Lezione 11: Python e Moduli, MongoDB e Db Relazionali

11.1 Funzioni di Callback

Prima di introdurre i moduli, viene chiarito un concetto utile:

- **Concetto:** In Python, le funzioni sono "oggetti di prima classe". Questo significa che possono essere trattate come qualsiasi altro dato: assegnate a variabili, inserite in liste e, soprattutto, passate come argomenti ad altre funzioni.
- **Callback:** Una funzione passata come argomento a un'altra funzione, con l'intenzione che quest'ultima la "richiami" (call back) in un momento specifico (es., al termine di un'operazione), è detta funzione di callback.
- **Esempio:**
  - Viene definita una funzione `stampa(sum)` che semplicemente stampa un valore.
  - Viene definita una funzione `main(a, b, callback=None)` che stampa `a` e `b`, calcola la loro somma, e *se* il parametro `callback` è stato fornito (cioè non è `None`), chiama la funzione `callback` passandole la somma `a+b`.
  - Chiamando `main(1, 2, stampa)`, la funzione `stampa` viene passata come callback. `main` eseguirà `callback(1+2)`, che a sua volta chiamerà `stampa(3)`, producendo l'output:

    ```
    adding 1 + 2
    Sum = 3
    ```

11.2 Moduli Python

- **Definizione:** I moduli sono contenitori di codice Python (variabili, funzioni, classi) che permettono di organizzare e riutilizzare il codice. Ogni file `.py` può essere considerato un modulo.
- **Incapsulamento ed Esportazione:** Un modulo incapsula il suo contenuto e "esporta" funzioni e classi per essere utilizzate da altri programmi.
- **Moduli Built-in:** Python include molti moduli pre-installati (es., `gc`, `json`, `math`). Non richiedono installazione separata.
- **Importazione:** Per usare un modulo, si usa l'istruzione `import nome_modulo`.
- **Prefissi (Namespace):** Dopo l'importazione, per accedere a funzioni o classi del modulo, si usa la notazione puntata: `nome_modulo.nome_funzione()`. Questo evita conflitti di nomi tra moduli

diversi (ogni modulo ha il suo "namespace").

11.2.1 Modulo `gc`

- Il modulo `gc` permette di interagire con il *garbage collector* (il meccanismo automatico di gestione della memoria di Python).
- Funzioni principali:
  - `gc.collect()` : Forza una ciclo di raccolta della memoria inutilizzata.
  - `gc.enable()` : Abilita il garbage collector (è lo stato predefinito).
  - `gc.disable()` : Disabilita il garbage collector.
  - `gc.isenabled()` : Restituisce `True` se il GC è attivo.

11.2.2 Modulo `json`

- Un modulo built-in fondamentale per lavorare con dati in formato JSON (JavaScript Object Notation).
- **Funzioni principali:**
  - **Serializzazione** (da oggetto Python a JSON):
    - `json.dumps(data)` : Converte un oggetto Python (tipicamente un dizionario o una lista) in una *stringa* JSON.
    - `json.dump(data, filehandler)` : Scrive l'oggetto Python serializzato come JSON direttamente su un *file* (aperto in modalità scrittura).
  - **Deserializzazione** (da JSON a oggetto Python):
    - `json.loads(stringa)` : Converte una *stringa* contenente JSON valido in un oggetto Python (tipicamente dizionario o lista).
    - `json.load(filehandler)` : Legge JSON da un *file* (aperto in modalità lettura) e lo converte in un oggetto Python.

11.2.3 File Handler

- Per interagire con i file (necessario per `json.dump` e `json.load` su file), Python usa oggetti *file handler*.
- **Apertura:** Si ottiene un file handler usando la funzione built-in `open()` :
  - `filein = open("nome_file.txt")` : Apre il file in modalità lettura (default).
  - `fileout = open("nome_file.txt", "w")` : Apre il file in modalità scrittura (`w` sta per write). Se il file esiste, viene sovrascritto. Altre modalità comuni sono `a` (append) e `r+` (lettura/scrittura).
- **Chiusura:** È **fondamentale** chiudere un file handler dopo aver finito di usarlo per rilasciare le risorse e assicurarsi che tutti i dati siano stati scritti. Si usa il metodo `.close()` :

