**Corutinas en Python**

**Introducción**

En el desarrollo de software moderno, especialmente en aplicaciones que requieren alta concurrencia (como servidores web, bots de chat o programas que interactúan con APIs), es fundamental comprender cómo manejar tareas que dependen de operaciones de entrada/salida (I/O). Tradicionalmente, estas tareas podían bloquear la ejecución del programa, afectando su eficiencia.

Las corutinas (del inglés coroutines) surgen como una solución para modelar asincronía de manera legible y estructurada, evitando el uso excesivo de callbacks y facilitando la escritura de programas concurrentes.

**Conceptos previos en el contexto de corutinas**

**Repaso de Concepto de Proceso y de Hilo**

**Proceso**

Un proceso es un programa en ejecución. Es la unidad básica de asignación de recursos en un sistema operativo.

Cada proceso tiene su propio espacio de memoria, archivos abiertos, estado de ejecución y otros recursos del sistema. Es independiente de otros procesos (es decir, que se ejecuta de forma aislada cada uno).

Un proceso puede tener uno o más hilos de ejecución.

**Hilo (o thread)**

Un hilo es la unidad más pequeña de ejecución dentro de un proceso. Es una secuencia de instrucciones que puede ejecutarse de forma concurrente con otros hilos del mismo proceso. Los hilos comparten el mismo espacio de memoria y recursos del proceso, pero tienen su propio contador de programa, pila y registros.

**Operación bloqueante y no bloqueante**

**Operación bloqueante**

Operación bloqueante: operación que detiene el flujo de ejecución hasta que se completa. El hilo que la ejecuta queda inactivo, esperando el resultado. Ejemplo: Leer un archivo o esperar una respuesta de red. Si se usa una función bloqueante, el hilo no puede hacer otra cosa hasta que esa operación termine. Como consecuencia, en sistemas con pocos hilos (como servidores), esto puede causar cuellos de botella y baja eficiencia.

**Operación no bloqueante**

Operación no bloqueante: una operación que inicia una tarea (como una lectura o escritura) y continúa ejecutando otras instrucciones sin esperar que esa tarea termine. Ejemplo: Enviar una solicitud HTTP y seguir ejecutando código. Como consecuencia, se mejora la eficiencia y la capacidad de respuesta, especialmente en aplicaciones con muchas operaciones de entrada/salida

**Forma de ejecutar varias instrucciones: secuencial, concurrente o en paralelo**

* Secuencial: varias instrucciones se ejecutan una tras otra, sin superposición.
* Concurrente: varias instrucciones o tareas que se ejecutan de manera solapada, es decir, avanzan intercalando su ejecución (no necesariamente en simultáneo). Por ejemplo: un solo hilo de ejecución que alterna entre tareas, como cuando una operación queda bloqueada (podría ser esperando una respuesta de red) y el sistema aprovecha para avanzar con otra tarea.
* Paralelo: varias instrucciones o tareas se ejecutan de manear simultánea (al mismo tiempo), típicamente en múltiples núcleos o hilos de procesamiento. Por ejemplo: dos tareas que realmente se ejecutan al mismo tiempo en distintos núcleos de CPU.

|  |  |  |
| --- | --- | --- |
| Concepto | Concurrencia | Paralelismo |
| Tiempo | Tareas se solapan | Tareas se ejecutan al mismo tiempo |
| Recursos | Puede usar un solo hilo | Requiere múltiples núcleos o hilos |
| Ejecución | Alternancia entre tareas | Ejecución simultánea |
| Ejemplo | Corutinas en un solo hilo | Procesos en paralelo |
| Objetivo | Mejorar eficiencia sin bloquear recursos | Acelerar procesamiento usando más hardware |

**Programa Asíncrono vs Programa Síncrono**

* Programa Síncrono: Conjunto de instrucciones que se ejecutan una tras otra, de forma secuencial. Cada operación debe completarse antes de que comience la siguiente.
* Programa Asíncrono: Conjunto de instrucciones que al ejecutarse pueden pausarse mientras esperan (por ejemplo, una respuesta de red), permitiendo que el programa ejecute otras tareas de manera concurrente, sin bloquearse.

