Multi-threading

Hvad er Multi-threading?

Multi-threading i Python er en teknik, der muliggør udførelse af flere opgaver parallelt ved at oprette flere tråde, der kører samtidig inden for et enkelt program. Dette er særligt nyttigt, når der arbejdes med I/O-tunge opgaver som netværksanmodninger eller filhåndtering, da det reducerer ventetid og forbedrer ydeevnen ved at udføre flere opgaver parallelt.

I et multi-threading miljø kan hver tråd køre uafhængigt af hinanden, men de deler samme hukommelse, hvilket kan føre til problemer såsom "race conditions", hvor flere tråde forsøger at skrive til den samme variabel på samme tid. Python's Global Interpreter Lock (GIL) forhindrer dog visse race conditions, men begrænser også ægte parallel udførelse af CPU-bundne opgaver.

Direkte brug af Multi-threading

For at oprette og arbejde med tråde i Python kan du bruge threading-modulet, der gør det muligt at oprette nye tråde, hvor hver tråd udfører en specifik opgave parallelt. Et simpelt eksempel ser sådan ud:

```python
import threading
import time
import random

def task(arg):
    print(f"Starting task with arg: {arg}")
    # Simulerer en arbejdsopgave
    time.sleep(random.randint(1,5))
    print(f"Completed task")

# Opret flere tråde
threads = []
args = ["argument 1", "argument 2"]

for args in args:
    thread = threading.Thread(target=task, args=(args,))
    threads.append(thread)
    thread.start()

# Vent på at alle tråde er færdige
for thread in threads:
    thread.join()
```

Her oprettes en tråd for hver arbejdsopgave der skal udføres, og vi bruger thread.join() til at sikre, at hovedtråden venter, indtil alle opgaver er færdige.

Custom Thread Pools via Queue

En anden elegant måde at håndtere threading på, er at lave en såkaldt "thread safe queue" af opgaver. Ideen er at man kan oprette x antal tråde og aktivt give dem opgaver ved at lægge dem i køen. Tråden kan så være lavet til at håndtere forskellige tasks i køen, eller man kan designe i kø for hver specifikke task. F.eks kan køen indeholde logbeskeder hvor trådende selv kan håndtere efter log type. Køen er trådsikker på den måde, at hvis en tråd tager en task fra den, så bliver den låst fra andre tråde, og tråden kan kun tage en anden besked, når den laver at tasken er done. Det betyder derfor, at du f.eks. instantierer tre tråde hvor deres argument er køen. Disse tre tråde bliver derfor ikke brugt til andet end at håndtere opgaver fra køen, og kan leve for evigt eller til at køen er tom (mere info herunder:)

Et simpelt script der laver tre tråde og bruger queues kan se således ud:

```
from queue import Queue
import threading



def task(queue: Queue) -> None:
    while True:
        message = queue.get()
        print(message)
        queue.task_done()



queue = Queue()
num_threads: int = 3
pool = []

for i in range(num_threads):
    worker = threading.Thread(target=task, args=(queue,))
    worker.start()
    pool.append(worker)

queue.put("hello world")
```

Læg mærke til at task har "`while True:`" dette sørger for at tråden lever for evigt. Man kan også bruge "`while not queue.empty():`" eller "`while queue:`" hvis man bare skal tømme køen. Man kan også sætte en timer til at få den til at leve i x antal minutter etc.

Thread Pools

Thread pools er en effektiv måde at administrere tråde på, da de genbruger eksisterende tråde fremfor at oprette nye tråde for hver opgave, hvilket kan reducere overheaden. Dette er ideelt, når du har mange små opgaver, der kan køre parallelt.

Python giver dig mulighed for at bruge concurrent.futures.ThreadPoolExecutor til at implementere en thread pool:

```python
from concurrent.futures import ThreadPoolExecutor
import time
import random

def task(arg):
    print(f"Starting task with arg: {arg}")
    # Simulerer en arbejdsopgave
    time.sleep(random.randint(1, 5))
    print(f"Completed task")

args = ["argument 1", "argument 2"]

with ThreadPoolExecutor() as executor:
    executor.map(task, args)
```

I dette eksempel bliver ThreadPoolExecutor brugt til at oprette en pulje af tråde, der kan håndtere flere opgaver på én gang. Det reducerer belastningen ved at administrere trådenes livscyklus, og er især effektivt når du arbejder med mange samtidige opgaver.

Asyncio

Asyncio er et bibliotek i Python, der giver dig mulighed for at skrive asynkron ikke-blokerende kode, hvilket er nyttigt, når du arbejder med I/O-bundne opgaver. I stedet for at oprette flere tråde, arbejder asyncio ved at lade programmet håndtere flere opgaver ved hjælp af såkaldt "cooperative multitasking". Dette betyder at asyncio reducerer overhead for flere tråde og kontekstskift.

Flow of Control med await

Når du bruger async- og await-syntaksen i asyncio, fortæller du Python, at programmet skal vente på en asynkron funktion, før det fortsætter med eksekveringen. Forskellen fra multi-threading er, at mens programmet venter, kan tråden arbejde på andre opgaver, hvilket gør det muligt at håndtere mange opgaver uden at blokere.

Et simpelt eksempel med asyncio kunne se sådan ud:

```python
import asyncio
Import random

async def task(arg):
    print(f" Starting task with arg: {arg}")
    # Simulerer asynkron arbejdsopgave
    await asyncio.sleep(random.randint(1, 5))
    print(f"Completed task")

async def main():
    args = ["argument 1", "argument 2"]
    tasks = [task(arg) for arg in args]
    await asyncio.gather(*tasks)

asyncio.run(main())
```

I dette eksempel bruges await til at indikere, at funktionen skal vente på, at asyncio.sleep() er fuldført, uden at blokere resten af programmet. Dette tillader, at flere opgaver kan startes og fuldføres parallelt.

Kilder og Yderligere Information

For mere information kan følgende ressourcer benyttes:

1. **Thread**
   https://docs.python.org/3.12/library/threading.html#threading.Thread
2. **ThreadPool**
   https://docs.python.org/3.12/library/concurrent.futures.html#processpoolexecutor
3. **Asyncio**
   https://docs.python.org/3.12/library/asyncio.html
4. **Concurrent futures**
   https://docs.python.org/3/library/concurrent.futures.html