

Multi-threading in Python

Python supporta il modello multi-threading, tuttavia come altri linguaggi interpretati (Ruby), la versione CPython soffre del problema del **Global Interpreter Lock (GIL)**. Il GIL può essere visto come un mutex che implementa un accesso in mutua esclusione all'interprete Python → *assenza di un effettivo parallelismo, anche in caso di applicazione multithreaded*.

Tuttavia il GIL di Python è fondamentale per garantire la safety di un'applicazione quando per esempio si va a modificare la reference a un oggetto.

Il problema della concorrenza verrà affrontato in corsi più avanzati.

Una soluzione parziale al problema del GIL è l'utilizzo del modello **multi-processing**.

288

Multi-threading in Python

Il modulo `threading` consente di creare e lanciare nuovi thread, il cui codice è definito all'interno di una funzione (o estendendo `Thread`). Un classico esempio è il seguente:

```
def count_to_five(times=1):
    for i in range(1, 6):
        print(f"{i} (thread {times})")
        # get the id of the current thread
        print(f"Thread id: {threading.get_ident()}")
        time.sleep(1)

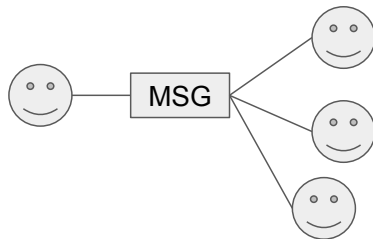
def main():
    t1 = threading.Thread(target=count_to_five, args=(1,))
    t1.start()
    print("Hello dal thread main")

if __name__ == "__main__":
    main()
```

289

Multicast UDP 1/

Una comunicazione basata su multicast permette di inviare uno stesso messaggio a più partecipanti, seguendo un pattern 1:molti.



Per poter inviare / ricevere messaggi multicast bisogna creare un socket UDP e utilizzare una classe di indirizzi IP apposita (224-239.x.x.x).

Multicast è molto utilizzato in reti wireless, al fine di poter raggiungere più riceventi con una sola trasmissione.

290

Multicast UDP 2/

Creo una socket DATAGRAM e effettuo il bind a un IP e porta specificata.

```
import socket
import struct
address = '224.0.0.5'
port = 45000
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((address, port))
```

291

Multicast UDP 3/

```
group = socket.inet_aton(address)
mreq = struct.pack('4sL', group, socket.INADDR_ANY)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP,
mreq)
```

Le seguenti istruzioni permettono di trasformare la socket UDP in una socket UDP multicast.

Il nodo viene registrato in un gruppo multicast, a quali si dovranno registrare tutti gli altri nodi interessati alla comunicazione.

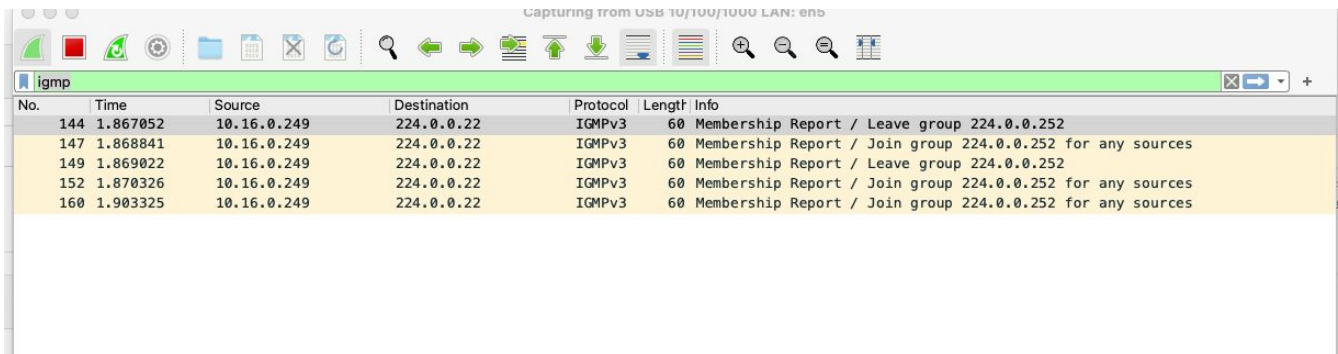
292

Multicast UDP 2/

```
import socket
import struct
address = '224.0.0.5'
port = 45000
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((address, port))
group = socket.inet_aton(address)
mreq = struct.pack('4sL', group, socket.INADDR_ANY)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP,
mreq)
```

293

Multicast UDP 4/



The image shows a Wireshark packet capture window. The title bar indicates it is capturing from USB 10/100/1000 LAN: enb. The packet list pane shows five packets, all of which are IGMPv3 Membership Reports. The first packet (No. 144) is a 'Leave group' message for 224.0.0.252. The subsequent four packets (Nos. 147, 149, 152, and 160) are 'Join group' messages for the same address. All packets have a source IP of 10.16.0.249 and a destination IP of 224.0.0.22. The packet details pane is currently empty.

