Table of Contents

Python und Datenbanken

Python bietet sehr einfach die Nutzung von Datenbankmanagementsysteme (DBMS) an. Im Folgenden wird anstelle des sperrigen Wortes einfach von Datenbanksystem gesprochen. Was ist der Vorteil eines Datenbanksystems?

Wir trennen dadurch Daten von Programm und müssen uns nicht kümmern um:

- die Speicherung der Daten
- die Verwaltung der Daten

Wir übergeben einfach unserer Datenbank unsere Daten und im Folgenden können wir dann über unser Programm bestimmen, was mit den Daten passieren soll. Meistens möchte man diese in irgendeiner Form auswerten oder einfach nur ausgeben. Wobei bei der Ausgabe gerne eine Auswahl (Selektion) stattfindet, damit nur gerade benötigte Daten angezeigt werden.

Konkretes Beispiel, das wir Schritt für Schritt umsetzen

Wie sieht das in einem konkreten Beispiel aus?

Wir wollen **einen Geburtstagswarner programmieren, der seine Daten in einer Datenbank speichert**. Dazu benötigen wir Vorname, Nachname und das Datum des Geburtstags. Wer mag, kann noch Telefonnummer und E-Mail mit in die Daten aufnehmen. Im ersten Schritt müssen die Daten über ein Formular eingegeben und gespeichert werden können.

Diese Daten können durchsucht werden.

Zusätzlich können die Daten ausgegeben werden mit der gewünschten Sortierreihenfolge, z.B. nach dem Vornamen oder Nachnamen oder Geburtsdatum.

Und als Bonbon sollen immer die kommenden 3 Geburtstage beim Start des Programms angezeigt werden. Wir haben hier also eine Selektion der Daten nach einem bestimmten Kriterium und eine sortierte Ausgabe.

Und zu guter Letzt wollen wir auch wieder Daten ändern (Tippfehler kommen vor) und auch löschen (manchen Menschen möchte man nicht mehr zum Geburtstag gratulieren) können.

Das zur Funktion. Was bietet uns Python zum Thema Datenbanken?

Verschiedene Datenbanksysteme

Mit Python können wir verschiedene Datenbanksysteme wie MySQL, SQLite, Oracle, PostgreSQL und weitere Datenbanken nutzen.

Über ein Datenbanksystem kann man sehr einfach Daten verwalten. Dazu gehört:

- Datensätze anlegen
- Datensätze löschen
- Daten ändern
- Daten suchen

- Daten sortieren

Im folgenden Kapitel wollen wir mit SQLite arbeiten. Diese Variante ist ein optimaler Einstieg in das Thema Datenbanken. Die grundsätzliche Programmierung ist bei allen Datenbanken die Gleiche. Der Vorteil bei SQLite ist, dass dieses bereits als Modul in Python vorhanden ist und direkt als Datei bei dem Softwareprojekt eingebettet ist. Wer im Dateisystem nachsehen mag – es ist eine „.db“ -Datei.

Der große Vorteil für den Einsteiger ist: Wir müssen keine Client-Server-Datenbank wie beispielsweise bei MySQL installieren. So kann auch sehr einfach die Anwendung mit der Datenbank und bereits erfasste Daten weitergegeben werden.

Aber Schritt für Schritt.

Unsere Lernanwendung für Datenbank-Nutzung

Im Folgenden wollen wir die Anwendung „Geburtstagswarner“ programmieren, die Geburtstage verwaltet. Dafür nutzen wir die folgenden Datenfelder:

- Vorname
- Nachname
- Geburtstag

Als Aktionen gibt es:

- Datensatz anlegen
- anzeigen
- ändern
- löschen
- suchen
- sortieren

Dies setzen wir Schritt für Schritt in den folgenden Kapiteln um und lernen so den Umgang mit Datenbanken in Python.

Grundsätzliche Vorgehensweise für Nutzung von Datenbankzugriffen (hier SQLite)

Für die Nutzung von Datenbanken mit Python gibt es eine grundsätzliche Vorgehensweise, die wir in 5 Schritten hier ansehen möchten. Dabei ist für die meisten verfügbaren Datenbanken (im folgenden SQLite) die Vorgehensweise nahezu identisch. Im folgenden Kapitel bauen wir eine Verbindung zur Datenbank auf und führen eine Aktion mit der Datenbank aus (im Beispiel erzeugen wir eine leere Tabelle).

Wir starten mit SQLite, weil es uns gleich mehrere Vorteile bietet (wie in der einführenden Beschreibung erwähnt). Es ist bereits in Python vorhanden und das Modul kann einfach importiert

werden. Die Daten selber werden in einer Datei mit der Dateiendung „.db“ gespeichert und können somit problemlos weitergegeben werden.

erster Schritt: SQL-Modul importieren

Als Erstes müssen wir unser SQL-Modul importieren:

```
import sqlite3
```

Die folgenden 5 Schritte sind für die Interaktion mit Datenbanken meistens notwendig:

1. Verbindung zur Datenbank herstellen
2. Cursor-Objekt (zum Zugriff auf die Datenbank)
3. SQL-Query übergeben
4. commit: wir bestätigen der Datenbank, dass wir die SQL-Anweisung wirklich ausführen lassen wollen
5. Schließen der Datenbankverbindung (Ordnung muss sein)

Schritt 1: Unsere Verbindung zur Datenbank aufbauen

Wir wollen nun eine Verbindung zu unserer Datenbank aufbauen. Dazu benötigen wir einen Datenbanknamen, auf den wir dann über eine Variable (Zeiger) zugreifen können.

Schauen wir uns erst den Befehl an und dann kommt die Erklärung:

```
verbindung = sqlite3.connect("geburtstage.db")
```

Wir bauen also eine Verbindung (engl. „connection“ und das Verb verbinden „connect“) zu unserer Datenbank „geburtstage.db“ auf. Gibt es diese Datenbank noch nicht, legt unser Datenbanksystem „SQLite“ automatisch beim ersten Aufruf eine Datei im selben Ordner an.

Möchte man es nicht im gleichen Ordner wie die Python-Programme haben, dann einfach den gewünschten Unterordner angeben (der Unterordner sollte bereits angelegt sein):

```
verbindung = sqlite3.connect("datenbank/geburtstage.db")
```

Ist der Ordner nicht angelegt, schlägt das Anlegen der Datenbank fehl und wir erhalten die Fehlermeldung: „sqlite3.OperationalError: unable to open database file“

Also einfach Ordner im Betriebssystem anlegen, bevor wir unseren `connect`-Befehl darauf loslassen!

Schritt 2: Cursor-Objekt cursor()

Im nächsten Schritt wird das Cursor-Objekt eingerichtet. Was ist ein Cursor eigentlich? Man verwendet schon immer auf dem Bildschirm den Cursor, hat sich aber noch nie gefragt, woher dieses Wort wohl kommt. Wie das meiste entweder aus dem Lateinischen oder Griechischen. Im Lateinischen hat das Wort Cursor die Bedeutung von „Läufer“. Mit dem Cursor wird die aktuelle Bearbeitungsposition auf dem Bildschirm gezeigt. Mit dem Datenbankcursor das gleiche – aber auf

den Datenbanksatz bezogen. Er zeigt also die aktuelle Position beim Lesen bzw. Schreiben von Datensätzen.

So ein Objekt wollen wir nun erzeugen:

```
import sqlite3
verbindung = sqlite3.connect("datenbank/geburtstage.db")
zeiger = verbindung.cursor()
```

Ab jetzt können wir unseren Zeiger verwenden.

