Python

Persistenza dei dati

Argomenti della lezione

- Gestione dei files
- Database & Python: MySQL
- Esercitazioni - CRUD su una semplice tabella “contatti”

Gestione dei files

I dati rappresentati dalle variabili di un programma in esecuzione risiedono nella RAM. Questa memoria è volatile e, quindi, quando il programma termina, o il computer viene spento, questi dati sono irrimediabilmente persi.

Un file serve a memorizzare dati in modo permanente su memoria di massa (hard disk, chiavetta USB, SSD,…) affinché siano disponibili per un uso successivo. I files sono archivi di dati organizzati in contenitori chiamati directory, o cartelle, secondo una struttura ad albero chiamata *file system*. Ogni file all'interno di una cartella è identificato da un nome univoco.

Python prevede la gestione di 2 tipologie di file: **testo** e **binario**.

Gestione di file di testo

Per permetterci di interagire con il filesystem, Python ci fornisce la funzione built-in `open()`.

La funzione `open()` accetta diversi argomenti ma i due più importanti sono il nome del file che vogliamo aprire e il modo di apertura; restituisce un **file object** tramite il quale è possibile effettuare diverse operazioni sul file (ad es. lettura/scrittura).

```
file_object = open("filename", "mode")
```

dove:

- **filename**: Il nome del file deve essere una stringa che rappresenta un percorso in grado di identificare la posizione del file nel filesystem. Il percorso può essere relativo alla directory corrente o assoluto

- **mode:** determina la modalità (lettura e/o scrittura) con cui vogliamo aprire il file. Il suo valore di default è la stringa 'r' (read).

Modalità	Descrizione
r	Apre un file di testo in lettura. Modo di apertura di default dei file.
w	Apre un file di testo in scrittura. Se il file non esiste lo crea, altrimenti cancella il contenuto del file.
a	Apre un file di testo in append. Il contenuto viene scritto alla fine del file, senza modificare il contenuto esistente.
x	Apre un file di testo in creazione esclusiva. Se il file esiste, restituisce un errore, altrimenti lo crea lo apre in scrittura.
r+	Apre un file di testo in modifica. Permette di leggere e scrivere contemporaneamente.
w+	Apre un file di testo in modifica. Permette di leggere e scrivere contemporaneamente. Cancella il contenuto del file.

Alcuni errori comuni coi file sono:

- operare su un file che è già stato chiuso;
- indicare in lettura o modifica il nome di un file non esistente (spesso sbagliando il suo nome o il suo percorso nel file system);
- lavorare con un file che si trova su una periferica che nel frattempo è stata rimossa;
- dimenticare di chiudere un file.

Per ovviare a questi problemi si dovrebbero includere le operazioni per gestire i file in blocchi **try...except**

Una volta ottenuto un riferimento al **file object** possiamo utilizzare i suoi metodi per interagire con il file

Metodo	Descrizione
read()	Legge e restituisce l'intero contenuto del file come una singola stringa.
read(n)	Legge e restituisce n caratteri (o byte).
readline()	Legge e restituisce una riga del file, spostando la posizione corrente alla riga successiva.
readlines()	Legge e restituisce l'intero contenuto del file come lista di righe (stringhe).
write(s)	Scrive nel file la stringa s e ritorna il numero di caratteri (o byte) scritti.
writelines(lines)	Scrive nel file la lista in righe lines
tell()	Restituisce la posizione corrente memorizzata dal file object.
seek(offset, pos)	Modifica la posizione corrente memorizzata dal file object.
flush()	Svuota i buffer di scrittura.
close()	Chiude il file.

```
import os.path
FILE_PATH = '/home/danilo/Documenti/contatti.txt'
if not os.path.isfile(FILE_PATH):
    contact_file = open(FILE_PATH, 'w') # Se il file non esiste lo creo
else:
    contact_file = open(FILE_PATH, 'a') # Se esiste lo apro in append
# Inserisco un contatto
identifier = input("Inserire un id:")
name = input("Inserire il nome:")
surname = input("Inserire il cognome:")
CONTACT_ROW = "{};{};{}\n"
contact_file.write(CONTACT_ROW.format(identifier, name, surname))
contact_file.close()
# Stampo la lista di contatti
contact_file = open(FILE_PATH, 'r')
print('ID', 'Nome', 'Cognome', sep='\t')
for line in contact_file:
    print(line.replace(';', '\t\t'))
contact_file.close()
```

