to other holders.

3. B `empty()` is a race; one sentinel per consumer ends each loop cleanly.

4. B `as_completed` yields by completion; sort its results before printing.

5. B Under the `spawn` start method the import re-runs every top-level statement.

6. B Hence module-level functions and picklable data only.

7. B Anything with `__await__`; `gather` and `create_task` produce them too.

8. B It creates and closes the loop; inside a running loop you `await` instead.

9. B A list or set of tasks, or a `TaskGroup`, keeps them alive.

10. B A bounded queue stops a fast producer from outrunning slow consumers.

11. B Cancellation is cooperative: `CancelledError` arrives at its next `await`.

12. B Collect into ordered slots, join or gather, then print from the main thread.

CodeKairo · Python study plan · Module 17: Concurrency — threads, processes and asyncio. Solve the exercises online to track your progress and earn XP.