Korawich Inthamusik 6410110625

Data Management With Python, SQLite, and SQLAlchemy

Using Flat Files for Data Storage

Flat files contain human-readable characters and are very useful for creating and reading data. Flat files use other structures for a program to parse text. For example, comma-separated value (CSV) files are lines of plain text in which the comma character separates the data elements. Below is an example of CSV

```
class,x_center,y_center,width,height,confidence_score
15,0.7455,0.561875,0.229,0.35375,0.518581
15,0.246,0.555625,0.342,0.42875,0.809506
```

The first line are the column names for the data in the remaining lines. The rest of the lines contain the data, with each line representing a single record.

Advantages of Flat Files

- Manageable and straightforward to implement.
- Helpful for creating the data file with a text editor.
- Helpful for examining the data and looking for any inconsistencies or problems.
- Transferable (can import or export).
- Have tools that make working with CSV files easier (built-in csv and pandas module)

Disadvantages of Flat Files

- Editing large flat files to create data or look for problems becomes a more difficult task. When application will change the data in the file, it would read the entire file into memory, make the changes, and write the data out to another file.
- Need to explicitly create and maintain any relationships between parts of your data and the application program within the file syntax by generate code in your application to use those relationships.
- People you want to share your data file with will also need to know about the structures and relationships in the data and need to understand the programming tools necessary for accessing it.

Using SQLite to Persist Data

There's redundant data file contained more related data. This data would be duplicated for each root data item. It's possible to create data this way, but it would be exceptionally unwieldy. Think about the problems keeping this data file current. What if you want to change name? You'd have to update multiple records containing that name and make sure there were no typos.

Worse than the data duplication would be the complexity of adding other relationships to the data. What if you decided to add phone numbers for the users, and they had phone numbers for home, work, mobile, and perhaps more? Every new relationship that you'd want to add for any root item would multiply the number of records by the number of items in that new relationship.

This problem is one reason that relationships exist in database systems. An important topic in database engineering is database normalization, or the process of breaking apart data to reduce redundancy and increase integrity. When a database structure is extended with new types of data, having it normalized beforehand keeps changes to the existing structure to a minimum.

The SQLite database is available in Python, and according to the SQLite home page, it's used more than all other database systems combined. It offers a full-featured relational database management system (RDBMS) that works with a single file to maintain all the database functionality.

It also has the advantage of not requiring a separate database server to function. The database file format is cross-platform and accessible to any programming language that supports SQLite.

Creating a Database Structure

The brute force approach to getting the sample_file.csv data into an SQLite database would be to create a single table matching the structure of the CSV file. Doing this would ignore a good deal of SQLite's power.

Relational databases provide a way to store structured data in tables and establish relationships between those tables. They usually use Structured Query Language (SQL) as the primary way to interact with the data.

An SQLite database provides support for interacting with the data table using SQL. Not only does an SQLite database file contain the data, but it also has a standardized way to interact with the data. This support is embedded in the file, meaning that any programming language that can use an SQLite file can also use SQL to work with it.

Interacting With a Database With SQL

SQL is a declarative language used to create, manage, and query the data contained in a database.

Structuring a Database With SQL

Conceptually, data is stored in the database in two-dimensional table structures. Each table consists of rows of records, and each record consists of columns, or fields, containing data.

The data contained in the fields is of pre-defined types, including text, integers, floats, and more. CSV files are different because all the fields are text and must be parsed by a program to have a data type assigned to them.

Each record in the table has a primary key defined to give a record a unique identifier similar to the key in a Python dictionary. The database engine itself often generates the primary key as an incrementing integer value for every record inserted into the database table.

If the data stored in a field is unique across all other data in the table in that field, then it can be the primary key. For example, a table containing data about books could use the book's ISBN (The International Standard Book Number) as the primary key.

Maintaining a Database With SQL

SQL provides ways to work with existing databases and tables by inserting new data and updating or deleting existing data.

Building Relationships

Databases that support relationships allow you to break up data into multiple tables and establish connections between them.

One-to-Many Relationships

A one-to-many relationship is like that of a customer ordering items online. One customer can have many orders, but each order belongs to one customer.

