



# Python and Databases

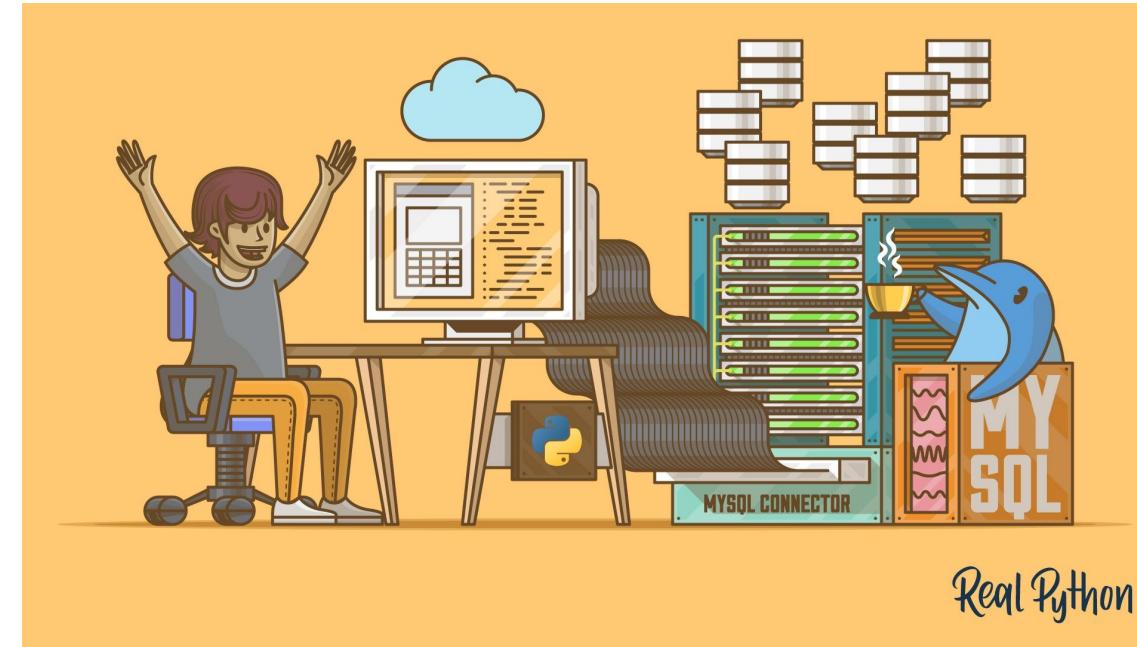
**Access, DAO Pattern, ORM, Identity Map, Pooling**

Fulvio Corno

Giuseppe Averta

Carlo Masone

Francesca Pistilli



Real Python

<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

<http://dilbert.com/strips/comic/1995-11-17/>



# 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

• Interfaccia coe database!

Graphical frontend to work with a database:

- Data Editor
- SQL Editor
- Task management
- Database maintenance tools



<https://dbeaver.io/>



mySQL-connector

# **DATABASE ACCESS IN PYTHON**

# Resources

↑  
Useremo il **base mySQL-connector** x fare l'accesso al database in Python!

