Python and Databases

Access, DAO Pattern, ORM, Identity Map, Pooling
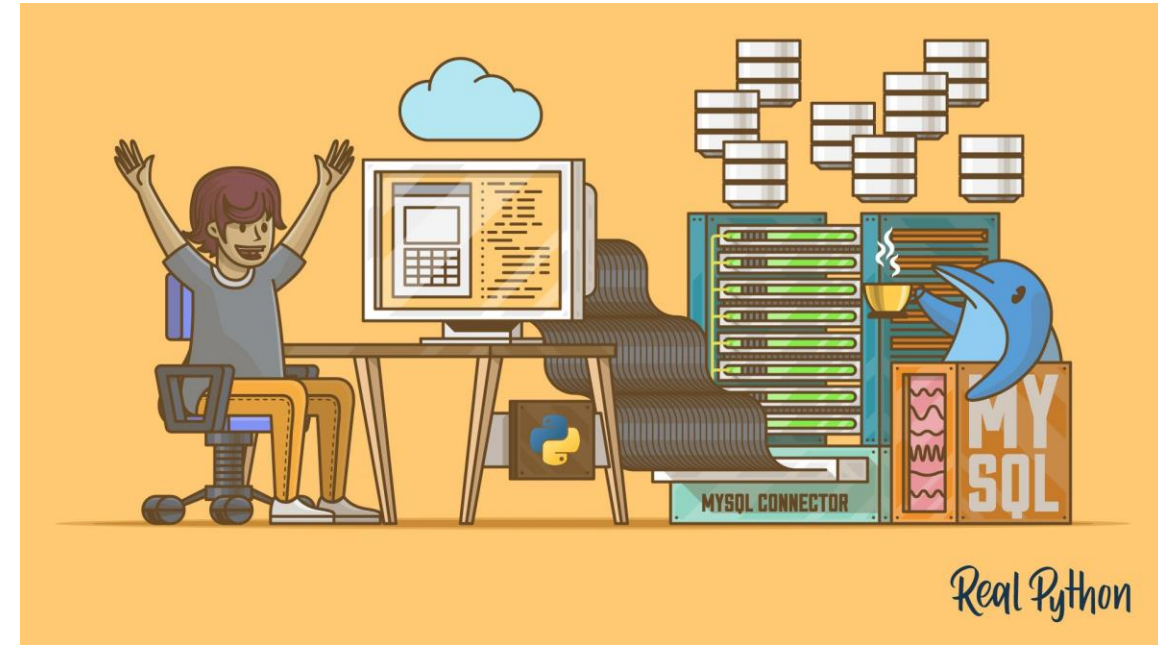
Giuseppe Averta

Carlo Masone

Francesca Pistilli

Davide Buoso

Gaetano Falco

https://realpython.com/python-mysql/
https://realpython.com/python-sql-libraries/
https://realpython.com/python-pyqt-database/

Outline

- Tools
  - MariaDB, MySQL, DBeaver
- Database access in Python
  - mysql-connector, Connection, Cursor, Statements
- Pattern DAO
- Object-Relational Mapping (ORM)
- Connection Pooling

Goal

- Enable Python applications to access data stored in Relational Databases
  - Query existing data
  - Modify existing data
  - Insert new data
- The data can be used by
  - The algorithms running in the application
  - The user, through the user interface

TOOLS

Tools: MariaDB / MySQL

Database Management Systems (DBMS) are software systems used to store, retrieve, and run queries on data, as well as administer the data. A DBMS allows end-users/applications to interact with a database.

https://mariadb.org/

https://www.mysql.com/

Tools: DBeaver

Graphical frontend to work with a database:
- Data Editor
- SQL Editor
- Task management
- Database maintenance tools
- NB: STICK WITH VERSION SUGGESTED
  - Dbeaver 23.x

https://dbeaver.io/

Tools: MySQL Workbench

Graphical frontend to work with a database:

- Data Editor

- SQL Editor

- Task management

- Database maintenance tools

- Slightly better supported

https://dev.mysql.com/downloads/workbench/

mySQL-connector

DATABASE ACCESS IN PYTHON

Resources

- Official **mySQL-connector** guide: https://dev.mysql.com/doc/connector-python/en/

- Useful tutorials: https://www.geeksforgeeks.org/how-to-connect-python-with-sql-database/

Connecting and interacting with DBMS

- **Database management system** (DBMS): software system that enables users to define, create, maintain and control access to the database

- Different flavours of SQL-based DBMSs: MySQL, MariaDB, PostgreSQL, SQLite, and SQL Server, …

  – All of these databases are compliant with the SQL standards but with varying degrees of compliance

