LE04 – Einführungs SQL und Python mit SQLite-Datenbank

Mit Python kann auch auf eine Datenbank zugegriffen werden. Die Kommunikation mit der Datenbank erfolgt über eine vereinheitlichte Schnittstelle. Als Abfragesprache verwenden wir auch hier SQL, welche wir in unserem Python-Code einbetten.

Als Standarddatenbank für Python gilt SQLite. Es können aber auch andere DBs wie MySQL, PostgreSQL, MS SQL Server etc. verwendet werden.

Voraussetzung für das Arbeiten mit der gewünschten DB ist die Verwendung eines passenden Python-Moduls.

Datenbank	Python-Modul
SQLite	sqlite3
MySQL	pymysql
PostgreSQL	postgresql, Psycopg
Microsoft SQL Server	pyodbc, pymssql

Einführungsbeispiel mit SQLite und chinook.db

Zeige alle Songs, welche zum Album *Facelift* gehören:

HelloDB.py

```
1  import os # (8)
2  import sqlite3 # (1)
3
4  # Datenbank öffnen und verbinden
5  conn = sqlite3.connect(os.path.join('C:/Users/tom/WORKLOCAL/PythonProj/UE04/','chinook.db'))
6  # (2)
7
8  # Cursor kreieren. Dieser wird im Zusammenhang mit einem Query verwendet
9  cur = conn.cursor() # (3)
10
11 # Query mit Cursor ausführen
12 cur.execute('SELECT * FROM songs WHERE album = \'Facelift\';') # (4)
13
14 # Bearbeite jede Zeile des Result-Sets
15 for row in cur.fetchall():  # (5)
16     print(row)
17
18 # Schliesse Cursor und Verbindung
19 cur.close() # (6)
   conn.close() # (7)
```

1. 🙋 Damit mit SQLite gearbeitet werden kann, muss das Modul `sqlite3` importiert werden

2. 🙋‍♂️ Aufbau der Datenbankverbindung mit `connect`-Funktion. Diese gibt ein `Connection`-Objekt zurück. Hier mit dem Namen `connection`. Wenn es die Datei `chinook.db` in diesem Verzeichnis nicht gibt, wird eine leere Datenbank mit diesem Namen erstellt.
3. 🙋‍♂️ Um mit der verbundenen Datenbank zu arbeiten, werden Cursors (Positionszeiger) verwendet. Mit Cursors können wir Datensätze verändern oder abfragen. Eine Datenbankverbindung kann beliebig viel Cursors haben. Ein neuer Cursor wird mit der `cursor`-Methode des `Connection`-Objektes erzeugt.
4. 🙋‍♂️ Das SQL-Statement wird mir der `execute`-Methode des `Cursor`-Objektes an die SQLite-Datenbank gesendet.
5. 🙋‍♂️ Das Result-Set der Abfrage wird mit `cursor.fetchall` zurückgegeben. Der Rückgabewert von `fetchall` ist eine Liste, die für jeden Datensatz ein Tupel mit den Werten der angeforderten Spalten enthält.
6. 🙋‍♂️ Close the cursor
7. 🙋‍♂️ Close the connection
8. 🙋‍♂️ wird benötigt für `os.path.join()`

Wenn sie mit `SELECT * ...` abfragen, richtet sich die Reihenfolge der Spaltenwerte nach der Reihenfolge, in welcher die Spalten mit `CREATE` definiert wurden.

Mit `fetchall` werden immer alle Ergebnisse einer Abfrage auf einmal aus der Datenbank geladen und dann gesammelt ausgegeben. Diese Methode eignet sich allerdings nur für relativ kleine Datenmengen, da erstens das Programm so lange warten muss, bis die Datenbank alle Ergebnisse ermittelt und zurückgegeben hat, und zweitens das Resultat komplett als Liste im Speicher gehalten wird. Für Operationen wie Bildschirmausgaben, die jeweils nur einen einzelnen Datensatz benötigen, ist dies bei sehr umfangreichen Ergebnissen eine Verschwendung von Arbeitsspeicher. Aus diesem Grund gibt es die Möglichkeit, die Daten zeilenweise, also immer in kleinen Portionen, abzufragen. Sie erreichen durch dieses Vorgehen, dass Sie nicht mehr auf die Berechnung der kompletten Ergebnismenge warten müssen, sondern schon währenddessen mit der Verarbeitung beginnen können. Ausserdem kann so der Arbeitsspeicher effizienter genutzt werden.