No.	Time	Source	Destination	Protocol	Length	Info
144	1.867052	10.16.0.249	224.0.0.22	IGMPv3	60	Membership Report / Leave group 224.0.0.252
147	1.868841	10.16.0.249	224.0.0.22	IGMPv3	60	Membership Report / Join group 224.0.0.252 for any sources
149	1.869022	10.16.0.249	224.0.0.22	IGMPv3	60	Membership Report / Leave group 224.0.0.252
152	1.870326	10.16.0.249	224.0.0.22	IGMPv3	60	Membership Report / Join group 224.0.0.252 for any sources
160	1.903325	10.16.0.249	224.0.0.22	IGMPv3	60	Membership Report / Join group 224.0.0.252 for any sources

Quando un nodo viene aggiunto a un gruppo multicast, viene generato del traffico IGMP (Internet Group Management Protocol), il protocollo per la gestione dei gruppi multicast.

294

Esercizio Multicast UDP

Si scriva un programma di chat in Python. Il programma deve creare una socket UDP multicast, utilizzando l'indirizzo 224.0.0.5 e la porta 10000.

Una volta avviato il programma, questo deve chiedere in input l'username dell'utente. Poi il programma deve:

- 1) continuare a richiedere all'utente di inserire un messaggio da inviare;
- 2) visualizzare i messaggi inviati dagli altri partecipanti alla chat;

295

Esercizio: Date Server

Sviluppare un'applicazione client/server (utilizzando le socket UDP) per ottenere da un Server in ascolto su una determinata PORTA la data e il tempo corrente. A tal fine, il Server deve rispondere alle richieste dei client che richiedono il tempo configurato sul sistema.

Per richiedere il servizio, il client invia un datagram contenente la stringa "date" al Server, che una volta ricevuto il messaggio esegue il comando **date**, ne legge l'output e lo invia al client che ha fatto richiesta.

296

Dual-stack IPv6

Con poche semplici istruzioni è possibile creare un server dual-stack, con supporto sia a IPv4 che IPv6.

```
import socket
```

```
addr = ("", 45000)
```

```
if socket.has_dualstack_ipv6():
```

```
    s =
```

```
socket.create_server(addr, family=socket.AF_INET6, dualstack_ipv6=True)
```

```
else:
```

```
    s = socket.create_server(addr)
```

```
s.listen(1)
```

Notare la differenza di sintassi con il codice precedente...

297

Multi-threading in Python

Una documentazione ricca e approfondita dell'utilizzo della classe Thread è disponibile nella documentazione ufficiale di Python:

<https://docs.python.org/3/library/threading.html>

298

Multi-processing in Python

Con multi-processing è possibile creare più sub-processi, ognuno con un proprio spazio di memoria, annullando così il problema del GIL.

```
from multiprocessing import Pool
import time
COUNT = 50000000
def countdown(n):
    while n>0:
        n -= 1

if __name__ == '__main__':
    pool = Pool(processes=4)
    r1 = pool.apply_async(countdown, [COUNT//4])
    r2 = pool.apply_async(countdown, [COUNT//4])
    r3 = pool.apply_async(countdown, [COUNT//4])
    r4 = pool.apply_async(countdown, [COUNT//4])
    pool.close()
    pool.join()
```

299

Server multi-processing in Python

Possiamo utilizzare il metodo visto nell'esempio per implementare un server multi-processo.

Il codice della gestione della richiesta andrà implementato **in una funzione di handling**.

Al ricevimento di una richiesta di connessione andiamo a creare un nuovo processo (*metodo **start** su un'istanza di **Process***) passando la socket client creata alla funzione di handling.

Vediamo il nostro EchoServer implementato con il modello multi-processing...

300

struct package

struct è un modulo utilissimo per effettuare la conversione tra valori Python e struct C rappresentate come degli oggetti bytes. Quindi possiamo utilizzare struct **per** leggere dati in formato binario che sono memorizzati o che provengono da una connessione di rete.