**Conclusión:**

**Un programa síncrono suele ser secuencial y bloqueante.**

* Las instrucciones se ejecutan una tras otra, en orden.
* Si una operación tarda (como leer un archivo o esperar una respuesta de red), bloquea el hilo hasta que se completa, lo que significa que el programa no puede avanzar hasta que esa tarea termine.

**Un programa asíncrono permite concurrencia con operaciones no bloqueantes.**

* Las tareas pueden iniciarse y suspenderse, permitiendo que otras se ejecuten mientras tanto.
* Las operaciones de entrada/salida (I/O) se hacen de forma no bloqueante, lo que permite concurrencia incluso en un solo hilo.
* Esto es especialmente útil en aplicaciones que manejan muchas tareas I/O (como servidores web, bots, entre otros).

|  |  |  |
| --- | --- | --- |
| Concepto | Bloqueante | No bloqueante |
| Ejecución | Detiene el hilo hasta que finaliza | El hilo sigue ejecutando otras tareas |
| Uso de corutinas | No aprovecha su potencial | Ideal para suspender y reanudar sin bloquear |
| Rendimiento | Menor eficiencia en sistemas concurrentes | Mayor escalabilidad y fluidez |

|  |  |  |
| --- | --- | --- |
| Concepto | Síncrono | Asíncrono |
| Ejecución | Secuencial | Concurrencia posible |
| Bloqueo | Sí (espera que termine la tarea) | No (puede suspender y continuar con otra) |
| Uso típico | Scripts simples, tareas CPU | Aplicaciones I/O intensivas, servidores, bots |

**Repaso de la definición de Clase:**

Una clase es una estructura de datos, una abstracción, que encapsula:

* Estado: representado por atributos o propiedades.
* Comportamiento: definido por métodos o funciones.

**Repaso de la definición de Instancia:**

Una instancia es una representación en memoria de la estructura definida por la clase, con valores específicos para sus atributos.

**Repaso de la definición de Objeto**

**Objeto en Programación Orientada a Objetos (POO)**

Un objeto es una instancia de una clase, que encapsula estado (atributos) y comportamiento (métodos).

Los objetos interactúan entre sí mediante el envío de mensajes (llamadas a métodos), y son la unidad básica de abstracción en POO.

Ejemplo: Si se tiene una clase Perro, un objeto sería mi\_perro = Perro("Firulais"). Ese objeto tiene atributos como nombre, edad, y métodos como ladrar().

**Objeto en Python**

En Python, **todo es un objeto**: funciones, clases, números, cadenas, listas, diccionarios, corutinas, incluso los propios módulos. Cada uno de estos elementos es una instancia de alguna clase definida en el sistema de tipos de Python.

Ejemplo:

* 3 es un objeto de la clase int
* "hola" es un objeto de la clase str
* def saludar(): ... es un objeto de la clase function
* async def tarea(): ... devuelve una **corutina**, que es un objeto de tipo coroutine

**Comparación: Objeto en POO y en Python**

|  |  |  |
| --- | --- | --- |
| **Concepto** | **POO tradicional** | **Python** |
| Qué es un objeto | Instancia de una clase que encapsula datos y métodos | Cualquier entidad que tenga tipo y comportamiento |
| ¿Todo es objeto? | No necesariamente (depende del lenguaje) | Sí, absolutamente todo |
| Métodos y funciones | Son parte de una clase, no objetos por sí mismos | Son objetos también (de tipo function) |
| Corrutinas | No existen como concepto estándar | Son objetos especiales que se pueden pausar y reanudar |

Cada elemento que se crea en Python es una **instancia de una clase**. En Python, **todo objeto hereda de** object, que es la clase base universal.

Esto permite tratar a cada elemento creado en Python como un dato, es decir, que se lo puede pasar como argumento, almacenarlo en alguna estructura, inspeccionarlo, etcétera.

Un diccionario, una lista, una función o una corutina son objetos en Python, mientras que en otros lenguajes no.

Una corutina es un objeto en Python porque está definida como una instancia de una clase interna (coroutine) e implementa el protocolo asíncrono en Python.

**Corutina**

En Python, una **corutina** es un objeto que encapsula código capaz de ejecutarse, y que durante su ejecución puede **pausarse y reanudarse** en distintos momentos.