https://troels.arvin.dk/db/rdbms/

Interacting with DBMS in Python
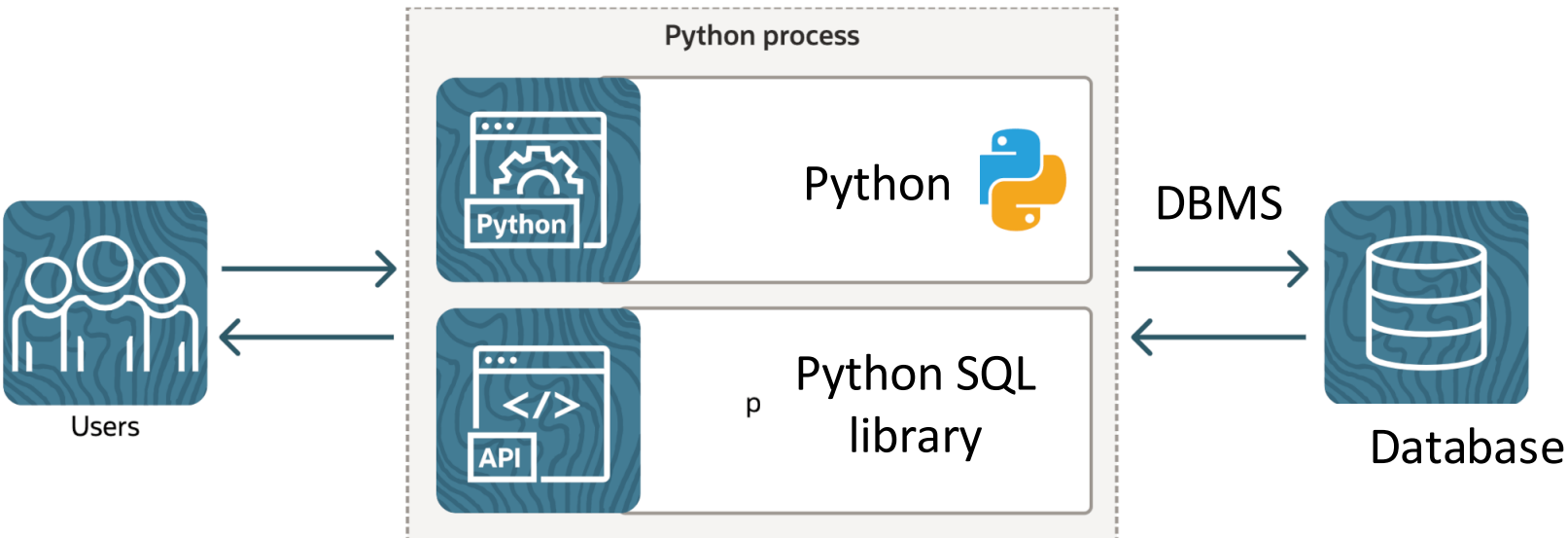
Many different Python libraries (**connectors**) that implement modules for interacting with different DBMS
- `mysql-connector-python`
- `SQLite`
- `Psycopg2`
- `mariadb-connector-python`
- …

https://realpython.com/python-sql-libraries

Python Database API Specification

The Python Database API (DB-API) defines a standard interface for Python database access modules, e.g., using **Connection** and **Cursor** objects.
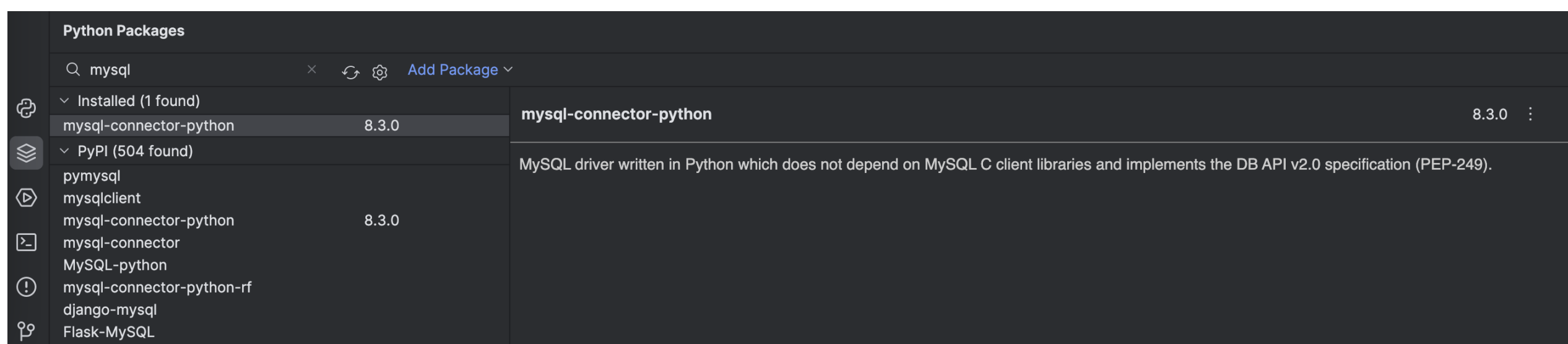Goals:
- Encourage similarity between the Python modules that are used to access databases.
- Achieve a consistency leading to more easily understood modules.
- Code that is generally more portable across databases.

https://peps.python.org/pep-0249/

mysql-connector-python

- It 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/

- Install via pip (`pip install mysql-connector-python`) , or directly in PyCharm in the virtual environment of the project

Python Packages

🔍 mysql ✕ ↻ ⚙ Add Package ∨

∨ Installed (1 found)

mysql-connector-python	8.3.0

mysql-connector-python 8.3.0 ⋮

∨ PyPI (504 found)

MySQL driver written in Python which does not depend on MySQL C client libraries and implements the DB API v2.0 specification (PEP-249).

pymysql
mysqlclient
mysql-connector-python   8.3.0
mysql-connector
MySQL-python
mysql-connector-python-rf
django-mysql
Flask-MySQL

Connection

- The first step in interfacing a Python application with a MySQL/MariaDB server is to establish a connection

- mysql-connector provides a `connect()` function that is used to establish connections to the MySQL server

```python
import mysql.connector

cnx = mysql.connector.connect(user='scott',
                              password='password',
                              host='127.0.0.1',
                              database='employees')

cnx.close()
```

https://dev.mysql.com/doc/connector-python/en/connector-python-example-connecting.html

Connection

```
import mysql.connector

cnx = mysql.connector.connect(user='scott',
                              password='password',
                              host='127.0.0.1',
                              database='employees')

cnx.close()
```

It is a `MySQLConnection` object.