Schritt 3: SQL-Code erstellen und an Datenbank senden: execute()

Jetzt können wir unsere SQL-Anweisung an die Datenbank übergeben. Mit dieser Anweisung sagen wir der Datenbank, was zu tun ist: Ob Beispielsweise ein neuer Datensatz angelegt wird oder die Datenbank bestimmte Datensätze zurückliefern soll, damit wir diese anzeigen können. Die Übergabe läuft als String ab. Mit der Anweisung `execute()` wird die SQL-Anweisung ausgeführt.

Bisher haben wir eine Datenbank mit dem Namen „geburtstage.db“. Allerdings haben wir noch keine Tabellen in der Datenbank. Grundsätzlich wäre der Aufbau unseres `execute`-Befehls:

```
zeiger.execute(SQL-Anweisung)
```

Was steht nun in unserer SQL-Anweisung?

Eine Tabelle besteht wie in Excel aus Zeilen und Spalten. In Excel wird für die Spalten automatisch die Benennung „A, B, C, … AA, AB“ verwendet. Das wäre für uns eher unpraktisch. Im Unterschied zu Excel vergeben wir also für unsere Datenbank für jede Spalte:

- einen Namen (damit wir die „Spalte“ ansprechen können)
- die Art der Inhalte (String, Integer, Datum etc.)
- die Feldlänge

Wir erzeugen (engl. „create“) eine Tabelle (engl. „table“) die einen Namen hat.

```
zeiger.execute("CREATE TABLE personen (vorname VARCHAR(20), nachname
VARCHAR(30), geburtstag DATE)")
```

Gerne wird eine SQL-Anweisung auch als String vorbereitet und dann dieser String der Anweisung `execute` übergeben. Das macht die SQL-Anweisung besser lesbar und somit können sich Fehler nicht so einfach einschleichen. Also nochmals die gleiche Anweisung wie oben:

```
sql_anweisung = """
CREATE TABLE personen (
vorname VARCHAR(20),
nachname VARCHAR(30),
geburtstag DATE
);"""

zeiger.execute(sql_anweisung)
```

Schritt 4: Anweisung ausführen lassen mit `commit()`

Beim Verändern von Daten in der Datenbank ist eine weitere Bestätigung zum Ausführen notwendig. Die übergebene SQL-Anweisung muss nun ausgeführt werden. Dazu wird über den Befehl `commit()` dem Datenbanksystem mitgeteilt, dass endgültig die Anweisung ausgeführt werden darf. In unserem Beispiel wird also die Tabelle angelegt:

```
import sqlite3
verbindung = sqlite3.connect("datenbank/geburtstage.db")
zeiger = verbindung.cursor()

sql_anweisung = """
CREATE TABLE personen (
vorname VARCHAR(20),
nachname VARCHAR(30),
geburtstag DATE
);"""

zeiger.execute(sql_anweisung)

verbindung.commit()
```

Schritt 5: Verbindung schließen mit close()

Aufräumen ist immer eine gute Sache. Also schließen wir die Verbindung nach getaner Arbeit über die Anweisung `close()`

```
verbindung.close()
```

kompletter Code zum Tabelle anlegen

Und hier unser kompletter Code:

```
import sqlite3
verbindung = sqlite3.connect("datenbank/geburtstage.db")
zeiger = verbindung.cursor()

sql_anweisung = """
CREATE TABLE personen (
vorname VARCHAR(20),
nachname VARCHAR(30),
geburtstag DATE
);"""

zeiger.execute(sql_anweisung)

verbindung.commit()
verbindung.close()
```

Jetzt können wir unser Programm ausführen lassen und erhalten unsere Datenbank angelegt. Auch wenn es noch keine Inhalte gibt, wird die Datei „geburtstage.db“ erzeugt und besitzt eine Dateigröße (es steckt jetzt die vorbereitete Struktur darin).

Lassen wir den Code nochmals ausführen, bekommen wir eine Fehlermeldung:

sqlite3.OperationalError: table personen already exists

Diesen Fehler können wir vermeiden, indem wir die SQL-Anweisung nur ausführen lassen, wenn noch keine Tabelle existiert. Wir erweitern dazu unsere SQL-Anweisung um „IF NOT EXISTS”:

```
sql_anweisung = """
CREATE TABLE IF NOT EXISTS personen (
vorname VARCHAR(20),
nachname VARCHAR(30),
geburtstag DATE
);"""
```

Daten in Datenbank speichern: INSERT INTO

Auch zum Eintragen von Daten in die Datenbank gibt es die entsprechende SQL-Anweisung. Diese lautet `INSERT INTO`. Als Nächstes muss die entsprechende Tabelle angegeben werden:

```
INSERT INTO tabellennamen
```

In unserem Beispiel in die Tabelle mit dem Namen „personen“:

```
INSERT INTO personen
```

Jetzt werden im nächsten Schritt die Daten (engl. „values“) übergeben – die Daten werden in Klammern geschrieben:

```
INSERT INTO personen VALUES ()
```

Die Daten werden in der Reihenfolge eingegeben, wie wir die Felder festgelegt haben. Also in unserem Beispiel: vorname, nachname, geburtstag

Wir wollen unseren „Johann Wolfgang von Goethe“ mit aufnehmen. Geboren wurde er am 28.8.1749.

Unsere SQL-Anweisung ergänzen wir entsprechend:

```
INSERT INTO personen VALUES (Johann Wolfgang von Goethe 28.8.1749)
```

So weit, so gut. In diesem Fall, nicht gut. Warum? Die Datenbank weiß nicht, was zusammengehört: wo hört der Vorname auf und wo fängt der Nachname an. Daher werden Daten, die zu einem Feld gehören, jeweils in Anführungszeichen gesetzt. Dabei nutzt man sehr gerne einfache Anführungszeichen, da unsere SQL-Anweisung selber oft als String erstellt wird.

Probieren wir es aus:

```
INSERT INTO personen VALUES ('Johann Wolfgang von', 'Goethe', '28.8.1749')
```

Ob das nun passt oder nicht, sagt uns einfach die Reaktion der Datenbank. Also testen!

Unser kompletter Python-Code bisher:

```
import sqlite3
```

```
verbindung = sqlite3.connect("datenbank/geburtstage.db")
zeiger = verbindung.cursor()

sql_anweisung = """
CREATE TABLE IF NOT EXISTS personen (
vorname VARCHAR(20),
nachname VARCHAR(30),
geburtstag DATE
);"""

zeiger.execute(sql_anweisung)

sql_anweisung = """
INSERT INTO personen VALUES ('Johann Wolfgang von', 'Goethe', '28.8.1749')
"""

zeiger.execute(sql_anweisung)

verbindung.commit()
verbindung.close()
```

Beim Ausführen bekommen wir keine Fehlermeldung. Scheint also geklappt zu haben. Spannend wird es, wenn wir die Daten wieder auslesen. Wenn die Daten exakt so wieder herauskommen, dann passt es. Oder doch nicht?

Wer auf die Auflösung der Anmerkung nicht warten kann, hier der Typ dazu. In SQLite unterstützt die folgenden Datentypen: INTEGER, REAL, TEXT, BLOB und NULL. Wenn mir mit DATE ums Eck kommen, kommt zwar keine Fehlermeldung aber effektiv wird es als TEXT gespeichert! Gespeichert ist es, nur mit Datum „rechnen“ können wir erst nach irgendwelchen Umwandlungsaktion über Python.

