Working with databases in Python

Programmazione Avanzata 2025-26

Outline

- Tools
- Database access in Python
  - mysql-connector, connection, cursor, statements
- DAO pattern
- Object-Relational Mapping (ORM)
- Connection pooling

Goal

- Enable Python applications to access data stored in relational databases, performing CRUD operations
  - Create (insert new data)
  - Read (query existing data)
  - Update (modify existing data)
  - Delete (remove existing data)
- Data can be used by
  - The algorithms running in the application
  - The user, through the graphics user interface

Working with databases in Python

- A Database Management System (DBMS) needs to be installed (locally, on a remotely accessible server)
  - E.g., MariaDB, MySql, etc.
- A tool with a graphics frontend for testing queries on the DBMS before actually moving to Python is recommended
  - E.g., phpMyAdmin, Dbeaver, HeidiSql, TablePlus, etc.
- They can be found (with other components) integrated in “all-in-one” tools like XAMPP

MariaDB / MySQL

- Database Management Systems (DBMS) are software systems used to store, retrieve, and run queries on data, as well as administer the data

https://mariadb.org/

https://www.mysql.com/

phpMyAdmin / DBeaver

- Graphics frontends to work with a DBMS:
  - Data editor
  - SQL editor
  - Task management
  - Database maintenance tools

phpMyAdmin

DBeaver

https://www.phpmyadmin.net/

https://dbeaver.io/

XAMPP

XAMPP Apache + MariaDB + PHP + Perl

What is XAMPP?

XAMPP is the most popular PHP development environment

XAMPP is a completely free, easy to install Apache distribution containing MariaDB, PHP, and Perl. The XAMPP open source package has been set up to be incredibly easy to install and to use.

Download
Click here for other versions

XAMPP for **Windows**
8.2.12 (PHP 8.2.12)

XAMPP for **Linux**
8.2.12 (PHP 8.2.12)

XAMPP for **OS X**
8.2.4 (PHP 8.2.4)

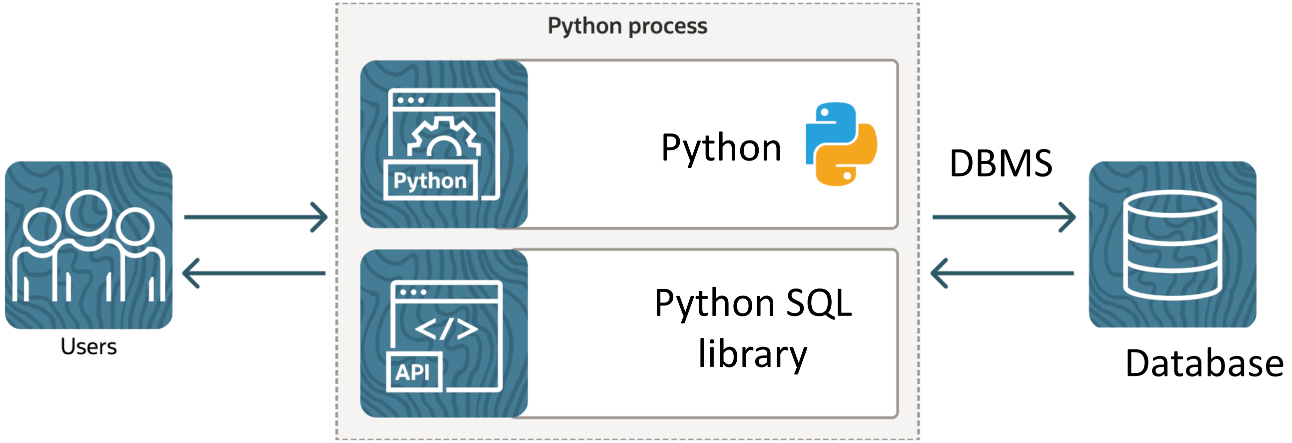
https://www.apachefriends.org/index.html

Connecting and interacting with DBMSs

- Through a DBMS, end-users/applications can interact with a database
- Different flavours of SQL-based DBMSs: MariaDB, MySQL, PostgreSQL, SQLite, SQL Server, etc.
- All of these databases
  - Are compliant with the SQL standards
  - But with varying degrees (https://troels.arvin.dk/db/rdbms/)
- Mechanism needed to let developers create programs ignoring these differences

