
LUT
University
LAND
OF THE
CURIOUS

CT60A4304 - BASICS OF DATABASE SYSTEMS

SQL AND PYTHON

Lecture

Jiri Musto, D.Sc.

TABLE OF CONTENTS

WHY DATABASES AND SOFTWARE

- As a reminder, databases are more efficient for storing and viewing data than hardcoding everything into a software
  - Also no reason to reinvent the wheel by redoing the data storage for every single software
- But databases require a software to view, analyse and modify data
  - DBMS if nothing else
- Software does not need to be complex
  - A basic to-do list application can be extremely simple but effective with a proper database

API AND ORM

- Application programming interface (API) offers a standardized way to communicate between two components
- Python has a database API that can connect to PostgreSQL, MySQL and SQLite databases
- Some object-oriented programming language support object relation mapping (ORM)
  - Python, Java, C#, C++
  - Enables direct transformation from objects to entities, classes to relations

PYTHON DB API

- https://www.python.org/dev/peps/pep-0249/
- Python DB API is a library that offers a simple interface for SQL databases
  - SQLite (by default)
  - MySQL (requires another library)
  - PosgreSQL (requires another library)
- Basic commands are:
  - Connect() Connect to the desired database, used to initialize the cursor
  - Cursor() The object that points to the database, used for all database commands
  - Execute() Execute the given SQL query

CONNECTION OBJECTS

- Creates the connection to a database
- Used to open and close the database as well as initialize the cursor() object
- Used to setting environmental settings
- Used to commit transactions

```
1  #Connect to RAM to create a database
2  db = sqlite3.connect(':memory:')
3
4  #Open or create an existing database (.db or .sqlite)
5  db = sqlite3.connect('hw2tennis.db')
```

TRANSACTIONS

- Python DB API handles SQL queries as transactions
- All modifications need to be committed to be made permanent to the database
  - If the modifications are not committed, the database will revert to before changes
- Transaction automatically starts with the connection object and ends with commit()
- Database can be reverted manually using rollback() command

CURSOR OBJECTS

- Cursor objects are used to manipulate data in the database
  - SELECT, INSERT, CREATE, UPDATE, etc.
- Created based on the connection object
- Has a basic execution command, as well as multiple specific commands

```
10  #Create cursor object
11  cursor = db.cursor()
12
13  #Execute an SQL query
14  cursor.execute('SELECT * FROM Ranking;')
```