Variablen nutzen für die `INSERT`-SQL-Anweisung

Gerade haben wir eine SQL-Anweisung über `INSERT INTO` aufgebaut und direkt in der SQL-Anweisung die Daten mitgegeben. Das wird in den seltensten Fällen zielführend sein. Meistens liegen uns Werte in Variablen vor, die z.B. über Nutzereingaben erfasst worden sind.

Bauen wir also unsere SQL-Anweisung über Variablen auf. Wir bleiben beim gleichen Beispiel oben und ergänzen weitere bekannte Namen in unserer Datenbank, die man schon immer mal gerne in seiner Adressliste gehabt hätte.

```
nachname   = "Schiller"
vorname    = "Friedrich"
geburtstag = "10.11.1759"
```

Und jetzt bauen wir eine SQL-Anweisung (die noch nicht optimal ist!):

```
zeiger.execute("INSERT INTO personen VALUES (vorname, nachname, geburtstag)")
```

Warum ist diese Vorgehensweise so nicht gut? Über solche Konstruktionen wird die Datenbank anfällig für SQL-Injektions. Sprich von außen wird schadhafter Code eingebracht, der dann unter Umständen direkt ausgeführt wird. Schlimmstenfalls könnte die komplette Datenbank gelöscht oder Daten ausgespäht werden.

Daher wollen wir die Daten nicht ungeprüft übergeben. Der folgende Aufbau filtert die größten Probleme heraus:

```
zeiger.execute("INSERT INTO personen VALUES (?,?,?)", (vorname, nachname,
geburtstag))
```

Man sieht an der oberen Zeile, dass die SQL-Anweisung sehr lang werden kann und man nach rechts scrollen muss. Das ist natürlich einerseits unpraktisch und andererseits verbessert es nicht die Lesbarkeit. Abhilfe schafft die Technik der 3 Anführungszeichen einsetzen. Somit hat Python kein Problem mehr, wenn wir unsere Anweisung in mehrere Zeilen verteilen. Das macht besonders bei SQL-Anweisungen Sinn, da wir dann die Felder von den Variablen besser getrennt darstellen können.

Hier die Anweisung von oben sinnvoll in mehrere Zeilen umgebrochen:

```
zeiger.execute("""
                INSERT INTO personen
                       VALUES (?,?,?)
                """,
               (vorname, nachname, geburtstag)
               )
```

Und nun der komplette Code:

```
import sqlite3
verbindung = sqlite3.connect("datenbank/geburtstage.db")
zeiger = verbindung.cursor()

nachname   = "Schiller"
vorname    = "Friedrich"
geburtstag = "10.11.1759"

zeiger.execute("""
                INSERT INTO personen
                       VALUES (?,?,?)
                """,
               (vorname, nachname, geburtstag)
               )

verbindung.commit()
verbindung.close()
```

Im folgenden Kapitel speichern wir mehrere Datensätze auf einen Rutsch – nicht nur einen wie bisher.

Viele Datensätze in Datenbank speichern

Im letzten Kapitel haben wir genau einen neuen Datensatz in unserer SQLite-Datenbank gespeichert. Dazu haben wir folgenden Python-Code erstellt:

```
import sqlite3
verbindung = sqlite3.connect("datenbank/geburtstage.db")
zeiger = verbindung.cursor()

nachname   = "Schiller"
```

```
vorname    = "Friedrich"
geburtstag = "10.11.1759"

zeiger.execute("""
                INSERT INTO personen
                       VALUES (?,?,?)
                """,
               (vorname, nachname, geburtstag)
               )

verbindung.commit()
verbindung.close()
```

Wenn uns allerdings mehrere Daten vorliegen, dann möchten wir diese so effektiv wie möglich in unserer Datenbank speichern können.

Als Erstes benötigen wir die Daten selber. Dazu bieten sich natürlich Listen an. Bisher haben wir einzelne Variablen verwendet, was sehr schnell ab einer gewissen Menge an verschiedenen Variablen sehr unübersichtlich wird. Daher nutzen wir eine Liste, in der wir in einer vorgegebenen Reihenfolge unsere Namen und Geburtstagsdaten abspeichern:

```
personendaten = ("Heinrich Hermann Robert", "Koch", "11.12.1843")
```

Diese Liste kann jetzt einfach unsere SQL-Anweisung `execute` übergeben werden.

```
personendaten = ("Heinrich Hermann Robert", "Koch", "11.12.1843")

zeiger.execute("""
                INSERT INTO personen
                       VALUES (?,?,?)
                """, personendaten)
```

Was passiert aber, wenn wir eine geschachtelte Liste haben – sprich eine Liste mit vielen Personendaten?

```
beruehmtheiten = [('Georg Wilhelm Friedrich', 'Hegel', '27.08.1770'),
                  ('Johann Christian Friedrich', 'Hölderlin', '20.03.1770'),
                  ('Rudolf Ludwig Carl', 'Virchow', '13.10.1821')]
```

Wenn wir nun die geschachtelte Liste unserer SQL-Anweisung `execute` übergeben, erhalten wir eine Fehlermeldung:

sqlite3.InterfaceError: Error binding parameter 0 - probably unsupported type.

Geschachtelte Listen per SQL ausführen lassen: executemany

Und jetzt kommt es. Es gibt mehr als die `execute`-Anweisung. Über die SQL-Anweisung `executemany` werden auch geschachtelte Listen auf einen Rutsch abgearbeitet.

```
import sqlite3
verbindung = sqlite3.connect("datenbank/geburtstage.db")
zeiger = verbindung.cursor()

beruehmtheiten = [('Georg Wilhelm Friedrich', 'Hegel', '27.08.1770'),
                  ('Johann Christian Friedrich', 'Hölderlin', '20.03.1770'),
                  ('Rudolf Ludwig Carl', 'Virchow', '13.10.1821')]
```

```
zeiger.executemany("""
                INSERT INTO personen
                       VALUES (?,?,?)
                """, beruehmtheiten)

verbindung.commit()
```

Es muss aber eine geschachtelte Liste sein. Diese kann aber auch nur einen Datensatz enthalten wie im folgenden Beispiel:

```
personendaten = [("Heinrich Hermann Robert", "Koch2", "11.12.1843")]
```

Im folgenden Kapitel wollen wir auch mal schauen, was in unserer Datenbank angekommen ist. Dazu lesen wir die Datenbank aus und zeigen die Inhalte an.

Datenbank auslesen und anzeigen: SELECT * FROM

Wir wollen eine Datenbank auslesen und anzeigen. Dazu müssen wir auswählen, was angezeigt werden soll. Wir selektieren (engl. „select“) also eine Teilmenge (oder auch die kompletten Datensätze) unserer Datenbank. Am Anfang sind wir nicht wählerisch und nehmen alle Felder – sprich „Alles“ wird über den Stern „*“ ausgewählt. Jetzt müssen wir noch unsere SQL-Anweisung mitgeben, von welcher Tabelle etwas ausgelesen wird („von“ = engl. „from“):

```
SELECT * FROM tabellennamen
```

In unserem bisherigen Beispiel wäre das:

```
zeiger.execute("SELECT * FROM personen")
```

Und nun „holen“ wir die Daten ab und übergeben diese einer Liste. Am Rande bemerkt, das englische Wort für holen bzw. abholen ist „fetch“. Mit den abgeholten in einer Liste gespeicherten Daten können wir weiterarbeiten – in unserem Fall erst einmal ausgeben um zu sehen, ob die in die Datenbank geschriebenen Daten auch wieder korrekt herauskommen.

```
inhalt = zeiger.fetchall()
```

Da wir nur Daten abholen, benötigen wir keine `commit()`-Anweisung.

