Національний університет біоресурсів і
природокористування України

Робота з базами даних у Python: SQLite та PostgreSQL. Підключення, отримання та запис даних.

Частина 2

Лектор: к.ф.-м.н., доцент Кириченко Віктор Вікторович

Огляд лекції

SQLite

- Огляд та можливості
- Підключення та створення таблиць
- Основні операції з даними

PostgreSQL

- Архітектура та переваги
- Встановлення та налаштування
- Робота з віддаленою БД

Робота з даними

- Виконання SQL-запитів
- Параметризовані запити
- Обробка результатів

Безпека та оптимізація

- Транзакції
- Налаштування SSL
- Практична вправа

Вступ до баз даних у Python

Що таке база даних?

- Структуроване сховище даних
- Ефективне керування інформацією
- Швидкий доступ та пошук даних
- Підтримка ACID-властивостей

Python та бази даних

- Стандартна бібліотека `sqlite3`
- Багато сторонніх бібліотек
- ORM-системи (SQLAlchemy, Django ORM)
- Універсальний API для роботи з БД

Переваги використання

- Надійне зберігання даних
- Масштабованість додатків
- Одночасний доступ багатьох користувачів
- Захист від втрати даних

Типи баз даних

- Реляційні (SQLite, PostgreSQL)
- NoSQL (MongoDB, Redis)
- Графові (Neo4j)
- Часові ряди (InfluxDB)

SQLite Огляд

Що таке SQLite?

- Легковагова, вбудована СУБД
- Не потребує окремого сервера
- Зберігає дані в одному файлі
- Підтримує стандарт SQL

Переваги

- Нульове адміністрування
- Швидка робота для малих/середніх БД
- Надійність (ACID-сумісна)
- Портативність між платформами

Технічні особливості

- Підтримка типів даних: NULL, INTEGER, REAL, TEXT, BLOB
- Повнотекстовий пошук (FTS)
- Тригери та представлення
- Обмежена підтримка ALTER TABLE

Випадки використання

- Мобільні додатки
- Десктопні програми
- Кешування даних
- Тестування та розробка

SQLite: Підключення та налаштування

Підключення до БД

- Імпорт модуля `sqlite3`
- Створення з'єднання: `sqlite3.connect()`
- Створення курсора: `connection.cursor()`
- Закриття з'єднання: `connection.close()`

Приклад підключення

```
import sqlite3

# Створення підключення
conn = sqlite3.connect('example.db')

# Створення курсора
cursor = conn.cursor()

# Виконання операцій...

# Закриття підключення
conn.close()
```

Режими підключення

- `:memory:` - БД в оперативній пам'яті
- `file.db` - Файл БД на диску
- `file::memory:` - Тимчасова БД
- `file?mode=ro` - Тільки для читання

Кращі практики

- Використання контекстного менеджера
- Обробка винятків
- Завжди закривайте з'єднання
- Використання `with` для автоматичного закриття

SQLite: Створення таблиць

Основні команди

- `CREATE TABLE` - створення таблиці
- `PRIMARY KEY` - первинний ключ
- `AUTOINCREMENT` - автоінкремент
- `NOT NULL` - обов'язкове поле

Приклад створення таблиці

```
cursor.execute('''
CREATE TABLE students (
id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL,
age INTEGER,
email TEXT UNIQUE
)'''
)

conn.commit()
```

Типи даних SQLite

- `NULL` - NULL значення
- `INTEGER` - цілі числа
- `REAL` - числа з плаваючою комою
- `TEXT` - текстові дані
- `BLOB` - бінарні дані

Додаткові опції

- `UNIQUE` - унікальність значень
- `DEFAULT` - значення за замовчуванням
- `CHECK` - перевірка умов
- `FOREIGN KEY` - зовнішній ключ

SQLite: Операції з даними (INSERT, SELECT)

INSERT - Додавання даних

```
# Простий INSERT
cursor.execute("INSERT INTO students (name, age) VALUES (?, ?)",
    ('Іван Петренко', 21))

# Багаторядковий INSERT
students = [
    ('Олена Коваль', 20),
    ('Тарас Шевченко', 22)
]
cursor.executemany("INSERT INTO students (name, age) VALUES (?, ?)",
students)
conn.commit()
```

