

Programmazione multi threaded in Python

Motivazione all'uso dei thread

- **Pro**
 - **Scrittura di applicazioni con molteplici eventi asincroni (GUI)**
 - **Riduzione della latenza di servizio mediante l'uso di un pool di server**
 - **Speedup di applicazioni intrinsecamente parallele**
- **Contro**
 - **Modello di programmazione complicato (gestione della mutua esclusione)**
 - **Debugging scomodo**

Il modulo threading

▪ ESEMPI:
thread.py

- Python mette a disposizione il modulo **threading** per la programmazione di applicazioni multithreaded

```
import threading
```

- Un thread è rappresentato dalla classe **Thread**, il cui costruttore riceve in ingresso il nome della funzione eseguente il thread

```
t = threading.Thread(target = worker)
```

- Il metodo **start()** esegue la funzione in un nuovo thread

```
t.start()
```

Passaggio argomenti

▪ ESEMPI:
thread_args.py

- È possibile passare un numero arbitrario di argomenti alla funzione da eseguire
- Si usa il parametro con nome args della classe Thread

```
t = threading.Thread(target = worker, \
    args = (TUPLA DI ARGOMENTI))
```

- Si definisce la funzione con la signature opportuna

```
def worker(num):
```

...

Identificazione

▪ ESEMPI:
thread_identify.py

- Non è strettamente necessario passare un numero intero quale identificatore di un thread
- Ciascun thread ha già un proprio identificatore interno, che può essere ottenuto come segue
 - Si accede all'oggetto rappresentante il thread corrente tramite il metodo `currentThread()`
 - Si invoca il metodo `getName()` su tale oggetto
- name = `threading.currentThread().getName()`
- È possibile impostare un nome arbitrario tramite l'argomento `name` del costruttore
 - In caso contrario, il nome assegnato di default è del tipo `Thread-<num>`

Identificazione

▪ ESEMPI:
thread_identify_log.py

- Il modulo logging supporta la stampa del nome del thread nel log degli eventi
- È necessario attivare la stampa tramite l'uso della variabile `threadName` nella stringa di formato del logger

```
logging.basicConfig(level=logging.DEBUG, \
    format='[%(levelname)s] (%(threadName)-10s)' \
    ' %(message)s',)
```

Thread demoni

▪ ESEMPI:
thread_daemon.py

- Negli esempi visti finora, il programma principale ha aspettato il termine dei thread prima di uscire
- È possibile marcare un thread come demone (daemon) in modo tale che il programma non ne aspetti il termine prima di uscire
- Si usa il metodo `setDaemon()` sull'oggetto Thread

```
d = threading.Thread(name='daemon', target=daemon)  
d.setDaemon(True)
```

Sincronizzazione

▪ ESEMPI:
thread_join.py
thread_join_timeout.py

- Invocando il metodo `join()` su un oggetto `Thread`, il programma principale aspetta che il thread associato esca

```
t = threading.Thread(name='non-daemon', \
    target=daemon)
```

```
t.join()
```

- È possibile limitare l'attesa ad un certo numero di secondi (passati in argomento a `join()`)

```
t.join(1)
```

- Il metodo `isAlive()` dell'oggetto `Thread` mi dice se il thread è ancora in esecuzione

```
is_alive = t.isAlive()
```

Enumerazione

▪ ESEMPI:
thread_enumerate.py

- Il metodo `enumerate()` ritorna una lista di tutti i thread attivi
- Tale lista contiene il thread attualmente attivo (`current`) su cui non si può effettuare un `join`
 - Va saltato

```
main_thread = threading.currentThread()  
for t in threading.enumerate():  
    if t is main_thread:  
        continue  
    logging.debug('joining %s', t.getName())  
    t.join()
```

-

Estensione di Thread

▪ ESEMPI:
thread_subclass.py

- Un oggetto di tipo Thread esegue il metodo run() per eseguire la funzione specificata
- Si può estendere la classe Thread ed inserire nel metodo run() la funzione da eseguire
`class MyThread(threading.Thread):`

```
def run(self):  
    logging.debug('running')  
    return
```

Estensione di Thread

▪ ESEMPI:
thread_subc_args.py

- Per passare argomenti al metodo run, è necessario creare un metodo `__init__()` in grado di salvare i parametri in altrettanti attributi della classe
 - Il costruttore della sottoclasse deve invocare esplicitamente il costruttore di Thread
 - Tali attributi saranno acceduti da run()

Segnalazione di eventi

▪ ESEMPI:
thread_events.py

- I thread possono comunicare fra loro usando oggetti di tipo Event
- Un oggetto di tipo Event espone un flag interno, impostabile con i metodi
 - set(): ad 1
 - clear(): a 0
- Un thread può aspettare l'impostazione ad 1 (set) del flag invocando il metodo wait() dell'oggetto Event corrispondente
 - wait() può accettare un timeout di attesa (sec.)
- Meccanismo primitivo di variabili condizione (monitor)
 - Le variabili sono in realtà dei flag

Mutua esclusione

▪ ESEMPI:
thread_lock.py

- Gli oggetti complessi del Python (liste, dizionari, etc.) sono già thread-safe
- I tipi di dato di base (int, float) non sono thread-safe
- Le classi scritte dal programmatore non sono thread-safe
- Per garantire la mutua esclusione, si usa l'oggetto Lock del modulo threading
 - Metodo acquire(): prova ad acquisire il lock
 - Metodo release(): rilascia il lock

Semafori

▪ ESEMPI:
thread_sem.py

- È possibile gestire un pool di risorse di dimensione n in mutua esclusione
- Si usa la classe **Semaphore()** che implementa la struttura dati semaforo
 - Metodo **acquire()**: prova ad acquisire il lock
 - Metodo **release()**: rilascia il lock
- Il costruttore di **Semaphore()** accetta un numero intero (il numero di slot del pool)

Thread local storage

▪ ESEMPI:
thread_local.py

- Ciascun thread può usufruire di un'area privata di memoria per la memorizzazione di informazioni
- Tale area è locale al thread; essa non viene condivisa con gli altri thread in esecuzione
- Si invoca la funzione `local()` del modulo `threading`, ottenendo un puntatore alla variabile locale
- Si modifica la variabile locale
`local_data = threading.local()`
`local_data.value = 1000`