Jetzt noch die Ausgabe:

```
print(inhalt)
```

Der dazu notwendige komplette Code:

```
import sqlite3
verbindung = sqlite3.connect("datenbank/geburtstage.db")
zeiger = verbindung.cursor()
zeiger.execute("SELECT * FROM personen")
inhalt = zeiger.fetchall()
print(inhalt)
verbindung.close()
```

Als Feedback bekommen wir folgende Ausgabe auf dem Bildschirm:

[('Johann Wolfgang von', 'Goethe', '28.8.1749'), ('Johann Wolfgang von', 'Goethe', '28.8.1749'), ('Friedrich', 'Schiller', '10.11.1759')]

Wir haben also eine Liste mit unseren einzelnen Elementen. Unseren Goethe haben wir versehentlich 2-mal in die Datenbank gespeichert. Interessanterweise scheint das Geburtsdatum korrekt übernommen worden zu sein, obwohl wir dies in deutscher Schreibweise gespeichert haben.

Nicht alle Felder auslesen

Möchte ich aus meiner Tabelle nicht alle Felder auslesen, kann ich dies in der `SELECT`-Anweisung mitgeben. Jetzt gebe ich anstelle des Sterns (sprich nicht mehr alle Felder) meine gewünschten Felder an.

Möchte ich aus meiner Datenbank „personen“ nur noch den Nachnamen und den Geburtstag erhalten, gebe ich nur diese beiden Feldnamen an:

```
SELECT nachname, geburtstag FROM personen"
```

Und im kompletten Code:

```
import sqlite3
verbindung = sqlite3.connect("datenbank/geburtstage.db")
zeiger = verbindung.cursor()
zeiger.execute("SELECT nachname, geburtstag FROM personen")
inhalt = zeiger.fetchall()
print(inhalt)
verbindung.close()
```

Als Ergebnis erhalten wir:

[('Goethe', '28.8.1749'), ('Goethe', '28.8.1749'), ('Schiller', '10.11.1759')]

Datensätze ändern/ergänzen: UPDATE

Falls in einem Datensatz eine Änderung ansteht oder einzelne Angaben eines Datensatzes ergänzt werden müssen gibt es die SQL-Anweisung `UPDATE`.

Wir wollen unseren Datensatz von Herrn Schiller korrigieren. Schiller ist zwar primär mit dem Vornamen Friedrich bekannt, aber seine kompletten Vornamen lauten: „Johann Christoph Friedrich“

Also wollen wir unseren Datensatz aktualisieren. Schauen wir uns dazu erst die komplette SQL-Anweisung an:

```
UPDATE personen SET vorname='Johann Christoph Friedrich' WHERE
nachname='Schiller'
```

Die Anweisung `UPDATE` ist logisch: Aktualisiere (update) in der Tabelle „personen“ folgende angegebene Daten. Bei den angegebenen Daten müssen wir nur die zu änderten Daten und das entsprechende Feld über die SQL-Anweisung `SET` angeben.

Nicht alles Datensätze ändern: Datensatz auswählen

Wir grenzen unsere Datensätze über die zusätzliche SQL-Anweisung `WHERE` ein. Hier können wir festlegen, dass unsere Aktion nur für die Datensätze mit der in der `WHERE`-Bedingung formulierten Einschränkung durchgeführt wird.

In unserem Fall wollen wir unseren Herrn Schiller auswählen, also `WHERE nachname='Schiller'`. Auf das Problem, dass es mehr als einen „Schiller“ in unserer Tabelle gibt, kommen wir später noch zu sprechen.

Bauen wir unsere SQL-Anweisung um, damit wir wie gewohnt mit Variablen arbeiten können und geschützt vor SQL-Injektionen sind.

```
nachname   = "Schiller"
vorname    = "Johann Christoph Friedrich"

zeiger.execute("UPDATE personen SET vorname=? WHERE nachname=?", (vorname,
nachname))
verbindung.commit()
```

Noch einmal, da es so wichtig ist! Hier ist die korrekte formulierte Bedingung in der SQL-Anweisung mit `WHERE` extrem wichtig. Ansonsten bekommen ALLE Datensätze den neuen Vornamen. Und das würde Goethe definitiv nicht gefallen.

Nach Ausführung der Anweisung haben wir als Ergebnis:

[('Johann Christoph Friedrich', 'Schiller', '10.11.1759')]

[('Johann Christoph Friedrich', 'Schiller', '10.11.1759'), ('Johann Wolfgang von', 'Goethe', '28.8.1749'), ('Johann Wolfgang von', 'Goethe', '28.8.1749')]

Und einfach, weil es so schön ist, einmal diesen fatalen Fehler zu machen, lassen wir die WHERE-Anweisung weg:

```
zeiger.execute("UPDATE personen SET vorname=? ", (vorname, ))
verbindung.commit()
```

Anmerkung: das Komma nach `(vorname, )` ist wichtig. Dazu später mehr.

Einfach einmal probieren, was dabei rauskommt!

Löschen von Datensätzen: DELETE FROM

Jetzt wollen wir unseren doppelten Datensatz aus dem letzten Kapitel mit „Goethe“ wieder löschen. Hierzu gibt es die SQL-Anweisung `DELETE FROM`.

In unserem Beispiel wäre das:

```
zeiger.execute("DELETE FROM personen WHERE nachname=?", ('Goethe',))
```

Diese Anweisung müssen wir über `commit()` „bestätigen“.

Hier der komplette Code mit Anzeige der Datensätze:

```
import sqlite3
verbindung = sqlite3.connect("datenbank/geburtstage.db")
zeiger = verbindung.cursor()

zeiger.execute("DELETE FROM personen WHERE nachname=?", ('Goethe',))
verbindung.commit()

zeiger.execute("SELECT * FROM personen")
inhalt = zeiger.fetchall()

print(inhalt)

verbindung.close()
```

Lassen wir es aufführen. Falls eine Fehlermeldung kommt, bitte das Komma nach „('Goethe',)" beachten.

Wir bekommen als Ergebnis angezeigt:

[('Friedrich', 'Schiller', '10.11.1759')]

Ungeschickterweise sind jetzt alle Goethes den Weg der Sterblichen gegangen. Wir haben keinen einzigen mehr in der Datenbank.

Unsere SQL-Anweisung wurde exakt so ausgeführt, wie wir diese geschrieben haben. Lösche alle Personen mit dem Nachnamen „Goethe", egal wie oft diese Vorkommen und welchen Vornamen diese haben. Wäre eine „Susanne Goethe" in der Datenbank vorhanden gewesen, wäre auch diese unserem Löschangriff zum Opfer gefallen.

Hier kommt nun ein grundsätzliches Problem in unserem Datenbankdesign zum Vorschein, dass wir in den nächsten Kapiteln angehen müssen.

Bisher haben wir keine Möglichkeit exakt den einen Datensatz auszuwählen, den wir wollen. Wir könnten zwar unsere `WHERE`-Bedingung noch weiter präzisieren mit Beispielsweise der Angabe von weiteren Feldern:

```
zeiger.execute("DELETE FROM personen WHERE nachname=?, geburtstag=?", ('Goethe',
'28.8.1749'))
```

Es würden aber weiterhin mehrere Datensätze betroffen sein!