The `MySQLConnection` class is used to open and manage a connection to a MySQL server. It also used to send commands and SQL statements and read the results.

The arguments of the `connect()` identify the database we want to connect, and the credentials of the user

There are many more arguments...

https://dev.mysql.com/doc/connector-python/en/connector-python-connectargs.html

Connection

```python
import mysql.connector

cnx = mysql.connector.connect(user='scott',
                              password='password',
                              host='127.0.0.1',
                              database='employees')
cnx.close()
```

The `close()` function closes the connection, when we don't need it anymore.

- The connection to the database is a resource!!!

Connection

- The `connect()` function may raise exceptions (for example if the connection fails due to wrong authentication)

```python
import mysql.connector
from mysql.connector import errorcode

try:
    cnx = mysql.connector.connect(user='scott',
                                  database='employ')
except mysql.connector.Error as err:
    if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
        print("Something is wrong with your user name or password")
    elif err.errno == errorcode.ER_BAD_DB_ERROR:
        print("Database does not exist")
    else:
        print(err)
else:
    cnx.close()
```

We can 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

This may also be rewritten using a with statement

https://docs.python.org/3/reference/compound_stmts.html#with

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 to use a separate config file.

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

Example config file

```
[client]
user=John
password=Wick
host=127.0.0.1
database=tests
raise_on_warnings=True
```

Using the connection

- When connected to the DBMS and we have the `MySQLConnection` object, we can interact in different ways
    - Create tables
    - Create/Update/Delete data
    - Read data

- This is achieved through the execution of **SQL statements**, using a handle structure known as **cursor**

Tables creation: 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	1st Record
101	Jyostna	6500.50	2nd Record
102	Jyothi	7550.50	3rd Record

Cursor on first record →

100	S N Rao	5500.50	1st Record
101	Jyostna	6500.50	2nd Record
102	Jyothi	7550.50	3rd Record

100	S N Rao	5500.50	1st Record
101	Jyostna	6500.50	2nd Record
102	Jyothi	7550.50	3rd Record

Cursor position after last 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

- There are several cursor classes that inherit from the `MySQLCursor`, and can be created by passing an appropriate argument to the `cursor()` function

```python
import mysql.connector

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

`MySQLCursorDict` cursor returns rows as dictionaries

`MySQLCursorNamedTuple` cursor returns rows as named tuples

`MySQLCursorPrepared` cursor is used for executing prepared statements

Cursor documentation: https://dev.mysql.com/doc/connector-python/en/connector-python-api-mysqlcursor.html

Statement execution

- 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

- <u>Example</u>: find item whose code is specified by the user

- <u>Method 1</u>: string interpolation (with concatenation or as an f-string)
  - ```
    query =
    "SELECT * FROM items
    WHERE code='"+user_code+"'" ;
    ```

Parametric queries

- SQL queries may depend on user input data

- <u>Example</u>: find item whose code is specified by the user

- <u>Method 1</u>: string interpolation (with concatenation or as an f-string)

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

Parametric queries

- SQL queries may depend on user input data

- <u>Example</u>: find item whose code is specified by the user

- <u>Method 1</u>: string interpolation (with concatenation or as an f-string)

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

- <u>Method 2</u>: use parametric Statements

  - Always preferable

  - Always

What's wrong with method 1?

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

  For example, string written by the user in a textbox in the GUI

- This may cause security problems

SQL injection

- SQL injection – syntax errors or privilege escalation
- Example
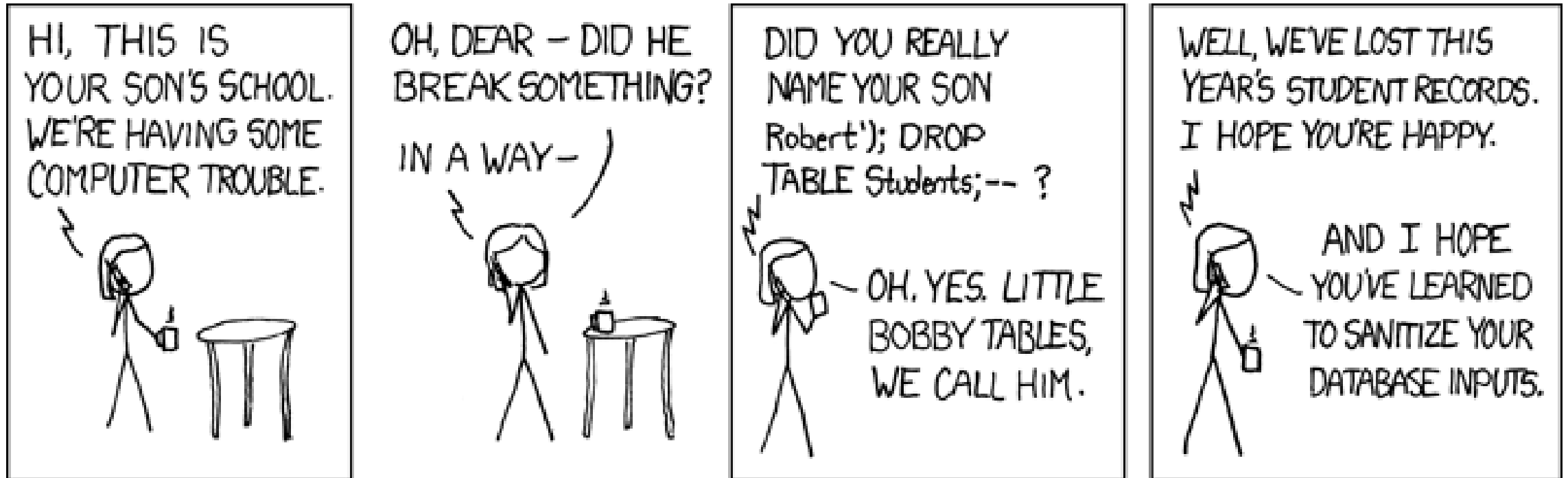 - username : `'; delete * from users ; --`

```
select * from users where
username=''; delete * from
users ; -- '
```

- **Must** detect or escape all dangerous characters!
  - Will **never** be perfect…

- **Never** trust user-entered data. Never. Not once. Really.