Interacting with DBMSs in Python

- Many Python libraries (connectors) available that implement modules for interacting with different DBMSs (https://realpython.com/python-sql-libraries)
  - `mysql-connector-python`
  - `mariadb-connector-python`
  - `SQLite`
  - `Psycopg2`
  - `...`



Python Database API Specification

- The Python Database API (DB-API) defines a standard interface for Python database access modules (https://peps.python.org/pep-0249/)
- Based on `Connection` and `Cursor` objects
- Goals:
  - Encourage similarity between the Python modules that are used to access databases
  - Achieve a consistency leading to more easily understood modules
  - Foster the creation of code that is generally more portable across databases

mysql-connector-python

- The **`mysql-connector-python`** package is a self-contained Python driver for communicating with MySQL servers, using an API that is compliant with the Python Database API Specification v2.0 (PEP 249)
- Documentation:
  - https://dev.mysql.com/doc/connector-python/en/
- Can be installed using the **`pip`** command or via PyCharm

Connection

- The first step in interfacing a Python application with a MySQL/MariaDB server is to establish a connection
- The `mysql-connector` provides a `connect()` function that is used to establish connections to the server
  - The function receives parameters that indicate which database to connect and the credentials to use
    - https://dev.mysql.com/doc/connector-python/en/connector-python-connectargs.html
  - The function returns a `MySQLConnection` object, which is used to send commands/SQL statements and read the results
    - https://dev.mysql.com/doc/connector-python/en/connector-python-example-connecting.html

Connection

```
import mysql.connector
cnx = mysql.connector.connect(user='admin',
                              password='',
                              host='localhost',
                              database='test2')
cnx.close()
```

Connection

- The `connect()` function may raise exceptions (for example if the connection fails due to wrong authentication)
- It is possible/recommended to handle these exceptions with a **`try`** – **`except`** – **`else`** – **`finally`** clause:
  1. Try to connect
  2. Handle exceptions
  3. If there was no exception, close the connection with `close()`
- This may also be rewritten using a **`with`** statement
  - https://docs.python.org/3/reference/compound_stmts.html#with

Connection

```
try:
    cnx = mysql.connector.connect(user='admin',
                                  password='',
                                  host='localhost',
                                  database='test2')
except mysql.connector.Error as err:
    if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
        print("Something is wrong w/ uname or password")
    elif err.errno == errorcode.ER_BAD_DB_ERROR:
        print("Database does not exist")
    else:
        print(err)
else:
    cnx.close()
```

Connection

- Writing the configuration of the database and authentication information in the code is not ideal, especially if the file is worked collaboratively (git)
- It is possible/advisable to use a separate configuration file and load it in the `connect()` function

```
cnx = mysql.connector.connect(option_files=
                              "/etc/mysql/connector.cnf")
```