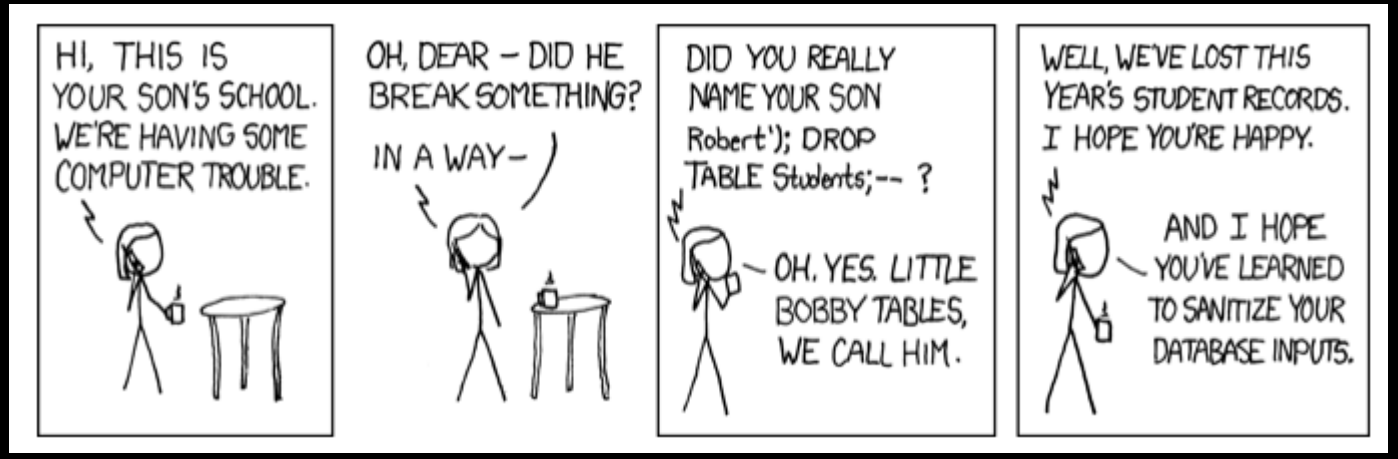
EXECUTING SQL QUERIES

- Execute SQL query
  - Execute() - Execute the SQL query
  - Executemany() - Execute multiple similar SQL queries
  - Executescript() – Executes the given sql_script, for example a text string and creates a new cursor object

```
13  #Execute an SQL query
14  cursor.execute("INSERT INTO Player VALUES ('Emil', 'Ruusuvuori', 'FIN', '01/01/2000');")
15
16  #Execute multiple SQL queries
17  data = [
18          (31, 'Emil', 'Ruusuvuori', 'FIN', '01/01/2000'),
19          (32, 'Jarkko', 'Nieminen', 'FIN', '23/07/1981'),
20          (33, 'Veli', 'Paloheimo', 'FIN', '13/12/1967')
21      ]
22  cur.executemany("INSERT INTO Player VALUES (?, ?, ?, ?, ?)", data)
23  db.commit()
```

SQL-INJECTIONS

- SQL queries are strings that end with ;
- Never use user input directly in a query



INSERT DATA TO DATABASE

- "Unsafe"
  - Put variables directly inside the SQL query
  - Treats everything as a string
  - Can be used when SQL queries are made by developer, should NOT be used with user input
- Safe
  - Recommended to be used everytime, SHOULD be used with user input
  - Use ? –marks in place of variables, have them as query parameters
  - The parameters are validated before inserting into the query and executed

```
33  #Unsafe insert
34  cursor.execute("INSERT INTO Player VALUES (31, 'Emil', 'Ruusuvuori', 'FIN', '01/01/2000');")
35  pid = 31
36  first_name = "Emil"
37  last_name = "Ruusuvuori"
38  nationality = "FIN"
39  birthdate ="01/01/2000"
40  cursor.execute("INSERT INTO Player VALUES ('"+pid+"', '"+first_name + "', '"+last_name + "', '"+nationality + "', '"+birthdate + "');")
41
42  #Safe insert
43  cursor.execute("INSERT INTO Player VALUES (?, ?,?,?,?);", (pid,first_name, last_name, nationality, birthdate))
```

```
46  userInput = "'; update users set admin = 'true' where username = 'hacker'; select true; --"
47  cursor.execute("INSERT INTO Player VALUES ('"userInput"')")
48  #vs
49  cursor.execute("INSERT INTO Player VALUES (?)", (userInput,))
```

EXAMPLE OF USING PYTHON AND SQL

```
58  #Basic template
59
60  #Open or create an existing database (.db or .sqlite)
61  db = sqlite3.connect('hw2tennis.db')
62
63  #Create cursor object
64  cursor = db.cursor()
65
66  #Execute an SQL query
67  cursor.execute("SELECT * FROM Player;")
68
69  #Retrieve all results and store them in a variable
70  results = cursor.fetchall()
71
72  #Print results
73  print(results)
```

RETRIEVING DATA FROM DB

- Fetch commands are used after executing the SQL query
  - Used after executing a SELECT query
  - Fetchone() - Retrieve the next result of the SQL query
  - Fetchmany() - Retrieve the next X amount of results of the SQL query
  - Fetchall() - Retrieve all (remaining) results of the SQL query

```
#Retrieve next result
cursor.fetchone()

#Retrieve 10 next results
cursor.fetchmany(10)

#Retrieve all remaining results
cursor.fetchall()
```