Quando analizziamo un pacchetto di rete, sappiamo che i suoi dati sono codificati in formato *big-endian* (o network order) che è differente da quella utilizzata sulla nostra macchina (*little-endian*).

struct ci permette di utilizzare delle format string per specificare sia il byte-ordering che il tipo di dato che vogliamo leggere (int, short, char, ...)

301

struct package

```
struct.pack(format, v1, v2, ...)
```

Ritorna un oggetto di tipo `bytes` che contiene i valori `v1`, `v2`, ... nella codifica specificata da `format`. Gli argomenti (`v1`, `v2`, ...) devono corrispondere ai valori specificati in `format`.

```
struct.unpack(format, buffer)
```

Decodifica da *buffer* una serie di valori specificata da *format*. I valori di ritorno vengono memorizzati in un oggetto di tipo tupla (anche nel caso di un singolo valore); la dimensione di *buffer* deve corrispondere a quella richiesta in *format*.

struct package

Il primo carattere di *format* può essere utilizzato per specificare il byte-ordering, il size e l'allineamento dei byte (eventuale presenza di padding).

Character	Byte order	Size	Alignment
@	native	native	native
=	native	standard	none
<	little-endian	standard	none
>	big-endian	standard	none
!	network (= big-endian)	standard	none

struct package

I restanti caratteri di format specificano invece il tipo di dato da leggere:

Format	C Type	Python type	Standard size
x	pad byte	no value	
c	char	bytes of length 1	1
b	signed char	integer	1
B	unsigned char	integer	1
?	_Bool	bool	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4

struct package

I	unsigned int	integer	4
l	long	integer	4
L	unsigned long	integer	4
q	long long	integer	8
Q	unsigned long long	integer	8

La tabella completa è disponibile nella documentazione di struct
<https://docs.python.org/3/library/struct.htm> o utilizzando l'helper inline del linguaggio.

struct Esempio

Supponiamo di dover decodificare 4 byte dal formato big-endian che contengono le informazioni relative a porta sorgente e destinazione di un pacchetto UDP.

SRC Port (16 bits)	DST Port (16 bits)
--------------------	--------------------

```
src_port, dst_port = struct.unpack('! H H', buf) # buf is a
4 bytes buffer

print('SRC: {} DST: {}'.format(src_port, dst_port))
```

struct Esempio

Proviamo a effettuare sia l'encoding che il decoding:

```
import struct

src, dst = 13455, 8080

buf = struct.pack('! H H', src, dst)

src_port, dst_port = struct.unpack('! H H', buf)

print('SRC: {} DST: {}'.format(src_port, dst_port))
```

Implementazione di una interfaccia ReST

Indice

- ReST
- Obiettivo
- Librerie, Framework e Strumenti
- Hands on: Creare un Client ReST
- Hands on: Creare un'API ReST

ReST

- Stile architetturale, non standard o protocollo
- ⇒ Alcuni dettagli sono specifici dell'implementazione
- Basato su HTTP
- Scambio di rappresentazioni di risorse
- Caratteristiche principali:
 - Client-server
 - Stateless
 - Cacheable

ReST

- Le risorse sono identificate da un URI, definito in maniera gerarchica
- Insiemi di risorse sono detti “collezioni”

URI				Cosa rappresenta
/users				La collezione “users”
/users	/1			L'utente con id “1”
/users	/1	/comments		La collezione dei commenti dell'utente “1”
/users	/1	/comments	/3	Il commento con id “3” dell'utente “1”

ReST

- Per accedere a una risorsa o collezione si deve effettuare una richiesta HTTP all'URI che la identifica
- L'azione che si vuole effettuare va specificata con il relativo metodo HTTP
- Le azioni fondamentali sono le cosiddette CRUD:
 - Create
 - Read
 - Update
 - Delete
- La gestione dei metodi HTTP è implementation-specific, si presenta una delle possibilità di gestione

ReST

- Azioni su una risorsa

URI	GET	PUT	POST	DELETE
/users/1	Recupera una rappresentazione della risorsa	Modifica una risorsa esistente, sostituendola	Non utilizzato	Elimina la risorsa
Possibili stati HTTP della risposta	200 OK	200 OK 201 Created 204 No Content	Non utilizzato	200 OK 202 Accepted 204 No Content

Stati in caso di errore: 404 Not Found, 400 Bad Request, 415 Unsupported Media Type