- 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

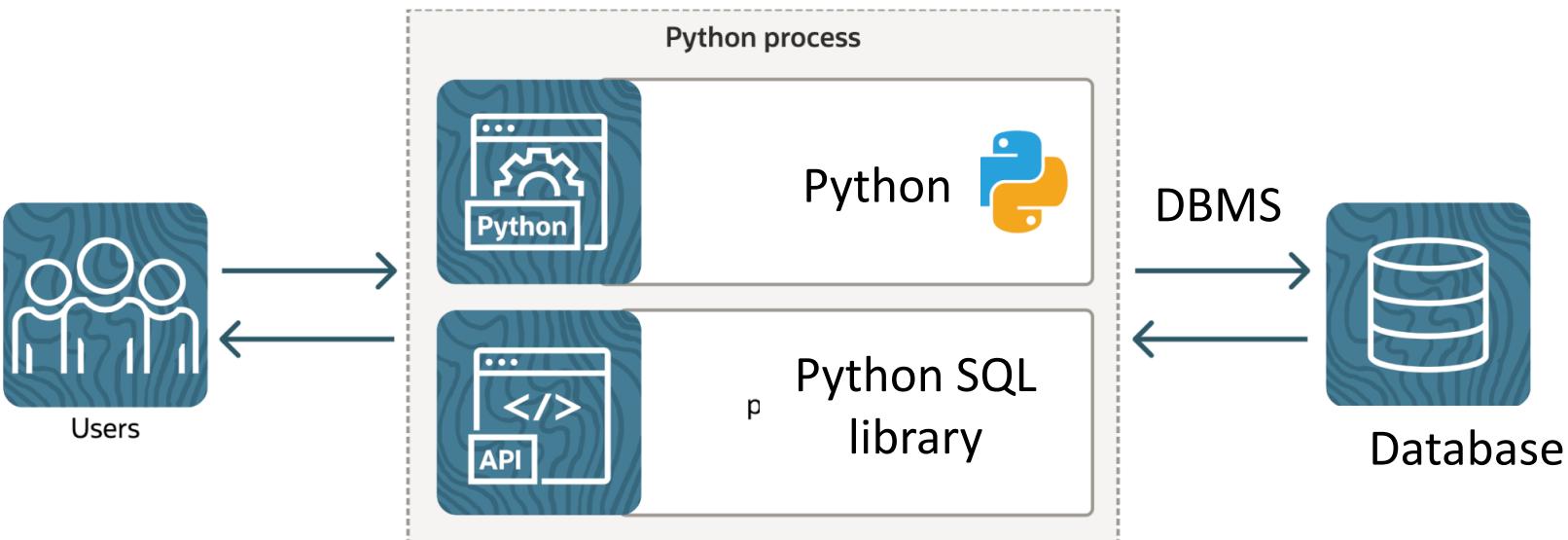
- **Database management system (DBMS)**: software system that enables users to define, create, maintain and control access to the database
- Different flavours of SQL-based DBMSs: MySQL, MariaDB, PostgreSQL, SQLite, and SQL Server, ...
  - All of these databases are compliant with the SQL standards but with varying degrees of compliance

#DBMS; noi useremo mySQL!

Come i DBMS implementano lo standard SQL!

<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`
- ...

Noi dovremo installare la libreria `mysql-connector-python` tramite pip in modo da rendere + easy l'interazione tra python e il database

<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.

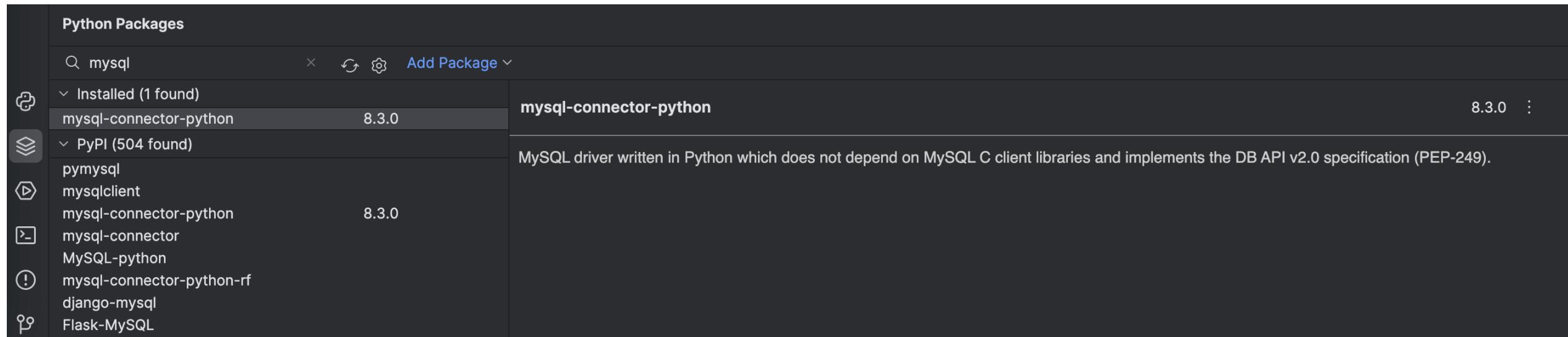
• Tramite la specifica **Python Database API** si definisce uno standard, contenente i vari criteri che un connettore, che dovrà far comunicare Python col database, dovrà rispettare.

