Національний університет біоресурсів і
природокористування України

Робота з базами даних у Python: SQLite та PostgreSQL

Підключення, отримання та запис даних

Частина 1

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

Вступ до реляційних баз даних

Реляційні бази даних

- Організація даних у вигляді таблиць
- Зв'язки між таблицями через ключі
- Структурований запит даних (SQL)
- ACID-властивості транзакцій

Роль СУБД у Python

- Зберігання структурованих даних
- Ефективний доступ до великих обсягів
- Інтеграція через спеціалізовані модулі
- Підтримка паралельних операцій

Переваги використання БД у Python-проектах

- Надійність та цілісність даних
- Масштабованість та продуктивність
- Стандартизація роботи з даними

Огляд СУБД SQLite

Особливості SQLite

- Безсерверна, файлова СУБД
- Нульове адміністрування
- Підтримка повного SQL
- Транзакції ACID
- Одне зберігання в одному файлі

Переваги SQLite

- Легковагова та швидка
- Надійна та стабільна
- Портативна між платформами
- Публічний домен (без ліцензії)
- Вбудована в багато мов

Типові сценарії використання

Мобільні додатки

Локальне зберігання даних на пристрої

Настільні ПЗ

Зберігання налаштувань та даних користувача

Вбудовані системи

IoT пристрої та промислове обладнання

Інсталяція та налаштування SQLite для Python

Модуль sqlite3

- Вбудований у стандартну бібліотеку Python
- Не потребує окремої інсталяції
- Підтримує стандартний SQL-синтаксис
- Сумісний з версією SQLite 3.7.15+

Імпорт модуля

```
import sqlite3

# Перевірка версії
print(sqlite3.sqlite_version)
```

- Модуль надає API для роботи з БД
- Підтримує контекстний менеджер

Основні кроки налаштування

Підключення

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

Створення курсора

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

Закриття з'єднання

```
conn.close()
```

Порада

Використовуйте контекстний менеджер `with` для автоматичного закриття з'єднання

Створення бази даних та таблиць

Основні типи даних SQLite

- `INTEGER` - цілі числа
- `REAL` - числа з плаваючою комою
- `TEXT` - текстові рядки
- `BLOB` - двійкові дані
- `NULL` - відсутність значення

Створення бази даних

```
import sqlite3

# Створення/підключення до БД
conn = sqlite3.connect('my_database.db')

# Створення курсора
cursor = conn.cursor()

# Закриття з'єднання
conn.close()
```

Створення таблиць

```
# SQL-запит для створення таблиці
create_table_query = '''
CREATE TABLE users (
id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL,
email TEXT UNIQUE NOT NULL,
age INTEGER,
registration_date TEXT DEFAULT CURRENT_TIMESTAMP
)
'''

# Виконання запиту
cursor.execute(create_table_query)
conn.commit()
```

PRIMARY KEY
Унікальний ідентифікатор запису

NOT NULL
Обов'язкове для заповнення поле

UNIQUE
Унікальне значення в таблиці

Підключення до бази даних та виконання запитів

Підключення до БД

```
import sqlite3

# Підключення до існуючої БД
conn = sqlite3.connect('my_database.db')

# Створення курсора
cursor = conn.cursor()
```

Отримання даних (SELECT)

```
sql = "SELECT * FROM users WHERE age > ?"

cursor.execute(sql, (20,))

# Отримання всіх результатів
results = cursor.fetchall()

# Отримання одного результату
result = cursor.fetchone()
```

Вставка даних (INSERT)

```
sql = "INSERT INTO users (name, email, age) VALUES (?, ?, ?)"

data = ('Іван Петренко', 'ivan@example.com', 25)

cursor.execute(sql, data)
conn.commit()
```

Оновлення даних (UPDATE)

```
sql = "UPDATE users SET age = ? WHERE name = ?"
cursor.execute(sql, (26, 'Іван Петренко'))
conn.commit()
```

Видалення даних (DELETE)

```
sql = "DELETE FROM users WHERE id = ?"
cursor.execute(sql, (1,))
conn.commit()
```

Безпека запитів

- Використовуйте параметризовані запити з `?` для запобігання SQL-ін'єкціям
- Завжди викликайте `commit()` після змін даних
- Використовуйте `with` для автоматичного закриття з'єднання

Обробка результатів запитів та робота з курсорами

Курсор в SQLite

- Об'єкт для виконання SQL-запитів
- Створюється з'єднанням з БД
- Зберігає стан запиту
- Надає методи для отримання даних

```
conn = sqlite3.connect('example.db')
cursor = conn.cursor()
cursor.execute("SELECT * FROM users")
```