Daher sind `DELETE FROM` außerordentlich gefährlich. Was passiert wohl bei der Anweisung:

```
zeiger.execute("DELETE FROM personen")
```

Korrekt – alles wird gelöscht! Nur doof, wenn wir das eigentlich nicht wollten oder irgendwas schieflief mit der `WHERE`-Bedingung!

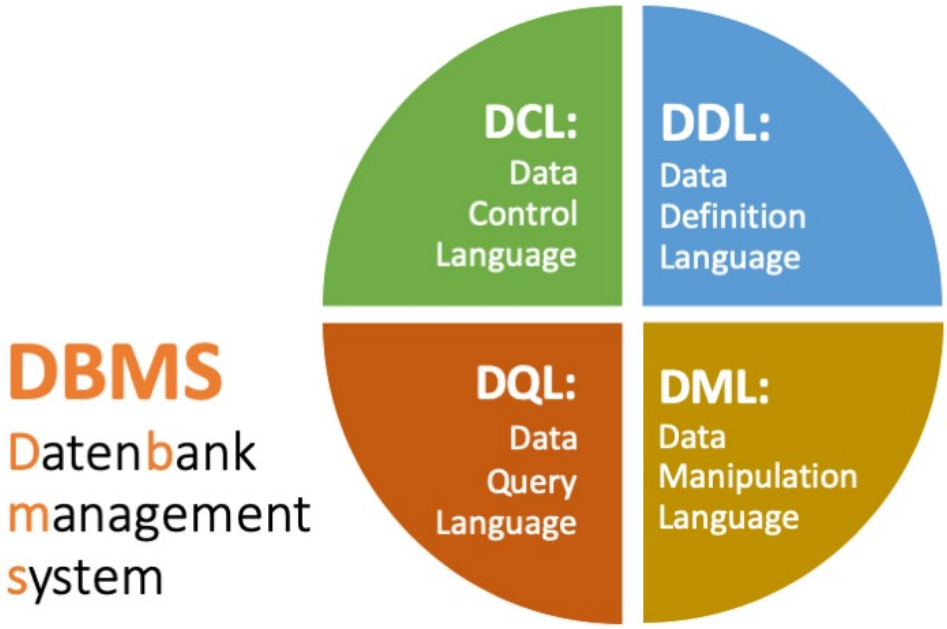
Daher benötigen wir ein exaktes einmaliges Kriterium zum Löschen, damit exakt der eine gewünschte Datensatz gelöscht werden kann.

Dafür brauchen wir zum Verständnis SQL-Grundlagen, die in den folgenden Kapiteln aufgebaut werden.

SQL Grundlagen lernen

Bisher haben wir mit Python ohne SQL-Grundlagenwissen einfach Datenbankerstellung und Datenbankzugriffe durchgeführt. Dabei kamen wir bei verschiedenen Aktionen schnell an unsere Grenzen durch die fehlenden Grundlagenkenntnisse. Grundlagen haben Vorteile und bringen Sicherheit in die Anwendung und Nutzung von Datenbanken bzw. SQL. Wenn wir von SQL sprechen, sprechen wir von einem **relationalem Datenbankmanagementsystem** (RDBMS: Relational Database Management System). Flapsig gesagt: Daten stehen in einer Beziehung – einer Relation. Diese Beziehungen können ausgewertet und ausgegeben werden.

Das Kürzel SQL stehen für „Structured Query Language“ – eine strukturierte Abfrage-Sprache.

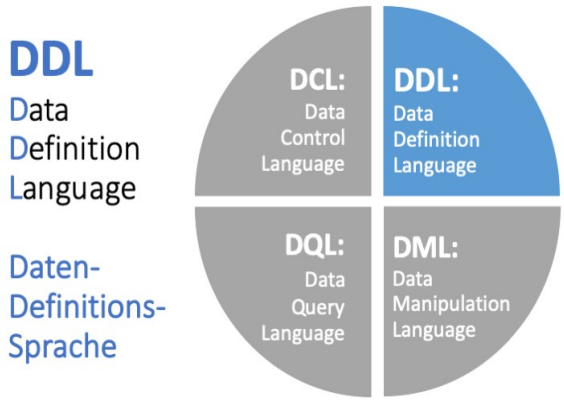


SQL selber hat 4 große Bereiche:

- DDL: Data Definition Language
- DML: Data Manipulation Language
- DQL: Data Query Language
- DCL: Data Control Language

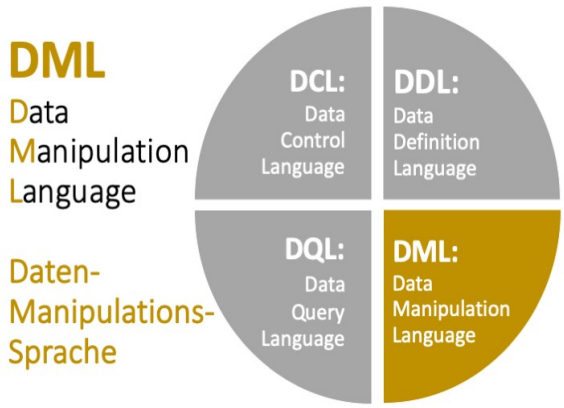
Was machen diese Bereiche für uns?

DDL: Data Definition Language – Daten-Definitions-Sprache



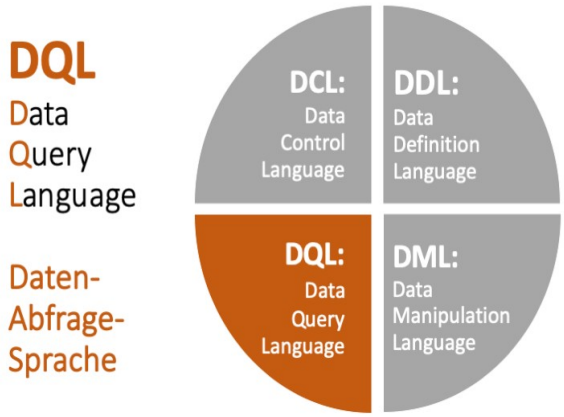
Bevor wir ein Datenbanksystem nutzen können, müssen wir erst unsere Datenbank mit deren Tabellen einrichten. Auf gut Deutsch: welche Felder gibt es und wie nennen wir diese, damit wir komfortable wieder darauf zugreifen können. Wir definieren also unsere Daten bzw. Datenfelder und können die gewünschte Struktur über eine SQL-Anweisung in der Datenbank erzeugen. Ohne eine Datenbank und Daten können wir auch nicht die folgenden Bereiche nutzen wie die Datenbank mit Daten zu füllen bzw. auszuwerten.

DML: Data Manipulation Language – Daten-Manipulations-Sprache



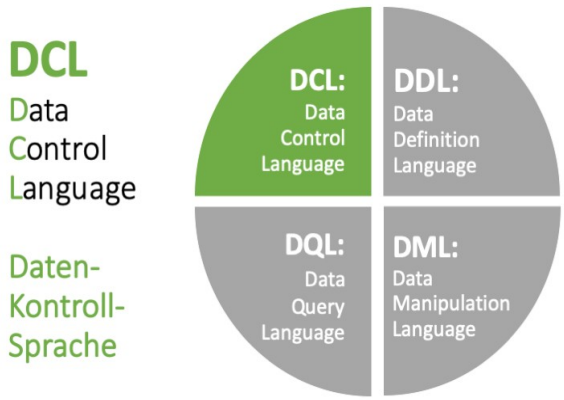
Nachdem wir unsere gewünschte Struktur über die DDL erstellt haben, wollen wir Daten in unser Datenbanksystem hineinschreiben, ändern und löschen können. Genau das macht man mit der DML.

