Pertemuan-6

Concurency

A. Concurency

Konsep Concurrency dalam OOP Python merujuk pada kemampuan program untuk mengeksekusi beberapa tugas secara bersamaan (atau serentak).

Dalam bahasa pemrograman Python, terdapat beberapa mekanisme yang mendukung concurrency, seperti :

1. Threading
2. Multiprocessing

1. Threading

Threading dalam Python memungkinkan program untuk mengeksekusi beberapa tugas secara bersamaan dalam satu thread. Thread adalah unit pemrosesan kecil yang dijalankan oleh sebuah program. Dalam OOP Python, konsep threading dapat diimplementasikan dengan membuat objek dari kelas Thread yang ada di dalam modul threading. Thread dapat dijalankan secara independen dan saling berbagi sumber daya dengan thread lain dalam program.

Berikut adalah contoh sederhana penggunaan threading dalam OOP Python:

```
import threading

class MyThread(threading.Thread):
    def __init__(self, name):
        threading.Thread.__init__(self)
        self.name = name

    def run(self):
        print("Thread {} is running".format(self.name))

t1 = MyThread("1")
t2 = MyThread("2")

t1.start()
t2.start()
```

Pada contoh di atas, kita membuat kelas MyThread yang merupakan turunan dari kelas Thread. Konstruktor kelas MyThread digunakan untuk menginisialisasi nama thread. Kemudian, kita mendefinisikan metode run yang akan dijalankan ketika thread dimulai. Pada metode run, kita hanya mencetak pesan "Thread <nama thread> is running".

Setelah kelas MyThread selesai didefinisikan, kita membuat objek t1 dan t2 dari kelas MyThread. Objek-objek ini kemudian dijalankan menggunakan metode start(). Kedua thread

akan berjalan secara bersamaan dan mencetak pesan "Thread 1 is running" dan "Thread 2 is running".

2. Multiprocessing

Multiprosesing dapat berguna dalam situasi di mana Anda memiliki tugas yang terikat CPU yang dapat dipecah menjadi bagian independen yang dapat dijalankan secara paralel.

Berikut adalah beberapa modul yang dapat digunakan dalam multiprocessing di Python:

1. **multiprocessing**: Modul bawaan Python yang menyediakan API untuk melakukan multiprocessing dengan memanfaatkan process-based “threading” yang memanfaatkan CPU yang tersedia secara optimal.
   Contoh:

```
import multiprocessing

def worker(num):
    """Fungsi untuk menjalankan tugas pada proses baru"""
    print('Worker', num)

if __name__ == '__main__':
    # Membuat objek Process sebanyak 5
    processes = []
    for i in range(5):
        p = multiprocessing.Process(target=worker, args=(i,))
        processes.append(p)
        p.start()

    # Menunggu proses selesai
    for p in processes:
        p.join()
```

   Pada contoh di atas, kita membuat fungsi worker yang akan dieksekusi pada setiap proses baru yang dibuat. Kemudian, kita membuat objek Process sebanyak 5 dan menambahkannya ke dalam list processes. Setiap objek Process akan memanggil fungsi worker dengan argument berupa nomor proses.

   Selanjutnya, kita menjalankan setiap proses dengan memanggil metode start pada objek Process. Setelah semua proses selesai, kita menunggu setiap proses selesai dengan memanggil metode join pada setiap objek Process.

   Dalam hal ini, kita menggunakan if __name__ == '__main__': untuk memastikan bahwa kode di dalam blok tersebut hanya dijalankan ketika file ini dijalankan sebagai program utama, bukan sebagai modul yang diimpor oleh program lain. Hal ini penting untuk menghindari konflik ketika menjalankan multiprocessing di lingkungan Windows.

2. **concurrent.futures**: Modul yang menyediakan antarmuka yang lebih sederhana untuk melakukan multiprocessing di Python. Modul ini dapat menggunakan multiprocessing maupun multithreading.
   Contoh:

```
import concurrent.futures

def worker(num):
    """Fungsi untuk menjalankan tugas pada proses baru"""
    print('Worker', num)

if __name__ == '__main__':
    # Membuat objek ThreadPoolExecutor
    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        # Memanggil fungsi worker sebanyak 5 kali
        for i in range(5):
            executor.submit(worker, i)
```

   Pada contoh di atas, kita membuat fungsi worker yang akan dieksekusi pada setiap proses baru yang dibuat. Kemudian, kita membuat objek ThreadPoolExecutor dengan maksimum worker sebanyak 5. Objek ThreadPoolExecutor akan menjalankan setiap tugas pada worker yang tersedia secara paralel.