ReST

- Azioni su una collezione

URI	GET	PUT	POST	DELETE
/users	Recupera l'elenco di tutte le risorse nella collezione	Sostituisce la collezione esistente con una nuova	Crea una nuova risorsa nella collezione	Elimina la collezione
Possibili stati HTTP della risposta	200 OK	200 OK 201 Created 204 No Content	200 OK 204 No Content	200 OK 202 Accepted 204 No Content

Stati in caso di errore: 404 Not Found, 400 Bad Request, 415 Unsupported Media Type

ReST

- A seconda dell'implementazione si possono supportare altri metodi HTTP:
 - PATCH → Modificare porzioni di una risorsa
 - HEAD → Ottenere informazioni riguardo una risorsa (versione, dimensione, etc.)
 - OPTIONS → Ottenere informazioni riguardo l'API (metodi, content-type supportati, etc.)
- Come formato di rappresentazione delle informazioni andremo a utilizzare JSON

Obiettivo

- Prima parte: creazione di un Client ReST che contatta un'API ReST “up and running” ([JSONPlaceholder](#))
- Seconda parte: creazione di un'API ReST che imita il comportamento di jsonplaceholder

Librerie, Framework e Strumenti

Librerie, Framework e Strumenti

- Linguaggio **Python** 
- Per il Client: Libreria **requests** per effettuare richieste HTTP
- Per il Server: Framework **Flask** 
- Per testare il nostro Server: **Insomnia**



Possiamo provare l'interfaccia ReST anche con Google Chrome e le sue estensioni

Librerie, Framework e Strumenti

Nota: si assume che la distribuzione Linux in uso sia Ubuntu

- Creare un **ambiente virtuale** nella cartella dove lavorerete:

```
cd /cartella/di/lavoro
```

```
python -m venv venv
```

- **Attivare** l'ambiente virtuale: (Nota: se si chiude il terminale va riattivato!)

```
source venv/bin/activate
```

- Dopo aver attivato l'ambiente virtuale, installare i pacchetti Python necessari, ovvero **requests** e **Flask**:

```
pip install requests Flask
```


Hands on: Creare un Client ReST

Hands on: Creare un Client ReST

- La libreria requests consente di effettuare richieste HTTP in maniera semplice
- Ad esempio, per effettuare una richiesta GET:

```
import requests

my_headers = {"Content-type": "application/json"}
res = requests.get("URL_DA_CONTATTARE", headers=my_headers)
print("Stato HTTP: "+str(res.status_code))
print("Risposta: "+str(res.json()))
```

Hands on: Creare un Client ReST - Esercizio 1

- Creare un Client ReST che contatta JSONPlaceholder (<https://jsonplaceholder.typicode.com>), e che effettua le seguenti richieste:

Metodo	URL	Significato
GET	/todos	Recuperare tutti i todo
GET	/todos/1	Recuperare il todo con id 1
POST	/todos	Inserire un nuovo todo, i dati da inserire vanno specificati nel body della richiesta (tranne l'id, che viene assegnato dal server)
PUT	/todos/1	Sostituire il todo con id 1 con un nuovo todo, del quale tutti i dati vanno specificati nel body della richiesta (compreso l'id)
DELETE	/todos/5	Eliminare il todo con id 5

Hands on: Creare un Client ReST - Esercizio 1

- Template di riferimento: Es-1/es-1.py
- Formato dei todo:

```
{
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
}
```

Hands on: Creare un Client ReST - Esercizio 2

- Aggiungere al Client creato nell'esercizio precedente le seguenti richieste:

Metodo	URL	Significato
GET	/users/1	Recuperare l'utente con id 1
POST	/users	Inserire un nuovo utente, i dati da inserire vanno specificati nel body della richiesta (tranne l'id, che viene assegnato dal server)
PUT	/users/1	Sostituire l'utente con id 1 con un nuovo utente, del quale tutti i dati vanno specificati nel body della richiesta (compreso l'id)

Hands on: Creare un Client ReST - Esercizio 2

- Template di riferimento: Es-2/es-2.py
- Formato degli users: →

```
{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",
  "address": {
    "street": "Kulas Light",
    "suite": "Apt. 556",
    "city": "Gwenborough",
    "zipcode": "92998-3874",
    "geo": {
      "lat": "-37.3159",
      "lng": "81.1496"
    }
  },
  "phone": "1-770-736-8031 x56442",
  "website": "hildegard.org",
  "company": {
    "name": "Romaguera-Crona",
    "catchPhrase": "Multi-layered client-server
neural-net",
    "bs": "harness real-time e-markets"
  }
}
```