Es decir, puede incluir múltiples operaciones asincrónicas, como llamadas a funciones que llevan adelante consultas de entrada/salida (I/O) o esperan respuestas de red (por ejemplo, llamada a la API de Discord y quedar a la espera de una respuesta para usarla) o leer archivos, entre otras, sin bloquear el flujo principal del programa.

Por lo tanto, las funciones corutinas permiten concurrencia, es decir, que las instrucciones se solapan, intercalan su ejecución, de manera que se queda sin congelar el programa.

Cuando se llama a una función corutina (que tiene async def en su definición), se crea y devuelve un objeto corutina que **no se ejecuta inmediata ni automáticamente.** Es decir, se lo debe iniciar, detener o retomar en distintos puntos del flujo de un programa. **De eso se encarga el event loop (bucle de eventos)**.

**Un event loop** es un motor que gestiona la ejecución de corutinas (se suele crear con la instrucción asyncio.run() en Python, es decir, que lo maneja el módulo asyncio, que permite la ejecución de instrucciones asíncronas desde el mundo síncrono). Decide cuándo se ejecutan, se suspenden y se reanudan.

**Ejemplo básico (para entender cómo se definen y se ejecutan las corutinas)**

Entonces, para ejecutar el código presente en una función corutina, es necesario que ésta sea gestionada por un bucle de eventos (event loop). Esto puede lograrse de dos formas:

1. Usando await dentro de otra corutina (asyn def) que ya esté siendo ejecutada por un event loop (asyncio.run()).

# módulo de Python para conectar mundo síncrono y asíncrono

import asyncio

# se define corutina poniendo async def

async def saludar():

print("Hola desde una corutina")

# usando await dentro una corutina para llamar a otra corutina

# se define corutina poniendo async def

async def main():

# await para llamar a una corutina dentro de otra

# pero no la ejecuta

await saludar()

# esto crea el evento loop y ejecuta la corutina

asyncio.run(main())

# await: No ejecuta la corutina, sino que se usa dentro de una corutina para esperar el resultado de otra corutina. No puede usarse en el nivel superior del código (fuera de una corutina).

# asyncio.run(): Es la forma más común de iniciar el event loop y ejecutar una corutina principal desde el nivel superior del código.

1. Ejecutando la corutina directamente desde el nivel superior usando asyncio.run(), que crea y gestiona el event loop automáticamente.

# módulo de Python para conectar mundo síncrono y asíncrono

import asyncio

# se define corutina poniendo async def

async def saludar():

print("Hola desde una corutina")

# usando asyncio.run() en el nivel superior

# esto crea el evento loop y ejecuta la corutina

asyncio.run(saludar())

# nivel superior: se refiere a*fuera* de cualquier función o corutina, es decir, el código que está directamente en el cuerpo principal del script o módulo de Python.

# asyncio.run(): Es la forma más común de iniciar el event loop y ejecutar una corutina principal desde el nivel superior del código.

**Más ejemplos.**

A continuación, se presentan 3 programas:

* síncrono secuencial
* asíncrono secuencial
* asíncrono concurrente

con el objetivo de destacar la importancia del uso adecuado de la librería asyncio para aprovechar la concurrencia y, por lo tanto, el uso de los recursos.

**Código síncrono secuencial (tradicional)**

import time

# función que no usa async def porque no corresponde

def tarea(nombre, segundos):

print(f"{nombre} empezando...")

# detiene la ejecución del programa cierta cantidad de “segundos”

time.sleep(segundos)

print(f"{nombre} terminó después de {segundos} segundos")

# se ejecuta directamente invocando el nombre de la función

tarea("Tarea 1", 2)

tarea("Tarea 2", 3)

tarea("Tarea 3", 1)

Cada llamada a la función tarea() **bloquea** el hilo principal. Esto es propio de un programa asíncrono, donde una tarea no comienza hasta que finaliza la anterior.

El tiempo total de ejecución es la suma del tiempo que tarda cada una de las tareas en ejecutarse: (**2 + 3 + 1) segundos = 6 segundos**

Las tareas no se solapan (no hay concurrencia), ni se llevan delante de manera paralela.

**Código asincrónico secuencial (con await)**

# se importa librería para ejecutar instrucciones asíncronas

import asyncio

# se declara una corutina con async def

async def tarea(nombre, segundos):

print(f"{nombre} empezando...")