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

# mysql-connector-python

→ Guida a l'installazione di 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/> → Documentazione RICCA! ~ Possiamo anche utilizzarla in sede d'esame!
- Install via pip (`pip install mysql-connector-python`) , or directly in PyCharm in the virtual environment of the project



# 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

• Se uso **mysql-connector** come libreria, la connessione avverà sempre nello stesso identico modo!

```
import mysql.connector  
  
cnx = mysql.connector.connect(user='scott',  
                                password='password',  
                                host='127.0.0.1',  
                                database='employees')  
  
cnx.close()  
  
Va sempre usata la funzione connect()!  
La funzione connect() ci restituisce un oggetto che chiamiamo cnx e che è un oggetto connessione  
Dopo aver lavorato e fatto le + query usando la connessione al DBMS, dovranno chiudere la connessione usando il metodo .close() sull'oggetto cnx!
```

La funzione `connect()` riceverà sempre in input:  
• `user`: solitamente è `'root'`.  
• `password`: password del DBMS  
• `host` = `'127.0.0.1'` (fisso!).  
• `database`: nome del database col quale voglio stabilire la connessione!

<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

```
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)

La connessione al database può anche fallire per tanti vari motivi... per esempio se metto le mie credenziali sbagliate, la connessione fallirà. Come vengono gestiti questi problemi? Vengono gestiti lanciando delle eccezioni. La libreria mysql-connector fornisce una serie di Error codes che vanno a distinguere diversi tipi di errori (credenziali sbagliate, ecc), quindi nel nostro programma dovremo gestire queste eccezioni, quindi la chiamata alla connessione dovremo sempre incapsularla in un blocco try-except!

```
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()

    ↗ Se non si verificano eccezioni, entra nel blocco else!
    ↗ CREDENZIALI ERRATE!
    ↗ ECCEZIONE X DATABASE INESISTENTE
```

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](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
```



Esempio di file contenente i dati d'accesso al database!

# 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**

• Una volta creata la connessione al database, potremo andare a modificare tabella (soltanente la creazione, cancellazione e modifica delle tabelle la faremo solo su DBeaver, senza mai usare Python!) oppure modificare/ leggere i dati delle tabelle!

