**Asyncio —Cheat Sheet**

**Asyncio = smart waiting.**  
When your program has to **wait** (for the internet, a file, or a database), it doesn’t sit idle. It says “I’ll pause here” (**that’s await**), and meanwhile it **does another job**. When the first thing is ready, it comes back and finishes it.

Think of a single cook in a kitchen:

* Puts Pizza A in the oven → **awaits** it to bake.
* While waiting, starts chopping veggies for Pizza B.
* Oven beeps → goes back, finishes Pizza A, then continues Pizza B.

So asyncio **doesn’t do two things at the exact same millisecond**; it just **switches tasks whenever one is waiting**, which makes everything feel fast and efficient when there’s lots of waiting involved.

# 1) Background of asyncio — pros & cons

What it is (one line): Asyncio lets one Python program juggle many waiting tasks (network, disk, DB) by switching at await points.

Pros (use when):

* I/O-bound work: HTTP requests, sockets, files, DB queries.
* Lightweight concurrency: thousands of tasks possible without many threads.
* Built-in tools: timeouts, cancellation, queues, locks.

Cons (avoid when):

* CPU-heavy tasks: use asyncio.to\_thread(...) or multiprocessing.
* Blocking calls freeze the loop (e.g., time.sleep, slow drivers).
* Learning curve: async/await, tasks, and choosing async-friendly libs.

One-liner for students: "Asyncio shines when your code spends most of its time WAITING."

# 2) Event Loop (the conductor)

The event loop is the conductor that decides which task runs next. It runs a task until it hits await, then switches to another ready task.

Cycle: pick ready task → run until await → switch to another → repeat.

Tiny demo:

import asyncio  
  
async def main():  
 print("start")  
 await asyncio.sleep(1) # pretend: waiting for network/file  
 print("end")  
  
asyncio.run(main())

# 3) Coroutines & Tasks

Coroutine: function declared with async def. Calling it returns a coroutine object (a plan). You await it to get the result.

Task: the scheduled coroutine managed by the loop. Use tasks for concurrency (several coroutines active in the same time window).

Analogy: Coroutine = recipe; Task = chef actually starting that recipe alongside others.

Mini demo (two tasks run concurrently):

import asyncio  
  
async def bake(name, secs):  
 await asyncio.sleep(secs)  
 return f"{name} ready"  
  
async def main():  
 t1 = asyncio.create\_task(bake("Margherita", 2))  
 t2 = asyncio.create\_task(bake("Veggie", 2))  
 # both 'bake' during the same 2 seconds  
 r1, r2 = await asyncio.gather(t1, t2)  
 print(r1, "|", r2)  
  
asyncio.run(main())

# 4) Futures & Synchronization

Future: a placeholder for a result that will arrive later. You usually await tasks (higher level); Futures power things under the hood.

Synchronization tools:

* Lock: only one task enters a critical section at a time.
* Semaphore(n): up to n tasks allowed (limit concurrency).
* Event: one task signals others to proceed.
* Queue: producer/consumer pattern, safe hand-off of items.

Quick peeks:

import asyncio  
  
# Lock  
lock = asyncio.Lock()  
async def update\_shared():  
 async with lock:  
 # only one task here at a time  
 ...  
  
# Semaphore  
sem = asyncio.Semaphore(5)  
async def download(url):  
 async with sem: # at most 5 downloads in parallel  
 ...  
  
# Queue (producer/consumer)  
q = asyncio.Queue()  
  
async def producer():  
 for i in range(3):  
 await q.put(i)  
 await q.put(None) # poison pill to stop consumer  
  
async def consumer():  
 while True:  
 item = await q.get()  
 if item is None:  
 break  
 # process item...  
 q.task\_done()

# 5) Practical Do’s & Don’ts

Do:

* Use await asyncio.sleep(), aiohttp, and async DB drivers.
* Wrap blocking/CPU work: await asyncio.to\_thread(blocking\_func, \*args).
* Use asyncio.gather(...) for parallel waits; consider return\_exceptions=True.
* Add timeouts with asyncio.wait\_for(coro, timeout=5).

Don’t:

* Don’t call blocking functions (time.sleep, heavy file ops) directly in coroutines.
* Don’t mix async and sync libraries without care (choose async-friendly ones).