# detiene el flujo de ejecución cierta cantidad de “segundos”

await asyncio.sleep(segundos)

print(f"{nombre} terminó después de {segundos} segundos")

# se declara otra corutina para usar la corutina anterior con await

async def main():

await tarea("Tarea 1", 2)#await para esperar resultado de corutina

await tarea("Tarea 2", 3)#se suele esperar operación asicnrónica

await tarea("Tarea 3", 1)

# se necesita asyncio.run() para crear el evento loop que se encarga de la gestión de corutinas, su ejecución y es la forma de pasar del mundo síncrono al asíncrono

asyncio.run(main())

En este ejemplo, aunque se use async y await, el flujo sigue siendo **secuencial**, ya que cada await espera que la tarea anterior finalice. Por lo tanto, el tiempo total de ejecución del programa es la suma de los tiempos individuales de ejecución de cada instrucción: (2 +3+1) segundos = **6 segundos.** No hay concurrencia. Ni paralelismo.

**Código asíncrono concurrente (create\_task(corutina()))**

# se importa librería para ejecutar instrucciones asíncronas

import asyncio

# se declara una corutina con async def

async def tarea(nombre, segundos):

print(f"{nombre} empezando...")

# detiene el flujo de ejecución de esa tarea cierta cantidad de “segundos” pero el asyncio.sleep() permite que se ejecuten otras de manera concurrente, lo que no pasa con time.sleep()

await asyncio.sleep(segundos)

print(f"{nombre} terminó después de {segundos} segundos")

# se declara otra corutina con async def para usar a la anterior corutina con await

async def main():

# asyncio.create\_task : envuelve una corutina, para que la gestione y la ejecute el event loop lo antes posible

# las tareas se ejecutan en ese orden, pero de manera concurrente

t1 = asyncio.create\_task(tarea("Tarea 1", 2))

t2 = asyncio.create\_task(tarea("Tarea 2", 3))

t3 = asyncio.create\_task(tarea("Tarea 3", 1))

# el orden del await solo determina qué resultado se espera primero

await t1

await t2

await t3

# asyncio.run() para crear el evento loop que se encarga de gestionar y ejecutar las corutinas

asyncio.run(main())

Las tres tareas se **intercalan**, el encargado de llevar a cabo el cambio entre una tarea y otra es el event loop. Mientras una “duerme”, la otra puede avanzar, todo en el mismo hilo.

Entonces, el tiempo total que se va a tener para ejecutar las tres tareas será de 3 segundos (coincide con el valor de la tarea que más tarda en ejecutarse). Ya no se trata de la suma de los tiempos porque se acoplan, se corren de manera concurrente.

Es importante destacar que el tiempo destinado al intercambio de tareas es en milisegundos, por lo que pareciera que la ejecución de las tareas se de forma paralela, pero en realidad es concurrente (una tras otras), mientras una está ocupada en una operación de entrada-salida (I/O), otra distinta se ejecuta.

Esto es importante, dado a que muchas tareas no necesitan bloquear el programa por acciones como:

* Esperar respuestas de APIs
* Leer archivos
* Consultar bases de datos
* Esperar eventos del usuario

Si se ejecutan secuencialmente, se pierde tiempo. Si se ejecutan concurrentemente, se aprovechan mejor los recursos.

|  |  |  |  |
| --- | --- | --- | --- |
| **Herramienta** | **Tipo de ejecución** | **¿Paralelismo real?** | **¿Concurrencia?** |
| await secuencial | Secuencial | ❌ | ❌ |
| asyncio.create\_task() | Concurrente | ❌ | ✅ |
| threading / multiprocessing | Paralelo | ✅ | ✅ |

En este cuadro, podemos observar, que para lograr paralelismo se necesita tener:

* Threading, es decir, múltiples hilos (lo que resulta útil cuando hay muchas operaciones de entrada salida (I/O-bound))
* Multiprocessing, es decir, múltiples procesos (que es útil aplicarlo cuando se presenta CPU-bound : uso intenso de la CPU)

**Otros conceptos asociados**

**Callbacks**

En Python, una **callback** es una [función](https://realpython.com/ref/glossary/function/) que se pasa como [argumento](https://realpython.com/ref/glossary/argument/) a otra función. La función receptora puede invocar la callback posteriormente, a menudo como respuesta a un evento o tras completar una tarea.