    - `filehandler.close()`
    - Se non si chiude, il file potrebbe rimanere bloccato ("locked") dal processo Python.
- **Lettura:** Metodi comuni del file handler aperto in lettura:
    - `filehandler.read()` : Legge l'intero contenuto del file come una singola stringa.
    - `filehandler.readline()` : Legge una singola riga dal file (fino al carattere `\n`).
    - `filehandler.readlines()` : Legge tutte le righe del file e le restituisce come una *lista* di stringhe.
    - Iterazione: Si può iterare direttamente sul file handler con un ciclo `for` per leggere le righe una alla volta: `for line in filehandler:` .
- **Scrittura:** Metodi comuni del file handler aperto in scrittura:
    - `filehandler.write(testo)` : Scrive la stringa `testo` sul file. Per andare a capo, inserire esplicitamente `\n` nella stringa.
    - `filehandler.writelines(lista)` : Scrive sul file tutti gli elementi (stringhe) contenuti nella `lista` , uno dopo l'altro. Non aggiunge automaticamente il carattere di fine riga.

11.2.4 Esempio JSON e File I/O

Viene mostrato un esempio che:

1. Importa il modulo `json` .
2. Chiede all'utente i nomi dei file di input (JSON) e output usando un ciclo `while` per assicurarsi che non siano vuoti.
3. Apre il file di input: `filein = open(nfilein)` .
4. Legge e deserializza il JSON dal file: `data = json.load(filein)` . Si suppone che `data` sia un dizionario contenente una chiave `"docs"` che è una lista di documenti (dizionari).
5. Crea un nuovo dizionario `nuovo` .
6. Popola `nuovo` con il numero di elementi in `data["docs"]` e una nuova lista `"docs"` .
7. Itera sui documenti in `data["docs"]` e appende una versione semplificata (solo con "name" e "age") di ciascuno alla lista `nuovo["docs"]` .
8. Apre il file di output in scrittura: `fileout = open(nfileout, "w")` .
9. Serializza il dizionario `nuovo` e lo scrive nel file di output: `json.dump(nuovo, fileout)` .
10. Chiude entrambi i file handler: `filein.close()` e `fileout.close()` .

Esempio di trasformazione: Se l'input JSON è:

```
{
  "Total": 1, "Selected": 10,
  "docs": [ { "_id": "...", "name": "\"Paolino\"", "age": "12" } ]
}
```

L'output JSON sarà:

```
{
  "Elements": 1,
  "docs": [ { "name": "\"Paolino\"", "age": "12" } ]
}
```

11.2.5 Istruzione `with` (Context Manager)

- L'istruzione `with` fornisce un modo più pulito e sicuro per gestire risorse che richiedono setup e teardown (come i file handler).
- **Context Manager Protocol:** Funziona con oggetti che implementano i metodi speciali `__enter__` (eseguito all'inizio del blocco `with`) e `__exit__` (eseguito alla fine del blocco, anche in caso di errori).
- **File Handler:** Gli oggetti restituiti da `open()` sono context manager. `__exit__` si occupa automaticamente di chiamare `close()`.
- **Sintassi:**

```
with open("file.txt", "w") as file_handler:
    # Codice che usa file_handler
    file_handler.write("...")
# Qui, file_handler è automaticamente chiuso, anche se si verifica un errore nel blocco wit
```

- **Vantaggio:** Non è più necessario ricordare di chiamare `filehandler.close()`, rendendo il codice più robusto. L'esempio JSON precedente viene riscritto usando `with` per aprire sia il file di input che quello di output, eliminando le chiamate esplicite a `.close()`.

11.2.6 Chiamate HTTP (`requests`)

- Per effettuare chiamate HTTP (es. per interagire con API web), si usa comunemente il modulo `requests`. (Nota: Le slide lo definiscono built-in, ma `requests` è una libreria di terze parti molto popolare che va installata, tipicamente con `pip install requests` o `conda install requests`. Potrebbe essere inclusa in distribuzioni come Anaconda.)
- **Importazione:** `import requests`
- **Metodo GET:** Per inviare una richiesta GET:
  - `res = requests.get(url, params=params_dict)`
  - `url`: L'indirizzo a cui inviare la richiesta.
  - `params` (opzionale): Un dizionario di parametri da aggiungere all'URL come query string (es. `?key1=value1&key2=value2`).