Connection

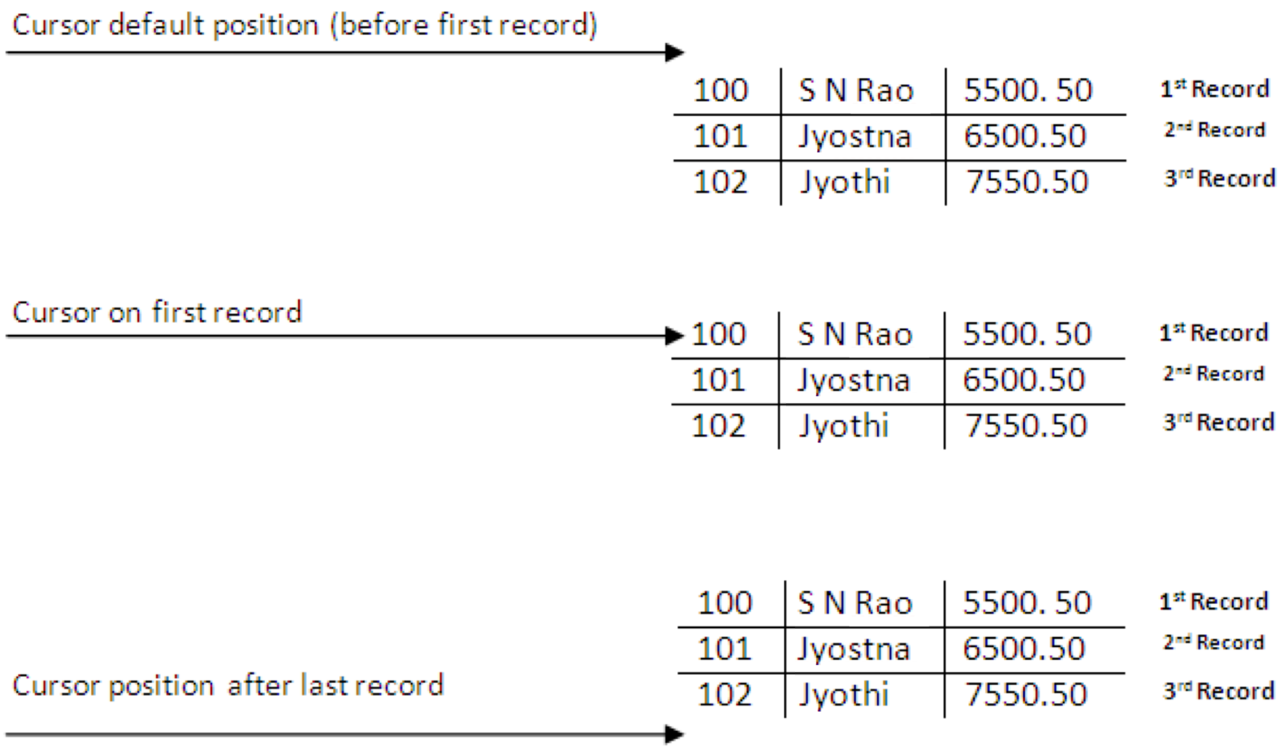
- Content of a possible configuration file

```
[client]
user='admin'
password=''
host='localhost'
database='test'
```

Using the connection

- When connected to the DBMS and a connection has been obtained, it is possible to use it to interact in different ways
  - Create tables
  - Create/Update/Delete data
  - Read data
- This is achieved through the execution of SQL statements (either DDL or DML), using a handle structure known as cursor
  - https://dev.mysql.com/doc/connector-python/en/connector-python-example-ddl.html

Cursor



Cursor default position (before first record)
100 | S N Rao | 5500. 50
101 | Jyostna | 6500.50
102 | Jyothi | 7550.50
1st Record
2nd Record
3rd Record
Cursor on first record
100 | S N Rao | 5500. 50
101 | Jyostna | 6500.50
102 | Jyothi | 7550.50
1st Record
2nd Record
3rd Record
Cursor position after last record
100 | S N Rao | 5500. 50
101 | Jyostna | 6500.50
102 | Jyothi | 7550.50
1st Record
2nd Record
3rd Record

Cursor

- The **`MySQLCursor`** class instantiates objects that can execute operations such as SQL statements
- A cursor is created from a **`MySQLConnection`** using the **`cursor()`** function
- Documentation:
  - https://dev.mysql.com/doc/connector-python/en/connector-python-api-mysqlcursor.html

Cursor

- There are several cursor classes that inherit from the **`MySQLCursor`**, and can be created by passing an appropriate argument to the **`cursor()`** function
  - **`MySQLCursorDict`** returns rows as dictionaries
  - **`MySQLCursorNamedTuple`** returns rows as named tuples
  - **`MySQLCursorPrepared`** is used for executing prepared statements

Cursor

```
import mysql.connector
import mysql.connector

cnx = mysql.connector.connect(...)
cursor = cnx.cursor()
cursor_dict = cnx.cursor(dictionary=True)
cursor_tuple = cnx.cursor(named_tuple=True)
cursor_prepared = cnx.cursor(prepared=True)

cnx.close()
```

Executing SQL statements

- A cursor object has a method **`execute()`** that allows to execute a SQL statement, expressed as a string

```
query = """SELECT id, name FROM user"""
cursor.execute(query)
```

Parametric queries

- SQL queries may depend on user input data
- Example: find item whose code is specified by the user
  - Method 1: string interpolation (with concatenation or as an f-string)

```
query = SELECT * FROM items
        WHERE code='"+user_code+"'" ;
```

  - May cause security issues, e.g., with strings typed by the user in the textbox of a GUI

SQL injection

SQL injection

- SQL injection, due to syntax errors or privilege escalation
- Example, with user typing

```
username : '; delete * from users ; --
```

- The query becomes

```
SELECT * FROM users
WHERE username=''; delete * from users ; --'
```

