Grundlagen von Datenbanken

Agenda

1. Was ist eine Datenbank?
2. Arten von Datenbanken
3. Datenbankaufbau und Schlüsselkonzepte
4. SQL: Strukturierte Abfragesprache
5. Praktische Anwendung von SQL

Was ist eine Datenbank?

- Eine **Datenbank** ist eine organisierte Sammlung von strukturierten Informationen oder Daten.
- Sie ermöglicht das Speichern, Verwalten und Abrufen von Daten effizient und strukturiert.
- Wird in nahezu allen modernen Anwendungen zur Datenverwaltung eingesetzt.

Arten von Datenbanken

1. **Relationale Datenbanken (RDBMS)**:
    - Daten werden in Tabellen organisiert.
    - Verwendet Beziehungen (Joins) zwischen Tabellen.
    - Beispiele: MySQL, PostgreSQL, SQLite.
2. **NoSQL-Datenbanken**:
    - Nicht-relational, für unstrukturierte Daten geeignet.
    - Kann Dokumente, Schlüssel-Werte, Graphen oder spaltenbasierte Daten speichern.
    - Beispiele: MongoDB, CouchDB, Cassandra.
3. **In-memory-Datenbanken**:
    - Daten werden im Arbeitsspeicher gehalten, um schnellen Zugriff zu ermöglichen.
    - Beispiele: Redis, Memcached.
4. **Graph-Datenbanken**:
    - Speichern Daten in Knoten und Kanten, um komplexe Beziehungen darzustellen.
    - Beispiel: Neo4j.

Datenbankaufbau

- **Tabellen**: Struktureinheiten, die Daten in Zeilen und Spalten organisieren.
- **Spalten**: Beschreiben Attribute eines Datensatzes (z.B. Name, Alter, etc.).
- **Zeilen (Datensätze)**: Enthalten individuelle Einträge in einer Tabelle.
- **Schlüssel**:
  - **Primärschlüssel**: Eindeutige Identifikation eines Datensatzes.
  - **Fremdschlüssel**: Verweist auf einen Primärschlüssel in einer anderen Tabelle, um eine Beziehung herzustellen.

Wichtige Datenbankkonzepte

- **Normalisierung**: Prozess der Organisation von Daten, um Redundanzen zu vermeiden und Datenintegrität zu sichern.
- **Denormalisierung**: Bewusste Aufhebung der Normalisierung, um Abfragen zu beschleunigen.
- **Transaktionen**:
  - Eine **Transaktion** ist eine Abfolge von Operationen, die als Einheit ausgeführt wird.
  - Eigenschaften: **ACID** (Atomicity, Consistency, Isolation, Durability).

SQL: Die Strukturierte Abfragesprache

- **SQL** (Structured Query Language) ist die Standardsprache für relationale Datenbanken.
- SQL ermöglicht:
  - Erstellen und Verwalten von Datenbanken und Tabellen.
  - Einfügen, Abrufen, Aktualisieren und Löschen von Daten.

SQL-Grundlagen: Daten abfragen

- **SELECT**: Abfrage von Daten aus einer Tabelle.
- Beispiel:

```
SELECT name, age FROM students;
```

- **WHERE**: Filtert die Ergebnisse nach einer Bedingung.

```
SELECT name FROM students WHERE age > 20;
```

SQL-Grundlagen: Daten hinzufügen, aktualisieren, löschen

- **INSERT INTO**: Fügt neue Daten in eine Tabelle ein.

```
INSERT INTO students (name, age) VALUES ('Max', 22);
```

- **UPDATE**: Aktualisiert bestehende Daten.

```
UPDATE students SET age = 23 WHERE name = 'Max';
```

- **DELETE**: Löscht Daten aus einer Tabelle.

```
DELETE FROM students WHERE name = 'Max';
```

SQL-Grundlagen: Tabellen erstellen

- **CREATE TABLE**: Erstellt eine neue Tabelle.

```
CREATE TABLE students (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    age INTEGER NOT NULL
);
```

- **ALTER TABLE**: Ändert die Struktur einer bestehenden Tabelle.

```
ALTER TABLE students ADD COLUMN email TEXT;
```

SQL-Grundlagen: Daten aus mehreren Tabellen abfragen (Joins)

- **INNER JOIN**: Verknüpft Zeilen aus zwei Tabellen basierend auf einer übereinstimmenden Bedingung.

```sql
SELECT students.name, courses.course_name
FROM students
INNER JOIN enrollments ON students.id = enrollments.student_id
INNER JOIN courses ON enrollments.course_id = courses.id;
```

- **LEFT JOIN**: Gibt alle Zeilen aus der linken Tabelle zurück und nur die passenden aus der rechten.

```sql
SELECT students.name, enrollments.course_id
FROM students
LEFT JOIN enrollments ON students.id = enrollments.student_id;
```

- Das geht aber schon in die Richtung von komplexeren Abfragen...

