PyLadies

Vienna 23.8.2020

Who?

International mentorship group with a focus on helping more women become active participants and leaders in the Python open-source community.

Our mission is to promote, educate and advance a diverse Python community through outreach, education, conferences, events and social gatherings.

Agenda for today

---

1. Intro into databases
2. Intro into SQL
3. Python and sqlite3 database
4. SQLAlchemy - object relational mapping ORM
5. Intro into NoSQL databases
6. Small own database project

Useful tutorial

- https://pynative.com/python-sqlite/
- https://www.w3schools.com/sql/sql_intro.asp
- https://www.omnisci.com/technical-glossary/relational-database

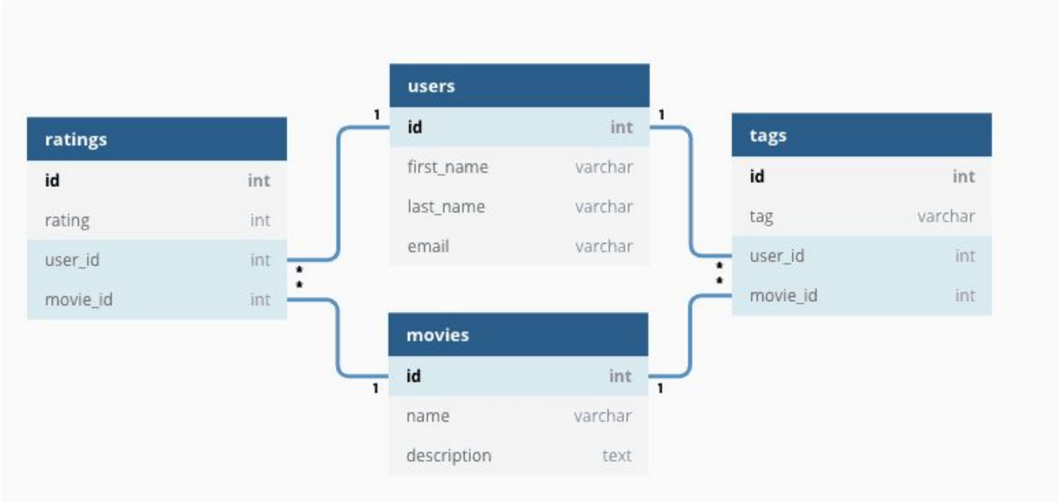
Databases introduction

- Collection of data organized in a certain way
- Classic example - tables with rows containing information and columns maintaining information about the field
- Standard way how to keep information organized

Examples of databases

- sqlite3
- PostgreSQL
- Oracle
- MySQL
- MS Access
- MongoDB
- Apache Ignite
- Neo4J

Relational Database



ratings	
id	**int**
rating	int
user_id	int
movie_id	int

users	
id	**int**
first_name	varchar
last_name	varchar
email	varchar

movies	
id	**int**
name	varchar
description	text

tags	
id	**int**
tag	varchar
user_id	int
movie_id	int

- (usually) deduplicated
- (usually) normalized data
- rows - actual info
- columns - what and data type
- primary key - unique identifier
- foreign key - for match to the other tables

NoSQL

- Key value: data is stored as attribute names or keys with values
- Document: contains many different key value pairs
- Graph: used to store data related to connections or networks
- Column: data is stored as columns instead of rows

SQL vs NoSQL

SQL	NoSQL
Schema designed at beginning	Flexible schema design
SQL	Many different languages
Vertically Scalable	Horizontally scalable
Best for complex queries	Best for complex, unstructured data

Install sqlite3 for this course

- In-process library that implements a serverless, zero configuration, self contained SQL database engine
- https://www.sqlite.org/download.html
- download DLL + CLI tools, extract to new folder and add the folder to PATH
- - test by typing **sqlite3** to cmd line
- Windows - needs download
- Mac + Ubuntu usually preinstalled

