Data storage & Persistence

Prof. Dr. Axel Polleres
Dr. Stefan Sobernig

2 April 2019

Unit 5: Data storage & Persistence

Connection to and loading data into and from a database system (vs. storing/loading from a file)

- Relational Databases Systems: SQLite
  - Querying a Database
- Python and Persistence:
  - Persisting objects in files: Pickle
  - Persisting objects in a Relational Database
  - Querying data from a Relational Database

Slides: This unit is also available in a PDF format and as a single HTML Page

Readings:

- Grus, J. (2015) Data Science from Scratch, O'Reilley, **Chapter 23** (available via the WU library, EBSCO)

Storing/persisting data to disc

Storing/persisting data to disc

We will briefly cover the following methods:

- writing to CSV (text)
- writing to JSON (text)
- using Pickle (binary)

All code snippets on the next slides are also available as notebook

Do you remember?

```
cityCodeFile="./data/cities.csv"
#Building the cityCode to Label map
cityCodeMap={}
with open(cityCodeFile) as f:
    csvfile = csv.reader(f)
    for i,row in enumerate(csvfile):
        cityCodeMap[row[3]]= row[1]
```

Storing/persisting data as CSV

Let's store the dictionary to a CSV file.

```
import csv
with open('cityNames.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, delimiter=',')
    for cityCode, cityName in cityCodeMap.items():
      writer.writerow( [ cityCode, cityName] )
```

The method **writerow()** expects a list of values. Each value in the list will be convert to its string representation and written to file

Loading the data back into a dictionary requires to parse the file as CSV and build the dictionary again (see our code before).

Storing/persisting data as JSON

Another way to persist our data structure is to store it to a JSON file.

```
import json
with open('data.json', 'w') as fp:
    json.dump(cityCodeMap, fp)
```

NOTE: Storing/persisting data as JSON

Storing and loading objects to and from JSON is normally fast and the preferable way.

HOWEVER :

- the json module handles JSON (JavaScript Object Notation), specified by RFC 7159
- That means that the following Python data types are performed and supported by default:
  - dict, list, str, int, float, True, False, None
  - other data types, e.g. date, datetime, are not supported by

default, i.e., they will not be preserved when storing and loading to/from a JSON file! This requires a custom JSONEncoder and JSONDecoder

NOTE: Storing/persisting data as PICKLE

Alternative: The pickle module implements binary protocols for serializing and de-serializing a Python **object structure**. That is, **any** Python data structure can be "pickled"

```
import pickle

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(cityCodeMap, f, pickle.HIGHEST_PROTOCOL)
```

Different protocols are supported.

```
with open('data.pickle', 'rb') as f:
    data=pickle.load( f)
```

To JSON or to Pickle

There are fundamental differences between the pickle protocols and JSON (JavaScript Object Notation):

- JSON is a **text serialization** format (it outputs unicode text, although most of the time it is then encoded to utf-8), while pickle is a **binary serialization** format;
- JSON is human-readable, while pickle is not;
- JSON is interoperable and widely used outside of the Python ecosystem, while pickle is Python-specific;
- JSON, by default, can only represent a subset of the Python built-in types, and no custom classes; pickle can represent an extremely large number of Python types (many of them automatically, by clever usage of Python's introspection facilities; complex cases can be tackled by implementing specific object APIs).

Question.

When should you use JSON and when Pickle as serialisation format?

see also official documentation

(Relational) Databases Systems

(Relational) Databases Systems

Question.

- What is a database?
- Why does a data scientist need databases?

What is a (Relational) Database?

- What is a *Database*?
  - A (potentially very large), integrated collection of data.
  - Typically the data models some real-world entities and their relations
  - But data could also be text/documents (e.g. abstract of the book, …) or binary (e.g. eBook in PDF, image of the cover), or semi-structured data
- A *Database (Management) System*, short DBMS is a software package designed to store and manage databases, e.g.

- A *relational* DBMS (RDBMS) is a DBMS adhering to the relational model (cf. BIS I)
  - data (typically) stored in relations, i.e. in "tables"

Why does a data scientist need databases?

- Why does a data scientist need databases?
  - persistent (on disk) storage of data and results of analyses
  - a lot o data is already stored in (relational) databases

- But we can store and load data in/from **files** already!
  - Right, but that is indirect, if you can access the data directly via an API (often using the SQL language) from the DBMS
  - Persisting data in DBMS gets you lots of additional functionality for free...

Question.

What features does a DBMS support that you'd need to take care of in your code otherwise?

- When storing/updating data?
- When retrieving data?

(Relational) Databases Systems: main features

RDBMSs shield some functionality from the user, which you'd need to take care of yourself when storing all data in files:

- **concurrency** (several users can read/write concurrently)
- **transaction control** (in a group of operations, either all-or-nothing is performed)
- **consistency** (only data consistent with the schema can be stored in the database)
- **automatic durability** (persistence on disk, you don't have to press the "save" button

Plus they offer efficient and **declarative access** to the data via a universal, *structured query language* (SQL):

- filtering, sorting, grouping, aggregation ... can all be done directly in SQL, without additional (Python) code once the data is in an RDBMS.