- One shall detect or escape all dangerous characters, but still might never be perfect
  - Never trust user-entered data, never ever

Parametric queries

- SQL queries may depend on user input data
- Example: find item whose code is specified by the user
  - Method 2: parametric statements, always preferable

Parametric statements

- Idea: separate statement creation from statement execution
  - At creation time: define SQL syntax (template), with placeholders for variable quantities (parameters)
  - At execution time: define actual quantities for placeholders (parameter values), and run the statement
- Parametric statements can be re-run many times
- Parameter values are automatically
  - Converted according to their primitive type
  - Escaped, if they contain dangerous characters
  - Handle non-character data (serialization)

Insert/Update/Delete data

- Using the cursor, it is possible to execute SQL INSERT, UPDATE and DELETE operations as parametric statements
  1. Define the statement (possibly using the Python multi-line block `""" """`); values may be written in the statement, or left unspecified as **`%s`** (because they may depend on user data)
  2. Execute the statement (setting all the unspecified values)
  3. Commit the changes to the database with **`commit()`**
  4. Close the cursor

Insert data

```
add_user = """ INSERT INTO user
               (id, name)
               VALUES (%s, %s) """
cursor.execute(add_user, (3, "John Doe"))
cnx.commit()
cursor.close()
```

Update data

```
update_user = """ UPDATE user
                  SET name = %s
                  WHERE id = %s """
cursor.execute(update_user, ("John Doe Jr.", 3))
cnx.commit()
cursor.close()
```

Delete data

```
delete_user = """ DELETE FROM user
                  WHERE id = %s """
cursor.execute(delete_user, (3, ))
cnx.commit()
cursor.close()
```

Read data

- It is also possible to use the cursor to execute an SQL SELECT statement that queries data from the database
  - Executing the query fetches results from the database
  - It is then possible to use the cursor as an iterator over the result set
  - There is no need to commit in this case, as the statement is not modifying the content of the database

Processing the result set

- Through the cursor, data is available one row at a time

```
query = """ SELECT * FROM user """
cursor.execute(query)
for (id, name) in cursor:
    print(id, name)
>>>
1 John Doe
2 Jane Smith
```

Processing the result set

- If, e.g., a **`MySQLCursorDict`** cursor is used, the result set is read as a dictionary, hence it is possible to access its field while iterating over its rows

```
cursor_dict = cnx.cursor(dictionary=True)
query = """ SELECT * FROM user """
cursor_dict.execute(query)
for row in cursor_dict:
    print(row["id"], row["name"])
>>>
1 John Doe
2 Jane Smith
```

fetchone, fetchmany, fetchall

- The cursor object also has other methods to fetch the results retrived by exceuing a query statement
  - **`fetchone()`** retrieves the next row of a query result set and returns a single sequence, or **`None`** if no more rows are available
  - **`fetchmany(N)`** fetches the next set of **`N`** rows of a query result and returns a list of tuples (or dictionaries or named tuples, if using other specialized cursors)
  - **`fetchall()`** fetches all (or all remaining) rows of a query result set and returns a list of tuples (or dictionaries or named tuples, if using other specialized cursors); if no more rows are available, it returns an empty list

fetchone, fetchmany, fetchall

```
cursor = cnx.cursor()
query = """ SELECT * FROM user """
cursor.execute(query)
rows = cursor.fetchall()
print(rows)
```

```
>>>
[(1, 'John Doe'), (2, 'Jane Smith')]
```

fetchone, fetchmany, fetchall

- When executing a query to read the data, the cursor expects the program to handle all the results
  - If not done, a **`mysql.connector.error.InternalError`** exception is raised

```
cursor = cnx.cursor()
query = """ SELECT * FROM user """
cursor.execute(query)
row = cursor.fetchone()
while row is not None:
    print(row)
    row = cursor.fetchone()
```

MySql to Python type conversion

- By default, MySQL types in result sets are converted automatically to Python types
  - For instance, a DATETIME column value becomes a **`datetime.datetime`** object
- To disable conversion, one can use a cursor with the option **`cursor(raw=True)`**
- It is possible to check the read type using the Python **`type()`** function

Problems when working with database

- Database code involves a lot of “specific” knowledge
  - Connection parameters
  - SQL commands
  - The structure of the database
- “Mixing” this low-level information with main application code is a bad practice
  - Reduces portability and maintainability
  - Creates more complex code
  - Breaks “one-class one-task” assumption
- What is a better code organization?