- **Metodo POST:** Per inviare una richiesta POST:
  - `res = requests.post(url, data=data_dict, json=json_dict)`
  - `url` : L'indirizzo a cui inviare la richiesta.
  - `data` (opzionale): Dati da inviare nel corpo della richiesta (tipicamente come form data).
  - `json` (opzionale): Un dizionario Python da inviare come corpo della richiesta, serializzato automaticamente in JSON e con l'header `Content-Type: application/json` .
- **Oggetto Risposta:** Entrambi i metodi restituiscono un oggetto *Response* (`res`) che contiene la risposta del server.
  - **Campi/Metodi Utili:**
    - `res.text` : Il contenuto della risposta come stringa testuale.
    - `res.json()` : Deserializza il corpo della risposta (se è JSON valido) in un oggetto Python (dizionario/lista). Solleva un errore se la risposta non è JSON valido.
    - `res.status_code` : Il codice di stato HTTP (es. 200, 404, 500).
    - `res.headers` : Un dizionario degli header della risposta.
  - **Documentazione:** `https://requests.readthedocs.io/en/master/api/#requests.Response`
- **Esempio Chiamata API Node.js/MongoDB:**
  - Viene mostrato un esempio Python che chiama un ipotetico servizio `/list` su `http://127.0.0.1:8080` (presumibilmente un server Node.js realizzato precedentemente) usando `requests.get` .
  - La risposta JSON viene deserializzata con `r.json()` in un dizionario `data` .
  - Il codice itera poi sul campo `data["docs"]` per stampare i campi `name` e `age` di ogni documento ricevuto.
  - Si sottolinea la facilità con cui si effettua la richiesta e si processa la risposta JSON, integrando diversi tecnologie (Python client, Node.js server, MongoDB database).

11.2.7 Creare Moduli Personalizzati

- **Come creare:** Un modulo personalizzato è semplicemente un file Python (`.py`) che contiene definizioni di funzioni e/o classi.
- **Esempio:**
  i. Si crea un file `pweb.py` con una singola funzione:

```python
# pweb.py
def pweb():
    return "Modulo PWEB"
```

  ii. In un altro file Python (nella stessa directory, o in una directory inclusa nel `PYTHONPATH`), si importa e si usa il modulo:

```
# main_program.py
import pweb

print(pweb.pweb()) # Output: Modulo PWEB
```

- **Importazione:** L'istruzione `import pweb` cerca il file `pweb.py` e rende disponibili le sue definizioni sotto il namespace `pweb`.

11.3 Connessione a MongoDB con `pymongo`

- Per connettere Python a un server MongoDB, si usa la libreria `pymongo`.
- **Installazione:** `pymongo` è una libreria esterna e va installata. Se si usa Spyder/Anaconda, il comando consigliato è:

  ```
  conda install pymongo
  ```

  (Alternativamente, si può usare `pip install pymongo`).
- **Importazione:** `import pymongo`
- **Stabilire la Connessione:** Si crea un'istanza di `MongoClient`, passando la stringa di connessione:

  ```
  conn_string = "mongodb://localhost:27017/" # URL standard per MongoDB locale
  myclient = pymongo.MongoClient(conn_string)
  ```

- **Acquisire Database e Collezione:** Dall'oggetto `MongoClient`, si accede a un database specifico e poi a una collezione come se fossero attributi o elementi di dizionario:

  ```
  mydb = myclient["MyDB_test"]    # Accede al database 'MyDB_test'
  mycol = mydb["MyCollection"] # Accede alla collezione 'MyCollection'
  # Alternativamente: mydb = myclient.MyDB_test; mycol = mydb.MyCollection
  ```

- **Eseguire Query (`find`) e Iterare Cursori:** Il metodo `find()` della collezione funziona in modo simile alla shell di Mongo, restituendo un oggetto *Cursor*:

  ```
  result = mycol.find({}) # Trova tutti i documenti nella collezione
  ```

  Si può iterare sul cursore con un ciclo `for` per accedere ai singoli documenti (che sono dizionari Python):

```
for item in result:
    print(item)
    # 'item' è un dizionario che rappresenta un documento
```

Il cursore viene trattato come una sequenza (lista) durante l'iterazione.