SELECT - Отримання даних

```
# Отримання всіх записів
cursor.execute("SELECT * FROM students")
rows = cursor.fetchall()

# Отримання одного запису
cursor.execute("SELECT * FROM students WHERE id = ?", (1,))
student = cursor.fetchone()

# Отримання з умовою
cursor.execute("SELECT * FROM students WHERE age > ?", (20,))
adult_students = cursor.fetchall()
```

Параметризовані запити

- Захист від SQL-ін'єкцій
- Використання `?` як плейсхолдер
- Передача параметрів як кортеж
- Автоматичне екранування значень

Увага! Ніколи не використовуйте f-рядки або конкатенацію для SQL-запитів!

Кращі практики

- Використання `with` для автоматичного закриття
- Завжди викликайте `commit()` після змін
- Обробка винятків `try/except`
- Використання `fetchone()` для одного запису
- Використання `fetchall()` для всіх записів

SQLite: Операції з даними (UPDATE, DELETE)

UPDATE - Оновлення даних

```
# Оновлення одного запису
cursor.execute(
  "UPDATE students SET age = ? WHERE id = ?",
  (22, 1)
)

# Оновлення кількох полів
cursor.execute(
  "UPDATE students SET name = ?, age = ? WHERE id = ?",
  ('Іван Іванов', 23, 1)
)

# Оновлення з умовою
cursor.execute(
  "UPDATE students SET age = age + 1 WHERE name LIKE ?",
  ('Іван%',)
)
conn.commit()
```

DELETE - Видалення даних

```
# Видалення одного запису
cursor.execute(
  "DELETE FROM students WHERE id = ?",
  (1,)
)

# Видалення з умовою
cursor.execute(
  "DELETE FROM students WHERE age < ?",
  (18,)
)

# Видалення всіх записів
cursor.execute(
  "DELETE FROM students"
)
conn.commit()
```

Важливі зауваження

- Завжди використовуйте `WHERE` для оновлення/видалення
- Перевіряйте результат операції `cursor.rowcount`
- Не забувайте `commit()` для збереження змін
- Використовуйте транзакції для безпеки даних

Увага! Операції UPDATE та DELETE незворотні! Завжди перевіряйте умову WHERE.

Приклад з транзакцією

```
try:
  cursor.execute("BEGIN TRANSACTION")
  cursor.execute(
    "UPDATE students SET age = ? WHERE id = ?",
    (22, 1)
  )
  conn.commit()
except Exception as e:
  conn.rollback()
  print(f"Помилка: {e}")
```

PostgreSQL Огляд

Що таке PostgreSQL?

- Потужна об'єктно-реляційна СУБД
- Відкритий код (ліцензія BSD)
- Клієнт-серверна архітектура
- Підтримка стандартів SQL

Переваги

- Масштабованість та продуктивність
- Розширена підтримка типів даних
- Повна ACID-сумісність
- Можливість розширення функціональності

Технічні особливості

- Підтримка складних запитів
- Транзакції та MVCC
- Реплікація та партиціонування
- Підтримка JSON, XML, геоданих

Випадки використання

- Великі веб-додатки
- Системи з високим навантаженням
- Бізнес-аналітика та звіти
- Геоінформаційні системи

PostgreSQL: Архітектура та можливості

Архітектура

- Клієнт-серверна модель
- Процес-менеджер `postmaster`
- Пули з'єднань
- Багаторівневе сховище даних

Процеси управління

- Процеси `backend` для кожного клієнта
- Система `shared buffers` для кешування
- Процеси `WAL` для журналювання
- Автовакуум для очищення

Розширені можливості

- Індекси: B-tree, Hash, GiST, SP-GiST, GIN, BRIN
- Повнотекстовий пошук
- Розширені типи даних: JSON, XML, Array, Hstore
- Просторові дані (PostGIS)

