

### 1. asyncio คืออะไร และทำไมมันถึงเหมาะกับการใช้งาน non-blocking I/O application?

= asyncio เป็นโมดูลใน Python ที่ช่วยจัดการกับการทำงานแบบ asynchronous concurrency (การทำงานพร้อมกันแบบไม่รอ) โดยเฉพาะในกรณีที่มีการทำงานกับ I/O แบบ non-blocking เช่น การรอรับข้อมูลจากเครือข่ายหรือการอ่าน/เขียนไฟล์ ซึ่งไม่ต้องให้โปรแกรมหยุดทำงานในขณะที่รอ ทำให้สามารถเพิ่มประสิทธิภาพในการรันโปรแกรมที่มี I/O หนัก ๆ ได้อย่างมีประสิทธิภาพ

### 2. อธิบายความหมายของ coroutine ในบริบทของ Python และการใช้งาน asyncio

= coroutine เป็นฟังก์ชันที่ถูกสร้างขึ้นด้วยคีย์เวิร์ด `async def` และสามารถหยุดทำงานชั่วคราวได้ด้วยคำสั่ง `await` เพื่อรอให้ฟังก์ชัน asynchronous อื่นทำงานเสร็จก่อนที่จะทำงานต่อ Coroutines ใน asyncio เป็นฟังก์ชันที่ทำงานอย่างไม่เป็นลำดับ เพื่อจัดการกับการทำงานแบบ concurrent

### 3. นิยาม task ใน asyncio และบอกหน้าที่หลักของมัน

= Task เป็น object ที่สร้างขึ้นเพื่อจัดการและดำเนินการ coroutine ในรูปแบบ asynchronous มันจะทำให้ coroutine ทำงานควบคู่ไปกับส่วนอื่น ๆ ของโปรแกรมโดยไม่ต้องรอให้เสร็จสิ้นก่อนจึงจะดำเนินการต่อ

### 4. อธิบายฟังก์ชัน `asyncio.run()` ทำหน้าที่อะไร และเมื่อใดที่ควรใช้

= `asyncio.run()` เป็นฟังก์ชันที่ใช้ในการเรียกใช้ coroutine หลักและควบคุมการรัน event loop โดยอัตโนมัติ โดยจะสร้างและจัดการ event loop สำหรับรัน coroutine เมื่อรันเสร็จแล้วจะปิด loop ทันที นิยมใช้เมื่อคุณต้องการรัน coroutine หลักในโปรแกรม


### 5. อธิบายการใช้ `await` ใน asyncio และมันทำงานอย่างไรกับ coroutine

= คำสั่ง `await` ใช้เพื่อระงับการทำงานของ coroutine จนกว่าจะมีการคืนค่าให้กับ asynchronous function (เช่น coroutine อื่น, task, หรือ future) คำสั่งนี้จะหยุดการทำงานชั่วคราวและอนุญาตให้รันส่วนอื่น ๆ ใน event loop ไปก่อน

### 6. อธิบายวิธีสร้าง coroutine ด้วยการใช้ `async def` และการรันด้วย `await`

คุณสามารถสร้าง coroutine ด้วยการใช้ `async def` ดังนี้:


python

 Copy code

```
async def my_coroutine():  
    await asyncio.sleep(1)  
    return "Done!"
```

และเพื่อรัน coroutine คุณสามารถใช้คำสั่ง `await` ใน event loop:

python


 Copy code

```
result = await my_coroutine()
```

### 7. อธิบายความหมายของ async comprehension และยกตัวอย่างการใช้งาน

Async comprehension คือการใช้ comprehension เช่น list หรือ set ที่ภายในใช้ `async for` เพื่อ iterate ข้อมูลจาก asynchronous generator ตัวอย่าง:

python


 Copy code

```
result = [x async for x in async_gen()]
```

## 8. นิยามของ async for-loop คืออะไร และมันต่างจาก for-loop ปกติอย่างไร?

Async for-loop ใช้ในการ loop ข้อมูลที่มาจาก asynchronous iterator หรือ asynchronous generator ซึ่งมีการใช้ await ในการดึงค่าจากแต่ละ step แตกต่างจาก for-loop ปกติที่ใช้กับ iterator ปกติ ตัวอย่าง:

python


 Copy code

```
async for item in async_gen():  
    print(item)
```

## 9. อธิบายการใช้งานของ asyncio.create\_task() ในการสร้างและจัดการ task

asyncio.create\_task() ใช้เพื่อสร้าง task ใหม่จาก coroutine และทำให้ coroutine นั้นทำงานแบบ asynchronous โดย task จะถูกรันควบคู่ไปกับ event loop อื่น ๆ โดยไม่ต้องรอให้ task นั้นเสร็จสิ้นก่อน:

python


 Copy code

```
task = asyncio.create_task(my_coroutine())
```

## 10. อธิบายฟังก์ชัน asyncio.ensure\_future() และเปรียบเทียบกับ asyncio.create\_task()

asyncio.ensure\_future() สร้างและจัดการ coroutine ในลักษณะเดียวกับ asyncio.create\_task() แต่มีความยืดหยุ่นในการรับทั้ง task ที่ถูกสร้างขึ้นแล้วหรือ coroutine ที่ยังไม่ได้สร้าง task การใช้ ensure\_future() จะมีประโยชน์ในสถานการณ์ที่ต้องการตรวจสอบวัตถุที่ส่งมาเป็น task อยู่แล้วหรือยังไม่ใช่:

python


 Copy code

```
future = asyncio.ensure_future(my_coroutine())
```

## 11. นิยามของ shielded task ใน asyncio คืออะไร และใช้งาน asyncio.shield() อย่างไร?

Shielded task คือ task ที่ถูกป้องกันจากการยกเลิกด้วยการใช้ asyncio.shield(), หากคุณต้องการให้ task ไม่ถูกยกเลิกเมื่อมีการยกเลิก parent task คุณสามารถใช้ shield():

python


 Copy code

```
task = asyncio.shield(my_coroutine())
```

## 12. อธิบายฟังก์ชัน asyncio.gather() และเมื่อใดที่ควรใช้งานมัน

asyncio.gather() ใช้เพื่อรันหลาย ๆ coroutine พร้อมกันและรวบรวมผลลัพธ์ของมันทั้งหมด เหมาะกับการทำงานที่ต้องรันหลาย task พร้อมกันและต้องการรอให้เสร็จทั้งหมด:

python


 Copy code

```
result = await asyncio.gather(coro1(), coro2(), coro3())
```

### 13. อธิบายวิธีการรอหลาย task พร้อมกันด้วย asyncio.wait()

asyncio.wait() ใช้เพื่อรอหลาย task หรือ coroutines พร้อมกัน คุณสามารถรอให้ task ทั้งหมดเสร็จหรือหยุดหลังจาก task แรกเสร็จ:

python

 Copy code

```
done, pending = await asyncio.wait([task1, task2])
```


### 14. นิยามของ asyncio.TimeoutError คืออะไร และใช้เมื่อใด?

asyncio.TimeoutError เป็นข้อผิดพลาดที่ถูกยกขึ้นเมื่อการรอ coroutine หรือ task เกินเวลาที่กำหนด (timeout) ใช้ในกรณีที่ต้องการตั้งเวลาจำกัดให้ทำงาน

### 15. อธิบายวิธีการใช้ asyncio.wait\_for() ในการจัดการ timeout ให้กับ coroutine

asyncio.wait\_for() ใช้เพื่อรัน coroutine หรือ task พร้อมกับกำหนดเวลา timeout หากเกินเวลา timeout ที่กำหนดไว้ task จะถูกยกเลิก:

python

 Copy code

```
result = await asyncio.wait_for(my_coroutine(), timeout=5)
```

คำตอบเกี่ยวกับ Debugging:

### 16. ใน asyncio จะทำอะไรเพื่อ ตรวจสอบสถานะ ของ task ที่กำลังทำงาน เช่น รันอยู่, เสร็จสิ้น หรือถูกยกเลิก?


คุณสามารถตรวจสอบสถานะของ task ได้ด้วย:

- task.done() ตรวจสอบว่า task เสร็จหรือยัง
- task.cancelled() ตรวจสอบว่า task ถูกยกเลิกหรือไม่
- task.exception() ตรวจสอบว่า task มีข้อผิดพลาดหรือไม่

### 17. จะ ยกเลิก task อย่างไรใน asyncio โดยใช้ฟังก์ชัน task.cancel() และตรวจสอบว่าถูกยกเลิกสำเร็จหรือไม่?

คุณสามารถยกเลิก task ได้โดยใช้ task.cancel() และตรวจสอบการยกเลิกด้วย task.cancelled():

python


 Copy code

```
task.cancel()
if task.cancelled():
    print("Task was cancelled")
```

### 18. อธิบายวิธีการตรวจสอบ unhandled exceptions ใน task โดยใช้ task.exception()

ใช้ task.exception() เพื่อดูข้อผิดพลาดที่เกิดขึ้นใน task หากไม่มีข้อผิดพลาดจะคืนค่า None หากมีข้อผิดพลาดจะคืนค่า exception:

python

 Copy code

```
error = task.exception()
if error:
    print(f"Error occurred: {error}")
```

19. จะเพิ่มฟังก์ชัน callback เมื่อ task เสร็จสิ้นได้อย่างไร โดยใช้ task.add\_done\_callback()?

ใช้ task.add\_done\_callback() เพื่อเพิ่ม callback ที่จะถูกเรียกเมื่อ task เสร็จ:

python

Copy code

```
def on_done(task):  
    print("Task is done")  
  
task.add_done_callback(on_done)
```

20. วิธีการ ลบ callback ออกจาก task ที่ถูกเพิ่มไว้ก่อนหน้านี้ใน asyncio ควรทำอย่างไร?

คุณสามารถลบ callback ออกได้โดยการไม่ใส่ callback อีกต่อไป และแทนที่ task ใหม่โดยไม่ใช้ callback

## 1. อธิบายการทำงานของ async และ await ใน Python และให้ตัวอย่างการใช้งานที่ถูกต้อง

- `async` และ `await` เป็นคีย์เวิร์ดใน Python ที่ใช้ในการทำงานแบบ asynchronous เพื่อให้สามารถทำงานหลาย ๆ อย่างพร้อมกันได้โดยไม่ต้องรอให้แต่ละงานเสร็จก่อนถึงจะเริ่มงานถัดไป
  - `async` ใช้ในการกำหนดฟังก์ชันว่าเป็น coroutine (การทำงานแบบ asynchronous)
  - `await` ใช้ในการหยุดชั่วคราวเพื่อรอให้ผลลัพธ์จากฟังก์ชัน asynchronous เสร็จก่อนที่จะดำเนินการต่อ
  - ตัวอย่างการใช้งาน:

python

Copy code

```
import asyncio  
  
async def my_coroutine():  
    print("Start")  
    await asyncio.sleep(2) # รอ 2 วินาที  
    print("End")  
  
asyncio.run(my_coroutine())
```

ในตัวอย่างนี้ ฟังก์ชัน `my_coroutine()` จะเริ่มต้นทำงาน พอเจอคำสั่ง `await asyncio.sleep(2)` ก็จะหยุดรอ 2 วินาทีก่อนจะดำเนินการต่อ