SQL injection attempt ☺

ZU 0666', 0, 0); DROP DATABASE TABLICE;

SQL injection attempt ☺

http://xkcd.com/327/

Parametric statements

- 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, we can execute INSERT, UPDATE and DELETE statements

```
import mysql.connector

cnx = mysql.connector.connect(user='John',
                              password='Wick',
                              host='127.0.0.1',
                              database='tests')

cursor = cnx.cursor()

add_test = """INSERT INTO tests
              (id, name, value)
              VALUES (%s, %s, %s)"""

cursor.execute(add_test, (1, "John Doe", 11.3))

cnx.commit()
cursor.close()
cnx.close()
```

- 1. Define the statement (using the Python multi-line block """ """). Values may be written in the statement, or left unspecified as %s (because it may depend on user data)

- 2. Execute the statement (setting all the unspecified values)

- 3. **Commit the changes to the database**

- 4. Close the cursor

Insert/Update/Delete data

- Using the cursor, we can execute INSERT, UPDATE and DELETE statements

```python
import mysql.connector

cnx = mysql.connector.connect(user='John',
                              password='Wick',
                              host='127.0.0.1',
                              database='tests')
cursor = cnx.cursor()

update_test = """UPDATE tests
                 SET value = %s
                 WHERE id = %s"""

cursor.execute(update_test, (99.9, 3))

cnx.commit()
cursor.close()
cnx.close()
```

```python
import mysql.connector

cnx = mysql.connector.connect(user='John',
                              password='Wick',
                              host='127.0.0.1',
                              database='tests')
cursor = cnx.cursor()

delete_test = """DELETE FROM tests
                 WHERE id = %s"""

cursor.execute(delete_test, (5,))

cnx.commit()
cursor.close()
cnx.close()
```

Query Data

- We can use a cursor also to execute a statement that queries data from the database

```python
import mysql.connector

cnx = mysql.connector.connect(user='John',
                              password='Wick',
                              host='127.0.0.1',
                              database='prova')
cursor = cnx.cursor()
query = """SELECT * FROM test"""
cursor.execute(query)

for (id, name, value) in cursor:
    print(id, name, value)

cursor.close()
```

- Executing the query fetches results from the database
- We can then use cursor as an iterator over the results set
- There is no commit, because we are not modifying the database

Query Data: process the results

- After executing the statement, we can use the cursor as a iterator to go through the results

Cursor default position (before first record)

100	S N Rao	5500. 50	1st Record
101	Jyostna	6500.50	2nd Record
102	Jyothi	7550.50	3rd Record

- Data is available a row at a time
- Rows are read as tuple, in the standard cursor. They can be read as dictionary or as named tuple using the corresponding cursor

Cursor on first record

100	S N Rao	5500. 50	1st Record
101	Jyostna	6500.50	2nd Record
102	Jyothi	7550.50	3rd Record

100	S N Rao	5500. 50	1st Record
101	Jyostna	6500.50	2nd Record
102	Jyothi	7550.50	3rd Record

Cursor position after last record

Query Data

- We can use a cursor also to execute a statement that queries data from the database

```python
import mysql.connector

cnx = mysql.connector.connect(user='John',
                              password='Wick',
                              host='127.0.0.1',
                              database='prova')

cursor = cnx.cursor(dictionary=True)
query = """SELECT * FROM test"""
cursor.execute(query)

for row in cursor:
    print(row["id"], row["name"], row["value"])

cursor.close()
```

- If we use a `MySQLCursorDict` the results set is read as a dictionary, so we can iterate through the data accordingly

Query Data: 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.

Query Data: fetchone, fetchmany, fetchall

Example
- 0. cursor.execute(query)
- 1. cursor.fetchone()
- 2. cursor.fetchall()

Cursor before the first row

100	S N Rao	5500. 50	1st Record
101	Jyostna	6500.50	2nd Record
102	Jyothi	7550.50	3rd Record

Cursor at the first row

100	S N Rao	5500. 50	1st Record
101	Jyostna	6500.50	2nd Record
102	Jyothi	7550.50	3rd Record

100	S N Rao	5500. 50	1st Record
101	Jyostna	6500.50	2nd Record
102	Jyothi	7550.50	3rd Record

Cursor after the last row

Query Data: fetchone, fetchmany, fetchall

Example

```python
import mysql.connector

cnx = mysql.connector.connect(user='John',
                              password='Wick',
                              host='127.0.0.1',
                              database='prova')
cursor = cnx.cursor(dictionary=True)
query = """SELECT * FROM test"""
cursor.execute(query)

rows = cursor.fetchall()
print(rows)

cursor.close()
```

Query Data: fetchone, fetchmany, fetchall

Warning: when executing a query to read the data, we are expected to handle all the results.

```python
query = """SELECT * from test"""
cursor.execute(query)


row1 = cursor.fetchone()
print(row1)


cursor.close()
cnx.close()
```