Import database

- Download database from https://www.sqlitetutorial.net/sqlite-sample-database/
- To import database into sqlite3:

```
sqlite3 C:\Users\tyna\Desktop\PyLadies\chinook\chinook.db
.tables
.schema table_name
```

When using SQL commands, end them with ; - SELECT title FROM albums;

Intro into SQL (Structured Query Language)

- SQL is syntax language, used by SQL databases
- CREATE TABLE
- INSERT
- SELECT
- UPDATE
- DELETE
- ALTER TABLE
- SQL has dialects, not every SQL version (PostgreSQL, MySQL) has same functions

CREATE TABLE

- `CREATE TABLE students (id INTEGER PRIMARY KEY, name TEXT, surname TEXT, age INTEGER);`

INSERT

Two options:

1. selected columns - INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);
2. all columns - INSERT INTO table_name VALUES (value1, value2, value3, ...);

- Order is important!
- ID if not given is set automatically

SELECT

- SELECT column1, column2, … FROM table_name;
- Selects only some columns you want is specified between select and from statement

- SELECT * FROM table_name;
- select all = *

WHERE, AND, OR, ORDER BY, LIMIT

- SELECT column1, column2, ...

  FROM table_name

  WHERE condition1 AND condition2 AND condition3 ORDER BY column1;
- LIMIT - limit number of results

AGGREGATE, SUM, MAX, AVG, COUNT

```
SELECT SUM(column_name)

FROM table_name

WHERE condition;
```

- Applied directly on desired column

GROUP BY

```
SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country;
```

- Grouped information directly based on column, usually with combination with aggregation functions
- DATA -> WHERE -> GROUP BY -> HAVING -> ORDER BY -> LIMIT.

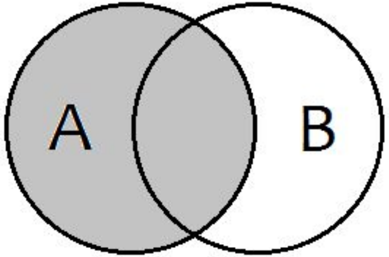
JOIN

- Two tables with different information

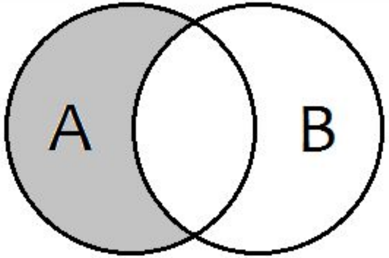
  ```
  SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate

  FROM Orders

  INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
  ```

- Both tables have to share one column (not necessary same name) - links them

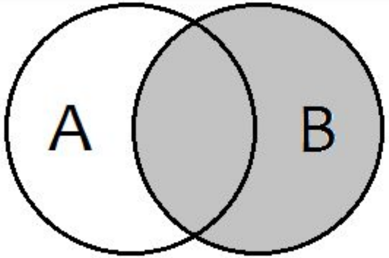
SQL JOINS



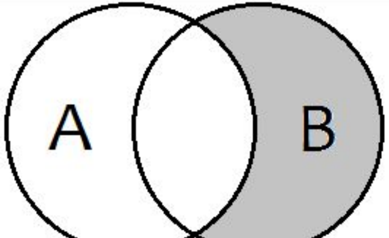
```
SELECT *
FROM TableA a
LEFT JOIN TableB b
ON a.Key = b.Key
```



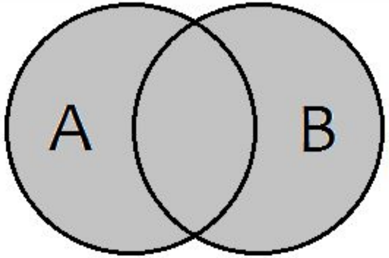
```
SELECT *
FROM TableA a
LEFT JOIN TableB b
ON a.Key = b.Key
WHERE b.Key IS NULL
```