Безпека та надійність

- Ролі та права доступу (RBAC)
- Рівні ізоляції транзакцій
- Шифрування даних (SSL)
- Реплікація та резервне копіювання

PostgreSQL vs SQLite Порівняння

Архітектура

PostgreSQL:	Клієнт-серверна
SQLite:	Вбудована, файлова

Продуктивність

PostgreSQL:	Висока для великих БД
SQLite:	Висока для малих БД

Багатокористувацький доступ

PostgreSQL:	Підтримує
SQLite:	Обмежена підтримка

Адміністрування

PostgreSQL:	Потрібне налаштування
SQLite:	Нульове адміністрування

Типи даних

PostgreSQL:	Багато типів, розширені
SQLite:	Базові типи даних

Випадки використання

PostgreSQL:	Великі проєкти, веб-сервіси
SQLite:	Мобільні/десктопні додатки

Встановлення PostgreSQL

Windows

- Завантажити з офіційного сайту
- Запустити інсталятор
- Вказати пароль для користувача `postgres`
- Вибрати порт (за замовчуванням 5432)

macOS

- Через Homebrew: `brew install postgresql`
- Запуск сервісу: `brew services start postgresql`
- Або через Postgres.app
- Створення користувача: `createuser -s postgres`

Linux (Ubuntu/Debian)

- `sudo apt update`
- `sudo apt install postgresql postgresql-contrib`
- Запуск сервісу: `sudo systemctl start postgresql`
- Автозапуск: `sudo systemctl enable postgresql`

Хмарні рішення

- Amazon RDS for PostgreSQL
- Azure Database for PostgreSQL
- Google Cloud SQL for PostgreSQL
- Heroku Postgres

Встановлення бібліотеки psycopg2

Що таке psycopg2?

- Найпопулярніший адаптер PostgreSQL для Python
- Реалізує специфікацію Python DB-API 2.0
- Підтримує асинхронні операції
- Висока продуктивність

Способи встановлення

- `pip install psycopg2-binary` (простий варіант)
- `pip install psycopg2` (з компіляцією)
- Через системний менеджер пакетів
- З вихідного коду

Приклад встановлення

```
# Встановлення через pip (рекомендовано)
pip install psycopg2-binary

# Перевірка встановлення
python -c "import psycopg2; print(psycopg2.__version__)"

# Для Linux може знадобитися:
sudo apt-get install libpq-dev python-dev
pip install psycopg2
```

Важливі зауваження

- `psycopg2-binary` містить скомпільовані бінарні файли
- Для продакшену краще використовувати `psycopg2`
- Для асинхронної роботи: `pip install psycopg2-binary`
- Перевіряйте сумісність версій Python та PostgreSQL

PostgreSQL: Налаштування підключення

Основні параметри підключення

- `host` - адреса сервера
- `port` - порт (за замовчуванням 5432)
- `database` - назва бази даних
- `user` - ім'я користувача
- `password` - пароль

Приклад підключення

```python
import psycopg2

# Підключення до БД
conn = psycopg2.connect(
  host="localhost",
  database="mydb",
  user="postgres",
  password="mypassword"
)

# Створення курсора
cursor = conn.cursor()

# Виконання операцій...

# Закриття підключення
cursor.close()
conn.close()
```

Альтернативні способи підключення

- Через рядок підключення:

  `psycopg2.connect("dbname=test user=postgres")`
- Використання файлу `.pgpass`
- Через змінні середовища
- Використання `connection_pool`

Кращі практики

- Використання контекстного менеджера
- Зберігання даних для підключення у змінних середовища
- Обробка винятків `try/except`
- Використання `with` для автоматичного закриття

PostgreSQL: Створення таблиць

Основні команди

- `CREATE TABLE` - створення таблиці
- `SERIAL` - автоінкремент
- `PRIMARY KEY` - первинний ключ
- `FOREIGN KEY` - зовнішній ключ

Приклад створення таблиці

```
cursor.execute('''
CREATE TABLE students (
id SERIAL PRIMARY KEY,
name VARCHAR(100) NOT NULL,
age INTEGER,
email VARCHAR(100) UNIQUE,
registration_date DATE DEFAULT CURRENT_DATE
)''')

conn.commit()
```