🐍 test_db ✕

```
  File "/Users/carlo/TdP2024/test_db/pythonProject/test_db.py", line 49, in <module>
    cursor.close()
  File "/Users/carlo/TdP2024/test_db/pythonProject/.venv/lib/python3.12/site-packages/mys
    self._cnx.handle_unread_result()
  File "/Users/carlo/TdP2024/test_db/pythonProject/.venv/lib/python3.12/site-packages/mys
    raise InternalError("Unread result found")
mysql.connector.errors.InternalError: Unread result found
(1, 'prova carico', 99.9)
```

Query Data: fetchone, fetchmany, fetchall

Type conversion MySQL -> Python

- By default, MySQL types in result sets are converted automatically to Python types. For example, a `DATETIME` column value becomes a `datetime.datetime` object. To disable conversion, one can use a cursor with the option cursor(raw=True)

- You can check the read type using the Python `type()` function

DATA ACCESS OBJECT (DAO)

Problems

- Database code involves a lot of «specific» knowledge
  - Connection parameters
  - SQL commands
  - The structure of the database
- Bad practice to «mix» this low-level information with main application code
  - Reduces portability and maintainability
  - Creates more complex code
  - Breaks the «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 DB 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

Pattern DAO

- 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 and you do not need to know how it does this, that's what an abstraction is.

- DAO is implemented in a separate file. Then, these methods are called in the main application.
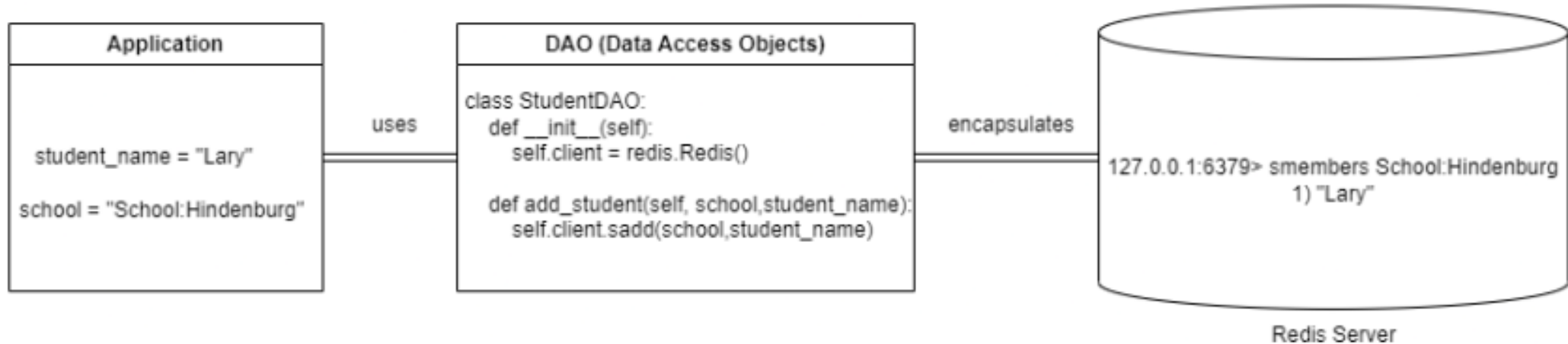
Pattern DAO

Image source https://www.analyticsvidhya.com/blog/2023/02/what-are-data-access-object-and-data-transfer-object-in-python/

Data Access Object (DAO) – 1/2

- «Client» classes:
  - Application code that needs to access the database
  - Ignorant of database details (connection, queries, schema, …)
- «DAO» classes:
  - Encapsulate all database access code (mysql-connector-python)
  - The only ones that will ever contact the database
  - Ignorant of the goal of the Client

