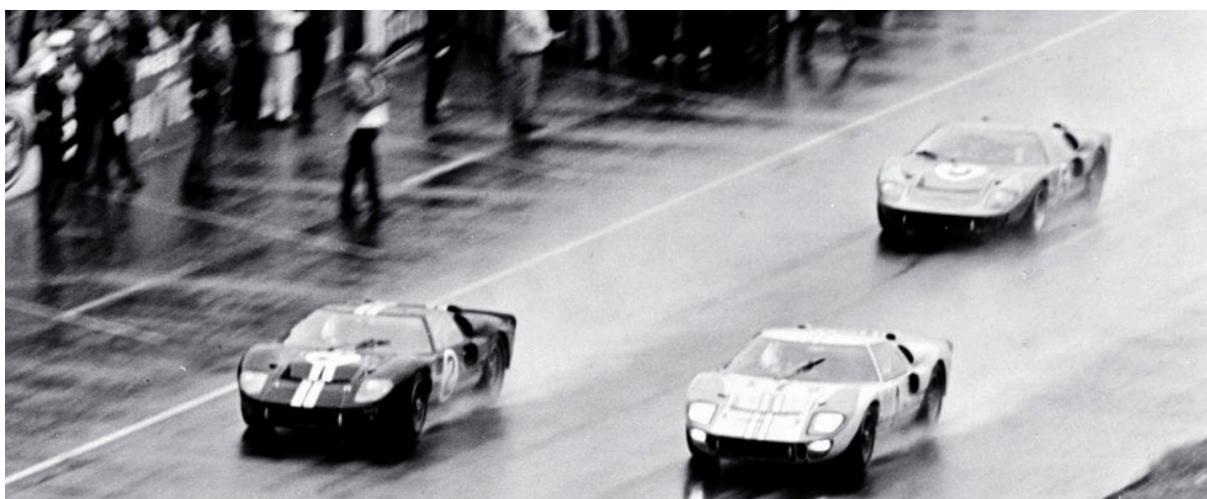




Python | Optimizando su velocidad

Ahora que conocemos Python, sabemos que es uno de los lenguajes más flexibles y entendibles que hay, lo que lo hace una herramienta muy querida para desarrolladores de muy diversos ámbitos y de campos tan distintos!

Desafíos que encuentra Python



Sin embargo, esta gran ventaja de Python que lo ha elevado a los mejores rankings de lenguajes, también le hace proclive a debilidades. En concreto, encontramos los

siguientes desafíos común a muchos otros lenguajes pero que en Python se entienden muy bien:

- 1. **Interpretación Lineal:** Python es un lenguaje interpretado, lo que significa que el código se ejecuta línea por línea por el intérprete de Python. Esto puede hacer que Python sea más lento en comparación con lenguajes compilados como C o C++.
- 2.  **Tipado Dinámico:** Python es tipado dinámicamente, lo que significa que los tipos de variables se determinan en tiempo de ejecución. Este tipado dinámico puede generar sobrecarga y rendimiento más lento en comparación con lenguajes tipados estáticamente.
 - Esto puede llevar a errores de tiempo de ejecución que podrían haberse detectado y corregido durante la fase de compilación en lenguajes con tipado estático.
 - El tipado dinámico puede requerir más recursos de computación durante la ejecución para realizar conversiones de tipo y verificaciones dinámicas.
 - Los compiladores y optimizadores pueden tener dificultades para realizar optimizaciones avanzadas en código Python debido al tipado dinámico, lo que podría limitar el rendimiento en comparación con lenguajes con tipado estático.
- 3.  **Bloqueo Global del Intérprete (GIL):** El propósito principal del GIL es garantizar que solo un hilo pueda ejecutar código de Python a la vez en un proceso de Python. En otras palabras, el trabajo de **GIL** es impedir que múltiples threads decrementen la referencia de algún objeto mientras otros están haciendo uso de ella. El GIL permite que espacios en memoria sean liberados cuando aún están siendo utilizados

¿Qué es el GIL?



El GIL (Global Interpreter Lock) es una característica específica de la implementación estándar de CPython, que es el intérprete de Python más comúnmente utilizado. Este bloqueo global del intérprete tiene un impacto importante en la ejecución de código en entornos multi-hilo.

El GIL simplifica significativamente la implementación del intérprete de Python, ya que evita la necesidad de coordinar la ejecución de código entre múltiples hilos para garantizar la coherencia de los datos y la seguridad de la memoria compartida. Sin embargo, también tiene implicaciones en términos de rendimiento y escalabilidad en ciertos tipos de aplicaciones.