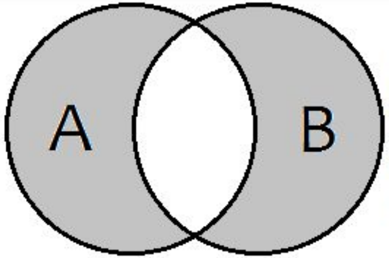
```
SELECT *
FROM TableA a
RIGHT JOIN TableB b
ON a.Key = b.Key
```



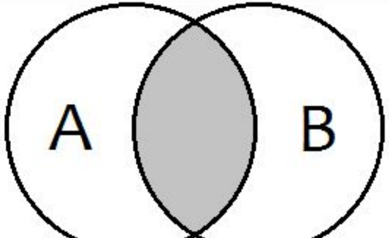
```
SELECT *
FROM TableA a
RIGHT JOIN TableB b
ON a.Key = b.Key
WHERE a.Key IS NULL
```



```
SELECT *
FROM TableA a
FULL OUTER JOIN TableB b
ON a.Key = b.Key
```



```
SELECT *
FROM TableA a
FULL OUTER JOIN TableB b
ON a.Key = b.Key
WHERE a.Key IS NULL
OR b.Key IS NULL
```



```
SELECT *
FROM TableA a
INNER JOIN TableB b
ON a.Key = b.Key
```

UPDATE, DELETE

- `UPDATE students SET age = 35 WHERE id = 1 LIMIT 1;`

- `DELETE FROM students WHERE id = 1 LIMIT 1;`

ALTER TABLE

- `ALTER TABLE students ADD COLUMN grade INTEGER;`
- `ALTER TABLE students DROP COLUMN grade;`

- `ALTER TABLE distributors ADD CONSTRAINT distaddr FOREIGN KEY (address_id) REFERENCES addresses (id);`

Best practices

- SELECT * FROM TABLE can take very long
- SQL commands written in CAPITAL LETTERS
- comments with **--comment line** or ***/ comment block */**
- UPDATE and DELETE LIMIT 1 for safety

DATA TYPES IN SQLITE3

- **NULL.** The value is a NULL value.
- **INTEGER.** The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
- **REAL.** The value is a floating point value, stored as an 8-byte IEEE floating point number.
- **TEXT.** The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).
- **BLOB.** The value is a blob of data, stored exactly as it was input.

SQL Tasks

1. How much is maximal invoice?
2. From which genre there are most tracks?
3. Which artist has most tracks?
4. What is the name of the artist who produce tranks on Purchased AAC audio file media_type and genre is jazz?
5. Find top 5 albums with most tracks
6. How many rock tracks are in database?

https://www.sqlitetutorial.net/sqlite-sample-database/

Testing database schema

Connecting to SQLite via Python

- **sqlite3** module included in standard library
- standardized DB-API (PEP 249), all other major database clients in Python use it (PostgreSQL, MySQL)

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
c.execute('SELECT name FROM books WHERE pages>400')
print(c.fetchall())
conn.commit()
```

Connecting to SQLite via Python DB-API

- Connection - should have **commit()** and **rollback()** for transactions - (single unit of work, can consist of many operations)
- Cursor - does actual traversing over DB content - should have **execute(operation, params)**, **fetchone()**, **fetchmany(rowcount)**, **fetchall()**, **rowcount**
- set of always present Exceptions (for example **DatabaseError**, **OperationalError**, **ProgrammingError**)
- you should do cursor.close(), connection.close()
- good idea to use **try/except** block for DB operations and closing connection in **finally** block

Connecting to PostgreSQL

- **psycopg2** package using same syntax as **sqlite3** (DB-API)
- not part of standard library - pip install psycopg2
- extra connection parameters need to be set up
- host (url), port (5432), database name, user, password

PostgreSQL extensions

- PostgreSQL is supporting usage of external extensions which helps with specific tasks
- PostGis - enables spatial data usage and fast operations, usage of geometry type
- PostPic - new Image type and image usage directly in database

SQL Alchemy and ORM - object relational mapping

- SQL Alchemy - standard interface over different database engines, to focus on implementing actual program logic instead of handling connections, syntax etc.
- most of times used for its optional ORM part
- ORM translated Python classes to tables and converts function calls to SQL statements, handles data types