Data Access Object (DAO) – 2/2

- Low-level database classes, to handle the connection (`MySQLConnection`, Pooled connection,…)
  - Used by DAO (only!) but invisible to Client
- «Transfer Object» (TO) or «Data Transfer Object» (DTO) classes
  - 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 Diagram
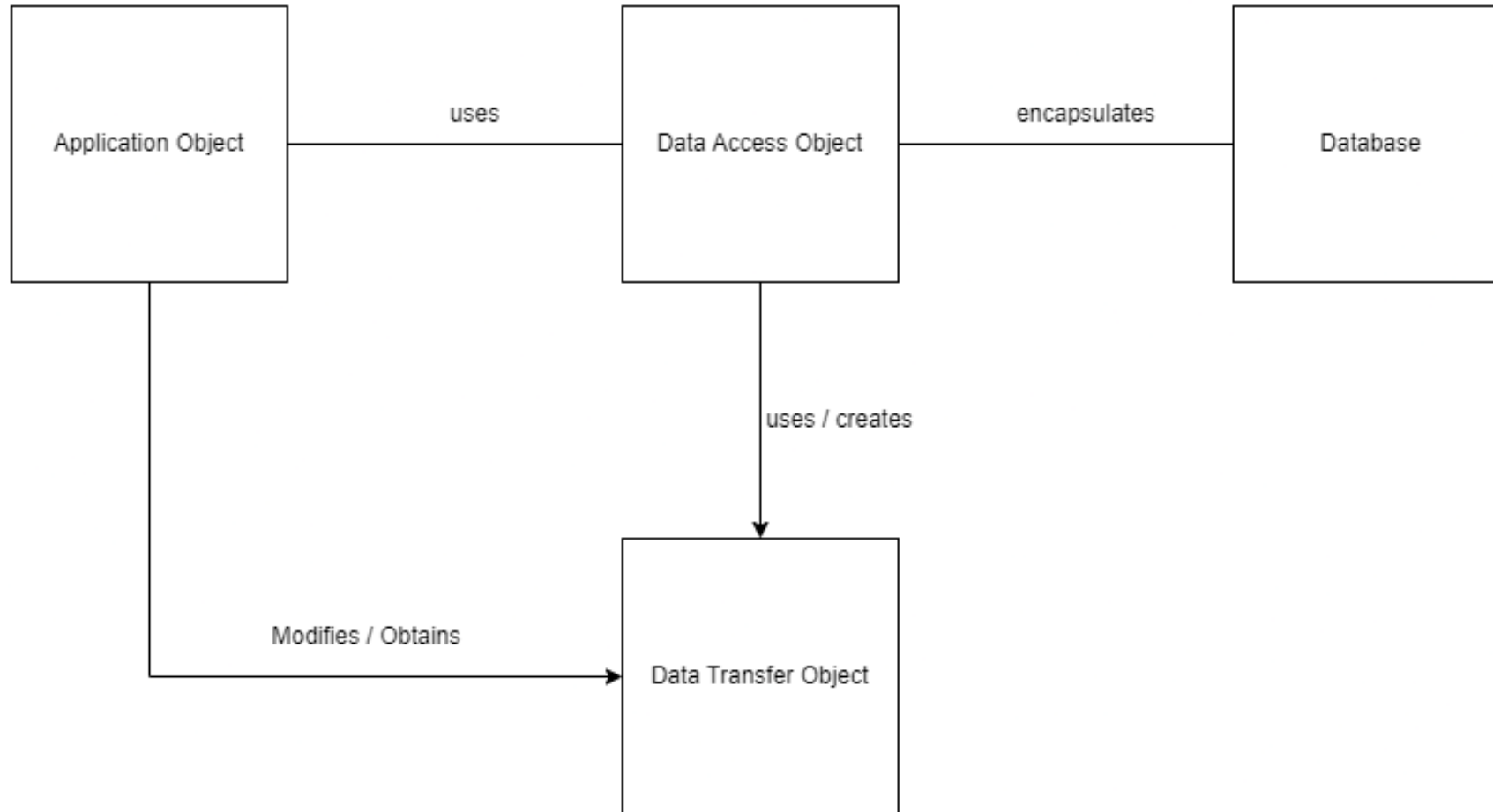
DAO: application example

```python
from dao import StudentDAO
from dto import StudentDTO

student_dao = StudentDAO()

student1= "Lary"
student2 = "Mike"
school = "School:Hindenburg"

stud1 = StudentDTO(school,student1)
stud2 = StudentDTO(school,student2)

student_dao.add_student(stud1)
student_dao.add_student(stud2)

student_names = student_dao.get_all_student("School:Hindenburg")
print(student_names)
```

For example a @dataclass

The DAO then implements the methods to interface with the student table in the database

50

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 DB tables
  - If you need more, create many DAO classes
- DAO offers CRUD methods
  - Create, Read, Update, Delete
- DAO may offer search methods
  - Returning collections of DTO
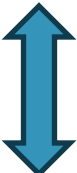
DAO: example

```python
class studentDAO:
    def __init__(self):
        #possibly a state to keep information about a connection
        # we will see this with pooling

    def get_methods(self):
        try:
            cnx = mysql.connector.connect(database='student')
        except mysql.connector.Error as err:
            print(err)
            result = None
        else:
            result = []
            cursor = cnx.cursor(dictionary=True)
            cursor.execute("SELECT * FROM students")
            for row in cursor:
                result.append(studentDTO(row["id"], row["name"]))
            cursor.close()
        finally:
            cnx.close()
            return result
```

OBJECT-RELATIONAL MAPPING

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, we represent entities as objects, and we need to represent their relationships

Object Relational Mapping (ORM)

Database,
table Pets

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

DTO Pet

Class Pet

Mapping Tables to Objects

- Goal: guidelines for creating a set of (data)classes to represent information stored in a relational database: will be used as DTO
- Goal: guidelines for designing the set of methods for DAO objects

Tables ➔ data class ORM rules

1. Create one dataclass per each main database entity
   - Except tables used to store n:m relationships!
2. Class names should match table names
   - In the singular form (Utente; User)
3. The Class should have one attribute for each column in the table, with matching names
   - According to Python naming conventions (NUMERO_DATI -> numero_dati)
   - Match the data type
   - Except columns used as foreign keys

Tables ➔ data class ORM rules

4. Add the getter (`@property`) and setter (`@attr.setter`) methods for the attributes, if needed. The setter method cannot be specified if the dataclass uses the `frozen=True` parameter.

5. Define `__eq__()` and `__hash__()` using the **exact** set of fields that compose the primary key of the table

Relationships, Foreign keys → Class

- Define additional attributes in the DTO classes, for every relationship that we want to easily navigate in our application

  – Not necessarily *all* relationships!

Cardinality-1 relationship

- A relationship with cardinality **1** maps to an attribute referring to the corresponding Python object
  - not the PK value
- Use singular nouns.

1:1 relationship

```
STUDENTE
--------
matricola (PK)
fk_persona

@dataclass
class Studente:
      persona: Persona
      codice_fiscale: str
```

```
PERSONA
-------
codice_fiscale (PK)
fk_studente

@dataclass
class Persone:
      studente: Studente
      matricola: int
```

Cardinality-N relationship

- 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 PK values).
  - Use plural nouns.
- The class should have methods for reading (get, ...) and modifying (add, ...) to the collection

1:N relationship

STUDENTE
----
matricola (PK)
fk_citta_residenza

```
@dataclass
class Studente:
        cittaResidenza: Citta
```

CITTA
----
cod_citta (PK)
nome_citta

```
@dataclass
class Citta:
        studentiResidenti: list[Studente]
```

1:N relationship

```
STUDENTE
-----------------
matricola (PK)
fk_citta_residenza

@dataclass
class Studente:
    cittaResidenza: Citta
```

```
CITTA
--------------
cod_citta (PK)
nome_citta

@dataclass
class Citta:
    studentiResidenti: list[Studente]
```

In SQL, there is no «explicit» Citta->Studente foreign key. The same FK is used to navigate the relationship in both directions.