Mit der Methode `fetchone` der `cursor`-Klasse fordern wir jeweils ein Ergebnis-Tupel an. Wurden bereits alle Datensätze der letzten Abfrage ausgelesen, gibt `fetchone` den Wert `None` zurück. Damit lassen sich auch grosse Datenmengen speichereffizient auslesen. Ersetzen Sie Zeilen 14 und 15 mit:

Variante mit fetchone

```
while row:= cur.fetchone():
    print(row)
```

Datentypen bei SQLite

Die automatische Datentypumwandlung von Daten aus SQLite in Python-Datentypen richten sich nach folgender Tabelle:

SQLite-Datentyp als Quelle	Python-Datentyp als Ziel
NULL	None

REAL	float
TEXT	str
BLOB	bytes

Frage

Wie können wir in den Rückübersetzungsprozess eingreifen, um Daten beim Auslesen aus der Datenbank unseren Vorstellungen entsprechend anzupassen? Diese Frage werden wir mit den Factory-Funktionen beantworten.

Factory-Funktionen mit Python und SQLite

Es kann sein, dass die Daten in der Datenbank nicht in der Form vorliegen, wie gewünscht. Deshalb kann man diese Rohdaten in der Datenbank in einer "Factory" mit Python aufbereiten. Wir schauen uns dazu zwei Factories an.

Connection.text_factory

Hier nehmen wir als Beispiel, dass wir die Daten alle in Grossbuchstaben erhalten wollen. Es sollen also alle Werte, welche in der Datenbank als TEXT abgespeichert sind in Grossbuchstaben umgewandelt werden.

Jede von `sqlite3.connect` erzeugte Connection-Instanz hat ein Attribut `text_factory`, das eine Referenz auf eine Funktion enthält, die immer dann aufgerufen wird, wenn `TEXT`-Spalten ausgelesen werden. Im Ergebnis-Tupel der Datenbankabfrage steht dann der Rückgabewert dieser Funktion. Standardmässig ist das `text_factory`-Attribut auf die Built-in Function `str` gesetzt. Dies können Sie sich durch Python mit Zeilenkommandos anzeigen lassen:

```
>>> import sqlite3
>>> conn=sqlite3.connect("chinnook.db")
>>> conn.text_factory
<class 'str'>
```

In unserer Beispielabfrage aller Songs eines Albums erstellen wir nun eine `text_factory`-Funktion, welche alle TEXT-Spalten-Werte in Grossbuchstaben umwandelt. Diese Funktion muss einen Parameter erwarten und den konvertierten Wert zurückgeben. Der Parameter ist ein byte-String, der die Rohdaten aus der Datenbank mit UTF-8-codiert enthält. Die Funktion muss den ausgelesenen Wert erst in einen String umwandelen und dann mit der `upper`-Methode alle Buchstaben zu Grossbuchstaben umwandeln.

text_factory.py

```
1
2
3
4
5
6
7
8
9
10
```

```
1112131415161718192021222324 25
```

```
import os
import sqlite3

# Datenbank öffnen und verbinden
conn =
sqlite3.connect(os.path.join('C:/Users/tom/WORKLOCAL/PythonProj/UE6

def my_text_factory_BIGLETTERS(value):
    return value.decode("utf-8", "ignore").upper()
    # Alternativ: return str(value.upper(),encoding="utf-8")

conn.text_factory = my_text_factory_BIGLETTERS

# Cursor kreieren. Dieser wird im Zusammenhang mit einem Query
verwendet
cur = conn.cursor()

# Query mit Cursor ausführen
cur.execute('SELECT * FROM songs WHERE album = \'Facelift\';')

# Bearbeite jede Zeile des Result-Sets
for row in cur.fetchall():
    print(row)

# Schliesse Cursor und Verbindung
cur.close()
conn.close()
```