Hands on: Creare un Client ReST - Esercizio 3

- Ogni utente ha associati i propri todo, accessibili tramite il path `/users/<id>/todos`
- Aggiungere al Client creato negli esercizi precedenti le seguenti richieste:

Metodo	URL	Significato
GET	<code>/users/1/todos</code>	Recuperare i todos dell'utente con id 1
POST	<code>/users/1/todos</code>	Inserire un nuovo todo per l'utente 1, i dati da inserire vanno specificati nel body della richiesta (tranne l'id, che viene assegnato dal server)

- Template di riferimento: `Es-3/es-3.py`

Hands on: Creare un'API ReST

Hands on: Creare un'API ReST

- Lo scopo di questa sezione consiste nel creare un'API ReST che simula il comportamento di jsonplaceholder
- Per fare ciò utilizzeremo il framework Flask, il quale consente di creare API ReST in maniera molto semplice
- Tramite i decoratori, Flask permette di mappare una richiesta HTTP (metodo e URL) a una funzione scritta dallo sviluppatore
- Per il momento, come sorgente di dati utilizzeremo una lista di dizionari scritta direttamente nel codice (già presente sui template forniti)

Hands on: Creare un'API ReST

- Ad esempio, per gestire la richiesta GET /todos (supponendo che l'oggetto "todos" sia una lista di todo), è sufficiente scrivere:

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.get("/todos") # <- Decoratore
def get_todos():
    return jsonify(todos), 200
```

- In questo caso si restituisce al Client la lista di todo in formato JSON (con stato HTTP 200)

Hands on: Creare un'API ReST

- Mentre per gestire la richiesta GET /todos/1:

```
@app.get("/todos/<id>")
def get_todo_by_id(id):
    to_return = _find_todo_by_id(id)
    if to_return is None:
        return "{}", 404
    else:
        return jsonify(to_return), 200
```

- Supponendo che il metodo `_find_todo_by_id`, data la lista di todo, recuperi quello con l'id specificato
- In questo caso si restituisce al Client il todo richiesto in formato JSON (con stato HTTP 200), oppure, se non è stato trovato, un oggetto JSON vuoto (con stato HTTP 404)

Hands on: Creare un'API ReST

- Per avviare la propria API ReST scritta con Flask, sempre dall'interno del proprio ambiente virtuale, dare i seguenti comandi:

```
export FLASK_APP=<nome-programma>.py
```

- Ad esempio, se il file Python contenente il codice dell'API è "server.py":

```
export FLASK_APP=server.py
```

- Infine, la API ReST può essere avviata con il comando:

```
flask run
```

Hands on: Creare un'API ReST - Esercizio 4

- Utilizzando il framework Flask, realizzare un'API ReST che risponde alle richieste HTTP nella seguente maniera:

URI	GET	PUT	POST	DELETE
/todos	Restituisce la lista di tutti i todo in formato JSON	Non implementare	Crea un nuovo todo con i dati passati dal Client e assegnando automaticamente l'id	Non implementare
/todos/<id>	Restituisce il todo con id <id> in formato JSON	Sostituisce il todo con id <id> con quello passato dal Client	Non implementare	Elimina il todo con id <id>

Hands on: Creare un'API ReST - Esercizio 4

- Template di riferimento: Es-4/es-4.py
- Per verificare la correttezza dei metodi implementati utilizzando il browser o curl

Hands on: Creare un'API ReST - Esercizio 5

- Aggiungere al Server creato all'esercizio precedente la gestione degli stessi metodi mostrati nell'Esercizio 4 per la collezione e per le risorse "users"
- Template di riferimento: Es-5/es-5.py
- Per testare la correttezza dei metodi implementati utilizzare Insomnia

Hands on: Creare un'API ReST - Esercizio 6

- Aggiungere al Server creato negli esercizi precedenti la gestione dei seguenti metodi:

URI	GET	POST
/users/<id>/todos	Restituisce la lista di tutti i todo dell'utente <id> in formato JSON	Crea un nuovo todo appartenente all'utente <id> con i dati passati dal Client e assegnando automaticamente l'id del todo

- Template di riferimento: Es-6/es-6.py
- Per testare la correttezza dei metodi implementati utilizzare Insomnia

Hands on: Creare un'API ReST - Esercizio 7

- Modificare il ReST Client creato nella sezione precedente in modo che contatti non più jsonplaceholder, ma l'API ReST creata in questa sezione
- Lanciare Client e Server e verificare che le richieste vengano inviate e gestite correttamente
- Inoltre si può utilizzare Insomnia per debuggare la propria API ReST