Методи отримання даних

fetchone()

Повертає один запис або None

```
row = cursor.fetchone()
```

fetchall()

Повертає всі записи як список

```
rows = cursor.fetchall()
```

fetchmany(size)

Повертає вказану кількість записів

```
rows = cursor.fetchmany(5)
```

Ітерація через результати

Пряма ітерація

```
cursor.execute("SELECT name, email FROM users")

for row in cursor:
    print(f"Ім'я: {row[0]}, Email: {row[1]}")
```

Ітерація з іменованими полями

```
cursor.execute("SELECT name, email FROM users")

# Налаштування доступу за іменем
cursor.row_factory = sqlite3.Row

for row in cursor:
    print(f"Ім'я: {row['name']}")
```

Поради щодо роботи з курсорами

- Закривайте курсори після використання
- Використовуйте `with` для автоматичного закриття
- Для великих обсягів даних використовуйте `fetchmany()`
- Використовуйте `row_factory` для зручного доступу до полів

Практична вправа: створення локальної бази даних

Завдання

- Створити базу даних для книжкового магазину
- Створити таблиці: books, authors, customers
- Додати дані до таблиць
- Виконати запити для отримання даних

Крок 1: Створення БД та таблиць

```
import sqlite3

conn = sqlite3.connect('bookstore.db')
cursor = conn.cursor()

# Створення таблиці books
cursor.execute('''CREATE TABLE books (
  id INTEGER PRIMARY KEY,
  title TEXT NOT NULL,
  author_id INTEGER,
  price REAL,
  FOREIGN KEY (author_id) REFERENCES authors(id)
)''')
```

Крок 2: Додавання даних

```
# Додавання автора
cursor.execute("INSERT INTO authors (name) VALUES (?)",
    ('Тарас Шевченко',))

# Отримання ID автора
author_id = cursor.lastrowid

# Додавання книги
cursor.execute("INSERT INTO books (title, author_id, price) VALUES (?, ?,
?)",
    ('Кобзар', author_id, 250.00))

conn.commit()
```

Крок 3: Отримання даних

```
# Отримання всіх книг з авторами
cursor.execute("SELECT b.title, a.name FROM books b JOIN authors a ON
b.author_id = a.id")

for row in cursor:
    print(f"Книга: {row[0]}, Автор: {row[1]}")
```

Додаткові завдання

Фільтрація даних

Знайдіть усі книги, ціна яких перевищує 200 грн

Оновлення даних

Змініть ціну книги з id=1 на 300 грн

Видалення даних

Видаліть книгу з назвою 'Стара книга'

Порада для виконання

Використовуйте контекстний менеджер `with` для автоматичного закриття з'єднання та обробки помилок:

```
with sqlite3.connect('bookstore.db') as conn:
    with conn:
        cursor = conn.cursor()
        # Ваші операції з БД
```

Вступ до PostgreSQL

Особливості PostgreSQL

- Об'єктно-реляційна СУБД
- Клієнт-серверна архітектура
- Підтримка складних запитів
- Розширюваність через функції та типи даних
- Повна підтримка ACID-транзакцій

Переваги PostgreSQL

- Висока надійність та стабільність
- Підтримка великих обсягів даних
- Потужна система безпеки
- Розширена підтримка SQL стандарту
- Відкритий код (ліцензія BSD)

Типові сценарії використання

Бізнес-додатки

ERP, CRM, фінансові системи

Веб-додатки

Складні веб-проекти з високим навантаженням

Аналітика

Сховища даних та бізнес-аналітика

Порівняння з SQLite

PostgreSQL

- Серверна СУБД
- Підтримка багатокористувацького доступу
- Складні запити та агрегації

SQLite

- Файлова СУБД
- Однокористувацький доступ
- Прості запити та операції

Інсталяція та налаштування PostgreSQL для Python

Встановлення PostgreSQL

- Завантаження з офіційного сайту postgresql.org
- Встановлення відповідно до ОС
- Налаштування пароля для користувача postgres
- Створення бази даних через pgAdmin або CLI

Команди для Linux

```
# Встановлення PostgreSQL
sudo apt-get install postgresql

# Запуск сервісу
sudo service postgresql start

# Вхід до CLI
sudo -u postgres psql
```

Модуль psycopg2

Встановлення

```
pip install psycopg2-binary
```

Імпорт та підключення

```
import psycopg2

try:
  conn = psycopg2.connect(
    host="localhost",
    database="mydb",
    user="postgres",
    password="yourpassword"
  )
except psycopg2.Error as e:
  print(f"Помилка: {e}")
```