Come si può notare nelle ultime righe di codice è stato utilizzato un ciclo for per scorrere e stampare le righe del file:

```
for line in contact_file:
    print(line.replace(';', '\t\t'))
```

Lo stesso risultato si potrebbe ottenere con **`readline()`** che, chiamato consecutivamente, restituisce tutte le righe una alla volta. Alla fine del file **`readline()`** restituisce una stringa vuota ‘’.

Inoltre è possibile ottenere un oggetto di tipo lista con tutte le righe del file invocato il metodo **`readlines()`.**

Come abbiamo visto è sempre necessario chiudere il file una volta terminate le operazioni in modo da non renderlo inaccessibile ad altri processi/funzioni. Per automatizzare questa operazione è possibile utilizzare il costrutto **`with`.**

```
contact_file = open(FILE_PATH, 'r')
with contact_file:
        print('ID', 'Nome', 'Cognome', 'cellulare', 'email', sep='\t\t')
        for line in contact_file:
            print(line.replace(';', '\t\t'))
        # contact_file.close() -> non c'è bisogno di invocarlo perché
verrà
        # chiuso automaticamente alla fine del costrutto with
        # grazie all'invocazione automatica di contact_file.__exit__()
```

`With` può essere utilizzato con tutti quegli oggetti che supportano il protocollo dei context manager.

All'apertura del blocco di codice viene invocato il metodo `contact_file.__enter__()` (l'object file in questo caso non fa nulla), mentre al termine del blocco di codice viene invocato il metodo `contact_file.__exit__()` nel quale viene chiuso il file.

Database & Python: MySQL

Per poter accedere ad un database MySQL in Python dobbiamo è necessario installare il driver “MySQL Connector”:

pip install mysql-connector-python

Al seguente indirizzo è possibile trovare la documentazione ufficiale del driver per mysql che utilizzeremo:

https://dev.mysql.com/doc/connector-python/en/connector-python-introduction.html

Apertura di una connessione verso un database MySQL:

```
import mysql.connector

config = {
  'user': 'root',
  'password': 'password',
  'host': '127.0.0.1',
  'database': contacts
}

cnx = mysql.connector.connect(**config)

cnx.close()
```

Naturalmente lo statement di apertura della connessione dovrebbe essere inserito all'interno di un costrutto try...except per gestire eventuali errori di connessione:

```
import mysql.connector
from mysql.connector import errorcode
try:
  cnx = mysql.connector.connect(**config)
except mysql.connector.Error as err:
  if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
    print("Something is wrong with your user name or password")
  elif err.errno == errorcode.ER_BAD_DB_ERROR:
    print("Database does not exist")
  else:
    print(err)
else:
  cnx.close()
```

La seguente tabella riporta una lista dei principali parametri che possono essere utilizzati in fase di apertura della connessione:

Nome Argomento	Default	Description
user		
password		
database		
host	127.0.0.1	
port	3305	
charset	UTF-8	
autocommit	False	
pool_size	5	

Per la lista completa dei parametri che possono essere utilizzati per l'apertura della connessione potete far riferimento alla documentazione ufficiale presente al link:

https://dev.mysql.com/doc/connector-python/en/connector-python-connectargs.html

Gestione delle transazione

Il valore di default per il parametro **autocommit** è False, in questo modo le transazioni non sono automaticamente committate. Per effettuare la commit di una transazione è necessario, dopo aver effettuato le dovuto operazioni di insert, update o delete, invocare il metodot commit() dell'istanza di MySQLConnection.

Si seguito sono riportati una serie di snippet di codice con esempi sull'interazione con il database MySQL tramite il driver.

Un server MySQL può gestire più database. Generalmente il database da utilizzare viene selezionato in fase di apertura della connessione con l'attributo **database** ma è possibile crearne uno nuovo o selezionarne uno esistente dopo aver aperto la connessione.