```python
class Book(Base):
    __tablename__ = 'books'
    id=Column(Integer, primary_key=True)
    title=Column('title', String(500))
    author=Column('author', String(500))
    in_stock=Column('in_stock', Boolean)
    quantity=Column('quantity', Integer)
    price=Column('price', Numeric)
```

SQL Alchemy ORM

- Declarative base Class - you describe actual database tables, that the classes will be mapped to
- After creating a session, you can add/query/delete instances of your Class from real database

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, Sequence('user_id_seq'), primary_key=True)
    name = Column(String(50))
ed_user = User(name='ed')
session = Session()
session.add(ed_user) # not created yet
our_user = session.query(User).filter_by(name='ed') # he was created, when we queried him
ed_user is our_user # returns True
session.commit() # to save changes manually
```

https://docs.sqlalchemy.org/en/13/orm/tutorial.html

SQL Alchemy ORM

- Adding more User objects via session.add_all([])

```
session.add_all([User(name='wendy'), User(name='fred')])
```

- Updating data ed_user.name='' # still not updated to DB
- List out non-commited updates via session.dirty or or non-commited added ones via session.new

SQL Alchemy ORM - rolling back

- Possible to revert wrong commits via **.rollback()**

```
ed_user.name = 'Edwardo'
fake_user = User(name='fakeuser')
session.add(fake_user)
session.query(User).filter(User.name.in_(['Edwardo', 'fakeuser'])).all() # gets flushed when queried
session.rollback()
ed_user.name # 'ed'
fake_user in session # False
```

SQL Alchemy ORM

```
# iterating over query results
for name in session.query(User.name):
    print(name)
```

Filtering - LIKE vs ILIKE (case insensitive), to be sure, as different DBs have different implementations of LIKE

```
query.filter(User.name.ilike('%ed%')) # get me all users containing ed in name
query.filter(~User.name.in_(['jane', 'tom'])) # ~ is negation, so it gets me users, which dont have name jane or tom
```

counting - with .count() in the end

Mongodb, example of NoSQL DB

- data are stored in documents (JSON-like), ids created by mongodb itself
- PyMongo - pip3 install PyMongo
- instead of tables, collections

```
import pymongo
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]


myquery = { "address": "Park Lane 38" }
mydoc = mycol.find(myquery)
for x in mydoc:
     print(x)
```

Project to practice **Python** & databases

- 3 .csv files to import into database provided in the github repository in /databases
- Read CSV
- Cleanup data (revenue $, rename columns, correct No Data values)
- Connect to sqlite3 database
- Create tables
- Import data
- Answer some questions about data

Project to practice **Python** & databases QUESTIONS

- Which series has got the most episodes (return name and number)
- Return average rating per series
- Which viewer watched for longest time on each day (10, 11, 12) (return name and how many minutes)
- Return the users which are registered on the platform the longest and shortest based on logins
- Order series by total minutes streamed descending
- Which series are most liked by men and women respectively based on minutes streamed
- ...

Sum it up

- Databases are useful for storing data in organized way
- Many many many types of them with different strengths
- For incrementally growing data in terms of structure, use NoSQL
- Python has great interface for communication with your data

Resources and materials general

- advent of code - adventofcode.com
- hackerrank - hackerrank.com
- https://www.practicepython.org
- Nice Python exercices at one place https://github.com/tystar86/python_exercises
- https://automatetheboringstuff.com
- https://diveintopython3.problemsolving.io

Next topics

Django

Graphics

GUI

fill the form regarding your interests please :) → https://forms.gle/UtfgVGe6AhhRwx539

Thank you and see you next time

Coding session - **9.9.2020,** 6PM-8PM

Next workshop - **19.9.2020,** Django