Damit Sie den Unterschied in der Ausgabe schön sehen und das ursprüngliche Ausgabeverhalten wieder herstellen können, setzen Sie in Zeile 11

```
conn.text_factory = str
```

Connection.row_factory

Row-Factory mit eigener Funktion

Ein ähnliches Attribut wie `text_factory` für `TEXT`-Spalten existiert auch für ganze Datensätze. In dem Attribut `row_factory` kann eine Referenz auf eine Funktion gespeichert werden, die Zeilen für das Benutzerprogramm aufbereitet. Standardmässig wird die Funktion `tuple` benutzt. Wir wollen beispielhaft eine Funktion implementieren, die uns auf die Spaltenwerte eines Datensatzes über die Namen der jeweiligen Spalten zugreifen lässt. Das Ergebnis soll folgendermassen aussehen:

```
{'song': 'We Die Young', 'album': 'Facelift', 'artist': 'Alice In Chains', 'genre': 'Rock',
'duration': 152084, 'price': 0.99}
{'song': 'Man In The Box', 'album': 'Facelift', 'artist': 'Alice In Chains', 'genre': 'Rock',
'duration': 286641, 'price': 0.99}
{'song': 'Sea Of Sorrow', 'album': 'Facelift', 'artist': 'Alice In Chains', 'genre': 'Rock',
'duration': 349831, 'price': 0.99}
{'song': 'Bleed The Freak', 'album': 'Facelift', 'artist': 'Alice In Chains', 'genre': 'Rock',
'duration': 241946, 'price': 0.99}
{'song': "I Can't Remember", 'album': 'Facelift', 'artist': 'Alice In Chains', 'genre': 'Rock',
'duration': 222955, 'price': 0.99}
{'song': 'Love, Hate, Love', 'album': 'Facelift', 'artist': 'Alice In Chains', 'genre': 'Rock',
'duration': 387134, 'price': 0.99}
{'song': "It Ain't Like That", 'album': 'Facelift', 'artist': 'Alice In Chains', 'genre': 'Rock',
'duration': 277577, 'price': 0.99}
{'song': 'Sunshine', 'album': 'Facelift', 'artist': 'Alice In Chains', 'genre': 'Rock',
'duration': 284969, 'price': 0.99}
{'song': 'Put You Down', 'album': 'Facelift', 'artist': 'Alice In Chains', 'genre': 'Rock',
'duration': 196231, 'price': 0.99}
{'song': 'Confusion', 'album': 'Facelift', 'artist': 'Alice In Chains', 'genre': 'Rock',
'duration': 344163, 'price': 0.99}
{'song': 'I Know Somethin (Bout You)', 'album': 'Facelift', 'artist': 'Alice In Chains', 'genre':
'Rock', 'duration': 261955, 'price': 0.99}
{'song': 'Real Thing', 'album': 'Facelift', 'artist': 'Alice In Chains', 'genre': 'Rock',
'duration': 243879, 'price': 0.99}
```

Um dies zu erreichen, benötigen wir das Attribut `description` der `Cursor`-Klasse, das uns Informationen zu den Spaltennamen der letzten Abfrage liefert. Das Attribut `description` enthält dabei eine Sequenz, die für jede Spalte ein Tupel mit 7 Elementen bereitstellt, von denen uns aber nur das erste, nämlich der Spaltenname, interessiert:

```
>>> import os
>>> import sqlite3
>>> conn = sqlite3.connect(os.path.join('C:/Users/tom/WORKLOCAL/PythonProj/UE04/','chinook.db'))
>>> cur = conn.cursor()
>>> cur.execute('SELECT * FROM songs WHERE album = \'Facelift\';')
<sqlite3.Cursor object at 0x000002443B320AC0>
>>> cur.description
(('song', None, None, None, None, None, None), ('album', None, None, None, None, None, None),
('artist', None, None, None, None, None, None), ('genre', None, None, None, None, None, None),
('duration', None, None, None, None, None, None), ('price', None, None, None, None, None, None))
>>>
```

