SQLite

`SQLite`:a käytetään Pythonissa tuomalla samannimisen moduuli (+versio 😀):

```
import sqlite3
```

Voidaan sen jälkeen luoda uusi tietokanta tai avata olemassaoleva tietokanta levystä:

```
conn = sqlite3.connect('tiedosto.db')
```

SQLite mahdollista myös luoda tilapainen tietokanta muistiin:

```
conn = sqlite3.connect(':memory:')
```

Seuravaavaksi luodaan kursori, jonka avulla SQL -kyselyjä suoritetaan:

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

Kursorin käyttö on tyyppillinen tapa käyttää tietokantoja yleisesti. Sitä käytetään eri tietokantajärjestelmissä (esim. MySQL, PostgreSQL, MariaDB, ...). Sen avulla saadaan suoritettua SQL -kyselyjä, haettua tuloksia ja käsiteltyä niitä.

SQL -kysely suoritetaan `execute` -metodilla:

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

Huomattava, että SQL -kyselyä ei tarvitse päättää puolipisteellä `;`.

Kursorilla on 3 metodia lukemaan tuloksia `SELECT` -kyselystä:

- `cursor.fetchone()`: Jos `cursor.row_factory` on None, palauttaa seuraavan tuloksen rivin **tuplena** (oletus). Jos `cursor.row_factory` on määritelty, välitetään rivin sille ja palautetaan sen tulos. Kursori siirtyy seuraavaan riviin. Metodi palauttaa `None`, jos tietoja/rivejä ei ole enää saatavilla.

```
while row := cursor.fetchone():
    print(row)  # Tulosta rivin: ('arvo1', 'arvo2', 'arvo3', ...)
```

- `cursor.fetchmany(n)`: Palauttaa **lista**, joka sisältää eniteen *n* riviä **tuplena**. *n*-parametrin oletusarvo on `cursor.arraysize`-arvo. Jos ei ole enää riviä saatavilla, se palauttaa **tyhjä lista**.

```
for row in cursor.fetchmany(5):
    print(row)  # Tulosta 5 riviä/tuplea
```

- `cursor.fetchall()`: Palauttaa **lista**, joka sisältää jäljellä olevat rivit **tuplena**. Jos ei ole enää riviä saatavilla, se palauttaa **tyhjä lista**.

```
for row in cursor.fetchall():
    print(row)
```

Huomattava, että rivit palautetaan tuplena vaikka kyseessä olisi vain yksi sarake:

```
cursor.execute("SELECT name FROM sqlite_master WHERE type='table'")
for row in cursor.fetchall():
    print(row)
```

`sqlite_master` on SQLiten sisäinen taulukko, joka sisältää tietoja luoduista rakenteista.

Tulokset sanakirjana

Voidaan saada SQLite-kyselyn tulokset sanakirjoina (eli avain–arvo-pareina) käyttämällä `row_factory` -ominaisuutta. Eli:

```
import sqlite3

# Luo yhteys ja ota käyttöön Row-factory
conn = sqlite3.connect("app.db")
conn.row_factory = sqlite3.Row  # Tämä tekee rivit sanakirjamaisiksi
cursor = conn.cursor()

# Kysely
cursor.execute("SELECT id, nimi, ika FROM henkilöt")

# Tulostetaan rivit dict-muodossa
for row in cursor.fetchall():
    print(dict(row))  # esim. {'id': 1, 'nimi': 'Aino', 'ika': 30}

conn.close()
```

`sqlite3.Row` palauttaa rivit **oliona**, joka toimii kuin sanakirja: avaimet = sarakkeiden nimet ja `dict(row)` muuntaa sen tavalliseksi Python-sanakirjaksi.

Vaihtoehtoksi voidaan luoda oma `row_factory` funktiota:

```
def dict_factory(cursor, row):
    return {col[0]: row[idx] for idx, col in enumerate(cursor.description)}

conn = sqlite3.connect("app.db")
conn.row_factory = dict_factory
cursor = conn.cursor()

cursor.execute("SELECT id, nimi, ika FROM henkilöt")
print(cursor.fetchall())  # lista sanakirjoja
```

SQL Injection

SQL-injektion välttäminen on erittäin tärkeää, ja Pythonin `sqlite3`-moduuli tarjoaa siihen valmiin ratkaisun: **parametrisoidut kyselyt**.