Creazione di un database di nome DB_NAME

```
try:
    cursor = cnx.cursor()
    cursor.execute(
        "CREATE DATABASE {} DEFAULT CHARACTER SET 'utf8'".format(DB_NAME))
except mysql.connector.Error as err:
    print("Failed creating database: {}".format(err))
    exit(1)
```

Selezione di un database di nome DB_NAME

```
try:
    cursor.execute("USE {}".format(DB_NAME))
except mysql.connector.Error as err:
    print("Database {} does not exists.".format(DB_NAME))
    if err.errno == errorcode.ER_BAD_DB_ERROR:
        create_database(cursor)
        print("Database {} created successfully.".format(DB_NAME))
        cnx.database = DB_NAME
    else:
        print(err)
        exit(1)
```

Inizializzazione di un dizionario con le DDL per la creazione delle tabelle

```
TABLES = {}
TABLES['contacts'] = (
    "CREATE TABLE contacts ("
    "  id int(11) NOT NULL AUTO_INCREMENT,"
    "  birth_date date NOT NULL,"
    "  first_name varchar(14) NOT NULL,"
    "  last_name varchar(16) NOT NULL,"
    "  gender enum('M','F') NOT NULL,"
    "  phone varchar(16) NOT NULL,"
    "  email varchar(100) NOT NULL,"
    "  PRIMARY KEY (id)"
    ") ENGINE=InnoDB")
```

Creazione di una serie di tabelle a partire da un dizionario contenente le definizioni

```
for table_name in TABLES:
    table_description = TABLES[table_name]
    try:
        print("Creating table {}: ".format(table_name), end='')
        cursor.execute(table_description)
    except mysql.connector.Error as err:
        if err.errno == errorcode.ER_TABLE_EXISTS_ERROR:
            print("already exists.")
        else:
            print(err.msg)
    else:
        print("OK")
cursor.close()
cnx.close()
```

Inserimento di un record all'interno della tabella contacts

```
from datetime import date, datetime, timedelta
data_contact = {
  'birth_date': date(1985, 3, 7),
  'first_name': 'Danilo',
  'last_name': 'Di Nuzzo',
  'phone': '3776900129',
  'email': 'danilo.dinuzzo@r-dev.it',
}
add_contact = ("INSERT INTO contacts "
               "(birth_date, first_name, last_name, phone, email) "
               "VALUES (%(birth_date)s, %(first_name)s, %(last_name)s, %(phone)s,
%(email)s)")
cursor = cnx.cursor()
cursor.execute(add_contact, data_contact)
cnx.commit()
cursor.close()
```

ATTENZIONE: nel caso in cui la connessione sia stata aperta con il valore di default per l'autocommit (False) l'istruzione **cnx.commit()** è necessaria per rendere effettive le modifiche. Nel sia necessario effettuare una roll back è possibile utilizzare il metodo **rollback()**.

```
import datetime
cursor = cnx.cursor()
query = ("SELECT first_name, last_name, phone FROM contacts "
        "WHERE first_name = %s")
cursor.execute(query, ('Danilo',)) # ATTENZIONE quando passate un solo parametro
for first_name, last_name, phone in cursor:
    print('Nome: {}, Cognome: {}, Telefono: {}'.format(first_name, last_name, phone))
cursor.close()
```

Esercitazioni - Crud su una semplice tabella contatti

Proviamo, con le competenze acquisite fino ad ora, a creare un programma in Python per la gestione di una rubrica interattiva.

Si richiede di implementare l'applicazione con un menu interattivo da console cercando di sfruttare tutte le competenze acquisite nel corso.

La nostra applicazione dovrà sfruttare un database MySQL per la gestione della persistenza. Inoltre si chiede di implementare un modulo che, a partire da un file di testo contenente lo script SQL per la creazione delle tabelle, all'avvio dell'applicazione crei il database e le relative tabelle se non presente.

Infine implementare la funzionalità di import/export su file di testo (txt con organizzazione dei dati a vostro piacere).