DQL: Data Query Language – Daten Abfrage Sprache



Auslesen der Datenbank auch mit bestimmten Bedingungen (man will ja nicht immer alle Daten). Hier können auch mehrere Tabellen miteinander verknüpft werden und daraus das Ergebnis ausgegeben werden.

DCL: Data Control Language – Daten-Kontroll-Sprache



Data Control Language: Daten-Kontroll-Sprache (Besser Zugriffs-Kontroll-Sprache)

Nicht jeder darf auf alle Daten zugreifen. Daher ergibt es Sinn, hier eine Kontrolle einzubauen.

Zur DCL gehört auch die Transaktionssteuerung, damit es auch sichergestellt ist, dass alle Änderungen bzw. Aktionen in der Datenbank stattfinden. Ansonsten (je nach System) gibt es automatisch ein zurück auf den definierten Ausgangspunkt. Einfach einmal eine Banküberweisung sich vorstellen, wo beim Bezahlenden abgebucht wird, aber nicht beim Empfänger gutgeschrieben wird. Das gibt Ärger! Daher entweder beides oder gar nichts!

ANSI-SQL

Hier der Begriff ANSI-SQL, damit man es mal gelesen hat und verorten kann. Das American National Standards Institut (ANSI) hat festgelegte Standards. Diese gibt es auch für SQL.

Allerdings gibt es da über die Jahre mehrere verschiedene, da es bei SQL Verbesserungen und Erweiterungen gegeben hat. Die verschiedenen SQL-Server bieten im Kern die gleiche Funktion, die aber in der Nutzung leicht unterschiedlich sein kann, was hier am Anfang über das von uns benötigte Wissen hinausgeht. Sollte man mal schlaflose Nächte haben, kann man sich mit den verschiedenen ANSI-SQL-Varianten die Nacht vertreiben.

MySQL, MariaSQL, Microsoft SQL Server (MSSQL), Oracle – verschiedene Datenbanksysteme

Es gibt verschiedene und am Anfang nutzt man eins um in die Materie hineinzukommen.

Lustig am Rande: MySQL – die ersten 2 Buchstaben vom Namen der Datenbank kommen von dem Vornamen „My“. Diese ist die Tochter des MySQL-AB-Mitgründers Michael Widenius (Monty genannt). Seine zweite Tochter heißt Maria (wurde auch als Namen für ein Datenbanksystem verwendet: MariaSQL).

Die verschiedenen Datenbankserver orientieren sich alle am Standard der aktuellen SQL-Version.

Begriffe rund um SQL

Anweisungen, Statement: SQL ist eine Anweisungssprache, keine Programmiersprache!

Schlüsselwörter:

einzelne SQL-Befehle

Anwendung:

- nicht case-sensitiv (Groß- und Kleinschreibung macht keinen Unterschied), es gibt aber Empfehlungen in Form von Schlüsselworten immer in Großbuchstaben, Rest in Kleinschreibung bzw. gemischte Schreibweise.
- Leerzeichen, Tabulatoren und Zeilenumbrüche dürfen nach Belieben genutzt werden. Hier ist die bestmögliche Lesbarkeit das Ziel!

Speicherung der Daten mit SQL

Die Daten werden in Tabellen gespeichert. Dabei wird jeder **Spalte** (man spricht auch von **Feldern bzw. Datenfelder**) mit einem Namen benannt um später darauf zugreifen zu können. Es wird festgelegt, ob in dieses Datenfeld nur Text, ganze Zahlen, Zahlen mit Nachkommastellen oder z.B. ein Datum zukünftig gespeichert werden soll. Hier spricht man von **Datentypen**, die verfügbar sind. Grob gesagt haben wir hier Zahlen, alphanumerische Zeichen (flapsig gesagt also Texte, die zusätzlich Zahlen enthalten können, mit denen wir aber nicht rechnen) und Datumstypen. Die Festlegung der Datentypen ist sehr wichtig, da nur mit Zahlen gerechnet werden kann und auch es auf Sortierungen Auswirkungen hat. In der Anwendung werden wir dann bei der Definition der Felder noch präzise den benötigten Datentyp definieren. Das dann später genauer im passenden Kapitel.

Zusätzlich wird auch für die meisten Datentypen eine Feldlänge vergeben.

Bezeichnung Datensatz: Die Zeilen in der Tabelle werden als Datensätze bezeichnet. Hier haben wir dann alle Einzeldaten, die zu diesem Datensatz gehört. Bei einem Telefonbuch gäbe es dann z.B. Vorname, Nachname, PLZ, Telefonnummer

Tabellen beherbergen somit gleichartige Informationen.

Ein Datensatz ist eine Zusammenstellung von einzelnen Informationen, die sinnvoll zusammengehören und daher schlecht getrennt werden können. In unserem Beispiel vom Telefonbuch macht es Sinn, dass der Vorname und der Nachname in der gleichen Tabelle als Datenfelder vorhanden sind.

Primärschlüssel (Primärschlüsselspalte): Um bei der Anwendung exakt auf einen gewünschten Datensatz zugreifen zu können, benötigen wir Primärschlüssel. Anhand des Primärschlüssels können wir den benötigten Datensatz bearbeiten. Hätten wir keinen Primärschlüssel, hätte z.B. der Name Müller öfters Pech, da ab einer gewissen Anzahl von Adressen mehrere Müller vorkommen können. Man kann also schlecht nach dem Nachnamen suchen und erwarten, dass dort immer exakt nur 1 Treffer zurückkommt. Daher benötigen wir eine eindeutige Zahl. Wird in unserer Tabelle ein Primärschlüssel definiert, müssen wir uns meistens nicht mehr darum kümmern, dass dieser einmalig ist. Versucht man einen Primärschlüssel mehrfach zu vergeben, erhalten wir eine Fehlermeldung vom DBMS – somit ist sichergestellt, dass es nur einen gibt. Die Vergabe des Primärschlüssels kann automatisch laufen und somit ist sichergestellt, dass eindeutige Zugriffe möglich sind.

Beziehungen in relationaler Datenbank: Schauen wir unser Beispiel mit der Telefonliste an. Neben dem Vor- und Nachnamen speichern wir die Postleitzahl – aber keinen Ort! Warum? Hier kommt der Vorteil von relationalen Datenbanken ins Spiel. Es geht um Beziehungen: Unser Postleitzahlensystem in Deutschland hat zu jeder Postleitzahl genau einen Ort. Es macht also wenig Sinn zusätzlich neben der PLZ den Ort in der Tabelle mit den Namen zu speichern. Für die Orte können wir eine weitere Tabelle anlegen, in der nur Postleitzahl und zugehörige Orte gespeichert werden. In dem Fall der neuen Tabelle ist unsere PLZ der Primärschlüssel unserer Tabelle „ortsnamen“ (wir gehen jetzt einfach nur von Orten in Deutschland aus). Unser Feld PLZ in der Tabelle „telefonbuch“ nennt sich auch „Fremdschlüssel“. Dieser Fremdschlüssel aus der Tabelle „telefonbuch“ zeigt auf den Primärschlüssel „PLZ“ in der Tabelle „ortsnamen“.