Las callbacks son una forma poderosa de personalizar el comportamiento de las funciones y se utilizan comúnmente en [programación asincrónica](https://realpython.com/ref/glossary/asynchronous-programming/) , manejo de eventos (event handling) y aplicaciones GUI.

Los callbacks pueden conducir a código difícil de leer y mantener (*callback hell*).

El módulo asyncio de Python permite manejar asincronía mediante corutinas, evitando la complejidad de las callbacks.

**Asyncio**

asyncio es una biblioteca de Python que proporciona un marco para escribir código asincrónico, usando corutinas (utilizando la sintaxis async/await), facilitando la ejecución concurrente de tareas, sin bloquear el hilo principal. Usa un *event loop* para coordinar múltiples operaciones I/O de forma eficiente.

Mejora la **legibilidad y mantenibilidad** respecto al uso de los callbacks, al mismo tiempo que evita el costo extra de crear muchos hilos en aplicaciones intensivas en entrada-salida (I/O bound).

En resumen:

* **Callbacks** fueron el primer paso hacia la asincronía, pero se vuelven caóticos en proyectos grandes.
* **Asyncio** y las corutinas ofrecen una forma moderna, clara y poderosa de manejar concurrencia sin bloquear el hilo principal.

**Future, task y coroutines**

**Future**

Objeto que representa un resultado que estará disponible en el futuro. Suele ser el resultado de una operación asincrónica (por ejemplo, una consulta a una API).

Es como una caja vacía que promete tener un resultado en el futuro (o un error).

# raramente se instancia un Future directamente con Future(). Lo más habitual es trabajar con asyncio.create\_task()

# o funciones de asyncio (asyncio.sleep, asyncio.gather, etc.) que ya manejan los Future internamente.

**Task**

Task = ejecuta una corutina en el event loop y expone su resultado como un Future

# toda Task es un Future, pero no todo Future es una Task

**Relación Future, Task y Coroutines**

**📌 Definiciones académicas (nivel universitario)**

* **Corutina:  
  Una corutina es una construcción de programación que generaliza las funciones, permitiendo suspender y reanudar su ejecución en puntos específicos. En Python, son objetos retornados por funciones declaradas con async def. Constituyen la unidad básica de trabajo en programación asíncrona.**
* **Future:  
  Un Future es un objeto de sincronización que representa el resultado eventual de una operación asíncrona. Puede estar en estado *pendiente*, *finalizado con éxito* o *finalizado con error*. Permite desacoplar el inicio de la computación de la obtención de su resultado.**
* **Task:  
  Una Task es una subclase de Future que encapsula la ejecución de una corutina dentro de un event loop. Su función es programar la corutina para su ejecución asíncrona, proveer acceso a su estado y resultado, y permitir su cancelación.**

**📌 Diferencias y asociaciones**

* **Corutina → Es el “qué” quiero ejecutar (el código).**
* **Future → Es el “resultado prometido” que aún no tengo.**
* **Task → Es el “cómo” ejecuto la corutina, gestionada por el event loop y asociada a un Future.**

**👉 Entonces:**

* **Una Task necesita una corutina.**
* **Una Task produce un Future que te permite saber cómo terminó esa corutina.**

**📌 Cuándo usar cada una**

* **Corutinas:  
  Cuando necesitás escribir código asíncrono legible y estructurado. Son tu forma natural de “expresar” tareas que pueden esperar (await).**
* **Futures:  
  Normalmente no los usás directamente en código de aplicación, salvo que estés implementando librerías. Son útiles para coordinar resultados, cancelaciones o callbacks.**
* **Tasks:  
  Cuando querés ejecutar corutinas en paralelo dentro del mismo event loop. Con asyncio.create\_task(coro) podés lanzar varias operaciones concurrentes.**

**📌 Cuándo ayudan y cuándo perjudican**

**✅ Ayudan cuando:**

* **Querés eficiencia en operaciones de I/O (red, archivos, bases de datos).**
* **Necesitás correr varias tareas “a la vez” sin bloquear el programa.**
* **Buscás un código más claro que usar callbacks anidados.**

**❌ Perjudican cuando:**

* **Los usás para cálculos muy pesados en CPU (el event loop se bloquea).**
* **Mezclás código síncrono y asíncrono sin entender bien cómo se coordinan.**
* **Lanzas muchas Tasks sin control → puede saturar memoria y dificultar el debugging.**