Типи даних PostgreSQL

- `SMALLINT, INTEGER, BIGINT` - цілі числа
- `REAL, DOUBLE PRECISION` - дробові числа
- `VARCHAR(n), TEXT` - текстові дані
- `DATE, TIME, TIMESTAMP` - дата/час
- `BOOLEAN` - логічні значення

Додаткові опції

- `UNIQUE` - унікальність значень
- `NOT NULL` - обов'язкове поле
- `DEFAULT` - значення за замовчуванням
- `CHECK` - перевірка умов
- `CONSTRAINT` - обмеження

PostgreSQL: Типи даних

Числові типи

- `SMALLINT` - 2 байти, від -32768 до 32767
- `INTEGER` - 4 байти, від -2147483648 до 2147483647
- `BIGINT` - 8 байт, великий діапазон
- `REAL, DOUBLE PRECISION` - числа з плаваючою комою
- `NUMERIC(p,s)` - точні числа з фіксованою точністю

Тт Текстові типи

- `VARCHAR(n)` - рядок з обмеженням довжини
- `CHAR(n)` - рядок фіксованої довжини
- `TEXT` - рядок необмеженої довжини
- `ENUM` - перелік можливих значень

Дата та час

- `DATE` - дата (рік, місяць, день)
- `TIME` - час (години, хвилини, секунди)
- `TIMESTAMP` - дата та час
- `INTERVAL` - проміжок часу

Спеціальні типи

- `BOOLEAN` - логічні значення (true/false)
- `JSON/JSONB` - дані у форматі JSON
- `ARRAY` - масиви значень
- `UUID` - унікальні ідентифікатори
- `HSTORE` - пари ключ-значення

Бінарні дані

- `BYTEA` - бінарні дані
- `OID` - ідентифікатори об'єктів
- `Large Objects` - великі об'єкти (до 2GB)

Географічні типи

- `POINT` - точка на площині
- `LINE` - нескінченна пряма
- `POLYGON` - замкнений контур
- `PostGIS` - розширення для геоданих

PostgreSQL: SELECT Запити

Основні методи отримання даних

- `fetchone()` - один запис
- `fetchall()` - всі записи
- `fetchmany(n)` - n записів
- `rowcount` - кількість записів

Приклад SELECT запиту

```
# Отримання всіх студентів
cursor.execute("SELECT * FROM students")
students = cursor.fetchall()

# Отримання з умовою
cursor.execute(
  "SELECT * FROM students WHERE age > %s",
  (20,)
)
adult_students = cursor.fetchall()
```

Складні SELECT запити

- `JOIN` - об'єднання таблиць
- `GROUP BY` - групування
- `ORDER BY` - сортування
- `LIMIT/OFFSET` - пагінація

Параметризовані запити

- Захист від SQL-ін'єкцій
- Використання `%s` як плейсхолдер
- Передача параметрів як кортеж
- Автоматичне екранування значень

PostgreSQL: INSERT Операції

Основні методи вставки

- `INSERT INTO` - додавання запису
- `VALUES` - значення для вставки
- `RETURNING` - повернення ID
- `executemany()` - множинна вставка

Приклад INSERT запиту

```
# Простий INSERT
cursor.execute(
  "INSERT INTO students (name, age) VALUES (%s, %s)",
  ('Іван Петренко', 21)
)

# INSERT з поверненням ID
cursor.execute(
  "INSERT INTO students (name, age) VALUES (%s, %s) RETURNING id",
  ('Олена Коваль', 20)
)
new_id = cursor.fetchone()[0]
```

Багаторядкова вставка

```
# Підготовка даних
students = [
  ('Тарас Шевченко', 22),
  ('Леся Українка', 19),
  ('Іван Франко', 23)
]

# Множинна вставка
cursor.executemany(
  "INSERT INTO students (name, age) VALUES (%s, %s)",
  students
)
conn.commit()
```