Основні кроки налаштування

Підключення

```
conn = psycopg2.connect(
  host="localhost",
  database="mydb",
  user="user",
  password="pass"
)
```

Створення курсора

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

Закриття з'єднання

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

Порада

Використовуйте контекстний менеджер `with` для автоматичного закриття з'єднання та обробки помилок

```
with psycopg2.connect(...) as conn:
  with conn.cursor() as cursor:
    # Ваші операції з БД
```

Підключення до баз даних PostgreSQL

Параметри підключення

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

Безпечне зберігання параметрів

- Використовуйте змінні середовища
- Зберігайте паролі у конфігураційних файлах
- Не зберігайте паролі у коді

Встановлення з'єднання

```python
import psycopg2
import os

# Отримання параметрів з оточення
db_params = {
  "host": os.getenv("DB_HOST", "localhost"),
  "database": os.getenv("DB_NAME", "mydb"),
  "user": os.getenv("DB_USER", "postgres"),
  "password": os.getenv("DB_PASSWORD"),
  "port": os.getenv("DB_PORT", "5432")
}

# Встановлення з'єднання
conn = psycopg2.connect(**db_params)
```

Пули з'єднань

Переваги використання пулів

- Підвищення продуктивності
- Зменшення навантаження на сервер
- Ефективне керування ресурсами

Використання psycopg2.pool

```python
from psycopg2 import pool

# Створення пулу з'єднань
connection_pool = pool.SimpleConnectionPool(
  1, # minconn
  10, # maxconn
  **db_params
)

# Отримання з'єднання з пулу
conn = connection_pool.getconn()
```

Найкращі практики

- Використовуйте контекстний менеджер `with`
- Обробляйте винятки підключення
- Завжди закривайте з'єднання після використання
- Використовуйте пули для веб-додатків

Робота з PostgreSQL в Python: виконання запитів

Вставка даних (INSERT)

```
sql = "INSERT INTO users (name, email, age) VALUES (%s, %s, %s)"

data = ('Олена Коваль', 'olena@example.com', 28)

cursor.execute(sql, data)
conn.commit()
```

Масова вставка

```
sql = "INSERT INTO users (name, email, age) VALUES (%s, %s, %s)"
users_data = [
  ('Іван Петренко', 'ivan@example.com', 25),
  ('Марія Сидоренко', 'maria@example.com', 32)
]
cursor.executemany(sql, users_data)
conn.commit()
```

Отримання даних (SELECT)

```
sql = "SELECT * FROM users WHERE age > %s"

cursor.execute(sql, (20,))

# Отримання всіх результатів
results = cursor.fetchall()

# Отримання одного результату
result = cursor.fetchone()
```

Оновлення даних (UPDATE)

```
sql = "UPDATE users SET age = %s WHERE name = %s"
cursor.execute(sql, (29, 'Олена Коваль'))
conn.commit()
```

Видалення даних (DELETE)

```
sql = "DELETE FROM users WHERE id = %s"
cursor.execute(sql, (1,))
conn.commit()
```

Відмінності від SQLite

PostgreSQL

- Використовує `%s` для параметрів
- Підтримує `executemany()` для масових операцій
- Потрібен явний `commit()` для збереження змін

SQLite

- Використовує `?` для параметрів
- Обмежена підтримка масових операцій
- Також потрібен явний `commit()`

Безпека запитів

- Використовуйте параметризовані запити з `%s` для запобігання SQL-ін'єкціям
- Використовуйте `with` для автоматичного закриття з'єднання
- Завжди викликайте `commit()` після змін даних
- Обробляйте винятки для надійності коду

Обробка результатів запитів PostgreSQL

Методи отримання даних

fetchone()

Повертає один запис або None

```
row = cursor.fetchone()
```

fetchall()

Повертає всі записи як список

```
rows = cursor.fetchall()
```

fetchmany(size)

Повертає вказану кількість записів

```
rows = cursor.fetchmany(5)
```

Ітерація через результати

Пряма ітерація

```
cursor.execute("SELECT name, email FROM users")

for row in cursor:
    print(f"Ім'я: {row[0]}, Email: {row[1]}")
```

Ітерація з іменованими полями

```
cursor.execute("SELECT name, email FROM users")

# Налаштування доступу за іменем
from psycopg2.extras import DictCursor
cursor = conn.cursor(cursor_factory=DictCursor)

for row in cursor:
    print(f"Ім'я: {row['name']}")
```

Форматування результатів

Перетворення у словники

```
cursor.execute("SELECT * FROM users")
columns = [desc[0] for desc in cursor.description]
results = []

for row in cursor:
    results.append(dict(zip(columns, row)))
```