Zusammenfassung

- Datenbanken sind eine wesentliche Komponente für die Verwaltung und Speicherung von Daten.
- Unterschiedliche Datenbankarten bieten flexible Lösungen für verschiedene Anwendungsfälle.
- SQL ist die Sprache der Wahl für relationale Datenbanken und ermöglicht effiziente Datenmanipulation und -abfrage.
- Das Verständnis von Tabellen, Schlüsseln und Joins ist zentral, um effektiv mit relationalen Datenbanken zu arbeiten.

Einführung in SQLite mit Python

Agenda

1. Datenbanken und SQLite
2. Installation und Einrichtung
3. Python-Code zur Interaktion mit SQLite
4. Arbeiten mit `venv`
5. Detaillierte Code-Erklärung
6. Praktische Beispiele

Was ist eine Datenbank?

- Eine **Datenbank** ist eine organisierte Sammlung von Daten.
- Sie ermöglicht das Speichern, Abrufen und Verwalten von Informationen.
- Relationale Datenbanken verwenden Tabellen, um Daten in Form von Zeilen und Spalten zu speichern.

SQLite - Eine leichtgewichtige Datenbank

- **SQLite** ist eine eingebettete, serverlose, selbstständige SQL-Datenbank.
- Ideal für kleine bis mittelgroße Anwendungen.
- Benötigt keine separate Installation – funktioniert direkt mit einer Datei.

Vorteile von SQLite

- Keine Konfiguration erforderlich.
- Kompatibel mit verschiedenen Plattformen.
- Ideal für Prototypen, Tests und kleinere Anwendungen.

Python-Integration mit SQLite

- Python bietet das Modul `sqlite3`, um mit SQLite-Datenbanken zu interagieren.
- Es ermöglicht das Erstellen, Verwalten und Abfragen von Datenbanken direkt aus dem Code heraus.

Setup einer virtuellen Umgebung (`venv`)

```
# Virtuelle Umgebung erstellen
python3 -m venv myenv

# Aktivieren der Umgebung (Linux/Mac)
source myenv/bin/activate

# Aktivieren der Umgebung (Windows) in Git Bash
source myenv\Scripts\activate

# Installation von Abhängigkeiten
pip install sqlite3
```

- `venv` ermöglicht es, eine isolierte Umgebung zu erstellen, in der spezifische Pakete für ein Projekt installiert werden können.
- So wird vermieden, dass globale Installationen durcheinander geraten.

Festhalten der Abhängigkeiten

```
# Erstellen einer requirements.txt-Datei
pip freeze > requirements.txt
```

Erstellen einer gitignore-Datei

```
# Erstellen einer .gitignore-Datei
echo "myenv/" > .gitignore
```

- Wichtig, um die virtuelle Umgebung und ihre Abhängigkeiten von der Versionskontrolle auszuschließen.
- `venv` ist spezifisch für das lokale System und sollte nicht geteilt werden.

Erstellen einer SQLite-Datenbank mit Python

```python
import sqlite3

# Verbindung zur SQLite-Datenbank herstellen (Datei wird erstellt, falls nicht vorhanden)
conn = sqlite3.connect('studenten.db')

# Cursor-Objekt zum Ausführen von SQL-Befehlen
cursor = conn.cursor()
```

Detaillierte Erklärung:

- **`sqlite3.connect('studenten.db')`**: Diese Zeile stellt eine Verbindung zu einer SQLite-Datenbank her. Falls die Datei `studenten.db` noch nicht existiert, wird sie erstellt.
- **`conn.cursor()`**: Der `cursor` ist ein Objekt, das SQL-Befehle ausführt und Ergebnisse zurückgibt. Es wird verwendet, um SQL-Anweisungen mit der Datenbank zu interagieren.