Goals

- Encapsulate database access into separate classes and modules, distinct from application ones
  - All other classes should be shielded from database details
- Database access should be independent from application needs
  - Potentially reusable in different parts of the application
- Develop a reusable development pattern that can be easily applied to different situations

DAO pattern

- DAO (Data Access Object) is a pattern that acts as an abstraction between the database and the main application
- It takes care of adding, modifying, retrieving, and deleting the data
  - One does not need to know how it does this, that is what an abstraction is
- DAO is implemented in a separate file, e.g., in a class with appropriate methods; then, these methods are called in the main application

DAO pattern

- Client class(es)
  - Application code that needs to access the database
  - Ignorant of database details (connection, queries, schema, ...)
- DAO class(es)
  - Encapsulate all database access code (**`mysql-connector`**)
  - The only ones that will ever contact the database
  - Ignorant of the goal of the client
- Low-level database classes
  - Handle the connection (**`MySQLConnection`**, etc.)
  - Used by DAO only, invisible to the client

DAO pattern

- Transfer Object (TO) or Data Transfer Object (DTO) class(es)
  - Contain data sent from client to DAO and/or returned by DAO to client
  - Represent the data model, as seen by the application
  - May use **`@dataclass`**
  - Ignorant of DAO, ignorant of database, ignorant of client
  - The DTO acts as a data store that moves the data from one layer to another
  - Should implement the **`__eq__()`** and **`__hash__()`** functions using the primary key
  - May implement **`__str__()`** and other dunder methods as needed

DAO design criteria

- DAO has no state
  - No instance variables (except `Connection`, maybe)
- DAO manages one “kind” of data
  - Uses a small number of DTO classes and interacts with a small number of database tables
  - If more are needed, other DAO classes can be created
- DAO offers CRUD methods
- DAO may offer search methods
  - Returning collections of DTO

Example: database

```
USER
----
id
name
```

Example: database_connect.py

```
import mysql.connector

class DatabaseConnect:
    def __init__(self):
        pass

    def get_connection(self):
        try:
            cnx = mysql.connector.
                connect(option_files="connector.cnf")
            return cnx
        except mysql.connector.Error as err:
            print(err)
            return None
```

Example: user_dto.py

```
class UserDTO:
    def __init__(self, id, name):
        self.id = id
        self.name = name

    def __str__(self):
        return f'{self.id} {self.name}'

    def __eq__(self, other):
        return self.id == other.id and
               self.name == other.name

    def __hash__(self):
        return hash(self.id)
```

Example: user_dao.py

```
from user_dto import UserDTO
from database_connect import DatabaseConnect

class UserDAO:
    def __init__(self):
        self.database_connect = DatabaseConnect()

    def get_users(self):
        cnx = self.database_connect.get_connection()
        cursor = cnx.cursor(dictionary=True)
        query = """ SELECT *
                    FROM user"""
        cursor.execute(query)
        result = []
        for row in cursor:
            result.append(UserDTO(row["id"],
                                  row["name"]))
        cursor.close()
        cnx.close()
        return result
```

Example: user_dao.py

```
def add_user(self, user : UserDTO):
    cnx = self.database_connect.get_connection()
    if cnx is None:
        print("Database connection failed")
        return

    cursor = cnx.cursor(dictionary=True)
    query = """ INSERT INTO user (id, name)
                VALUES (%s, %s)"""
    cursor.execute(query, (user.id,
                           user.name))
    cnx.commit()
    cursor.close()
    cnx.close()
```

Example: user_dao.py

```
if __name__ == "__main__":
    user_dao = UserDAO()
    users = user_dao.get_users()
    for user in users:
        print(user)
```

Example: main.py

```
from user_dao import UserDAO
from user_dto import UserDTO

user_dao = UserDAO()

user1 = UserDTO(1, "John Doe")
user2 = UserDTO(2, "Jane Smith")

user_dao.add_user(user1)
user_dao.add_user(user2)

students = user_dao.get_users()
for student in students:
    print(student)
```

Object Relational Mapping (ORM)

- Object Relational Mapping is programming pattern that enables for moving data between objects and a database while keeping them independent of each other
  - In the database, entities are represented as rows of a table, and they can be related to entries of other tables
  - In the Python application, entities are represented as objects, together with their relationships