   Selanjutnya, kita memanggil fungsi worker sebanyak 5 kali dengan memanggil metode submit pada objek ThreadPoolExecutor. Metode submit akan mengirimkan tugas ke worker yang tersedia dan mengembalikan objek Future yang dapat digunakan untuk memantau status tugas.

   Dalam hal ini, kita tidak perlu menggunakan if __name__ == '__main__': karena modul concurrent.futures sudah mengatasi masalah konflik ketika menjalankan multiprocessing di lingkungan Windows.

3. **subprocess**: Modul bawaan Python yang menyediakan fungsi untuk membuat proses baru, dan memungkinkan komunikasi antar proses.
   Contoh:

```
import subprocess

def worker(num):
    """Fungsi untuk menjalankan tugas pada proses baru"""
    print('Worker', num)

if __name__ == '__main__':
    # Membuat list untuk menyimpan setiap proses
    processes = []
    # Memulai setiap proses dengan menggunakan subprocess.Popen
```

```
for i in range(5):
    p = subprocess.Popen(['python', '-c',
         'import multiprocessing; '
         'multiprocessing.Process(
        target=worker,args=('+str(i)+',)).start()'],
    stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    processes.append(p)

# Menunggu proses selesai dan mendapatkan output dari setiap proses
for p in processes:
    out, err = p.communicate()
    print(out.decode('utf-8'), err.decode('utf-8'))
```

Pada contoh di atas, kita membuat fungsi worker yang akan dieksekusi pada setiap proses baru yang dibuat. Kemudian, kita membuat list processes untuk menyimpan setiap proses yang dibuat.

Selanjutnya, kita memulai setiap proses dengan menggunakan subprocess.Popen. Fungsi Popen akan memulai proses baru dengan mengirimkan perintah python -c dan kode Python yang akan dieksekusi pada setiap proses. Kode Python tersebut akan memanggil fungsi worker dengan nomor proses yang diberikan sebagai argument.

Setelah semua proses selesai, kita menunggu setiap proses selesai dengan memanggil metode communicate pada setiap objek Popen. Metode communicate akan mengembalikan output dari proses dan kode error jika terdapat error pada proses tersebut.

Dalam hal ini, kita tidak perlu menggunakan if __name__ == '__main__': karena kita menggunakan subprocess.Popen untuk memulai setiap proses, bukan multiprocessing.Process.

4. **threading**: Modul bawaan Python yang menyediakan API untuk melakukan multithreading. Modul ini dapat digunakan bersama dengan multiprocessing untuk melakukan parallelism pada level thread.

   Modul threading pada Python tidak sepenuhnya mendukung multiprocessing karena GIL (Global Interpreter Lock) membatasi hanya satu thread yang bisa mengeksekusi kode Python pada satu waktu tertentu. Oleh karena itu, penggunaan modul threading pada Python lebih disarankan untuk keperluan concurrency daripada multiprocessing.

   Namun, kita masih bisa melakukan multiprocessing dengan menggunakan modul threading dengan memanfaatkan submodul threading.Thread. Berikut ini adalah contoh penggunaan modul threading untuk melakukan multiprocessing pada Python:

```
import threading

def worker(num):
    """Fungsi untuk menjalankan tugas pada thread baru"""
    print('Worker', num)

if __name__ == '__main__':
    # Membuat list untuk menyimpan setiap thread
    threads = []
    # Memulai setiap thread dengan menggunakan threading.Thread
    for i in range(5):
        t = threading.Thread(target=worker, args=(i,))
        t.start()
        threads.append(t)

    # Menunggu thread selesai
    for t in threads:
        t.join()
```

Pada contoh di atas, kita membuat fungsi worker yang akan dieksekusi pada setiap thread baru yang dibuat. Kemudian, kita membuat list threads untuk menyimpan setiap thread yang dibuat.

Selanjutnya, kita memulai setiap thread dengan menggunakan threading.Thread. Fungsi Thread akan membuat thread baru dan mengembalikan objek thread. Objek thread tersebut kemudian dieksekusi dengan memanggil metode start.

Setelah semua thread selesai, kita menunggu setiap thread selesai dengan memanggil metode join pada setiap objek thread di list threads.

Dalam hal ini, kita perlu menggunakan if __name__ == '__main__': karena penggunaan submodul threading.Thread masih mengandalkan GIL dan tidak menjamin multiprocessing.