Erstellen einer Tabelle

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

Detaillierte Erklärung:

- **`CREATE TABLE IF NOT EXISTS`**: Dieser SQL-Befehl erstellt eine Tabelle namens `students`, falls sie nicht bereits existiert.
- **`id INTEGER PRIMARY KEY AUTOINCREMENT`**: Die `id`-Spalte dient als eindeutiger Bezeichner (Primärschlüssel) für jeden Datensatz und wird automatisch erhöht.
- **`name TEXT NOT NULL`**: `name` speichert den Namen des Studenten, der als Text (`TEXT`) definiert ist und nicht leer sein darf (`NOT NULL`).
- **`age INTEGER NOT NULL`**: `age` speichert das Alter des Studenten als Ganzzahl.
- **`course TEXT NOT NULL`**: `course` speichert den Studiengang des Studenten.

Hinzufügen von Daten

```
def student_hinzufuegen(name, age, course):
    cursor.execute('''
    INSERT INTO students (name, age, course)
    VALUES (?, ?, ?)
    ''', (name, age, course))
    conn.commit()
    print(f'Student {name} hinzugefügt.')
```

Detaillierte Erklärung:

- **Funktion**: `student_hinzufuegen` fügt neue Datensätze (Studenten) in die `students`-Tabelle ein.
- **`INSERT INTO students (name, age, course)`**: Dieser SQL-Befehl fügt neue Werte in die Tabelle ein.
- **`VALUES (?, ?, ?)`**: Die Platzhalter `?` werden durch die übergebenen Parameter (`name`, `age`, `course`) ersetzt.
- **`conn.commit()`**: Speichert die Änderungen in der Datenbank, ähnlich wie ein "Save" in einer Anwendung.
- **`print()`**: Gibt eine Erfolgsmeldung aus, wenn der Student erfolgreich hinzugefügt wurde.

Daten anzeigen

```
def studenten_anzeigen():
    cursor.execute('SELECT * FROM students')
    studenten = cursor.fetchall()
    print("Alle Studenten:")
    for student in studenten:
        print(student)
```

Detaillierte Erklärung:

- **Funktion**: `studenten_anzeigen` holt alle Datensätze aus der `students`-Tabelle und gibt sie auf der Konsole aus.
- **`SELECT * FROM students`**: SQL-Befehl, der alle Spalten (`*`) und Zeilen aus der `students`-Tabelle abruft.
- **`fetchall()`**: Diese Methode ruft alle abgerufenen Zeilen in einer Liste ab.
- **`for student in studenten`**: Eine Schleife, die jeden Studenten einzeln ausgibt.

Beispiel: Hinzufügen von Studenten

```
student_hinzufuegen("Max Mustermann", 21, "Informatik")
student_hinzufuegen("Anna Schmidt", 22, "Mathematik")
student_hinzufuegen("Lisa Müller", 20, "Physik")
```

Detaillierte Erklärung:

- In diesem Beispiel werden drei Studenten mit unterschiedlichen Namen, Altersangaben und Kursen in die Datenbank eingefügt.

Beispiel: Alle Studenten anzeigen

```
studenten_anzeigen()
```

Detaillierte Erklärung:

- Diese Funktion zeigt alle aktuell gespeicherten Studenten in der Datenbank an, indem sie die zuvor gespeicherten Datensätze abruft und auf der Konsole ausgibt.

Schließen der Verbindung

```
conn.close()
```

Detaillierte Erklärung:

- `conn.close()`: Beendet die Verbindung zur SQLite-Datenbank. Es ist wichtig, die Verbindung zu schließen, um Ressourcen freizugeben und sicherzustellen, dass alle Änderungen korrekt gespeichert sind.

Fazit

- SQLite ist eine einfache, dateibasierte Datenbank, die ideal für kleinere Projekte und das Testen von Datenbankkonzepten ist.
- Mit Python und dem Modul `sqlite3` kann man Datenbanken direkt in Python-Skripten verwalten.
- Das Arbeiten in virtuellen Umgebungen (`venv`) ist wichtig, um Abhängigkeiten sauber und isoliert zu halten.

Aber warte... es gibt mehr!