Mitä on SQL-injektio?

Jos rakennetaan SQL-lauseen suoraan käyttäjän syötteestä, esim.:

```
nimi = "Liisa"
cursor.execute(f"SELECT * FROM henkilöt WHERE nimi = '{nimi}'")
```

Jos käyttäjä syöttää `Liisa' OR 1=1 --`, kysely muuttuu:

```
SELECT * FROM henkilöt WHERE nimi = 'Liisa' OR 1=1 --'
```

palauttaa kaikki rivit (tai pahempaa: voi tuhota tietokannan).

Oikea tapa: parametrisoidut kyselyt

Käytetään **paikkamerkkejä (`?`)** ja annetaan arvot erillisenä **tuplena** (vaikka olisi vain yksi parametri):

```
nimi = "Liisa"
cursor.execute("SELECT * FROM henkilöt WHERE nimi = ?", (nimi,))
```

`?` -merkin yhteydessä ei käytetä hipsua. Tärkeä idea on kuitenkin, ettei koskaan laadita SQL-kyselyn vain yhdistämällä käyttäjän syötettä omaan koodiin. `sqlite3` huolehtii arvon oikeasta escapesta ja tyypityksestä.

Toinen esimerkki:

```
cursor.execute("INSERT INTO henkilöt (nimi, ika) VALUES (?, ?)", ('Aino', 30))
```

On myös mahdollista käyttää sanakirjaa parametreille:

```
cursor.execute("INSERT INTO henkilöt (nimi, ika) VALUES (:name, :age)", {'name': 'Aino', 'age': 30})
```

Huomattava on:

- Käytetään `?`-paikkamerkkejä (SQLite) aina kuin mahdollista.
- Muissa TKHJ:ssa voidaan käyttää jotain muuta merkkiä, kuten `%s`.
- Ei luoteta käyttäjän syötteeseen – Ei koskaan lisätä sitä suoraan SQL-lauseeseen.
- `?` voidaan käyttää vain sijainneissa, jossa vaaditaan arvoja. Jos käyttäjän syöte pitää käyttää jossain muussa, olisi hyvää validoida sitä vaika whitelistillä. Esim:

```
allowed_columns = {"nimi", "ika", "id"}
sarake = "nimi"  # käyttäjän syöte

if sarake not in allowed_columns:
    raise ValueError("Virheellinen sarake")

cursor.execute(f"SELECT {sarake} FROM henkilöt")
```

Aiheeseen liittyvä 😃: https://xkcd.com/327/

Yhteys ja kursorin elinkaari

Kursorit luodaan yhdelle loogiselle operaatiojoukolle (kyselyt + haut). Niitä pitää sulkea kun ei enää tarvitse. Yhteyden sulkeminen sulkee taustalla myös kursorit.

```
cursor.close()
```

Toisaalta yhteys jätetään auki siihen asti ettei tietokantaa enää tarvita. Eli tyyppillisesti yhteys avataan ja suljetaan vain kerran. Yhteys suljetaan samalla tavalla kuin kursoria:

```
conn.close()
```

Yhteyden (`conn`) sulkeminen sulkee myös siihen liittyvät kursorit, mutta **on silti suositeltavaa sulkea kursori erikseen**, jos se on ollut aktiivisessa käytössä.

Committia

Commit tarkoittaa tietokantatermein **transaktion vahvistamista**. Kun tehdään muutoksia tietokantaan (esim. `INSERT` , `UPDATE` , `DELETE` , `CREATE TABLE`), ne eivät tallennu pysyvästi heti, vaan jäävät avoimeen *transaktioon*.