Es importante destacar que el GIL solo afecta la ejecución de código Python. Las operaciones de E/S (entrada/salida), como lectura/escritura de archivos o solicitudes de red, no se ven tan afectadas porque liberan el GIL mientras esperan la finalización de la operación de E/S. En cambio, en aplicaciones intensivas en CPU, donde se desea aprovechar al máximo los múltiples núcleos de la CPU, el GIL puede convertirse en un obstáculo.

De hecho, **Guido van Rossum** creador del lenguaje también conocido como el **BDFL** (*Benevolent Dictator For Life*) ha especificado en varios correos en la lista de desarrolladores de Python que el GIL está aquí **para quedarse**. Pueden leerse sus argumentaciones en este [hilo](#) y en [este otro](#) donde explica que eliminar el GIL **no es tan sencillo** y anima a la Comunidad de desarrolladores a mantener un *fork* sin GIL si alguien se anima.

Veamos un ejemplo del efecto sobre un script



Como hemos dicho ya, con Python al utilizar Threads nunca seremos capaces de lograr un verdadero paralelismo, ya que el lenguaje está diseñado para que un thread y solo un thread pueda ejecutarse a la vez.

Veamos un ejemplo. En este caso he definido una función que lo único que hace es decrecer su parametro(number) hasta llegar a 0.

```
import time

def countdown(number):
    while number > 0:
        number -=1

if __name__ == '__main__':
    start = time.time()

    count = 100000000
    countdown(count)
```

```
print(f'Tiempo transcurrido {time.time() - start }')
```

Si ejecuto este script, tal y como se encuentra ahora, con un solo thread y de forma secuencial, le tomará un aproximado de 6 segundos finalizar. Todo bien, nada nuevo.

Ahora, hagamos los mismo pero con dos threads. En teoría, al los threads ejecutarse de forma concurrente al script le debería tomar poco más de 6 segundos finalizar, quizás, unos milisegundos más, en teoria. 😬

```
import time
import threading

def countdown(number):
    while number > 0:
        number -=1

if __name__ == '__main__':
    start = time.time()

    count = 100000000

    t1 = threading.Thread(target=countdown, args=(count,))
    t2 = threading.Thread(target=countdown, args=(count,))

    t1.start()
    t2.start()

    t1.join()
    t2.join()

print(f'Tiempo transcurrido {time.time() - start }')
```

A mi escript, de forma concurrente le toma 12 segundos finalizar, sí, el doble de tiempo a pesar de utilizar threads. 😬 Esto gracias al **GIL**.

Estrategias de Optimización:



1. Multihilos (Multithreading) ⭐:

- Utilizar hilos para operaciones de E/S o tareas que liberan el GIL. Ideal cuando se necesita concurrencia sin depender totalmente del rendimiento de la CPU.

2. Multiprocesos (Multiprocessing) 💥:

- Emplear procesos independientes para aprovechar al máximo los múltiples núcleos de la CPU. Beneficioso en escenarios donde la paralelización es clave para optimizar el rendimiento.

3. Programación Asincrónica (Asyncio) 🚀:

- Adoptar la programación asincrónica para gestionar eficientemente operaciones de E/S y mejorar la concurrencia, ideal para aplicaciones altamente concurrentes.

Estas estrategias, aplicadas con discernimiento, permiten a los desarrolladores abordar eficazmente los desafíos de rendimiento en Python, garantizando una ejecución ágil y eficiente de las aplicaciones. Con la combinación adecuada de estas herramientas, Python se convierte en una opción aún más poderosa para la creación de software de alto rendimiento.

1 | Multihilos > concurrencia



Lo voy a explicar con un ejemplo: imaginémonos a una persona la cual trabaja en múltiples tareas al mismo tiempo, y que rápidamente cambia de una tarea a otra. Por ejemplo, imaginemos a una persona la cual se encuentra programando, realizando cálculos en excel y contestando correos electrónicos, todo esto al mismo tiempo. Dedica un par de segundos a cada tarea, y rápidamente, con un ágil cmd + shift cambia de tarea.

Concluimos que la persona trabaja de forma concurrente. Las tareas que realiza no necesariamente deben seguir un orden, quizás, después de contestar un correo regresa con los cálculos en excel, le dedica un par de segundo, regresa a responder otro correo y finaliza con la codificación del programa, u, otro escenario pudiera ser que después finalizar ciertos cálculos, la persona continua codeando un par de segundos para después responder un par de correos y regresar con los cálculos.