5. **mpi4py**: Modul yang menyediakan antarmuka untuk mengakses Message Passing Interface (MPI) dari Python. MPI adalah standar de facto untuk parallel computing pada sistem yang terdiri dari banyak prosesor.
   Contoh:

```
from mpi4py import MPI

def worker(rank, size):
    """Fungsi untuk menjalankan tugas pada proses MPI"""
    print('Worker', rank, 'of', size)

if __name__ == '__main__':
    # Inisialisasi MPI
```

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Panggil fungsi worker pada setiap proses MPI
worker(rank, size)
```

Pada contoh di atas, kita mengimport modul MPI dari mpi4py dan membuat fungsi worker yang akan dieksekusi pada setiap proses MPI yang dibuat. Fungsi worker menerima dua argument, yaitu rank dan size. rank adalah nomor proses MPI, sedangkan size adalah jumlah total proses MPI yang dibuat.

Kemudian, kita menginisialisasi MPI dengan memanggil MPI.COMM_WORLD. Objek COMM_WORLD adalah objek communicator yang digunakan untuk berkomunikasi antara setiap proses MPI.

Selanjutnya, kita mendapatkan nomor proses MPI dan jumlah total proses MPI dengan memanggil comm.Get_rank() dan comm.Get_size(). Dalam hal ini, setiap proses MPI memiliki nomor proses yang berbeda dari 0 hingga size-1.

Terakhir, kita panggil fungsi worker dengan nomor proses dan jumlah total proses sebagai argument. Setiap proses MPI akan mengeksekusi fungsi worker dengan argument yang berbeda-beda.

Untuk menjalankan program ini, kita perlu menjalankan mpiexec dengan jumlah proses yang diinginkan, seperti mpiexec -n 4 python program.py untuk menjalankan program dengan 4 proses MPI.

6. **joblib**: Modul yang menyediakan fungsi untuk memproses tugas-tugas terpisah secara paralel pada komputer. Modul ini menyediakan beberapa algoritma parallel computing, termasuk parallel computing dengan menggunakan multiprocessing. Modul joblib menyediakan API yang mirip dengan API modul multiprocessing sehingga mudah digunakan.
   Contoh:

```
from joblib import Parallel, delayed

def worker(num):
    """Fungsi untuk menjalankan tugas pada proses multiprocessing"""
    print('Worker', num)

if __name__ == '__main__':
    # Panggil fungsi worker secara parallel
    Parallel(n_jobs=2)(delayed(worker)(i) for i in range(5))
```

Pada contoh di atas, kita mengimport modul Parallel dan delayed dari joblib. Fungsi Parallel digunakan untuk melakukan parallel computing dengan jumlah proses yang

diinginkan, sedangkan fungsi delayed digunakan untuk menunda pemanggilan fungsi sehingga dapat dijalankan secara parallel.

Kemudian, kita membuat fungsi worker yang akan dieksekusi pada setiap proses multiprocessing yang dibuat. Fungsi worker menerima satu argument, yaitu num, yang merepresentasikan nomor proses.

Selanjutnya, kita memanggil Parallel dengan n_jobs=2 untuk mengatur jumlah proses multiprocessing menjadi 2. Di dalam Parallel, kita menggunakan delayed untuk menunda pemanggilan fungsi worker pada setiap proses multiprocessing.

Akhirnya, kita menggunakan for loop untuk memanggil worker dengan nomor proses yang berbeda-beda.

Pada contoh di atas, kita menggunakan if __name__ == '__main__': karena penggunaan modul joblib juga memanfaatkan modul multiprocessing yang memiliki GIL.

7. **dask**: Modul yang menyediakan framework untuk parallelism dan distribusi data pada komputer cluster. Modul dask sangat berguna untuk melakukan parallel computing pada data yang besar dan kompleks.
   Contoh:

```
import dask.bag as db

def worker(num):
    """Fungsi untuk menjalankan tugas pada proses multiprocessing"""
    print('Worker', num)

if __name__ == '__main__':
    # Buat dask bag dengan 5 item
    bag = db.from_sequence(range(5))