Кращі практики

- Використання параметризованих запитів
- Завжди викликайте `commit()`
- Обробка винятків `try/except`
- Використання транзакцій для цілісності
- Перевірка `cursor.rowcount` для результату

PostgreSQL: UPDATE та DELETE Операції

UPDATE - Оновлення даних

```
# Оновлення одного запису
cursor.execute(
  "UPDATE students SET age = %s WHERE id = %s",
  (22, 1)
)

# Оновлення кількох полів
cursor.execute(
  "UPDATE students SET name = %s, age = %s WHERE id = %s",
  ('Іван Іванов', 23, 1)
)
conn.commit()
```

DELETE - Видалення даних

```
# Видалення одного запису
cursor.execute(
  "DELETE FROM students WHERE id = %s",
  (1,)
)

# Видалення з умовою
cursor.execute(
  "DELETE FROM students WHERE age < %s",
  (18,)
)
conn.commit()
```

Важливі зауваження

- Завжди використовуйте `WHERE` для оновлення/видалення
- Перевіряйте результат операції `cursor.rowcount`
- Не забувайте `commit()` для збереження змін
- Використовуйте транзакції для безпеки даних

Увага! Операції UPDATE та DELETE незворотні! Завжди перевіряйте умову WHERE.

Приклад з транзакцією

```
try:
  cursor.execute("BEGIN")
  cursor.execute(
    "UPDATE students SET age = %s WHERE id = %s",
    (22, 1)
  )
  conn.commit()
except Exception as e:
  conn.rollback()
  print(f"Помилка: {e}")
```

PostgreSQL: Параметризовані запити

Захист від SQL-ін'єкцій

- Автоматичне екранування значень
- Розділення коду та даних
- Запобігання виконанню шкідливого коду
- Вимога стандарту безпеки OWASP

Приклад параметризації

```
# НЕБЕЗПЕЧНО!
query = f"SELECT * FROM users WHERE name = '{user_input}'"
cursor.execute(query)

# БЕЗПЕЧНО
query = "SELECT * FROM users WHERE name = %s"
cursor.execute(query, (user_input,))
```

Синтаксис psycopg2

- Використання `%s` як плейсхолдера
- Передача параметрів як кортеж
- Для одного параметра: `(value,)`
- Для декількох: `(val1, val2, val3)`

Поширені помилки

- Використання f-рядків для формування SQL
- Конкатенація рядків з даними користувача
- Використання `?` замість `%s`
- Забування коми для одного параметра

PostgreSQL: Обробка результатів запитів

[] Методи отримання даних

- `fetchone()` - один запис у вигляді кортежу
- `fetchall()` - список кортежів
- `fetchmany(n)` - n записів
- `rowcount` - кількість оброблених рядків

<> Приклад обробки результатів

```
# Отримання та обробка даних
cursor.execute("SELECT id, name FROM students")
students = cursor.fetchall()

# Перетворення у словники
columns = [desc[0] for desc in cursor.description]
result = [dict(zip(columns, row)) for row in students]

# Вивід результатів
for student in result:
  print(f"ID: {student['id']}, Ім'я: {student['name']}")
```

Робота з великими наборами даних

- Використання `fetchmany()` для пагінації
- Ітерація по курсору: `for row in cursor`
- Обмеження результатів: `LIMIT` у SQL
- Серверний курсор для великих обсягів

Кращі практики

- Перевірка `cursor.rowcount` після запитів
- Використання `cursor.description` для метаданих
- Обробка `None` значень для NULL
- Використання `namedtuple` або `DictCursor`

PostgreSQL: Транзакції

Що таке транзакція?