- Der Code kann durch Funktionen zum **Aktualisieren**, **Löschen** und **Suchen** erweitert werden.
- CRUD-Operationen (Create, Read, Update, Delete) sind in SQLite mit Python möglich.
- Create: Hinzufügen von Daten
- Read: Anzeigen von Daten
- Update: Aktualisieren von Daten
- Delete: Löschen von Daten
- Suchen: Suchen nach bestimmten Daten

Erweiterung: Aktualisieren von Daten

- Manchmal ist es notwendig, Daten zu aktualisieren.
- Wir können SQL-UPDATE-Befehle verwenden, um bestimmte Datensätze zu ändern.

```
def student_aktualisieren(id, name, age, course):
    cursor.execute('''
    UPDATE students
    SET name = ?, age = ?, course = ?
    WHERE id = ?
    ''', (name, age, course, id))
    conn.commit()
    print(f'Student mit ID {id} wurde aktualisiert.')
```

Beispiel: Aktualisieren eines Studenten

```
student_aktualisieren(1, "Max Mustermann", 22, "Informatik")
```

- Hier aktualisieren wir die Daten des Studenten mit der ID `1` und passen den Kurs und das Alter an.

Erweiterung: Löschen von Daten

- Manchmal müssen Datensätze aus der Datenbank entfernt werden.
- Dazu verwenden wir SQL-DELETE-Befehle.

```
def student_loeschen(id):
    cursor.execute('''
    DELETE FROM students WHERE id = ?
    ''', (id,))
    conn.commit()
    print(f'Student mit ID {id} wurde gelöscht.')
```

Beispiel: Löschen eines Studenten

```
student_loeschen(2)
```

- Dies entfernt den Studenten mit der ID `2` aus der Datenbank.

Erweiterung: Suchen nach einem bestimmten Studenten

- Um bestimmte Datensätze zu suchen, nutzen wir SQL-Befehle mit `WHERE`.

```
def student_suchen(name):
    cursor.execute('''
    SELECT * FROM students WHERE name = ?
    ''', (name,))
    studenten = cursor.fetchall()
    print(f'Suche nach Student {name}:')
    for student in studenten:
        print(student)
```

Beispiel: Suchen nach einem Studenten

```
student_suchen("Lisa Müller")
```

- Dies gibt alle Datensätze von Studenten mit dem Namen "Lisa Müller" aus.

Das ist ja großartig! Aber wie geht es weiter?

- Wir wollen im nächsten Schritt das Beispiel erweitern, indem wir mehrere Tabelle und Beziehungen zwischen ihnen erstellen.
- Außerdem wollen wir komplexere Abfragen durchführen, um Daten aus mehreren Tabellen zu kombinieren.
- Dabei werden wir **Joins** verwenden, um Daten aus verschiedenen Tabellen zu verknüpfen.

Komplexere Abfragen mit SQLite und Python

Agenda

1. Einführung in relationale Datenbanken mit mehreren Tabellen
2. Erstellung mehrerer Tabellen in SQLite
3. Komplexere Abfragen (Joins)
4. Beispiel einer Universitätsdatenbank
5. Praktische Implementierung in Python

Einführung in relationale Datenbanken

- Relationale Datenbanken bestehen aus mehreren Tabellen.
- Diese Tabellen sind über Beziehungen (z.B. Fremdschlüssel) miteinander verknüpft.
- **Joins** werden verwendet, um Daten aus mehreren Tabellen zu kombinieren.

Tabellenstruktur für unser Beispiel

- **Students**: Informationen über die Studenten.
- **Courses**: Informationen über die Kurse.
- **Enrollments**: Eine Zuordnungstabelle, die speichert, welche Studenten in welchen Kursen eingeschrieben sind.

Erstellen der Tabellen

Tabelle: Students

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

- Diese Tabelle speichert die Basisinformationen über die Studenten.

Tabelle: Courses

```
cursor.execute('''
CREATE TABLE IF NOT EXISTS courses (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    course_name TEXT NOT NULL,
    course_code TEXT NOT NULL
)
''')
```

- Die Tabelle `courses` speichert die verfügbaren Kurse.

Tabelle: Enrollments

```
cursor.execute('''
CREATE TABLE IF NOT EXISTS enrollments (
    student_id INTEGER,
    course_id INTEGER,
    FOREIGN KEY (student_id) REFERENCES students(id),
    FOREIGN KEY (course_id) REFERENCES courses(id),
    PRIMARY KEY (student_id, course_id)
)
''')
```

- Die Tabelle `enrollments` verknüpft Studenten mit Kursen.