In Python, both directions (if needed) must be represented explicitly.

N:M relationship

ARTICLE

----

id_article (PK)
Article data...

AUTHORSHIP

----

id_article (FK,PK*)
id_creator (FK,PK*)
id_authorship (PK$^\#$)

CREATOR

----

id_creator (PK)
Creator data...

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

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

N:M relationship

ARTICLE

----

id_article (PK)

Article data…

AUTHORSHIP

----

id_article (FK,PK*)

id_creator (FK,PK*)

id_authorship (PK#)

CREATOR

----

id_creator (PK)

Creator data…

> In SQL, there is an extra table just for the N:M relationship .

> The PK may be an extra field (#) or a combination of the FKs (*)

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

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

> The extra table is not represented.
> The PK is not used.

Storing Keys vs Objects

`id_citta_residenza: int`

- Store the *value* of the foreign key
- Easy to retrieve
- Must call a read method from the DAO to get all the data
- Tends to perform more queries

`citta_residenza: Citta`

- Store a *fully initialized object,* corresponding to the matching foreign row
- Harder to retrieve (must use a Join or multiple/nested queries)
- Gets all data at the same time (eager loading)
- All data is readily available
- Maybe such data will not be needed

Storing Keys vs Objects (3rd way)

```
citta_residenza : Citta = field(default_factory=lambda: []) // lazy
citta_residenza : Citta = None                               // lazy
```

- Store a *partially initialized object*, with only the 'id' field set
  - Or even a null field
- Easy to retrieve
- Must ask the DAO to have the real data (lazy loading), but only once
- Loading details may be hidden behind getters

Identity problem

- It may happen that a single object gets retrieved many times, in different queries
  - Especially in the case of N:M relationships

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

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

Identity problem

- It may happen that a single object gets retrieved many times, in different queries
  - Especially in the case of N:M relationships

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

```
…
authors = []
for row in cursor:
    authors.append(Creator(…))
    ...uthors
```

If the same Creator is author of many articles, we would repeatedly query the database to create new objects with identical information

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 can't be compared for identity (== or !=)
  - You can't store additional information in those objects
- Solution: avoid creating pseudo-identical objects
  - Store all retrieved objects in a map (for example, using a dictionary)
  - Don't create an object if it's already in the map
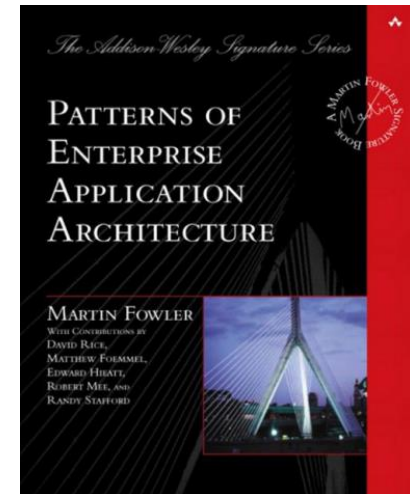
Identity Map pattern

- Ensures that each object gets loaded only once, by keeping every loaded object in a map

- Looks up objects using the map when referring to them.
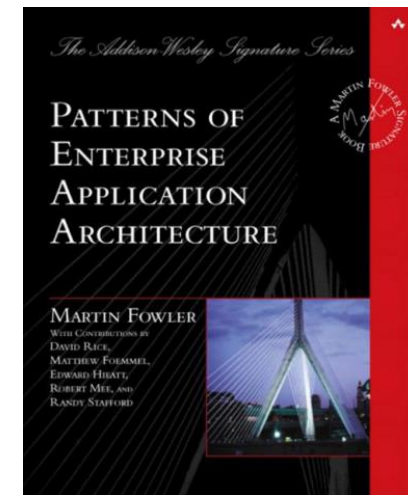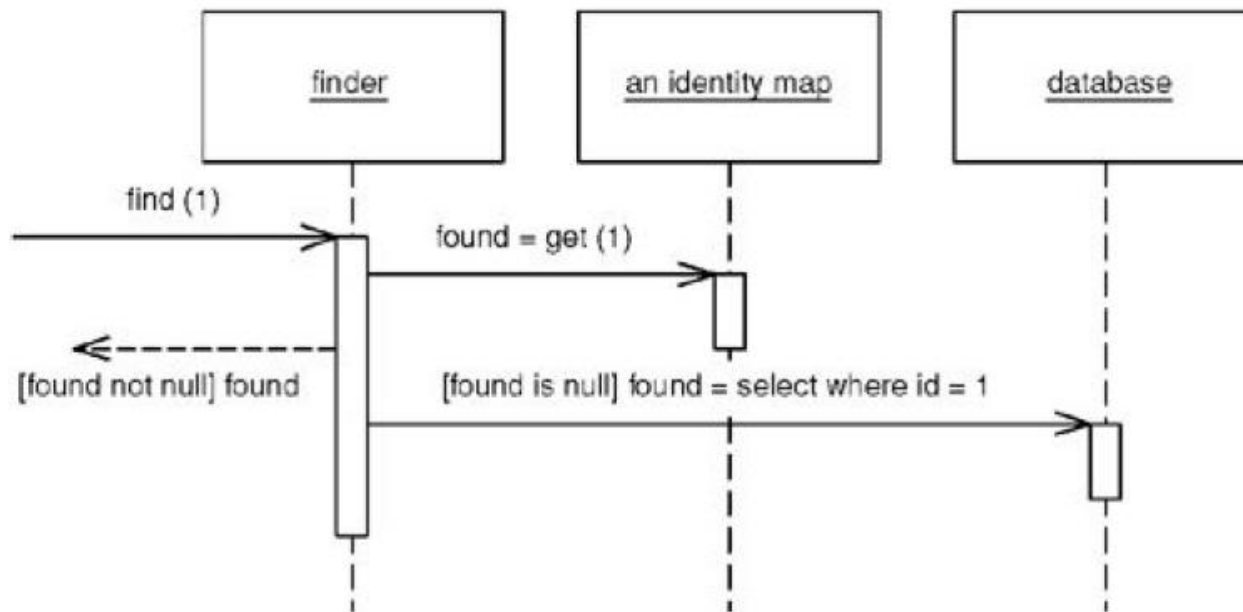
Creating an Identity Map