Mit dieser Information können wir nun die Spaltennamen in der Ausgabe aufbereiten, indem wir eine Funktion als eigene Factory verwenden, hier `row_with_column_name()`.

row_factory.py

```
1
2
3
4
5
6
7
8
9
10
11
12
```

```
13141516171819202122232 4
```

```python
import os
import sqlite3

# Datenbank öffnen und verbinden
conn =
sqlite3.connect(os.path.join('C:/Users/tom/WORKLOCAL/PythonProj/UE04/','c

def row_with_column_name(cursor, zeile):
    r = {}
    for spaltennr, spalte in enumerate(cursor.description):
        r[spalte[0]] = zeile[spaltennr]
    return r

conn.row_factory = row_with_column_name
cur = conn.cursor()

# Query mit Cursor ausführen
cur.execute('SELECT * FROM songs WHERE album = \'Facelift\';')

for row in cur.fetchall():
    print(row)

# Schliesse Cursor und Verbindung
cur.close()
conn.close()
```

Hinweis

`enumerate` erzeugt einen Iterator, der für jedes Element der übergebenen Sequenz ein Tupel zurückgibt, das den Index des Elements in der Sequenz und seinen Wert enthält. Siehe

Row-Factory mit eingebauter Funktion sqlite3.Row

Die eingebaute Funktion `sqlite3.Row` erlaubt es auf Spalten zuzugreifen via Namen oder numerischen Index. Die sqlite3.Row-Factory-Funktion sollte einer eigenen Funktion vorgezogen werden, es sei denn, Sie wollen etwas ganz Spezielles erreichen.

import_pandas.py

```
import os
import sqlite3

# Datenbank öffnen und verbinden
conn = sqlite3.connect(os.path.join('C:/Users/tom/WORKLOCAL/PythonProj/UE04','chinook.db'))

conn.row_factory = sqlite3.Row # sqlite3.Row ist eine interne Factory
cur = conn.cursor()

# Query mit Cursor ausführen
cur.execute('SELECT * FROM songs WHERE album = \'Facelift\';')

for row in cur.fetchall():
# Zugriff mit Spaltennamen oder via numerischen Index. Die Spalte artist hat index=2, daher
doppelte Ausgabe.
    print(row['artist'],row[2])
    print(type(row))  # Ausgabe des Klasse

# Schliesse Cursor und Verbindung
cur.close()
conn.close()
```

Import Data aus SQLite in Pandas-Dataframe

Hier ein Beispiel, wie man Daten aus einer SQLite-Datenbank in ein Pandas-Dataframe `df` importiert. Beachten Sie hier auch die Anwendung, wie man SQL-Statements als Variable einsetzen kann. Auch ein Cursor ist dank der Pandas-Funktion `read_sql()` nicht notwendig.

Die Bearbeitung des Datenframes kann dann auf gewohnte Art und Weise erfolgen.

row_factory.py

```
1  import os
2  import sqlite3
3  import pandas as pd
4
5  # Datenbank öffnen und verbinden
6  connection =
7  sqlite3.connect(os.path.join('C:/Users/tom/WORKLOCAL/PythonProj/UE04/','chinook.db'))
8  sqlquery = "SELECT * FROM employees;"
9  df = pd.read_sql(sqlquery,con=connection) # (1)
10 print(df)
11 # Schliesse Cursor und Verbindung
   connection.close()
```

1. 🙋 read_sql() ist eine Funktion (Methode) von Pandas, welche als Argument das SQL-Statement und die DB-Verbindung benötigt.