Diese Beziehungen können in Datenbankdiagrammen dargestellt werden.

[bild[Beispiel Datenbankdiagrammen]]

Den Primärschlüssel sieht man i.d.R. anhand eines Schlüsselsymbols und die Art der Verknüpfung der Daten ist sofort sichtbar anhand einer Linie. Die Enden der Linien zeigen die Art der Relation (Verknüpfung):

- Schlüssel: Primärschlüssel
- Unendlichzeichen: es können 0, 1 oder mehrere Datensätze betroffen sein

Referentielle Integrität (RI): die Referentielle Integrität dient zur Sicherstellung der Datenintegrität zwischen den Tabellen. Durch die Verknüpfung von Fremdschlüssel und Primärschlüssel kann das Datenbanksystem kontrollieren, ob ein Datensatz vorhanden ist und der

Fremdschlüssel verwendet werden darf. Auch ist es nicht einfach möglich den Primärschlüssel zu verändern, wenn dieser als Fremdschlüssel bereits verwendet wird.

Es kann nur auf die formale Korrektheit kontrolliert werden – inhaltlich ist immer noch der Nutzer in der Pflicht mitzudenken!

Schauen wir uns diese Grundlagen in konkreten Anwendungen an. Dann wird es auch greifbar. Im folgenden Kapitel nutzen wir den Primärschlüssel für die eindeutige Auswahl eines Datensatzes.

Tools für SQLite3 – SQLite Shell

SQLite bietet eine Shell. Diese kann direkt im Terminalfenster gestartet werden über

```
sqlite3
```

Danach erscheint im Terminalfenster dann:

Axels-MBP:Python-lernen.de axel$ sqlite3
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.

Jetzt können wir mit `.help` uns alle Befehle ansehen, die uns die sqlite3-Shell bietet.

Wir können alle Funktionen, die wir über den Umweg von Python kennengelernt haben auch direkt in der SQLite3-Shell ausführen. Wollen wir unsere erstellte Datenbank „geburtstage.db“ nutzen, müssen wir diese laden. Da wir diese in einem Unterordner gespeichert haben, müssen wir beim Pfad den Unterordner mit angeben:

```
.open datenbank/geburtstage.db
```

Bitte nicht wundern, wenn wir kein Feedback von der Shell bekommen:

sqlite> .open datenbank/geburtstage.db
sqlite>

Wir erhalten automatisch die Möglichkeit, den nächsten Befehl anzugeben.

Jetzt können wir uns die Datenbanken anzeigen lassen über:

```
.databases
```

Als Rückmeldung erhalte ich auf meinem System dann:

sqlite> .databases
main: /Users/axel/Documents/Python-lernen.de/datenbank/geburtstage.db

Haben wir die erstellten Tabellen vergessen, erhalten wir über den Befehl `.tables` eine Auflistung aller in der Datenbank vorhandenen Tabellen:

sqlite> .tables

adressen personen

Und jetzt können wir bereits direkt mit weiteren SQL-Anweisungen die Tabelle auslesen. Lassen wir uns den Inhalt komplett anzeigen:

sqlite> SELECT * FROM personen;

Johann Christoph Friedrich|Schiller|10.11.1759

Wir bekommen den Datensatz von „Schiller“ angezeigt.

…> - was geht?

Bekommen wir allerdings versehentlich nur als Ausgabe „…>“ haben wir ein abschließendes Semikolon vergessen. Einfach einmal probieren:

```
sqlite> SELECT * FROM adressen
...>
...> ;
```

Um wieder die normalen Shell-Eingaben machen zu können, müssen wir das anschließende Semikolon angeben!

```
sqlite> SELECT * FROM adressen
...>
...> ;
sqlite>
```

SQLite-Shell beenden: .quit

Um die Shell zu beenden, einfach in der Eingabeaufforderung `.quit` eingeben. Somit sind wir im normalen Terminal zurück.

```
sqlite> .quit
Axels-MBP:Python-lernen.de axel$
```

Bessere Ausgabe der Daten in SQLite3-Shell aktivieren

Bei unserem letzten Test kamen die Daten sehr unübersichtlich in der Ausgabe. Wir können aber eine wesentlich bessere Übersicht bekommen, wenn wir den Spalten-Modus aktivieren über `.mode column` und uns zusätzlich die Spaltenbezeichnungen ausgeben lassen über die Anweisung `.headers on`.

Schauen wir es uns an:

```
sqlite> .open datenbank/geburtstage.db
sqlite> .headers on
sqlite> .mode column
sqlite> SELECT * FROM personen;
vorname nachname geburtstag
```

------------------------ ---------- ----------
Johann Christoph Friedrich Schiller 10.11.1759
sqlite>

Einstellungen (und Änderungsmöglichkeiten)

Über die Shell-Anweisung `.show` bekommen wir weitere Informationen über die Anzeige und über die aktuell gewählte Datenbank:

sqlite> .show
echo: off
eqp: off
explain: auto
headers: on
mode: column
nullvalue: ""
output: stdout
colseparator: "|"
rowseparator: "\n"
stats: off
width:
filename: datenbank/geburtstage.db

sqlite>

Hier können wir nach Bedarf auch Änderungen durchführen. Beispielsweise können wir den Trenner, der als Zeichen „|“ verwendet umstellen über die Anweisung `.separator ^`