- 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 (and don't 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
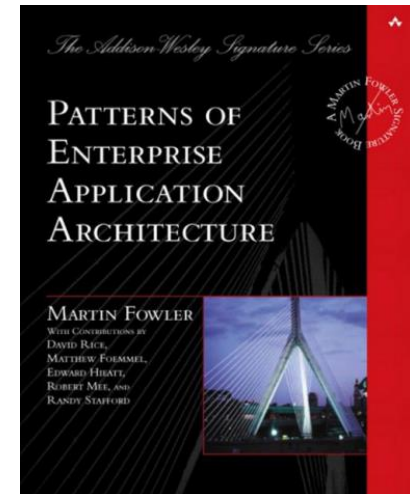
ORM Libraries

- There are many **Object Relational Mappers** in Python, that are libraries that implement the ORM logic and usually much more (they integrate the connector, implement DAO)
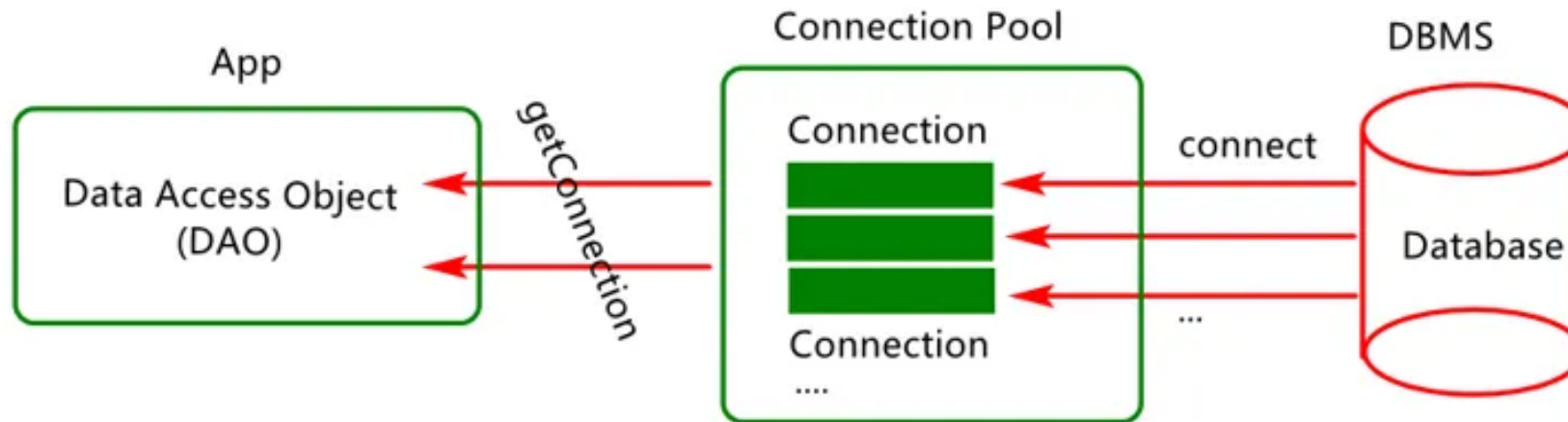
CONNECTION POOLING

Connection pooling

- Opening and closing DB connection is expensive
  - Requires setting up TCP/IP connection, checking authorization, …
  - 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 conceptual model

Connection pooling with mysql-connector

- 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` is raised.

Creating a pool

```
cnxpool = mysql.connector.pooling.MySQLConnectionPool(pool_name = "mypool",
                                    pool_size = 3,
                                    database = "test",
                                    user = "John",
                                    password = "Wick",
                                    host = "127.0.0.1")
```

The cnxpool object is an instantiation of the class PooledMySQLConnection .
Differently from MySQLConnection objects, PooledMySQLConnection objects
cannot be used directly as connections, but we must lend a connection from them

Lending a connection

- We can ask a connection from the pool using the `get_connection()` method
  - Warning: if there are no connections available this method raises a `PoolError` exception

```
cnxpool = mysql.mysql.connector.pooling.MySQLConnectionPool(pool_name = "mypool",
                                                            pool_size = 3,
                                                            database = "test",
                                                            user = "John",
                                                            password = "Wick",
                                                            host = "127.0.0.1")

cnx = cnxpool.get_connection()

//do operations

cnx.close()
```

`cnx` 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!

Benchmarks

# Iterations	1	10	100	1000	10000
Non-Pooling	0.012s	0.081s	0.717s	8.81s	92.2s
Pooling	0.031s	0.039s	0.081s	0.351s	2.42s

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/
  - essential!
- 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

License

- These slides are distributed under a Creative Commons license "**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**"
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/