## 2. อธิบายและเปรียบเทียบการใช้ `asyncio.gather()` และ `asyncio.wait()` ในการจัดการ task พร้อมกัน

- `asyncio.gather()` และ `asyncio.wait()` ใช้ในการจัดการ task หลาย ๆ task พร้อมกัน แต่มีความแตกต่างดังนี้:
  - `asyncio.gather()` จะรันหลาย ๆ coroutine พร้อมกันและรอจนทุก coroutine เสร็จสมบูรณ์ จากนั้นจะคืนค่าผลลัพธ์ทั้งหมดในรูปแบบของลิสต์:

```
python                                                                    Copy code

async def task1():
    await asyncio.sleep(1)
    return "Task 1 done"

async def task2():
    await asyncio.sleep(2)
    return "Task 2 done"

results = await asyncio.gather(task1(), task2())
print(results) # Output: ['Task 1 done', 'Task 2 done']
```

- `asyncio.wait()` ใช้เพื่อรอหลาย task พร้อมกันโดยสามารถเลือกว่าจะรอให้ task ทั้งหมดเสร็จหรือรอแค่ task ที่เสร็จก่อน:

```
python                                                                    Copy code

done, pending = await asyncio.wait([task1(), task2()])
for task in done:
    print(task.result()) # Task ที่เสร็จก่อนจะถูกคืนค่าก่อน
```

ความแตกต่างหลัก:

- `gather()` จะรวบรวมผลลัพธ์ของทุก task ที่เสร็จแล้วไว้ในลิสต์ทันที ในขณะที่ `wait()` จะแยก task ที่เสร็จแล้วและ task ที่ยังค้างอยู่ในสองชุด คือ `done` กับ `pending`

### 3. อธิบายการใช้ TaskGroup ใน asyncio และยกตัวอย่างสถานการณ์ที่เหมาะสมในการใช้งาน

- `TaskGroup` เป็นกลไกใหม่ใน Python 3.11 ที่ช่วยจัดการกลุ่มของ tasks โดยให้สามารถรัน tasks พร้อมกัน และรอจนทุก task เสร็จพร้อมกันอย่างปลอดภัย นอกจากนี้ หาก task ใดเกิดข้อผิดพลาด `TaskGroup` จะจัดการยกเลิก tasks อื่น ๆ ทันที
- ตัวอย่าง:

```
python Copy code

import asyncio

async def task1():
    await asyncio.sleep(1)
    return "Task 1 finished"

async def task2():
    await asyncio.sleep(2)
    return "Task 2 finished"

async def main():
    async with asyncio.TaskGroup() as tg:
        t1 = tg.create_task(task1())
        t2 = tg.create_task(task2())
    print(t1.result(), t2.result())

asyncio.run(main())
```

`TaskGroup` เหมาะสมในการใช้งานเมื่อคุณต้องการจัดการ task หลายตัวพร้อมกันและต้องการให้ task เหล่านั้นมีการควบคุมในกรณีที่ มีข้อผิดพลาดหรือการยกเลิก task อื่นๆ ตัวอย่างเช่น:

- การดาวน์โหลดไฟล์หลายไฟล์พร้อมกัน ถ้าการดาวน์โหลดไฟล์ใดไฟล์หนึ่งล้มเหลว คุณต้องการให้ไฟล์ที่เหลือยกเลิกทันที
- การประมวลผลงานหลายขั้นตอนพร้อมกัน และทุกขั้นตอนต้องสำเร็จพร้อมกันเพื่อผลลัพธ์ที่ต้องการ

ในกรณีนี้ ถ้า task หนึ่งล้มเหลว `TaskGroup` จะทำการยกเลิก tasks อื่นๆ เพื่อหลีกเลี่ยงการใช้ทรัพยากรที่ไม่จำเป็น