Daten hinzufügen

Hinzufügen von Kursen

```
def course_hinzufuegen(course_name, course_code):
    cursor.execute('''
    INSERT INTO courses (course_name, course_code)
    VALUES (?, ?)
    ''', (course_name, course_code))
    conn.commit()
    print(f'Kurs {course_name} hinzugefügt.')
```

Student zu einem Kurs anmelden

```
def enrollment_hinzufuegen(student_id, course_id):
    cursor.execute('''
    INSERT INTO enrollments (student_id, course_id)
    VALUES (?, ?)
    ''', (student_id, course_id))
    conn.commit()
    print(f'Student mit ID {student_id} in Kurs mit ID {course_id} eingeschrieben.')
```

Daten abrufen

Abfrage: Alle Studenten in einem Kurs

```
def studenten_im_kurs(course_name):
    cursor.execute('''
    SELECT students.name, students.age
    FROM students
    JOIN enrollments ON students.id = enrollments.student_id
    JOIN courses ON enrollments.course_id = courses.id
    WHERE courses.course_name = ?
    ''', (course_name,))
    studenten = cursor.fetchall()
    print(f'Studenten im Kurs {course_name}:')
    for student in studenten:
        print(student)
```

Erklärung:

- Hier verwenden wir einen **INNER JOIN**, um die Studenten zu finden, die in einem bestimmten Kurs eingeschrieben sind.

Abfrage: Alle Kurse eines Studenten

```
def kurse_von_student(student_name):
    cursor.execute('''
    SELECT courses.course_name, courses.course_code
    FROM courses
    JOIN enrollments ON courses.id = enrollments.course_id
    JOIN students ON enrollments.student_id = students.id
    WHERE students.name = ?
    ''', (student_name,))
    kurse = cursor.fetchall()
    print(f'Kurse von {student_name}:')
    for course in kurse:
        print(course)
```

Erklärung:

- Diese Abfrage gibt alle Kurse aus, in die ein bestimmter Student eingeschrieben ist.

Beispielaufrufe

Beispiel: Kurse hinzufügen

```
course_hinzufuegen("Informatik", "INF101")
course_hinzufuegen("Mathematik", "MAT102")
```

Beispiel: Studenten einschreiben

```
enrollment_hinzufuegen(1, 1)  # Student mit ID 1 in Kurs mit ID 1
enrollment_hinzufuegen(2, 2)  # Student mit ID 2 in Kurs mit ID 2
```

Erweiterung: Einschreibung basieren auf dem Namen des Studenten

```
def enrollment_hinzufuegen_via_name(student_name, course_id):
    # Suche die ID des Studenten anhand des Namens
    cursor.execute('''
    SELECT id FROM students WHERE name = ?
    ''', (student_name,))
    student = cursor.fetchone()

    if student:
        student_id = student[0]
        cursor.execute('''
        INSERT INTO enrollments (student_id, course_id)
        VALUES (?, ?)
        ''', (student_id, course_id))
        conn.commit()
        print(f'Student {student_name} in Kurs mit ID {course_id} eingeschrieben.')
    else:
        print(f'Student {student_name} nicht gefunden.')
```

Beispiel: Studenten einschreiben basierend auf dem Namen

```
enrollment_hinzufuegen_via_name("Max Mustermann", 1)
```

- Dasselbe können wir auch für den Kurs machen...

```
studenten_im_kurs("Informatik")
```

Erweiterung: Einschreibung basieren auf dem Namen des Studenten

```
def enrollment_hinzufuegen_via_name(student_name, course_id):
    # Suche die ID des Studenten anhand des Namens
    cursor.execute('''
    SELECT id FROM students WHERE name = ?
    ''', (student_name,))
    student = cursor.fetchone()
    # Suche die ID des Kurses anhand des Namens
    cursor.execute('''
    SELECT id FROM courses WHERE course_name = ?
    ''', (course_name,))
    course = cursor.fetchone()

    if student and course:
        student_id = student[0]
        course_id = course[0]
        cursor.execute('''
        INSERT INTO enrollments (student_id, course_id)
        VALUES (?, ?)
        ''', (student_id, course_id))
        conn.commit()
        print(f'Student {student_name} in Kurs {course_id} eingeschrieben.')
    elif !student:
```

puuh... das war eine Menge Code!

- Aber es ist wichtig, um zu verstehen wie SQLite und Python zusammenarbeiten.
- Mit SQLite können wir komplexe Datenstrukturen und Abfragen erstellen.
- Python bietet eine einfache Möglichkeit, auf SQLite-Datenbanken zuzugreifen und sie zu verwalten.

Zusammenfassung

- **Mehrere Tabellen** ermöglichen komplexere Datenstrukturen und Abfragen.
- **Joins** sind wichtig, um Daten aus verschiedenen Tabellen zu kombinieren.
- Mit SQLite und Python können wir leicht mehrere Tabellen verwalten und darauf zugreifen.