- **Differenze Oggetti `pymongo` (Collection, Cursor):** L'API di `pymongo` è leggermente diversa da quella della shell standard di MongoDB:
  - **Metodi Cursor:** L'oggetto Cursor restituito da `find` in `pymongo` *non* ha il metodo `.count()`.
  - **Metodi Collection:** L'oggetto Collection ha metodi con nomi diversi per le operazioni CRUD:
    - `insert_one()`, `insert_many()` (invece di `insertOne`, `insertMany`)
    - `update_one()`, `update_many()` (invece di `updateOne`, `updateMany`)
    - `delete_one()`, `delete_many()` (invece di `deleteOne`, `deleteMany`)
    - `count_documents(query)`: Questo è il metodo da usare sull'oggetto *Collection* per contare i documenti che corrispondono a una `query` (sostituisce il `.count()` del cursore della shell).

11.4 Connessione a Database Relazionali (PostgreSQL) con `psycopg2`

- **Introduzione a PostgreSQL:** Viene presentato come un DBMS relazionale gratuito, potente e considerato dall'autore migliore di MySQL. Implementa anche funzionalità object-relational. Sito ufficiale: `https://www.postgresql.org/`.
- **Installazione (Windows):** Si consiglia di usare l'installer "Interactive installer by EDB". Dopo l'installazione base, usare "StackBuilder" per installare i driver necessari. L'interfaccia grafica comune per Postgres è `pgAdmin`.
- **Driver Python (`psycopg2`):** Il driver più suggerito per connettere Python a PostgreSQL è `psycopg2`.
- **Installazione Driver:** Va installato come libreria esterna. Con Anaconda/Spyder:

```
conda install psycopg2
```

(Alternativamente: `pip install psycopg2-binary`).

- **Passi per l'Interazione:** La sequenza tipica di operazioni è:
  i. Creare una connessione al database.
  ii. Creare un cursore dalla connessione.
  iii. Eseguire una query SQL tramite il cursore.
  iv. Recuperare ("fetch") le righe risultanti dal cursore.

v. Elaborare ("scandire") le righe recuperate.

vi. Chiudere la connessione.

- **Creare Connessione e Cursore:**

```
import psycopg2

# Creare la connessione
conn = psycopg2.connect(
    database="MyDB",      # Nome del database
    user="MyUser",        # Nome utente
    password="MyPwd",     # Password
    host="localhost",     # Indirizzo del server DB (default: localhost)
    port="5432"           # Porta del server DB (default: 5432)
)

# Creare un cursore
cursor = conn.cursor()
```

- **Eseguire Query:** Si usa il metodo `execute()` del cursore, passando la stringa SQL:

```
query = 'SELECT "Name", "Age" FROM "Names";' # Nota: "" per nomi con maiuscole/minuscole
cursor.execute(query)
```

- **Recuperare Risultati:**
  - `results = cursor.fetchall()` : Recupera *tutte* le righe rimanenti dal risultato della query e le restituisce come una **lista di tuple**. Ogni tupla rappresenta una riga, e gli elementi della tupla sono i valori delle colonne nell'ordine specificato dalla SELECT.
  - `r = cursor.fetchone()` : Recupera *una sola* riga alla volta come tupla. Restituisce `None` quando non ci sono più righe. Utile per risultati molto grandi per non caricarli tutti in memoria.
- **Scandire il Risultato (`fetchall`):**

```
results = cursor.fetchall()
for r in results:
    # 'r' è una tupla, es: ('Pippo', 25)
    nome = r[0] # Accede al primo campo (Name)
    eta = r[1]  # Accede al secondo campo (Age)
    print(f"Nome: {nome}, Età: {eta}")
```

I campi nella tupla si accedono solo tramite indice posizionale.

- **Scandire il Risultato (`fetchone`):**

```
r = cursor.fetchone()
while r is not None: # Controlla se fetchone ha restituito una riga
    nome = r[0]
    print(f"Nome: {nome}")
    r = cursor.fetchone() # Legge la riga successiva
```

- **Chiudere la Connessione:** È importante chiudere la connessione per rilasciare le risorse sul database:

```
conn.close()
```

- **Gestione delle Transazioni:**
  - I DBMS relazionali supportano le transazioni (gruppi di operazioni atomiche).
  - **Comportamento Default (`psycopg2`):** Per impostazione predefinita, `psycopg2` inizia una transazione implicitamente alla prima query eseguita (SELECT, INSERT, UPDATE, DELETE). Questa transazione rimane aperta. Le modifiche (INSERT, UPDATE, DELETE) non diventano permanenti finché la transazione non viene esplicitamente "committata". Se la connessione viene chiusa senza commit, le modifiche vengono perse (rollback implicito).
  - **Commit Esplicito:** Per rendere permanenti le modifiche, si deve chiamare:

    ```
    conn.commit()
    ```

  - **Modalità Autocommit:** Si può impostare la connessione in modalità autocommit, dove ogni singola istruzione SQL viene eseguita e committata automaticamente in una sua transazione. Questo si fa *subito dopo* aver creato la connessione:

    ```
    conn.autocommit = True
    ```

    In questo modo, non è necessario chiamare `conn.commit()` dopo ogni modifica. Tuttavia, si perde la possibilità di raggruppare più operazioni in un'unica transazione atomica.
  - **Rollback Esplicito:** Se l'autocommit è disattivato (`conn.autocommit = False` , che è il default), si può annullare esplicitamente tutte le modifiche fatte dall'ultimo commit (o dall'inizio della transazione) chiamando:

    ```
    conn.rollback()
    ```

11.5 Esempio Complesso: Integrazione MongoDB e PostgreSQL

Viene presentato un esempio completo che integra le tecnologie viste:

- **Obiettivo:**
  i. Eseguire una query sulla tabella `Names` in PostgreSQL.
  ii. Prendere i risultati (righe) e inserirli come documenti in una collezione MongoDB.
  iii. Eseguire una nuova query sulla collezione MongoDB.
  iv. Produrre un documento JSON finale contenente i documenti estratti da MongoDB, salvandolo su file.
- **Fase 1: Connessioni:**
  - Importa `json`, `pymongo`, `psycopg2`.
  - Stabilisce la connessione a MongoDB (`myclient`, `mydb`, `mycol`).
  - Stabilisce la connessione a PostgreSQL (`conn`).
  - Imposta la connessione PostgreSQL in modalità **autocommit** (`conn.autocommit = True`) per semplicità in questo esempio.
- **Fase 2: Query SQL e Trasformazione Dati:**
  - Crea un cursore PostgreSQL (`cursor = conn.cursor()`).
  - Definisce ed esegue la query SQL: `SELECT "Name", "Age" FROM "Names";`.
  - Recupera tutti i risultati: `results = cursor.fetchall()`. `results` è una lista di tuple, es. `[('Pippo', 25), ('Pluto', 30)]`.
  - Crea una lista Python vuota `l`.
  - Itera sulla lista di tuple `results`. Per ogni tupla `r`:
    - Crea un nuovo dizionario `o`.
    - Popola il dizionario: `o["Name"] = r[0]`, `o["Age"] = r[1]`.
    - Appende il dizionario `o` alla lista `l`.
  - Alla fine, `l` sarà una lista di dizionari, es. `[{'Name': 'Pippo', 'Age': 25}, {'Name': 'Pluto', 'Age': 30}]`.
- **Fase 3: Inserimento in MongoDB:**
  - Conta e stampa i documenti *prima* dell'inserimento: `count1 = mycol.count_documents({})`.
  - Inserisce *tutti* i dizionari preparati nella lista `l` nella collezione MongoDB: `mycol.insert_many(l)`.
  - Conta e stampa i documenti *dopo* l'inserimento: `count2 = mycol.count_documents({})`.
  - Viene sottolineato l'uso di `count_documents` e `insert_many` (API `pymongo`).
- **Fase 4: Query in MongoDB con Proiezione:**
  - Crea un dizionario vuoto `out` e una lista vuota `l`.

    - Esegue una query `find` su MongoDB per recuperare tutti i documenti, ma **escludendo il campo `_id`** usando la proiezione: `result = mycol.find({}, {"_id": 0})`. Escludere `_id` è importante perché il suo tipo (`ObjectId`) non è direttamente serializzabile in JSON standard e spesso non è rilevante per l'output desiderato.
    - Itera sul cursore `result`. Per ogni documento `x` (che è un dizionario senza `_id`):
        - Appende `x` alla lista `l`.
    - Aggiunge la lista `l` al dizionario `out` con la chiave `"list"`: `out["list"] = l`.
- **Fase 5: Serializzazione JSON e Chiusura Connessioni:**
    - Apre un file JSON (`myjson.json`) in scrittura usando `open()` (si potrebbe usare `with` per maggiore sicurezza).
    - Serializza il dizionario `out` (che contiene la lista dei documenti MongoDB) nel file: `json.dump(out, fileout)`.
    - Chiude il file handler: `fileout.close()`.
    - Chiude la connessione PostgreSQL: `conn.close()`.
    - Chiude la connessione MongoDB: `myclient.close()`.