    # Panggil fungsi worker pada setiap item secara parallel
    bag.map(worker).compute()
```

   Pada contoh di atas, kita mengimport modul dask.bag sebagai db. Fungsi from_sequence digunakan untuk membuat dask bag dari urutan item, sedangkan fungsi map digunakan untuk memanggil fungsi pada setiap item secara parallel.

   Kemudian, kita membuat fungsi worker yang akan dieksekusi pada setiap proses multiprocessing yang dibuat. Fungsi worker menerima satu argument, yaitu num, yang merepresentasikan nomor proses.

   Selanjutnya, kita membuat dask bag dengan 5 item menggunakan from_sequence. Setiap item pada bag akan diproses oleh setiap proses multiprocessing secara parallel dengan menggunakan map.

Terakhir, kita panggil compute untuk mengeksekusi dask bag secara parallel. Fungsi compute mengembalikan hasil pengolahan pada setiap item pada dask bag.

Pada contoh di atas, kita tidak perlu menggunakan if __name__ == '__main__': karena dask sudah mengatur penggunaan multiprocessing dan GIL secara otomatis.

8. **pathos**: Modul yang menyediakan antarmuka yang lebih sederhana untuk melakukan multiprocessing dan multithreading pada Python. Modul pathos sangat berguna untuk melakukan parallel computing pada fungsi yang kompleks dan tidak dapat di-serialize.
   Contoh:

```
from pathos.multiprocessing import ProcessPool

def worker(num):
    """Fungsi untuk menjalankan tugas pada proses multiprocessing"""
    print('Worker', num)

if __name__ == '__main__':
    # Buat ProcessPool dengan 2 proses
    pool = ProcessPool(2)

    # Panggil fungsi worker pada setiap item secara parallel
    pool.map(worker, range(5))
```

   Pada contoh di atas, kita mengimport ProcessPool dari modul pathos.multiprocessing. ProcessPool digunakan untuk membuat pool proses multiprocessing dengan jumlah proses yang diinginkan.

   Kemudian, kita membuat fungsi worker yang akan dieksekusi pada setiap proses multiprocessing yang dibuat. Fungsi worker menerima satu argument, yaitu num, yang merepresentasikan nomor proses.

   Selanjutnya, kita membuat ProcessPool dengan 2 proses menggunakan ProcessPool(2). Setelah itu, kita memanggil map untuk memanggil fungsi worker pada setiap item secara parallel.

   Terakhir, kita tidak perlu memanggil join atau close karena ProcessPool sudah menangani pengaturan tersebut secara otomatis.

   Pada contoh di atas, kita tidak perlu menggunakan if __name__ == '__main__': karena pathos sudah mengatur penggunaan multiprocessing dan GIL secara otomatis.
9. **ray**: Modul yang menyediakan framework untuk melakukan distributed computing pada cluster atau cloud. Modul ini menyediakan beberapa algoritma parallel computing, termasuk parallel computing dengan menggunakan multiprocessing, thread, dan actor-

based computing. Modul ray sangat berguna untuk melakukan parallel computing pada aplikasi skala besar dan kompleks.

Contoh:

```
import ray

@ray.remote
def worker(num):
    """Fungsi untuk menjalankan tugas pada proses multiprocessing"""
    print('Worker', num)

if __name__ == '__main__':
    # Inisialisasi Ray
    ray.init()

    # Buat task untuk setiap item secara parallel
    results = [worker.remote(i) for i in range(5)]

    # Ambil hasil task secara parallel
    ray.get(results)
```

Pada contoh di atas, kita mengimport ray untuk melakukan parallel computing. @ray.remote digunakan untuk mendeklarasikan fungsi worker sebagai task yang akan dijalankan pada proses multiprocessing.

Kemudian, kita membuat fungsi worker yang akan dieksekusi pada setiap proses multiprocessing yang dibuat. Fungsi worker menerima satu argument, yaitu num, yang merepresentasikan nomor proses.

Selanjutnya, kita memanggil ray.init() untuk menginisialisasi Ray. Setelah itu, kita membuat task untuk setiap item secara parallel menggunakan worker.remote(i).

Terakhir, kita menggunakan ray.get(results) untuk mengambil hasil task secara parallel. ray.get akan mengembalikan hasil task yang dieksekusi pada proses multiprocessing.

Pada contoh di atas, kita tidak perlu menggunakan if __name__ == '__main__': karena ray sudah mengatur penggunaan multiprocessing dan GIL secara otomatis.

Semua kode program dicoba dan hasilnya diupload ke

github masing-masing di folder : **latihan6**

Tugas :

Buat gambar desain 2 buah aplikasi yang membutuhkan jalan bersamaan. Hasil gambar diupload ke github masing-masing di folder: **tugas6**