- Логічна одиниця роботи з БД
- Виконується повністю або не виконується взагалі
- Забезпечує цілісність даних
- Підтримує ACID-властивості

ACID-властивості

- **A**tomicity - Атомарність
- **C**onsistency - Узгодженість
- **I**solation - Ізольованість
- **D**urability - Стійкість

Приклад транзакції

```
try:
  # Початок транзакції
  cursor.execute("BEGIN")

  # Операції з даними
  cursor.execute("UPDATE accounts SET balance = balance - 100 WHERE id =
%s", (1,))
  cursor.execute("UPDATE accounts SET balance = balance + 100 WHERE id =
%s", (2,))

  # Завершення транзакції
  conn.commit()
except Exception as e:
  # Відкіт змін
  conn.rollback()
  print(f"Помилка: {e}")
```

Рівні ізоляції

- `READ UNCOMMITTED` - читання незафіксованих даних
- `READ COMMITTED` - читання зафіксованих даних
- `REPEATABLE READ` - повторюване читання
- `SERIALIZABLE` - серіалізація

Підключення до віддаленої бази даних

Основні параметри підключення

- `host` - IP-адреса або домен сервера
- `port` - порт (за замовчуванням 5432)
- `database` - назва бази даних
- `user` - ім'я користувача БД
- `password` - пароль користувача

Приклад підключення

```python
import psycopg2

# Підключення до віддаленої БД
conn = psycopg2.connect(
  host="db.example.com",
  port=5432,
  database="mydb",
  user="dbuser",
  password="dbpassword"
)

# Перевірка підключення
cursor = conn.cursor()
cursor.execute("SELECT version()")
print(cursor.fetchone())
```

Безпека підключення

- Використання SSL-шифрування
- Зберігання облікових даних у змінних середовища
- Обмеження доступу за IP-адресами
- Використання SSH-тунелів для додаткового захисту

Поширені проблеми

- Брандмауер блокує порт
- Неправильні облікові дані
- Відсутність прав доступу до БД
- Проблеми з DNS-розв'язанням
- Обмеження кількості підключень

Безпека та налаштування SSL

Основи безпеки PostgreSQL

- Ролі та права доступу (RBAC)
- Шифрування паролів
- Обмеження мережевого доступу
- Аудит та логування

SSL для PostgreSQL

- Шифрування трафіку клієнт-сервер
- Перевірка автентичності сервера
- Автентифікація клієнтів за сертифікатами
- Режими SSL: вимкнено, дозволено, вимагати

Налаштування SSL на сервері

- У `postgresql.conf`:

```
ssl = on
ssl_cert_file = 'server.crt'
ssl_key_file = 'server.key'
```

- У `pg_hba.conf`:

```
hostssl all all 0.0.0.0/0 cert
```

SSL-підключення в Python

```python
import psycopg2
import ssl

# SSL-підключення
conn = psycopg2.connect(
  host="db.example.com",
  database="mydb",
  user="dbuser",
  password="dbpassword",
  sslmode="require"
)

# Перевірка SSL
print(conn.get_dsn_parameters())
```

Оптимізація запитів

Індекси

- Створення індексів для частіших пошуків
- Композитні індекси для кількох полів
- Використання `EXPLAIN` для аналізу
- Видалення невикористовуваних індексів

Приклад індексу

```
# Створення індексу
cursor.execute(
  "CREATE INDEX idx_students_name ON students(name)"
)

# Композитний індекс
cursor.execute(
  "CREATE INDEX idx_students_age_name ON students(age, name)"
)

# Аналіз запиту
cursor.execute(
  "EXPLAIN SELECT * FROM students WHERE name = 'Іван'"
)
```

Оптимізація запитів

- Уникання `SELECT *` - вказувати поля
- Використання `LIMIT` для обмеження результатів
- Оптимізація `JOIN` - правильні індекси
- Використання `EXPLAIN ANALYZE` для профілювання

Оптимізація сервера

- Налаштування `shared_buffers`
- Оптимізація `work_mem`
- Налаштування `effective_cache_size`
- Регулярне виконання `VACUUM` та `ANALYZE`

Практична вправа: Налаштування

Необхідне програмне забезпечення

- Python 3.8+
- PostgreSQL 12+
- Бібліотека psycopg2-binary
- pgAdmin або DBeaver (опційно)

Підготовка середовища

```
# Створення віртуального середовища
python -m venv db_env
source db_env/bin/activate # Linux/Mac
db_env\Scripts\activate # Windows

# Встановлення бібліотеки
pip install psycopg2-binary

# Перевірка встановлення
python -c "import psycopg2; print('OK')"
```