Al contrario que una estructura secuencial, con la concurrencia el orden en que se ejecutan las tareas importa muy poco.

En código:

```

import time
import threading

def codificar():
    time.sleep(2)

    print(f'Codificando')

def responder_correos():
    time.sleep(2)

    print(f'Respondiendo correos')

def realizar_calculos():
    time.sleep(2)

    print(f'Realizar los calculos')

threading.Thread(target=codificar).start()
threading.Thread(target=responder_correos).start()
threading.Thread(target=realizar_calculos).start()

```

En este caso nuestro programa realiza tres tareas al mismo tiempo. A cada tarea le tomó un máximo de dos segundos ser completada, como se ejecutan de forma concurrente (al mismo tiempo) al programa le toma dos segundo finalizar su ejecución. Por otro lado, si ejecutamos el mismo código, pero ahora de forma secuencial, al programa le tomaría seis segundos finalizar. El tiempo es sin duda considerable.

```

#secuencial

codificar()
responder_correos()
realizar_calculos()

```

Por medio de los multihilos un procesador puede ejecutar diversos hilos de forma concurrente. En un simple CPU esto se consigue activando y desactivando hilos

(context switching). En context switching, el estado de un hilo se guarda en cuanto se interrumpe, y se carga otro hilo. Context switching es tan frecuente que parece que diversos hilos se ejecutan de forma paralela.

2 | Multiprocesos > paralelismo



Bien, ya tenemos claro que es la estructura secuencial y que es la concurrencia, ahora, ¿De qué va el paralelismo? El paralelismo es el poder de ejecutar dos o más acciones de forma simultánea, en lugar de concurrentemente. Si recordamos, en nuestro ejemplo anterior, el desarrollador realiza tres tareas al mismo tiempo, realizaba cálculos programaba y contestaba correos, sin embargo, ninguna de estas tareas se realizaba de forma simultánea.

En nuestro ejemplo, si queremos que las tareas se realicen de forma paralela tendríamos que tener a tres personas trabajando, vaya, una persona por cada tarea, una persona encargada de los cálculos, otra de la codificación y otra respondiendo correos, tres personas trabajando simultáneamente.

Si realmente queremos que las tareas se ejecuten de forma paralela debemos optar por el multiprocesamiento sobre el multithreading; de esta forma cada proceso tendrá su propio intérprete y podrá ejecutarse de manera independiente, logrando así evitar el cuello de botella de **GIL** y aprovechando todo el potencial de nuestros equipos.

La mejor forma de trabajar procesos en Python es sin duda con la librería **multiprocessing**.

Aquí un pequeño ejemplo de como crear nuestros propios Procesos. Haz la prueba por ti mismo y verás el resultado.

```
import time
import multiprocessing

def countdown(number):
    while number > 0:
        number -=1

if __name__ == '__main__':
    start = time.time()

    count = 100000000

    t1 = multiprocessing.Process(target=countdown, args=(count,))
    t2 = multiprocessing.Process(target=countdown, args=(count,))

    t1.start()
    t2.start()

    t1.join()
    t2.join()

    print(f'Tiempo transcurrido {time.time() - start }')
```

3 | Programación asíncrona



 motorsport
IMAGES

La programación asíncrona no tiene nada que ver con distintos hilos o procesos... En este tipo de programación vamos a tener un solo hilo y un solo proceso, pero seremos capaces de hacer varias cosas a la vez. ¿Entonces, cuál es el truco?

Veamos un ejemplo 

Imagina que eres el camarero en un restaurante muy concurrido. Tienes que tomar las órdenes de los clientes, entregar la comida y asegurarte de que todos estén satisfechos. Pero aquí está el truco: solo puedes atender a un cliente a la vez. Si alguien te pide unas papas fritas, no puedes atender a otros clientes hasta que entregues esas papas fritas. ¡Sería un caos y muy lento!

Aquí es donde entra en juego la **programación asíncrona**. Vamos a desglosar los conceptos:

Instrucción bloqueante :

- Imagina que un cliente te pide un plato de pasta, pero el cocinero aún no ha terminado de prepararlo. Tú, como camarero, no puedes hacer nada más hasta que el cocinero te entregue el plato. Esto es una acción bloqueante.
- En programación, una instrucción bloqueante sería como esperar a que se complete una operación antes de continuar con la siguiente.

Instrucción no bloqueante :

- Ahora, en lugar de quedarte parado esperando, decides ser más eficiente. Cuando un cliente te pide una pizza, le entregas la orden al chef y él te dice: "Puedes atender a otros clientes mientras preparo la pizza". ¡Genial!
- En programación, una instrucción no bloqueante permite que el programa continúe ejecutando otras tareas mientras espera que algo suceda en segundo plano.