**📌 Algo extra que deberías saber**

* **Todo esto vive dentro del event loop, que es el motor que agenda y ejecuta corutinas/tareas, y resuelve futures.**
* **Python asyncio ofrece herramientas como gather, wait, as\_completed para coordinar múltiples tareas.**
* **Un error común es pensar que “asíncrono = paralelo”. No siempre: podés tener asíncrono (concurrente en un solo thread) sin ser paralelo (varios hilos/núcleos).**

**👉 Lourdes, ¿querés que te arme un esquema visual comparativo (tipo tabla o diagrama) que muestre corutinas, futures y tasks en una sola vista para que quede grabado?**

**Dijiste:**

**Diferencia entre task y future**

**ChatGPT dijo:**

**Muy buena pregunta Lourdes 🙌, porque ahí suele estar la mayor confusión.**

**📌 Diferencia clave entre Task y Future**

* **Future:**
  + **Es solo una promesa de resultado (o error) que estará disponible en el futuro.**
  + **No ejecuta nada por sí mismo.**
  + **Podés pensar en él como una caja vacía a la espera de que alguien (el event loop u otro código) ponga un valor.**
* **Task:**
  + **Es un Future especializado que, además, ejecuta una corutina en el event loop.**
  + **Cuando la corutina termina, la Task guarda su resultado dentro del Future que hereda.**
  + **Es la forma de “arrancar” una corutina y que corra en paralelo con otras.**

**📌 Asociación**

* **Toda Task es un Future, pero no todo Future es una Task.**
* **Podés crear un Future manualmente y llenarlo con un resultado más tarde.**
* **Una Task, en cambio, se crea a partir de una corutina y automáticamente se encarga de ejecutarla.**

**📌 Ejemplo en Python**

**import asyncio**

**async def trabajo():**

**await asyncio.sleep(1)**

**return "hecho"**

**async def main():**

**# Future creado manualmente**

**loop = asyncio.get\_running\_loop()**

**fut = loop.create\_future()**

**fut.set\_result("resultado manual")**

**print("Future:", fut.result())**

**# Task creada a partir de una corutina**

**task = asyncio.create\_task(trabajo())**

**print("Task:", await task)**

**asyncio.run(main())**

**👉 El Future (fut) no ejecutó nada: simplemente le pusimos un valor.  
👉 La Task (task) sí ejecutó la corutina trabajo().**

**📌 En resumen**

* **Future: recipiente para un resultado.**
* **Task: recipiente + motor que ejecuta una corutina y guarda el resultado en su Future interno.**

**¿Querés que te arme un esquema gráfico comparativo (Future vs Task vs Coroutine) para que te quede visualmente más claro?**

Conclusión final

(relacionando todo)

En corutinas: Las corutinas permiten suspender y reanudar funciones, lo que facilita la concurrencia. Aunque solo haya un hilo, se puede cambiar de una tarea a otra sin bloquear el hilo principal.

En corutinas: Aunque las corutinas por sí mismas no implican paralelismo, pueden ser combinadas con múltiples hilos o procesos para lograrlo. Por ejemplo, lanzar varias corutinas en distintos hilos,

En el marco de asincronía y corutinas, asincronía se refiere a que una tarea no bloquea el flujo principal. Por ejemplo, una llamada a una API puede ser asíncrona, permitiendo que el programa continúe ejecutando otras instrucciones.

Si además se tuvieran varias tareas corriendo en distintos núcleos, se lograría paralelismo.

**Links de interés:**

<https://www.pythontutorial.net/python-concurrency/python-asyncio-future/>

<https://www.luisllamas.es/que-es-una-clase-y-una-instancia-programacion/>

<https://programacion.top/orientada-objetos/clase/>

<https://docs.python.org/es/3.8/library/asyncio-task.html>

<https://docs.python.org/3/library/asyncio-task.html>

<https://realpython.com/async-io-python/>

<https://realpython.com/ref/glossary/coroutine/>

<https://realpython.com/ref/glossary/coroutine-function/>

<https://realpython.com/python-concurrency/>

<https://www.pythontutorial.net/python-concurrency/python-asyncio-create_task/>