#### 4. การใช้ `await asyncio.sleep()` ใน coroutine มีข้อควรระวังหรือความผิดพลาดที่พบบ่อยอย่างไรบ้าง?

- `await asyncio.sleep()` ใช้สำหรับหยุดชั่วคราวใน coroutine โดยไม่บล็อก event loop ซึ่งหมายความว่าระหว่างที่รอเวลาอื่น ๆ ใน event loop ยังสามารถทำงานได้
- ข้อควรระวังและความผิดพลาดที่พบบ่อย:

##### 1. ลืมใส่ `await` ก่อน `asyncio.sleep()`:

- หากลืมใส่ `await` คำสั่ง `asyncio.sleep()` จะไม่ทำงานและผลลัพธ์ที่ได้จะไม่เป็นไปตามที่คาดหวัง เช่น coroutine จะไม่หยุดชั่วคราว
- ตัวอย่าง:

```
python Copy code  
  
async def wrong_sleep():  
    asyncio.sleep(1) # ผิด: ต้องใส่ await  
    print("This runs immediately")
```

##### 2. ใช้ `asyncio.sleep()` แทนการรอ task อื่นๆ:

- `asyncio.sleep()` มีไว้สำหรับการหยุดชั่วคราว ไม่ใช่การรอ task อื่น ถ้าคุณต้องการรอ task อื่นเสร็จ ให้ใช้ `await` หรือฟังก์ชันเช่น `gather()` หรือ `wait()`

##### 3. ใช้งานในบริบทที่ไม่จำเป็น:

- การใช้ `asyncio.sleep()` ในที่ที่ไม่จำเป็นอาจทำให้โปรแกรมทำงานช้าเกินไป โดยเฉพาะเมื่อใช้ในการทดสอบหรือจำลองเงื่อนไขแบบ synchronous

## 5. อธิบายการใช้งาน Queue ใน `asyncio` เพื่อการสื่อสารระหว่าง coroutines และยกตัวอย่างการใช้ Queue ในงานจริง

- `asyncio.Queue` ใช้สำหรับการสื่อสารข้อมูลระหว่าง coroutines โดยไม่บล็อก coroutine ใด ๆ ทำให้สามารถส่งต่อข้อมูลระหว่างงานต่างๆ ได้อย่างมีประสิทธิภาพ
  - การทำงานของ `Queue` :
    - Producer ใส่ข้อมูลลงใน queue
    - Consumer รับข้อมูลจาก queue โดยไม่ต้องรอให้ producer ทำงานเสร็จ
    - รองรับทั้งขั้นตอนการทำงานแบบ FIFO (First In, First Out)
  - ตัวอย่าง:

```
python Copy code

import asyncio

async def producer(queue):
    for i in range(5):
        await asyncio.sleep(1) # จำลองการผลิตข้อมูล
        await queue.put(1)
        print(f"Produced {i}")

async def consumer(queue):
    while True:
        item = await queue.get()
        print(f"Consumed {item}")
        queue.task_done()

async def main():
    queue = asyncio.Queue()
    # สร้าง producer และ consumer
    producer_task = asyncio.create_task(producer(queue))
    consumer_task = asyncio.create_task(consumer(queue))

    await producer_task
    await queue.join() # รอให้ queue ว่าง
    consumer_task.cancel() # ยกเลิก consumer

asyncio.run(main())
```

ในตัวอย่างนี้ producer จะผลิตข้อมูลลงใน queue และ consumer จะคอยดึงข้อมูลจาก queue ไปใช้งานต่อ วิธีนี้ช่วยให้สามารถสื่อสารระหว่าง coroutines ได้อย่างมีประสิทธิภาพและปลอดภัยจากปัญหาการบล็อก

การใช้งานในงานจริง:

- Web Scraping: Producer รวบรวม URLs ที่ต้อง scrape แล้วใส่ลงใน queue ส่วน Consumer จะดึง URLs จาก queue ไปประมวลผล
- งานประมวลผลข้อมูล: Producer สร้างข้อมูลหรืออ่านจากไฟล์ ส่วน Consumer คอยประมวลผลข้อมูลที่ได้รับ