- the RDBMS provides **efficient indexing** techniques, for faster access of data in the database through SQL
- a lot o data is already stored in relational databases, you can process it directly there in situ! (instead of processing a dumpfile)

(Relational) Databases Systems: SQLite

SQLite: Overview

In today's lecture we use a popular Open Source database engine: SQLite

- requires no server - Database is stored in a single file
- no set-up or installation necessary
- ACID-compliant, implements most of the SQL standard
- can be embedded directly in programms

... Due to the SQL standard, working with other RDBMS (e.g. PostgreSQL) is pretty similar!

SQLite: Resources

- Install SQLite
- Working with SQLite (Tutorial)
- DB Browser for SQLite
- SQLite & Python:
  - The sqlite3 Python library
  - other libraries for SQLite with Python

SQLite: Creating a table

```
CREATE TABLE table (
column_name1 data_type(size) constraint,
column_name2 data_type(size) constraint,
column_name3 data_type(size) constraint,
....
);
```

Example SQLite:

```
CREATE TABLE `person` (
        `personID`      INTEGER PRIMARY KEY AUTOINCREMENT,
        `name`  TEXT NOT NULL,
        `PLZ`   NUMERIC,
        `city`  TEXT,
        `country`       TEXT
);
```

SQLite: Inserting records in a table

Note: SQLite uses simplified data types. Other RDBMS provide more precise specification.

```
INSERT INTO table (column1,column2,...)
VALUES (value1,value2,...);
```

Example SQLite:

```
INSERT INTO person (personID, name, PLZ, city, country)
VALUES (1, "Peter", "1220", "Vienna", "Austria"),
       (2, "Jenny", "1220", "Vienna", "Austria");
```

SQLite: Updating records in a table

Note: SQLite uses simplified data types. Other RDBMS provide more precise specification.

```
UPDATE table
```

```
SET column_1 = new_value_1, column_2 = new_value_2 ...
WHERE search_condition;
```

Example SQLite:

```
UPDATE person
SET name="Claire", PLZ="1020"
WHERE personID=2;
```

SQLite: Deleting records from a table

```
DELETE FROM table
WHERE search_condition;
```

Example SQLite:

```
DELETE FROM person
WHERE name LIKE "C%";
```

SQLite: Querying Data

- projection: filtering columns
- selection: filtering columns
- join: merging tables

Examples SQLite:

```
SELECT column1, column4
FROM table
WHERE search_condition;
```

```
SELECT name, city
FROM person
WHERE PLZ < 1000;
```

SQLite: Querying Data - Merging Data

Connecting multiple tables using a relationship between two of their attributes, typically the primary key of one table and a foreign key of another.

Examples:

```
SELECT person.name, data.total
FROM person, data
WHERE person.personID=data.personID
         AND data.year < 2000;
```

```
SELECT person.name, data.total
FROM person JOIN data ON person.personID=data.personID
WHERE data.year > 2000;
```

SQLite: Querying Data - Sorting

Note that many things we did on Python, can be done in SQL as well:

- We saw already filtering (selection/projection) and merging (join)
- Clauses ORDER BY (DESC), LIMIT

Example:

```
SELECT person.name, data.total
FROM person, data
WHERE person.personID=data.personID
ORDER BY name DESC
LIMIT 10 OFFSER 31;
```

SQLite: Querying Data - Grouping/Aggregation

You can also do grouping (using the keyword `GROUP BY`) and aggregation, e.g. counting.

Example:

```
SELECT person.name, SUM(data.total) as TotalSum
FROM person, data
WHERE person.personID=data.personID
GROUP BY data.year;
```

- Other aggregation functions, except SUM: AVG, SUM, MIN, MAX, COUNT

SQL/RDB Disclaimer

We skipped a lot of stuff important for Relational Databases & SQL:

- normal forms
- how to define keys and integrity constraints in tables
- how to define indexes to make SQL queries more efficient!
- How to write more complex queries including computations, etc.

⇒ Recommended courses: Database Systems (BSc) or Database Systems (IS Master)

Python and SQLite

Example: let's work it through in an example in our Notebook! `notebooks/SQLite+Python.ipynb`

Summary: Python and SQLite (or another DBMS)

- 2 main reasons why you want to integrate SQLite into your (Python) data workflows:
    - **Load** data into Python for further processing
        - a whole database table, or
        - results of a complex SQL query
    - **Store** data into a database table, e.g.
        - persist data in a table
        - persisting complex data structures **(Note that many databases also support persisting JSON, e.g. PostgreSQL)**
        - enjoy advanced features: transactions, concurrency,...