Many-to-Many Relationships

Many-to-many relationships exist in the database between books and publishers. One book can be published by many publishers, and one publisher can publish many books.

Entity Relationship Diagrams

An entity-relationship diagram (ERD) is a visual depiction of an entity-relationship model for a database or part of a database.

Working With SQLAlchemy and Python Objects

The Model

The models are Python classes defining the data mapping between the Python objects returned as a result of a database query and the underlying database tables.

Table Creates Associations

The SQLAlchemy Table class creates a unique instance of an ORM mapped table within the database. The first parameter is the table name as defined in the database, and the second is Base.metadata, which provides the connection between the SQLAlchemy functionality and the database engine. The rest of the parameters are instances of the Column class defining the table fields by name, their type, and in the example above, an instance of a ForeignKey.

ForeignKey Creates a Connection

The SQLAlchemy ForeignKey class defines a dependency between two Column fields in different tables. A ForeignKey is how you make SQLAlchemy aware of the relationships between tables.

Conclusion

At this point, the advantages of using SQLAlchemy instead of plain SQL might not be obvious, especially considering the setup required to create the models representing the database. The results returned by the query is where the magic happens. Instead of getting back a list of lists of scalar data, you'll get back a list of instances of objects with attributes matching the column names you defined.

Behind the scenes, SQLAlchemy turns the object and method calls into SQL statements to execute against the SQLite database engine. SQLAlchemy transforms the data returned by SQL queries into Python objects.

Reference

https://realpython.com/python-sqlite-sqlalchemy/

SQLAlchemy –Tutorial

Filtering data

where

- SQL:

  SELECT * FROM student

  WHERE sex = F

- SQLAlchemy :

  db.select([student]).where(student.columns.sex == 'F')

- Explanation :

  เลือกทั้งหมด จาก table student ที่ column ‘sex’ มีค่าเป็น ‘F’

in

- SQL :

  SELECT address, sex

  FROM student

  WHERE address IN (Songkhla, Bangkok)

- SQLAlchemy :

  db.select([student.columns.address, student.columns.sex]).where(student.columns.address.in_([‘Songkhla’, ‘Bangkok’]))

- Explanation :

  เลือกคอลัมน์ ที่อยู่, เพศ จาก table student ที่ column address อยู่ใน list [‘Songkhla’, ‘Bangkok’]

and, or, not

- SQL :

  SELECT * FROM student

  WHERE address = 'Pattani' AND NOT sex = 'M'

- SQLAlchemy :

  db.select([student]).where(db.and_(student.columns.address == 'Pattani', student.columns.sex != 'M'))

- Explanation :

  เลือกทั้งหมด จาก table student ที่ column address == ‘Pattani’ และ column sex != ‘M’

order by

- Explanation :

  sorted data, have 2 options -> ASC : ascending, DESC : descending order

- Example :

  db.select([census]).order_by(db.desc(census.columns.state), census.columns.pop2000)

sum

- Explanation :

  summation of data

- Example :

  db.select([db.func.sum(census.columns.pop2008)])

group by

- Explanation :

  group data by selected column

- Example :

  db.select([db.func.sum(census.columns.pop2008).label('pop2008'),census.columns.sex]).\

  group_by(census.columns.sex)

distinct

- Explanation :

  no duplicate

- Example :

  db.select([census.columns.state.distinct()])

joins

- Explanation :

  If you have two tables that already have an established relationship, you can use that relationship by just adding the columns you want from each table to the select statement.

- Example :

  select([census.columns.pop2008, state_fact.columns.abbreviation])

Inserting Data into Tables

- Inserting record one by one

  query = db.insert(emp).values(Id=1, name='naveen', salary=60000.00, active=True)

- Inserting many records at ones

  query = db.insert(emp) .values([{'Id':'2', 'name':'ram', 'salary':80000, 'active':False},\

  {'Id':'3', 'name':'ramesh', 'salary':70000, 'active':True}])

Updating data in Databases

db.update(table_name).values(attribute = new_value).where(condition)

Delete Table

db.delete(table_name).where(condition)

Dropping a Table

table_name.drop(engine) #drops a single table

metadata.drop_all(engine) #drops all the tables in the database

Reference

https://towardsdatascience.com/sqlalchemy-python-tutorial-79a577141a91