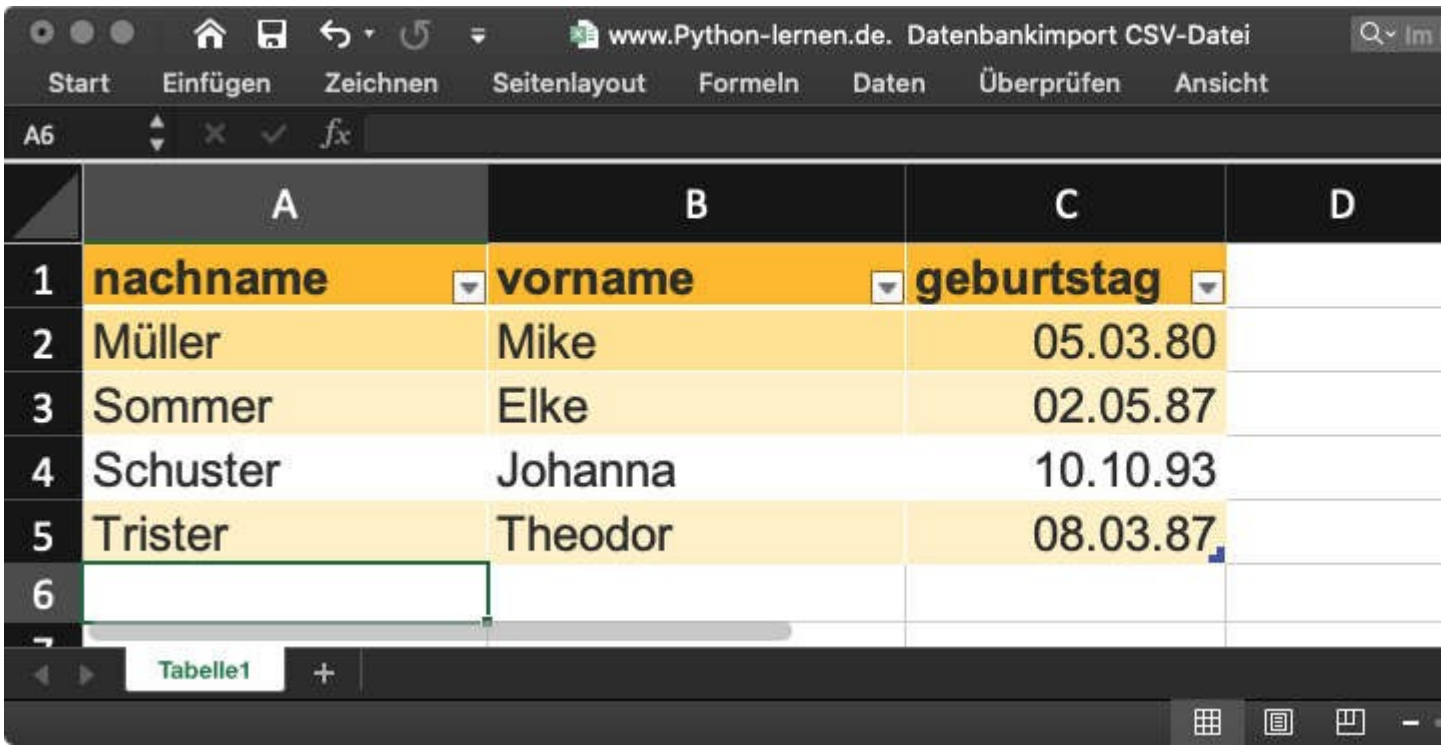
Direkt aus Shell starten

Gerade sind wir in mehreren Schritten zu unserem Ergebnis gekommen. Wir können auch direkt aus unserem Terminal (CMD-Line unter Windows) eine komplette SQL-Ausgabe erreichen. Dazu nutzen wird folgenden Aufbau:

```
sqlite3 datenbankname.db "SELECT * FROM tabellennamen"
```

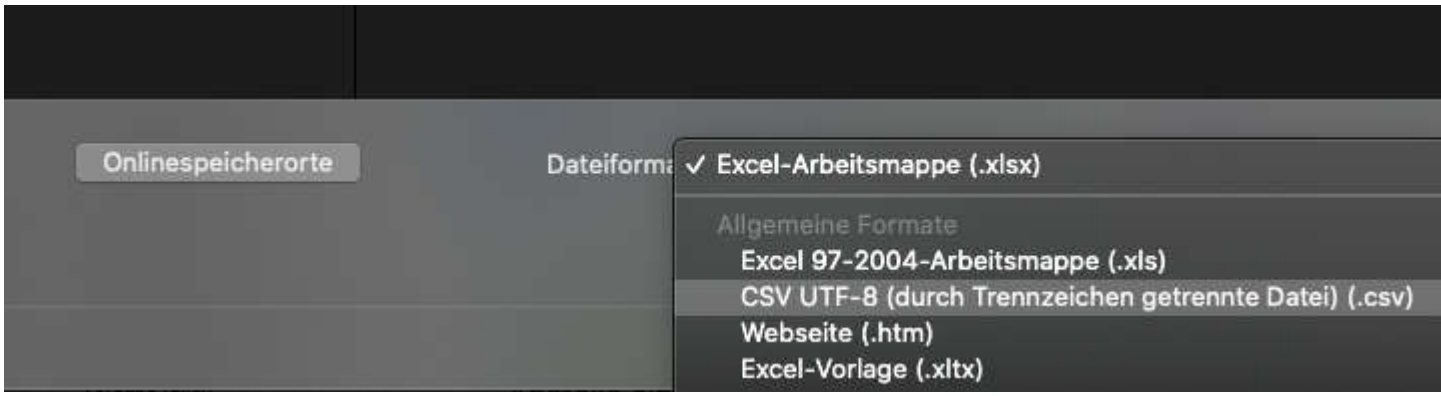
CSV-Datei in SQLite3 Datenbank importieren

Schauen wir uns den Inhalt einer CSV-Datei an. Meistens liegen uns Daten in Excel vor. In Excel kann der Export direkt in eine CSV-Datei exportiert werden:



Der Export aus Excel wird über:

Datei -> Speichern unter -> Dateiformat



Wir erhalten dann eine Textdatei, deren Datenfelder der Semikolons getrennt sind:

nachname;vorname;geburtstag
Müller;Mike;05.03.80
Sommer;Elke;02.05.87
Schuster;Johanna;10.10.93
Trister;Theodor;08.03.87

Und die CSV-Datei über Python ausgelesen und angezeigt:

```
import csv
with open("adressen.csv") as csvdatei:
    csv_reader_object = csv.reader(csvdatei, delimiter=';')
    for row in csv_reader_object:
        print(row)
```

Als Ergebnis erhalten wir die Daten als Datentyp "Liste":

['ufeffnachname', 'vorname', 'geburtstag']
['Müller', 'Mike', '05.03.80']
['Sommer', 'Elke', '02.05.87']
['Schuster', 'Johanna', '10.10.93']
['Trister', 'Theodor', '08.03.87']

Die ersten Zeichen 'ufeff' kommen daher, dass wir in der entsprechenden UTF8-Variante abgespeichert haben. Da wir die Überschriften nicht benötigt, kann man diesen kleinen Schönheitsfehler ignorieren und diese erste Zeile mit den Überschriften einfach mit einem Texteditor aus der CSV-Datei löschen. Dann passt alles beim richtigen Import.

Jetzt wollen wir eine Verbindung zur Datenbank „adressen.db“ aufbauen und diese gegebenenfalls neu anlegen, falls diese noch nicht existiert:

```
import csv
import sqlite3
verbindung = sqlite3.connect("adressen.db")
zeiger = verbindung.cursor()

sql_anweisung = """
CREATE TABLE IF NOT EXISTS adressen (
nachname VARCHAR(30),
vorname VARCHAR(20),
geburtstag DATE
);"""

zeiger.execute(sql_anweisung)

with open("adressen.csv") as csvdatei:
    csv_reader_object = csv.reader(csvdatei, delimiter=';')
    for row in csv_reader_object:
        print(row)
```

Ab jetzt können wir unsere SQLite3 Datenbank befüllen. Wir benötigen die entsprechende SQL-Anweisung:

```
import csv
import sqlite3
verbindung = sqlite3.connect("adressen.db")
zeiger = verbindung.cursor()

sql_anweisung = """
CREATE TABLE IF NOT EXISTS adressen (
nachname VARCHAR(30),
vorname VARCHAR(20),
geburtstag DATE
);"""

zeiger.execute(sql_anweisung)

sql_anweisung = """
INSERT INTO adressen (nachname, vorname, geburtstag)
VALUES (:nachname, :vorname, :geburtstag)
"""
```

```
with open("adressen.csv") as csvdatei:
    csv_reader_object = csv.reader(csvdatei, delimiter=';')

    with sqlite3.connect("adressen.db") as verbindung:
        zeiger = verbindung.cursor()
        zeiger.executemany(sql_anweisung, csv_reader_object)
```

Und zum Test können wir jetzt die Datenbank auslesen und den Inhalt anzeigen:

```
import sqlite3
verbindung = sqlite3.connect("adressen.db")
zeiger = verbindung.cursor()
zeiger.execute("SELECT * FROM adressen")
inhalt = zeiger.fetchall()
print(inhalt)
verbindung.close()
```

Wir haben alle Daten auf einen Rutsch in unsere SQLite3-Datenbank importiert und können damit weiterarbeiten.

So kann man schnell SQLite-Datenbanken mit Inhalt füllen.