GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura i Esport

ceedcv CENTRE ESPECÍFIC D'EDUCACIÓ A DISTÀNCIA DE LA COMUNITAT VALENCIANA

Unidad 5.2
Programación asíncrona

Índice

Licencia

Reconocimiento – NoComercial – CompartirIgual (by-nc-sa): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

1 La librería asyncio

En las versiones anteriores de Python, la concurrencia se abordaba principalmente mediante el uso de hilos (`Threads`), bloqueos (`Locks`), semáforos (`Semaphores`), eventos (`Events`), temporizadores (`Timers`), y colas (`Queues`). Estas herramientas permitían gestionar la ejecución de múltiples tareas de forma concurrente, pero conllevaban desafíos asociados a la complejidad y la posibilidad de condiciones de carrera.

Transición a Python 3.4 (2014):

Con el lanzamiento de Python 3.4 en 2014, se introdujo un cambio significativo en la forma en que Python aborda la concurrencia. Por un lado tenemos el enfoque tradicional de multithreading y por otro lado se introduce la idea de un único hilo principal que ejecuta y gestiona corrutinas y tareas, adoptando así el paradigma de programación asíncrona. Este cambio trajo consigo la introducción de **asyncio,** esta librería es especialmente valiosa cuando nos enfrentamos a operaciones que pueden consumir mucho tiempo, como solicitudes a servidores, operaciones de entrada/salida en archivos o eventos de red.

Corrutinas y AsyncIO:

Como hemos dicho Asyncio facilita la escritura de código asíncrono mediante el uso de corrutinas, que son **funciones especiales capaces de ser pausadas y luego reanudadas**. Al emplear palabras clave como `async` y `await`, asyncio permite a los desarrolladores estructurar su código de manera que las operaciones bloqueantes no impidan la ejecución de otras tareas.

- **Async**: Al marcar una función con `async`, estamos indicando al intérprete de Python que esa función puede contener operaciones asíncronas. Es un modificador que señala que la función puede ser suspendida y luego reanudada, permitiendo que otras operaciones se ejecuten mientras tanto.
- **Await**: La palabra clave `await` se utiliza dentro de la función asíncrona para indicar puntos en los cuales la ejecución puede detenerse temporalmente. Esta pausa permite que otras tareas se ejecuten mientras esperamos la finalización de una operación asíncrona. Es esencial para asegurar que la ejecución del programa no se bloquee durante la espera de resultados asíncronos.

El patrón `async/await` en Python tiene como objetivo fundamental permitir que el programa principal continúe ejecutándose y realice otras tareas mientras aguarda la conclusión de tareas asíncronas que podrían ser prolongadas.

Bucles de Eventos No Bloqueantes y Callbacks:

En la programación asíncrona se suele usar un bucle principal que está permanentemente encargado de ejecutar las corrutinas y recoger los eventos que estas le devuelven. Funciona como un planificador que coordina la ejecución de múltiples corrutinas y atiende a los eventos que éstas le van enviando para permitir que el programa principal realice otras tareas mientras espera la finalización de operaciones asíncronas. Un evento puede ser que una corrutina ha finalizado la operación de E/S.

Podríamos pensar que ese bucle principal es como el kernel de Linux que va gestionando las diferentes tareas, con la gran diferencia que las tareas devuelven el control al kernel **de forma voluntaria** cuando se bloquean por alguna razón (espera de una E/S).

Por ejemplo, en el código que vamos a trabajar:

- asyncio.run(main()) - Es el bucle principal que se ejecuta mientras hayan corrutinas ejecutándose y eventos a los que atender.

Callbacks: es una función ejecutable «A» que se usa como argumento de otra función «B». En este caso tenemos corrutinas implementadas por la librería **asyncio** y usamos el concepto de callback para pasarles por argumento corrutinas definidas por nosotros. De esta forma personalizamos el funcionamiento de la librería **asyncio** con nuestro código y nuestras necesidades específicas.

Por ejemplo, en el código que vamos a trabajar:

- servidor = await asyncio.start_server(**corutina_servidor**, host=HOST, port=PORT)

  Tenemos una corrutina implementada por la librería **asyncio** que se llama **start_server**. A esta corrutina le pasamos por parámetro un callback llamado **corutina_servidor**, que es una corrutina implementada por nosotros y que queremos que se ejecute simultáneamente con el resto de corrutinas.

2 Cosas a tener en cuenta cuando se programa asíncronamente

- Todas las funciones que realizan operaciones bloqueantes (E/S, solicitudes a servidores, descargas de recursos, procesamiento de datos, etc.) deben definirse con **async def**
- Cada vez que se llama a una corrutina que realiza operaciones bloqueantes hay que llamarlas con await. Ejemplo: **await** asyncio.sleep(…)
- Si tenemos la necesidad de ejecutar código síncrono de forma paralela a nuestra ejecución asíncrona, se crearán threads secundarios al thread principal. En qué situaciones se puede dar esta necesidad:
  - Cuando la API que usamos no tiene las funciones implementadas de forma asíncrona y sólo ofrece funciones síncronas. Las llamadas a esta API se harán en hilos paralelos que se ejecutarán de forma síncrona.
  - Cuando nuestro programa necesita realizar cálculos intensivos en CPU como por ejemplo factorizar números primos. Este tipo de operaciones secuestran la CPU y no dejan que otras corrutinas más pequeñas o ligeras se ejecuten. Es por ello que se pondría esta parte en un hilo secundario y se permitiría en el hilo principal ejecutar de forma asíncrona todas estas corrutinas más ligeras.
- **Cómo se hace**: await **asyncio.to_thread(**nombre_función**,** argumento1**,** argumento2**,** **...**) de esta forma incorporamos una corrutina que el programa principal irá vigiando y teniendo en cuenta en su planificación asíncrona, que resulta ser un hilo secundario. Cuando el hilo acabe, lanzará un evento de fin y se incorporarán sus resultados a las corrutinas que los esperaban.
  - **Cuidado no**: ~~await asyncio.to_thread(nombre_función(argumento1, argumento2, ...))~~

3 Referencias

https://docs.python.org/3/library/asyncio.html

https://en.wikipedia.org/wiki/Async/await