MariaDB connector

MariaDB on suosittu avoimen lähdekoodin relaatiotietokanta, joka on kehitetty MySQL-tietokannan pohjalta. MariaDB tarjoaa tehokkaan ja skaalautuvan ratkaisun tietokantojen hallintaan, ja se tukee monia ominaisuuksia, kuten ACID-yhteensopivuutta, replikaatiota ja laajennettavuutta.

MariaDB:n käyttö Python-ohjelmissa edellyttää MariaDB connectorin asentamista. Tämä connector mahdollistaa yhteyden muodostamisen MariaDB-tietokantoihin ja SQL-kyselyiden suorittamisen Pythonin kautta. Tämä connector toimii samalla tavalla kuin SQLite-moduuli, mutta se on suunniteltu erityisesti MariaDB-tietokantojen kanssa työskentelyyn.

MariaDB connectorin asentaminen

MariaDB connectorin asentaminen onnistuu helposti pip-paketinhallinnan avulla. Komentorivissa suoritetaan seuraava komento:

```
pip install mariadb
```

Nykyään kuitenkin suosittu tapa on käyttää uv-pakettihallinnan ohjelmaa:

```
uv add mariadb
```

Viralliset modulin ohjeet: https://mariadb-corporation.github.io/mariadb-connector-python/index.html

Yhteyden muodostaminen MariaDB-tietokantaan

MariaDB:n tietokantahallintajärjestelmään yhdistäminen on vähän monimutkaisempia kuin SQLiten tapauksessa:

```
import mariadb

conn = mariadb.connect(
    user = 'db_user',
    password = 'salasana',
    host = 'localhost',   # tai IP/DNS
    port = 3306,          # Tämä on MariaDB/MySQL:n oletusportti. Ei tarvitse käyttää sitä jos se on oletusportti.
    database = 'opetuskanta',
    autocommit = False     # Suositus: hallitse transaktiot itse. Tämä on kuitenkin oletusarvo.
)
```

Aika usein kuitenkin tätä kirjoitetaan käyttämällä sanakirjaa:

```
import mariadb

options = {
    user: 'db_user',
    password: 'salasana',
    host: 'localhost',   # tai IP/DNS
    port: 3306,          # Tämä on MariaDB/MySQL:n oletusportti. Ei tarvitse käyttää sitä jos se on oletusportti.
    database: 'opetuskanta',
    autocommit: False     # Suositus: hallitse transaktiot itse. Tämä on kuitenkin oletusarvo.
}

conn = mariadb.connect(**options)
```

Kursori

MariaDB:n kursori toimii samalla tavalla kuin SQLiten kursori, mutta sillä on jotain eri attribuuttia ja metodia.

```
cursor = conn.cursor()
```

Kyselyt suoritetaan samalla tavalla:

```
cursor.execute('SELECT * FROM table_name')
```

Ja on myös samat metodit käsittelemään tuloksia:

- `cursor.fetchone()`
- `cursor.fetchmany()`
- `cursor.fetchall()`

Kuitenkin on mahdollista huomata eroa. Tulokset käyttävät Python tietotyyppiä tarkemmin. Esimerkiksi päivämäärät ovat Pythonin `date` ja tarkat lukumäärät ovat Pythonin `Decimal`.

Viralliset ohjeet: https://mariadb-corporation.github.io/mariadb-connector-python/cursor.html

Tulokset sanakirjoina

MariaDB Connector/Python tukee sanakirjamaisia tuloksia helposti. **cursorin** luodessa lisätään parametri `dictionary=True`. Tällöin jokainen rivi palautetaan **dict**-muodossa, jossa avaimet ovat sarakkeiden nimet.

Esimerkki:

```
import mariadb

conn = mariadb.connect(
    user="db_user",
    password="salasana",
    host="localhost",
    database="opetuskanta"
)

cursor = conn.cursor(dictionary=True)  # HUOM: tämä tekee rivit dict-muotoon

cursor.execute("SELECT id, nimi, email FROM kayttajat LIMIT 3")

for row in cursor:
    print(row)  # esim. {'id': 1, 'nimi': 'Matti', 'email': 'matti@example.com'}

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

Kyselyn parametria

Samalla tavalla kuin SQLiten connector, MariaDB:n connector tukee parametria SQL kyselyissä, mutta MariaDB hyväksy `%s` paikkamerkkinä, `?` paikkamerkin lisäksi.

```
import mariadb
conn = mariadb.connect(host="localhost", user="app", password="salasana", database="appdb")
cur = conn.cursor()
cur.execute("INSERT INTO users(name, age) VALUES (%s, %s)", ("Ada", 37))
conn.commit()
```

ORM

ORM tarkoittaa **Object-Relational Mapping** eli olio-relaatiomäppäystä. Se on tekniikka, jolla yhdistetään ohjelmointikielen oliot ja relaatiotietokannan taulut niin, että tietokantakyselyt voidaan tehdä olioiden avulla ilman suoraa SQL-koodia. Eli:

- Muuntaa tietokannan rivit **olioiksi** ohjelmassa.
- Hoitaa **CRUD-operaatiot** (Create, Read, Update, Delete) automaattisesti.
- Piilottaa SQL:n yksityiskohdat, jolloin kehittäjä voi keskittyä liiketoimintalogiikkaan.

Hyödyt:

- Vähemmän käsin kirjoitettua SQL:ää.
- Koodin ylläpito helpottuu.
- Tietoturva paranee (vähemmän riskiä SQL-injektioille, kun käytetään ORM:n parametrisointia).

Haitat:

- Vähemmän kontrollia SQL-tason optimoinnista.
- ORM voi olla hitaampi kuin käsin optimoitu SQL.
- Monimutkaiset kyselyt voivat olla hankalia ORM:n kautta.

Suosittuja Python ORM-kirjastoja:

- **SQLAlchemy** (yleisin ja tehokas)
- **Django ORM** (sisäänrakennettu Django-sovelluskehykseen)
- **Peewee** (kevyt ORM)
- **Tortoise ORM** (async-tuki)
- **SQLModel** (yksinkertainen ja helppokäyttöinen)

SQLAlchemy-esimerkki:

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.orm import declarative_base, sessionmaker

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    age = Column(Integer)

engine = create_engine('sqlite:///app.db')
Base.metadata.create_all(engine)

Session = sessionmaker(bind=engine)
session = Session()

# Lisää käyttäjä
new_user = User(name="Ada", age=37)
session.add(new_user)
session.commit()
```