Multithreading in Python

El multithreading en Python permite la ejecución concurrente de tareas dentro de un único proceso. Esto es útil para operaciones que pueden realizarse en paralelo, como la E/S de archivos o la descarga de datos, pero no siempre mejora el rendimiento en tareas pesadas debido al GIL (Global Interpreter Lock) en CPython.

1. Creación de hilos con la clase Thread

```
import threading

def tarea():
    print("Tarea en ejecución")

# Crear un hilo
hilo = threading.Thread(target=tarea)

# Iniciar el hilo
hilo.start()

# Esperar a que el hilo termine
hilo.join()
```

En este ejemplo, se crea un hilo que ejecuta la función tarea(). El método start() inicia el hilo y join() espera a que termine.

2. Creación de hilos utilizando clases

```
import threading

class MiHilo(threading.Thread):
    def run(self):
        print("Tarea del hilo en ejecución")
```

```
# Crear e iniciar el hilo
hilo = MiHilo()
hilo.start()
hilo.join()
```

Aquí, se define una clase MiHilo que hereda de threading.Thread y sobreescribe el método run() con la tarea a ejecutar en el hilo.

3. Ejecución de múltiples hilos

```
import threading

def tarea(num):
    print(f"Hilo {num} en ejecución")

# Crear e iniciar varios hilos
hilos = []
for i in range(5):
    hilo = threading.Thread(target=tarea, args=(i,))
    hilos.append(hilo)
    hilo.start()

# Esperar a que todos los hilos terminen
for hilo in hilos:
    hilo.join()
```

Este código inicia múltiples hilos, cada uno ejecutando la función tarea() con un número de identificación diferente.

4. Bloqueo de hilos (threading.Lock)

```
import threading

contador = 0
bloqueo = threading.Lock()

def incrementar():
    global contador
    with bloqueo:
```

```
        local_contador = contador
        local_contador += 1
        contador = local_contador

# Crear e iniciar varios hilos
hilos = []
for _ in range(10):
    hilo = threading.Thread(target=incrementar)
    hilos.append(hilo)
    hilo.start()

# Esperar a que todos los hilos terminen
for hilo in hilos:
    hilo.join()

print(f"Valor final del contador: {contador}")
```

El uso de threading.Lock garantiza que solo un hilo acceda a una sección crítica del código a la vez, evitando condiciones de carrera.

5. Uso de threading.RLock

```
import threading

bloqueo = threading.RLock()

def tarea1():
    with bloqueo:
        print("Tarea 1 adquirió el bloqueo")
        tarea2()

def tarea2():
    with bloqueo:
        print("Tarea 2 adquirió el bloqueo")

# Crear e iniciar el hilo
hilo = threading.Thread(target=tarea1)
hilo.start()
hilo.join()
```

threading.RLock permite que un hilo adquiera un bloqueo que ya posee, lo cual es útil para evitar interbloqueos.

El multithreading es ideal para aplicaciones de E/S intensiva. Para CPU-bound tasks, se recomienda usar multiprocessing para aprovechar múltiples núcleos.