Перетворення у DataFrame

```
import pandas as pd

cursor.execute("SELECT * FROM users")
data = cursor.fetchall()
columns = [desc[0] for desc in cursor.description]

df = pd.DataFrame(data, columns=columns)
```

Поради щодо роботи з результатами

- Для великих обсягів даних використовуйте `fetchmany()`
- Перевіряйте `cursor.rowcount` для отримання кількості рядків
- Використовуйте `DictCursor` для зручного доступу до полів
- Для аналізу даних перетворюйте результати у DataFrame

Порівняння SQLite та PostgreSQL

SQLite

- Файлова база даних
- Безсерверна, нульове адміністрування
- Ідеальна для мобільних та настільних додатків
- Обмежена паралельність запису
- Вбудована в Python (модуль sqlite3)

PostgreSQL

- Клієнт-серверна база даних
- Потужна СУБД з повним набором функцій
- Ідеальна для веб-додатків та бізнес-систем
- Висока паралельність та масштабованість
- Потребує встановлення (psycopg2)

Ключові відмінності

Параметр	SQLite	PostgreSQL
Архітектура	Файлова	Клієнт-серверна
Масштабованість	Обмежена	Висока
Паралельність	Обмежена (читання/запис)	Повна (MVCC)
Складність запитів	Базова	Розширена
Типи даних	Базові (TEXT, INTEGER, REAL, BLOB)	Різноманітні (включаючи JSON, XML, геометричні)

Коли використовувати SQLite

- Локальні додатки
- Мобільні додатки
- Прототипування
- Невеликі проекти з одним користувачем

Коли використовувати PostgreSQL

- Веб-додатки з високим навантаженням
- Багатокористувацькі системи
- Складні бізнес-додатки
- Проекти, що потребують масштабування

Найкращі практики для роботи з базами даних у Python

Керування з'єднаннями

- Використовуйте контекстний менеджер `with`
- Завжди закривайте з'єднання після використання
- Використовуйте пули з'єднань для веб-додатків
- Уникайте довготривалих з'єднань

Безпека

- Використовуйте параметризовані запити
- Не зберігайте паролі у коді
- Використовуйте змінні середовища
- Обмежуйте права доступу користувачів БД

Обробка помилок

- Використовуйте блоки `try-except`
- Логуйте помилки для діагностики
- Надавайте зрозумілі повідомлення користувачам
- Перевіряйте з'єднання перед виконанням запитів

Продуктивність

- Використовуйте індекси для пошуку
- Вибирайте тільки потрібні поля
- Використовуйте `fetchmany()` для великих обсягів
- Оптимізуйте запити за допомогою EXPLAIN

Архітектурні практики

Розділення шарів

Відокремте логіку роботи з БД від бізнес-логіки

Міграції

Використовуйте інструменти керування схемою БД

Резервне копіювання

Регулярно створюйте резервні копії даних

Приклад безпечної роботи з БД

```
import os
from contextlib import closing

# Отримання параметрів з оточення
db_config = {
  "host": os.getenv("DB_HOST"),
  "database": os.getenv("DB_NAME"),
  "user": os.getenv("DB_USER"),
  "password": os.getenv("DB_PASSWORD")
}

# Безпечне виконання запиту
try:
  with closing(psycopg2.connect(**db_config)) as conn:
    with conn.cursor() as cursor:
```

Висновки

Реляційні бази даних у Python

- Python надає потужні інструменти для роботи з БД
- Вибір СУБД залежить від вимог проекту
- Стандартний SQL працює у всіх системах
- Параметри підключення та синтаксис можуть відрізнятись

SQLite та PostgreSQL

- SQLite - ідеальна для локальних та мобільних додатків
- PostgreSQL - потужна рішення для веб-проектів
- SQLite вбудована в Python, PostgreSQL потребує встановлення
- Обидві системи підтримують стандартні операції CRUD

Робота з даними

- Використання курсорів для виконання запитів
- Параметризовані запити для безпеки
- Обробка результатів через fetchone, fetchall, fetchmany
- Важливість транзакцій та commit

Ключові принципи

- Правильне керування з'єднаннями
- Безпечна робота з даними та паролями
- Обробка помилок та виняткових ситуацій
- Оптимізація запитів для продуктивності

Підсумок

Робота з базами даних у Python є важливою навичкою для розробників. Розуміння особливостей SQLite та PostgreSQL, вміння створювати структури даних, виконувати запити та обробляти результати дозволяє ефективно працювати з даними у різних типах проектів.