`conn.commit()` tekee seuraavaa:

- **Kirjoittaa kaikki avoimen transaktion muutokset pysyvästi tietokantaan**.
- Vapauttaa lukot, jotta muut prosessit voivat käyttää tietokantaa.
- Varmistaa, että muutokset eivät katoa, jos yhteys suljetaan.

```
conn.commit()  # Tallentaa muutokset
conn.close()
```

Jos kursori tai yhteys suljetaan ilman committia, oletuksena mutokset **häviävät**!.

Operaatiot, jotka vaativat `commit()` :

- **INSERT** – lisää rivejä tauluihin
- **UPDATE** – muuttaa olemassa olevia rivejä
- **DELETE** – poistaa rivejä
- **CREATE TABLE**, **ALTER TABLE**, **DROP TABLE** – skeeman muutokset
- **PRAGMA**-komennot, jos ne muuttavat asetuksia (esim. journal_mode)

Operaatiot, jotka **eivät vaadi committia**:

- **SELECT** – pelkkä lukeminen ei muuta tietokantaa
- PRAGMA-komennot, jotka vain lukevat asetuksia

Rollback

`ROLLBACK` tarkoittaa transaktion **peruuttamista**. Kun tehdään muutoksia tietokantaan (INSERT, UPDATE, DELETE), ne tapahtuvat ensin avoimessa transaktiossa. Jos `rollback()` kutsutaan, kaikki **vahvistamattomat muutokset** kumotaan ja tietokanta palautuu tilaan, jossa se oli ennen transaktion alkua.

Tyyppillisesti tätä käytetään:

- Jos tapahtuu virhe ennen `commit()`.
- Jos syystä tai toiseesta halutaan perua kaikki muutokset, joita ei ole vielä vahvistettu.

```
import sqlite3

conn = sqlite3.connect("app.db")
cursor = conn.cursor()

try:
    cursor.execute("INSERT INTO henkilöt(nimi, ika) VALUES (?, ?)", ("Aino", 30))
    cursor.execute("INSERT INTO henkilöt(nimi, ika) VALUES (?, ?)", ("Matti", 41))
    raise ValueError("Simuloitu virhe!")  # Jokin meni pieleen
    conn.commit()  # Ei koskaan saavuteta
except Exception as e:
    print("Virhe:", e)
    conn.rollback()  # Peruuttaa molemmat INSERTit
finally:
    conn.close()
```

Autocommit

Autocommit on tila, jossa **ei ole avointa transaktiota**. Jokainen muutos **vahvistuu heti** SQL-lauseen päätyttyä, ellei ole nimenomaisesti aloitettu `BEGIN`-transaktiota.

SQLite + Python (`sqlite3`) tekee tämän helpoksi säätämällä `autocommit`-parametrin yhteydessä:

- **Autocommit PÄÄLLÄ**: `sqlite3.connect(..., autocommit=True)`
  - Ei avointa transaktiota: jokainen `INSERT/UPDATE/DELETE/DDL` tallentuu heti.

    - Jos transaktiota halutaan, pitää se aloitta **eksplisiittisesti**: `conn.execute("BEGIN")` → `conn.commit()` / `conn.rollback()`.
- **Autocommit POIS** *(oletus)*: `sqlite3.connect("app.db")` tai `sqlite3.connect("app.db", autocommit=False)`
    - Python avaa transaktion **automaattisesti** ensimmäisestä kirjoittavasta komennosta (oletuksena DEFERRED-tilassa).
    - Vaatii `conn.commit()` (tai `with`-lohko hoitaa automaattisesti).

Transaktion tila voidaan tarkista:

```
conn.in_transaction  # True/False
```

Kontekstinhallinnan käyttö

Kontekstinhallinta `with`-lauseella on kätevä tapa hallita transaktioita SQLite:ssä Pythonissa. Kun käytetään `with conn:`, Python hoitaa automaattisesti transaktion aloituksen ja lopetuksen:

```
import sqlite3
conn = sqlite3.connect("app.db")
with conn:
    cursor = conn.cursor()
    cursor.execute("INSERT INTO henkilöt (nimi, ika) VALUES (?, ?)", ("Aino", 30))
# Tässä `with conn:` -lohko hoitaa automaattisesti commitin tai rollbackin.
```

Etu kotekstinhallinnasta on, **ettei tarvitse tehdä committia tai rollbackia erikseen**, niitä tehdään myös automaattisesti. Jos ei tullut virheitä, `with` päättymiseen tehdään committia. Jose `with` päätyy virheeseen, tehdään rollbackia.