Налаштування бази даних

- Створення бази даних `python_db`
- Створення користувача з правами доступу
- Налаштування підключення через `pg_hba.conf`
- Перевірка підключення до БД

Завдання практичної роботи

- Створення таблиць для зберігання даних
- Реалізація CRUD-операцій
- Використання параметризованих запитів
- Робота з транзакціями
- Оптимізація запитів

Практична вправа: Реалізація

1 Крок 1: Підключення до БД

```python
import psycopg2

# Функція підключення
def connect_to_db():
  try:
    conn = psycopg2.connect(
      host="localhost",
      database="python_db",
      user="postgres",
      password="your_password"
    )
    return conn
  except Exception as e:
    print(f"Помилка: {e}")
    return None
```

2 Крок 2: Створення таблиць

```python
def create_tables(conn):
  cursor = conn.cursor()
  cursor.execute('''
CREATE TABLE IF NOT EXISTS students (
id SERIAL PRIMARY KEY,
name VARCHAR(100) NOT NULL,
age INTEGER,
email VARCHAR(100) UNIQUE
)''')
  conn.commit()
  cursor.close()
```

3 Крок 3: CRUD-операції

```python
# Додавання студента
def add_student(conn, name, age, email):
  cursor = conn.cursor()
  cursor.execute(
    "INSERT INTO students (name, age, email) VALUES (%s, %s, %s) RETURNING
id",
    (name, age, email)
  )
  student_id = cursor.fetchone()[0]
  conn.commit()
  return student_id
```

4 Крок 4: Отримання даних

```python
# Отримання всіх студентів
def get_all_students(conn):
  cursor = conn.cursor()
  cursor.execute("SELECT * FROM students ORDER BY name")
  students = cursor.fetchall()
  cursor.close()
  return students

# Пошук за віком
def get_students_by_age(conn, min_age):
  cursor = conn.cursor()
  cursor.execute("SELECT * FROM students WHERE age >= %s", (min_age,))
  students = cursor.fetchall()
  return students
```

Практична вправа: Реалізація

5 Крок 5: Оновлення та видалення

```
# Оновлення віку
def update_student_age(conn, student_id, new_age):
  cursor = conn.cursor()
  cursor.execute(
    "UPDATE students SET age = %s WHERE id = %s",
    (new_age, student_id)
  )
  conn.commit()

# Видалення студента
def delete_student(conn, student_id):
  cursor = conn.cursor()
  cursor.execute("DELETE FROM students WHERE id = %s", (student_id,))
  conn.commit()
```

6 Крок 6: Використання транзакцій

```
# Переведення студентів між курсами
def transfer_students(conn, student_ids, new_course):
  try:
    cursor = conn.cursor()
    # Початок транзакції
    cursor.execute("BEGIN")

    # Оновлення курсу для всіх студентів
    for student_id in student_ids:
      cursor.execute(
        "UPDATE students SET course = %s WHERE id = %s",
        (new_course, student_id)
      )

    # Завершення транзакції
    conn.commit()
    return True
  except Exception as e:
    # Відкіт змін
    conn.rollback()
    print(f"Помилка: {e}")
    return False
```

Висновки

- SQLite є легковаговою вбудованою СУБД, ідеальною для локальних додатків та малих проєктів
- PostgreSQL — потужна клієнт-серверна СУБД, що підтримує складні операції та великі обсяги даних
- Підключення до баз даних у Python реалізується через стандартні бібліотеки (sqlite3) та сторонні (psycopg2)
- Використання параметризованих запитів є обов'язковим для захисту від SQL-ін'єкцій
- Транзакції забезпечують цілісність даних при виконанні складних операцій
- Оптимізація запитів та належне індексування значно покращують продуктивність бази даних
- Безпечне підключення до віддалених баз даних вимагає налаштування SSL та правильного управління обліковими даними
- Вибір між SQLite та PostgreSQL залежить від масштабу проєкту, вимог до продуктивності та кількості користувачів