03_SQLite

March 16, 2020

1 Acesso a bases de dados SQLite com Python

Pedro Cardoso
(ISE/UAlg - pcardoso@ualg.pt)

1.1 Pre-requisitos

1.1.1 Conectores instalados

Verificar que todos os módulos necessários estão instalados, nomeadamente, pode de ter de instalar o `sqlite3` [https://docs.python.org/2/library/sqlite3.html]
(Dependendo do seu sistema poderá ter de substituir pip3 por pip, pip3.7, pip3.8, . . .)

```
In [ ]: !pip3 install pysqlite3
```

1.2 Estabelecimento de conexão à base de dados usando um Connector/Python

Criar a conexão usando o método `connect`, que tem como parâmetro o caminho para o ficheiro que contém a base de dados

```
In [ ]: import sqlite3

        # se não existir a base de dados (ficheiro) exemplo.db será criada
        conn = sqlite3.connect('exemplo.db')

        conn.close()
```

1.3 Criação de uma base de dados

Para a criação das tabelas e relacionamentos podemos construiro o sql ou, como alternativa, podemos usar ferramentas como sejam o MySQL Workbench, o Phpmyadmin, o SQlite Browser, o DataGrip, etc.
Consideremos o caso em que contruímos o sql. . .

```
In [ ]: sql = '''
        create table Location
        (
            idLocation integer
                constraint Location_pk
```

```
            primary key autoincrement,
    name TEXT not null,
    description text not null
);


create table Unit
(
    unit text
        constraint Unit_pk
            primary key,
    description text not null
);

create table Sensor
(
    idSensor integer
        constraint Sensor_pk
            primary key autoincrement,
    idLocation integer not null
        constraint Sensor_Location_idLocation_fk
            references Location
                on update cascade on delete cascade,
    name text not null,
    unit text not null
        constraint Sensor_Unit_unit_fk
            references Unit
                on update cascade on delete cascade
);

create table Reading
(
    idReading integer
        constraint Reading_pk
            primary key autoincrement,
    idSensor integer
        constraint Reading_Sensor_idSensor_fk
            references Sensor
                on update cascade on delete cascade,
    timestamp datetime default CURRENT_TIMESTAMP,
    value real not null
);


create table Alert
(
    idAlert integer
        constraint Alert_pk
```

```
                primary key autoincrement,
            idSensor integer
                constraint Alert_Sensor_idSensor_fk
                    references Sensor
                        on update cascade on delete cascade,
            description text not null,
            cleared integer

        )
        '''
```

```
In [ ]: conn = sqlite3.connect('sensors.db')

        cursor = conn.cursor()

        # executescript is a nonstandard convenience method for executing multiple SQL stateme
        cursor.executescript(sql)

        cursor.close()
        conn.close()
```

1.4 Operações CRUD

1.4.1 `INSERT`

Aberta a conexão em sqlite

```
In [ ]: cnx = sqlite3.connect('sensors.db')
        cursor = cnx.cursor()
```

inserir uma nova localização na base de dados e obter o id correspondente

```
In [ ]: # prepare the sql query for the new location
        sql = '''
        INSERT INTO location
            (name, description)
        VALUES
            (?, ?)
        '''

        data = ('Prometheus Server', 'Prometheus Server @ lab. 163 / ISE /UAlg')

        #execute the sql query and get the new location id
        cursor.execute(sql, data)

        location_id = cursor.lastrowid
        location_id
```

Quando estamos a usar um sistema transacional, temos de efetuar o `commit` depois de fazer um `INSERT`, `DELETE`, ou `UPDATE`.

Note-se que: - o `commit` confirma a transação atual. Se não se chamar, tudo o que fez desde a última chamada do `commit()` não será visível às outras conexões. - quando a BD é acedida por várias conexões e um dos processos modifica-a, a BD SQLite fica bloqueada até que a transação seja confirmada (*commited*). - podemos desfazer as alterações desde o último `commit` chamando o método `rollback()`

```
In [ ]: cnx.commit()
```

Inserir uma nova `Unit`

```
In [ ]: sql = '''
        INSERT INTO Unit
            (unit, description)
        VALUES
            ("percent", "percentage of usage")
        '''

        cursor.execute(sql)

        cnx.commit()
```

Inserir um novo sensor e obter o seu id: - preparar o sql, note-se os *placeholders* com nome usados neste caso - preparar os dados - executar o query

```
In [ ]: sql = '''INSERT INTO `sensor` (`idLocation`, `name`, `unit`)
                VALUES (:idLocation, :name, :unit);'''

        data = {
                'idLocation': location_id,
                'name' : 'cpu_sensor_01',
                'unit' : 'percent'
               }

        cursor.execute(sql, data)
        sensor_id = cursor.lastrowid
        cnx.commit()
        sensor_id
```

E agora, obter alguns dados e enviar para a base de dados

```
In [ ]: import psutil

        sql = '''
        INSERT INTO reading
            (idSensor, value)
        VALUES
            (:idSensor, :value)
        '''
```

```
        for _ in range(20):
            data = {
                   'idSensor' : sensor_id,
                    'value' : psutil.cpu_percent(interval=1)
                   }
            cursor.execute(sql, data)
            cnx.commit()
            print('.', end='')
```

```
In [ ]: cursor.close()
        cnx.close()
```

1.5 Selecionar dados

```
In [ ]: import sqlite3

        cnx = sqlite3.connect('sensors.db')
        cursor = cnx.cursor()
```

```
In [ ]: sql = '''
        SELECT idLocation, name, description
        FROM location
        WHERE description LIKE "%163%"'''

        cursor.execute(sql)

        for (idLocation, name, description) in cursor:
            print("id: {}\n\t name: {} \n\t description: {}".format(idLocation, name, descripti
```

```
In [ ]: sql = '''
        SELECT idReading, idSensor, timestamp, value
        FROM reading
        WHERE value BETWEEN ? and ?
        '''
        data = (5, 50)

        cursor.execute(sql, data)

        for (idReading, idSensor, timestamp, value) in cursor:
            print("idReading: {}\n\t idSensor: {} \n\t time: {} \n\t value: {}".format(idReadi
```

```
In [ ]: sql = '''
            select *
            from Location
                inner join Sensor S on Location.idLocation = S.idLocation
                inner join Unit U on S.unit = U.unit
                inner join Reading R on S.idSensor = R.idSensor
            where value between :low and :high
            order by value
```

```
        '''

        data = {
            'low': 5,
            'high': 40
        }

        cursor.execute(sql, data)
```

Podemos obter os nomes das colunas

```
In [ ]: cursor.description
```

```
In [ ]: lista_de_colunas = [linha[0] for linha in cursor.description]
        lista_de_colunas
```

```
In [ ]: for linha in cursor:
            print('\t'.join([f'|{coluna}: {valor}' for valor, coluna  in zip(linha, lista_de_c
```

Usando o comando `fetchall` podemos obter todos os resultados de uma única vez como uma lista de tuplos

```
In [ ]: # é necessario voltar a correr o select pois o cursor foi esvaziado
        cursor.execute(sql, data)

        cursor.fetchall()
```

Podemos também converter para um dicionário mas **nosso caso NÃO é boa ideia** pois duas colunas "têm o mesmo nome" (e.g., `nome`), pelo que se perdem colunas.

```
In [ ]: # é necessario voltar a correr o select pois o cursor foi esvaziado
        cursor.execute(sql, data)

        for linha in cursor:
            print({coluna: valor for valor, coluna  in zip(linha, lista_de_colunas)})
```

```
In [ ]: cursor.close()
        cnx.close()
```

```
In [ ]:
```