Pragmat

PRAGMA on SQLite-tietokannan erityinen komento, jolla voit lukea tai muuttaa **tietokannan asetuksia ja sisäisiä ominaisuuksia**. Se ei ole osa standardia SQL:ää, vaan SQLite-spesifinen laajennus.

PRAGMA voidaan:

- Säätää **käyttäytymistä** (esim. transaktiot, lukitus, journalointi)
- Palauttaa **tietoa tietokannasta** (esim. taulujen skeema, indeksit)
- Käytetään usein optimointiin, debuggaamiseen tai tietoturva-asetuksiin

Tyyppillinen PRAGMA-komento on:

```
PRAGMA foreign_keys = ON;
```

koska SQLite:ssa ei ole oletuksen *Foreign key*-tuki päällä.

Sitä käytettään tavallisella tavalla:

```
cursor.execute('PRAGMA foreign_keys = ON')
```

Lisää esimerkkiä:

- Journal mode (parempi rinnakkaisuus):

  ```
  PRAGMA journal_mode = WAL;
  ```

  Vaihtaa Write-Ahead Logging -tilaan (nopeampi ja turvallisempi monikäyttö).
- Näyttää tietokannan taulut:

  ```
  cursor.execute("PRAGMA table_info(henkilöt)")
  ```

  Palauttaa sarakkeiden tiedot taulusta `henkilöt`.
- Näyttää indeksit:

  ```
  PRAGMA index_list('henkilöt');
  ```

- Näyttää tietokannan koko:

  ```
  PRAGMA page_size;
  PRAGMA page_count;
  ```

  Varsinainen koko voidaan laskea: `page_size * page_count`.

`cursor.lastrowid`

`cursor.lastrowid` on SQLite-kursorin ominaisuus, joka kertoo **viimeksi lisätyn rivin automaattisen ID:n** (eli arvon, joka annettiin `INTEGER PRIMARY KEY AUTOINCREMENT` -sarakkeelle).

Kun tehdään `INSERT` - tai `REPLACE` -operaation tauluun, jossa on automaattinen ID, voidaan heti sen jälkeen hakea uuden rivin tunnisteen:

```
import sqlite3

conn = sqlite3.connect("app.db")
cursor = conn.cursor()

cursor.execute("""
CREATE TABLE IF NOT EXISTS henkilöt (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    nimi TEXT,
    ika INTEGER
)
""")

cursor.execute("INSERT INTO henkilöt (nimi, ika) VALUES (?, ?)", ("Aino", 30))
print("Uuden rivin ID:", cursor.lastrowid)  # esim. 1

conn.commit()
conn.close()
```

Jos taulussa ei ole `INTEGER PRIMARY KEY AUTOINCREMENT` -saraketta, voi palauttaa `None`.

`cursor.rowcount`

`cursor.rowcount` kertoo **kuinka monta riviä viimeisin SQL-operaatio vaikutti**:

- **INSERT, UPDATE, DELETE**: kuinka monta riviä lisättiin, päivitettiin tai poistettiin.
- **SELECT**: monissa tietokannoissa palauttaa rivien määrän, mutta **SQLite:ssa** se on **aina -1** SELECT-kyselyille, koska SQLite ei laske rivejä etukäteen.

```
import sqlite3

conn = sqlite3.connect("app.db")
cursor = conn.cursor()

cursor.execute("UPDATE henkilöt SET ika = ika + 1 WHERE ika >= ?", (30,))
print("Muokattuja rivejä:", cursor.rowcount)  # esim. 2

cursor.execute("DELETE FROM henkilöt WHERE ika < ?", (20,))
print("Poistettuja rivejä:", cursor.rowcount)  # esim. 0

cursor.execute("SELECT * FROM henkilöt")
print("SELECT rowcount:", cursor.rowcount)  # SQLite: -1
```