Object Relational Mapping (ORM)

Database table
`Pet`

name	species	age	weight_in_kg	favorite_food
Pocket	dog	3	22.5	Kibbles & Bits
Mittens	cat	7	8.0	Fancy Feast: Salmon Edition
Mrs. Birdy III	bird	22	0.5	Froot Loops

Class
`Pet`

```
Pet
--------------
name
species
age
weight_in_kg
favorite_food
--------------
…
```

Object Relational Mapping (ORM)

- Needs
  - Guidelines to create a set of (data)classes for representing information stored in a relational database (will be used as DTO)
  - Guidelines to design the set of methods for DAO objects

Mapping tables

- Create one dataclass per each main database entity
  - Except tables used to store N:M relationships
- Class names should match table names
  - In the singular form
- The class
  - Should have one attribute for each column in the table
    - With matching names, according to Python naming conventions
    - And matching data types
- Except columns used as foreign keys

Mapping tables

- Add the getter (`@property`) and setter (`@attr.setter`) methods for the attributes, if needed
  - The setter method canno be specified if the dataclass uses the `frozen=True` parameter
- Define `__eq__()` and `__hash__()` using the exact set of fields that compose the primary key of the table

Mapping relationships

- Define additional attributes in the DTO classes, for every relationship to be navigated in the application
  - Not necessarily all relationships

Cardinality-1 relationships

- A relationship with cardinality 1 maps to an attribute referring to the corresponding Python object
  - Not the primary key value
- Use singular nouns

1:1 relationships

```
STUDENT
-------
student_id (PK)
fk_person

@dataclass
class Student:
  person: Person
  security_number : str
```

```
PERSON
------
security_number (PK)
fk_student

@dataclass
class Person:
  student: Student
  student_id : int
```

Cardinality-N relationships

- A relationship with cardinality N maps to an attribute containing a collection
  - The elements of the collection (for example list or set) are corresponding Python objects (not primary key values)
  - Use plural nouns.
- The class should have methods for reading (get, …) and modifying (add, …) the collection

1:N relationships

```
STUDENT
-------
student_id (PK)
fk_city_of_residence

@dataclass
class Student:
  city_of_residence: City
```

```
CITY
----
city_id (PK)
city_name

@dataclass
class City:
  students: list[Student]
```

1:N relationships

- In SQL, there is no explicit foreign key (e.g., CITY to STUDENT): the same foreign key is used to navigate the relationship in both directions
- In Python, both directions (if needed) must be represented explicitly

N:M relationships

```
ARTICLE
-------
article_id
Article data …
```

```
AUTHORSHIP
----------
article_id (FK, PK*)
creator_id (FK, PK*)
authorship_id (PK#)
```

```
CREATOR
-------
creator_id (PK)
Creator data ...
```

```
@dataclass
class Article:
  creators: set[Creator]
```

```
@dataclass
class Creator:
  articles: set[Article]
```

N:M relationship

- In SQL, there is an extra table just for the N:M relationship
  - The primary key may be an extra field (#) or a combination of the foreign keys (*)
- In Python, the extra table is not represented, and the primary key is not used

Storing keys vs objects

- Store the value of the foreign key

```
city_of_residence_id: int
```

- Pros
  - Easy to retrieve
- Cons
  - Must call a read method from the DAO to get all the data
  - Tends to perform more queries

Storing keys vs objects

- Store a fully initialized object, corresponding to the matching foreign row

```
city_of_residence: City
```

- Pros
  - Gets all data at the same time (eager loading)
  - All data is readily available
- Cons
  - Harder to retrieve data (must use a join or multiple/nested queries)
  - Maybe loaded data will not be needed

Storing keys vs objects

- Store a partially initialized object, with only the id field set (or even a null field)

```
city_of_residence :
      City = field(default_factory=lambda: []) //
lazy
city_of_residence : City = None // lazy
```

- Pros
  - Easy to retrieve
  - Loading details may be hidden behind getters
- Cons
  - Must ask the DAO to have the real data (lazy loading), but only once

Identity problem

- It may happen that a single object gets retrieved many times, in different queries
  - Especially in the case of N:M relationships
- Different “identical” objects will be created
  - They can be used interchangeably
  - They waste memory space
  - They cannnot be compared for identity (`==` or `!=`)
  - It is not possible to store additional information in those objects