ITERATING

- It is possible to go through each result row by row using iterating
  - When executing a SELECT query
  - After fetching multiple results

```
77  #Iterate after execute
78  for row in cursor.execute('SELECT * FROM Player'):
79      print(row)
80
81  #Iterate after fetching all
82  cursor.execute('SELECT * FROM Player')
83  results = cur.fetchall()
84  for row in results:
85      print(row)
```

PROGRAMMING IS NOT THE MAIN SUBJECT OF THIS COURSE

- The purpose of this course is to learn to use databases
  - Connecting a database into a simple software is part of the process
- No need to create a beautifully working graphical user interface
  - Basic console structure is enough

```
Menu options:
1: Print Players
2: Print Ranking
3: Print Matches
4: Search for one player
0: Quit
What do you want to do?
```

```python
def main():
    userInput = -1
    while(userInput != "0"):
        print("Menu options:")
        print("1: Print Players")
        print("2: Print Ranking")
        print("3: Print Matches")
        print("4: Search for one player")
        print("0: Quit")
        userInput = input("What do you want to do? ")
        print(userInput)
        if userInput == "1":
            printPlayers()
        if userInput == "2":
            printRanking()
        if userInput == "3":
            printMatches()
        if userInput == "4":
            searchPlayer()
        if userInput == "5":
            insertPlayers()
        if userInput == "0":
            print("Ending software...")
    db.close()

main()
```

EXAMPLE: CREATE DATABASE

- You can create the whole database in multiple ways
  1. Create all necessary commands inside the software (bad choice, not reusable)
  2. Create all necessary commands in a separate file (good choice, reusable)

SQL commands in a file

```sql
CREATE TABLE  Player (
    playerid INTEGER PRIMARY KEY AUTOINCREMENT,
    first_name varchar(50) NOT NULL,
    last_name varchar(50) NOT NULL,
    nationality varchar(50) NOT NULL,
    birthdate varchar(50) NOT NULL
);


CREATE TABLE  Matches (
    matchid INTEGER PRIMARY KEY AUTOINCREMENT,
    FK_playerOne int NOT NULL,
    FK_playerTwo int NOT NULL,
    resultSets varchar(50),
    matchdate varchar(50),
    winnerID int,
```

Initialize DB using the separate file

```python
def initializeDB():
    f = open("sqlcommands.sql", "r")
    commandstring = ""
    for line in f.readlines():
        commandstring+=line
    cur.executescript(commandstring)

```

EXAMPLE: USING RETRIEVED ROWS

- If you want to access specific attributes, there are two possible ways
- Using array notation
  - row[n], n=the attribute position you want to access starting from 0
  - Need to know the order of the attributes
  - Works without a changes to anything else
- Using key-value pair
  - row[‘key’], key=attribute name
  - Intuitive and easy to use
  - Requires some changes to the Python code
  - Use something called “row factory”

```
90  db = sqlite3.connect('newDB.sqlite')
91  db.row_factory = sqlite3.Row    #This is needed if using key-value pair
92  cur = db.cursor()
93  cur.execute("SELECT * FROM Player WHERE last_name ='Nadal'")
94  oneRow = cur.fetchone()
```

```
96  #Array form
97  print("ID: " + str(oneRow[0]))
98  print("First name: " + oneRow[1])
99  print("Last name: " + oneRow[2])
100 print("Birthdate: " + oneRow[4])
101 print("Nationality: " + oneRow[3])
102
103 #Key-value form
104 print("ID: " + str(oneRow['playerid']))
105 print("First name: " + oneRow['first_name'])
106 print("Last name: " + oneRow['last_name'])
107 print("Birthdate: " + oneRow['birthdate'])
108 print("Nationality: " + oneRow['nationality'])
```

```
108 #Changes needed if using the key-value
109 #Each fetched result will now be an object and handled as such
110 for row in cur.fetchall():
111     print(row)              #Normal, when row_factory is disabled
112     print(tuple(row))       #If using key-value, row_factory is  enabled
```

CT60A4304 - BASICS OF DATABASE SYSTEMS

TRANSACTIONS

Lecture

Jiri Musto, D.Sc.