Ejemplo en Python

- Supongamos que tienes una aplicación web que consulta una base de datos para mostrar información al usuario. Si hicieramos esto de manera bloqueante, la aplicación se quedaría esperando a que la base de datos responda antes de mostrar cualquier otra cosa.
- Con la programación asíncrona, puedes enviar la consulta a la base de datos y mientras esperas su respuesta, atender otras solicitudes de usuarios. Cuando la base de datos responda, vuelves a la consulta original y muestras los resultados. ¡Eficiente y rápido!

En resumen, la programación asíncrona no se trata solo de optimizar recursos del CPU, sino de **optimizar el tiempo de espera**. Permite que las tareas se ejecuten de manera no bloqueante, como un camarero que atiende a varios clientes mientras espera que algo suceda en segundo plano. ¡Así logramos ser camareros eficientes en el mundo de la programación! 🍔🚀

En definitiva

Imagina que tienes varios clientes en el restaurante:

- Cliente A quiere una hamburguesa (tarea 1).
- Cliente B pide una ensalada (tarea 2).
- Cliente C desea una sopa (tarea 3).

1. Concurrencia (Multihilos):

- Implica **múltiples hilos** de ejecución dentro de un proceso.
- Los hilos comparten recursos como memoria.
- Ejemplo: Varios camareros atienden a diferentes clientes al mismo tiempo.

2. Paralelización (Multiprocesos):

- Implica **múltiples procesos** independientes que se ejecutan simultáneamente.
- Cada proceso tiene su **propia memoria** y recursos.
- Ejemplo: Varios camareros trabajan en diferentes mesas sin compartir platos ni recursos.

3. Programación Asíncrona:

- Se centra en **optimizar el tiempo de espera** y permitir que el programa realice otras tareas mientras espera que ocurra algo en segundo plano.
- Utiliza palabras clave como `async` y `await` en Python.
- Ejemplo: Un camarero toma más pedidos mientras el chef cocina.

Vuestro turno



Haced grupos de dos:

- Multihilos en Python

- Relación entre procesos y sus hilos
- Explicación sobre Context Switching
- Librería como `threading` en Python.
 - Cómo importarlo
 - Cómo crear un hilo. Qué parámetros necesita tomar como argumentos de entrada
 - Cómo empezar un hilo. ¿Qué método se necesita?
 - ¿Hay otros métodos?
 - ¿Cómo finalizar un hilo?
 - Ejemplo

Glosario

- **Programa** : Un archivo ejecutable que contiene un conjunto de instrucciones para realizar una tarea específica.
 - En otras palabras, es como una receta de cocina.
- **Proceso** : Es un programa que se ha cargado en la memoria junto con todas las dependencias y ejecutables necesarios para funcionar. Cada proceso tiene su propio espacio de memoria.
 - En otras palabras, es como un chef en la cocina, con todas sus cazuelas, su sombrero de cocinero, y sus especias. Incluso con su radio para cocinar tranquilo.
- **Hilo** : Los hilos son unidades de ejecución dentro de un proceso. Pueden trabajar en paralelo, compartiendo el espacio de memoria del proceso.
 - Los hilos son como los ayudantes del chef. Imagina que un chef tiene varios ayudantes, cada uno ocupado en una tarea diferente. Por ejemplo, un hilo puede estar preparando la salsa mientras otro corta las verduras.
- **Multihilo** : Varios hilos son creados por un proceso para realizar diferentes tareas, aproximadamente al mismo tiempo. Aunque no se ejecutan verdaderamente en paralelo debido al **Bloqueo Global del Intérprete (GIL)** en Python, dan la ilusión de que lo hacen.

- Esta técnica es como tener varios chefs trabajando juntos en la misma cocina. Es como si los chefs estuvieran cocinando en una cocina muy bien coordinada.
- **Multiproceso** : La paralelización es real. Múltiples procesos se ejecutan en diferentes núcleos de CPU, sin compartir recursos entre sí. Cada proceso puede tener muchos hilos trabajando en su propio espacio de memoria.
 - Aquí sí tenemos varios chefs en cocinas separadas. Es como si tuvieras varios restaurantes, cada uno con su propio equipo de cocina.

En resumen, la programación asíncrona, la concurrencia y la paralelización son como las diferentes formas de organizar la cocina para preparar una gran comida. Cada una, tiene sus ventajas e inconvenientes. 