Identity problem

```
...
articles = dao.list_articles()
for article in articles:
   authors = dao.get_creators_for(article)
   article.creators(authors)
...


def get_creators_for(article):
   ...
   authors = []
   for row in cursor:
      authors.append(Creator(...))
   ...
   return authors
```

Identity map pattern

- Solution: avoid creating pseudo-identical objects
  - Store all retrieved objects in a map (for example, using a dictionary)
  - Looks up objects using the map when referring to them: object shall not be created if it is already in the map

Creating an identity map

- One identity map per database table
- The identity map stores a dictionary
  - Key: field(s) of the table that constitute the primary key
  - Value: object representing the table

Using the identity map

- Create and store the identity map in the model
- Pass a reference to the identity map to the DAO methods
- In the DAO, when loading an object from the database, first check the map
  - If there is a corresponding object, return it (do not create a new one)
  - If there is no corresponding object, create a new object and put it into the map, for future reference
- If possible, check the map before doing the query

ORM libraries

- There are many Object Relational Mappers in Python, i.e., librariesthat implement the ORM logic and usually much more (they integrate the connector, implement DAO)

Connection pooling

- Opening and closing DB connection is expensive
  - Requires setting up TCP/IP connection, checking authorization, etc.
  - After just 1-2 queries, the connection is dropped and all partial results are lost in the DBMS
- Connection pool
  - A set of “already open” database connections
  - DAO methods “lend” a connection for a short period, running queries
  - The connection is then returned to the pool (not closed) and is ready for the next DAO needing it

Connection pooling

- The `mysql.connector.pooling` module implements pooling
- A pool opens a number of connections and handles thread safety when providing connections to requesters
- A connection pool has several properties:
  - `size`: indicates the number of connections available in the pool; it is configurable at pool creation time and cannot be resized thereafter
  - `name`: can be retrieved from the connection pool or connections obtained from it
- It is possible to have multiple connection pools; this enables applications to support pools of connections to different MySQL servers, for example
- For each connection request, the pool provides the next available connection
  - No round-robin or other scheduling algorithm is used: if a pool is exhausted, a `PoolError` exception is raised

Creating a pool

- To create a pool, the following code needs to be used

```
cnxpool = mysql.connector.pooling.
MySQLConnectionPool(pool_name="mypool",
                    pool_size=3,
                    user="admin",
                    host="localhost",
                    database="test')
```

- The returned object is an instantiation of the class **`PooledMySQLConnection`**
  - Differently from **`MySQLConnection`** objects, **`PooledMySQLConnection`** objects cannot be used directly as connections, but it is necessary to lend a connection from them

Lending a connection

- It is possible to ask a connection from the pool using the **`get_connection()`** method

```
cnx = cnxpool.get_connection()
```

- If there are no connections available, the method raises a **`PoolError`** exception
- The returned object is an instantiation of the class **`PooledMySQLConnection`**
  - It is similar to a **`MySQLConnection`** object, but with one notable difference: the **`close()`** method return the connection to the pool, does not terminate it

References

- mysql-connector-python
  - Coding examples: https://dev.mysql.com/doc/connector-python/en/connector-python-examples.html
  - Tutorial: https://dev.mysql.com/doc/connector-python/en/connector-python-tutorials.html
  - Connection arguments and option files: https://dev.mysql.com/doc/connector-python/en/connector-python-connecting.html
  - API reference: https://dev.mysql.com/doc/connector-python/en/connector-python-reference.html

References

- Comparison of different SQL implementations
  - http://troels.arvin.dk/db/rdbms/
- DAO pattern
  - https://en.wikipedia.org/wiki/Data_access_object
  - https://www.analyticsvidhya.com/blog/2023/02/what-are-data-access-object-and-data-transfer-object-in-python/

References

- ORM patterns and identity map
  - Patterns of Enterprise Application Architecture, By Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, Randy Stafford, Addison Wesley, 2002, ISBN 0-321-12742-0
  - https://en.wikipedia.org/wiki/Object%E2%80%93relational_mapping#:~:text=Object%E2%80%93relational%20mapping%20(ORM%2C,from%20within%20the%20programming%20language

References

- Connection pooling
  - https://dev.mysql.com/doc/connector-python/en/connector-python-connection-pooling.html