ACID PROPERTIES

- ACID properties contains the integrity requirements for (relational) databases
- ***Atomicity***: The transaction is atomic, either everything is executed correctly or everything fails (roll back to before the transcation was started)
- **Consistency**: The transaction changes the database from one consistent state to another consistent state. Database is consistent when it follows the internal rules and logic of the database
- **Isolation**: Transactions are done in isolation and the intermediate events will not be visible to other transactions, regardless if the transactions are concurrent or serialized.
- **Durability**: The transaction is durable. If the transaction is committed successfully, the results will not disappear.

TRANSACTIONS

- There are some commands for transaction control
  - Databases can be operated without transactions
  - Basic commands are BEGIN, END, COMMIT and ROLLBACK
- Database is used in parallel by multiple users
  - The real need for transactions is visible during actual usage
- With transactions, closely related queries are made into one package
  - Either all succeeds or fails
  - Database stays consistent

CONCURRENCY CONTROL

- Databases operate simultaneously
  - Multiple users, databases, tables, rows
- Parallel transactions are serialized into a timeline
  - Timestamps, locking tables
  - Transactions operated independently from each other but one may prevent another from working
- Transactions indicate that the queries depends on each other and need to be separated

- Example: Bank

LOCKING MECHANISMS

- Binary lock
  - Used record (database, table, row) is either locked or unlocked.
  - Has some benefits to it compared to read/write locking
  - SQLite uses binary locking on database (only one write allowed at a time) but transactions have their own mechanism
- Shared / exclusive lock
  - Read (shared lock): Reading is allowed by multiple people at the same time
  - Write (exclusive lock): Writing blocks everyone else from using the record
  - Used extensively in distributed databases
  - Most used locking mechanism
- There are different protocols for locking
  - Intent locking, simplistic lock, pre-claiming locking
  - Two-phase locking
  - Etc.

ISOLATION LEVELS

- Serializable: Transactions results are fully isolated from each other. The record-in-use and all other rows used are locked until transaction is completed.
- Repeatable read: Record-in-use is exclusively locked until transaction is over. Referenced rows have shared locks and modified rows are exclusively locked.
- Read committed: Transaction locks the current row in use. Uncommitted changes made by other transactions are not visible.
- Read uncommitted: Transactions are not isolated. Transactions may read uncommitted changes made by other transactions.

SQL AND TRANSACTION CONTROL LANGUAGE (TCL)

BEGIN TRANSACTION	Start the transaction
ROLLBACK	Revert the transaction
COMMIT	Save (commit) the results of transaction.
END TRANSACTION	End transaction

- SQLite offers different behaviours for transactions
- DEFERRED: Default behaviour, transaction starts after database is accessed.
- IMMEDIATE: Transaction starts immediately after connection is established.
- EXCLUSIVE: Similar to immediate but prevents others from reading the database

SQLITE TRANSACTIONS

```
1  BEGIN
2  INSERT INTO Player
3  VALUES (31, 'Veli', 'Paloheimo', 'FIN', '13/12/1967')
4  INSERT INTO Ranking
5  VALUES (31, 100, 1, 'W: 10 - L: 0', 31)
6  COMMIT
7
```

PYTHON AND TRANSACTIONS

- Connection.commit() and connection.rollback() commands are possible
- Transaction begins automatically after connecting with the first write, ends with commit()

```
117
118 try:
119     c = sql.cursor()
120     c.execute("BEGIN IMMEDIATE")
121     c.execute("SELECT COUNT(*) FROM course WHERE courseCode=?", userInput)
122     count = cursor.fetchone()
123
124     c.execute("COMMIT")
125 except sql.Error:
126     print("Error!")
127     c.execute("ROLLBACK")
```


LUT
University