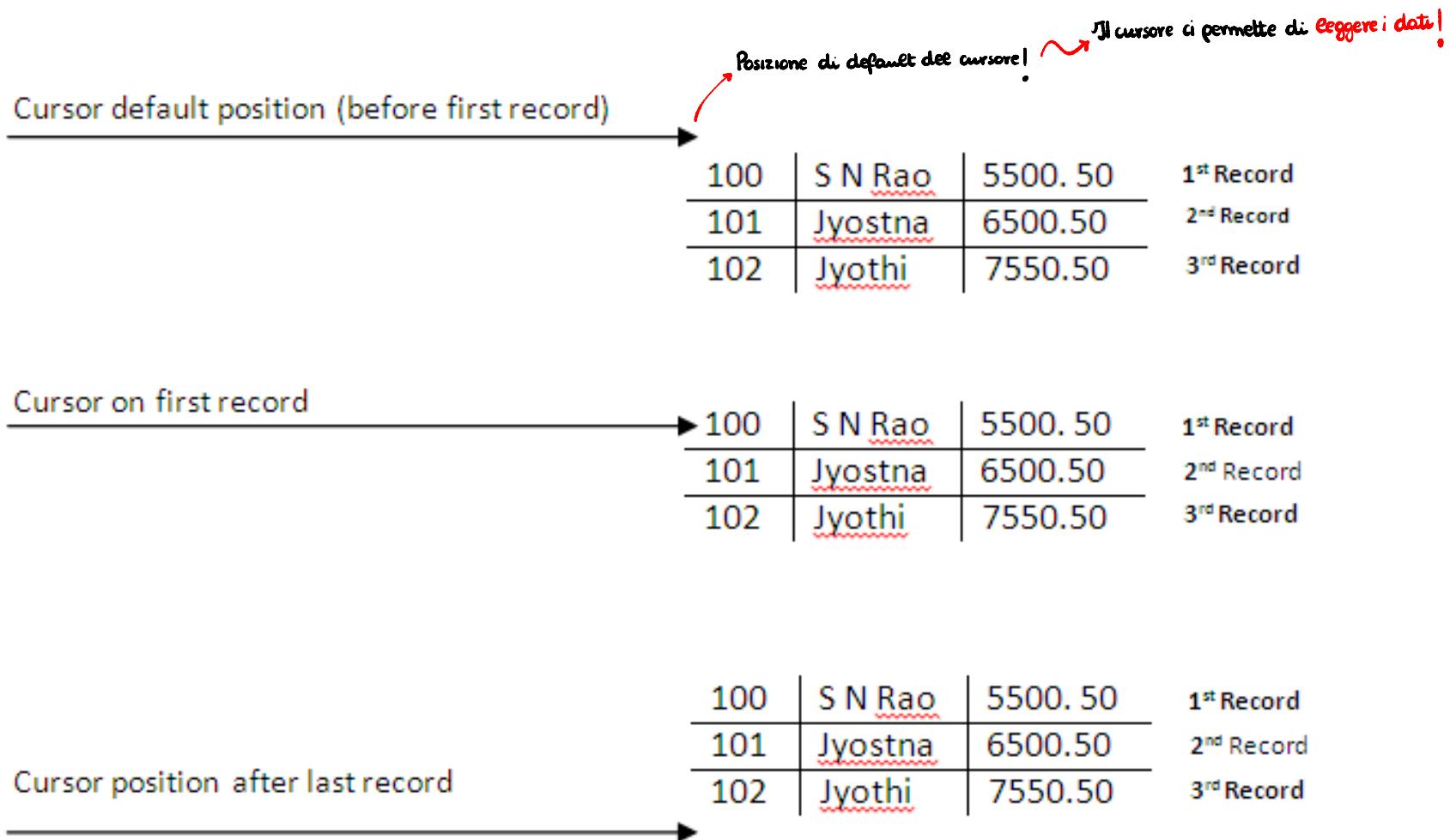
INSERT, SET, DELETE!      SELECT

Per le operazioni sui dati delle tabelle avremo sempre bisogno di:

- Statement
- Cursore

Tables creation: <https://dev.mysql.com/doc/connector-python/en/connector-python-example-ddl.html>

# Cursor



# Cursor

La funzione cursor() può ricevere dei parametri in input. A seconda di cosa metto = True, definisco il tipo di risultati che voglio ottenere dal cursore scorrendo la tabella.  
Se non metto nessun parametro, le righe della tabella ci viene restituita come una tupla.  
Se metto dictionary=True, le righe ci vengono restituite come se fossero dizionari.  
Tutti i tipi di cursori posso trovarli all'interno della documentazione!

- The **MySQLCursor** class instantiates objects that can execute operations such as SQL statements. A cursor is created from a **MySQLConnection** using the **cursor()** function *Usando la funzione cursor() sulla connessione, posso creare un oggetto cursore!*
- There are several cursor classes that inherit from the **MySQLCursor**, and can be created by passing an appropriate argument to the **cursor()** function

```
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

L'oggetto cursore avrà un metodo `.execute()`, che riceve in input una query e restituisce in output le righe della tabella (nel caso di `SELECT`)

La query viene scritta come se fosse una normale stringa!

```
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+"'" ;`
- Le query possono dipendere dall'input fornito dall'utente...  
Posso usare quindi la concatenazione di stringhe x fare le query usando forniti in input... ma questo metodo FA CAGARE!**

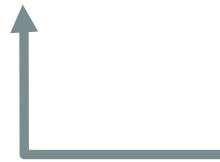
# Parametric queries

- 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+"'" ;
- Method 2: use parametric Statements
  - Always preferable
    - No concatenazione stringhe => Problemi di sicurezza (SQL Injection)
    - Bisogna usare i **parametric statements**!
  - 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 ; --`

 **SQL Injection** → Cancellazione di tutti i dati della tabella!



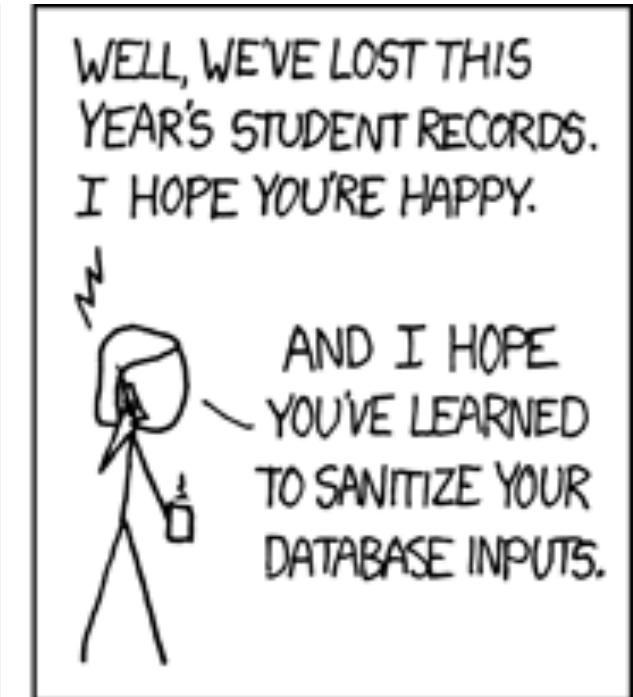
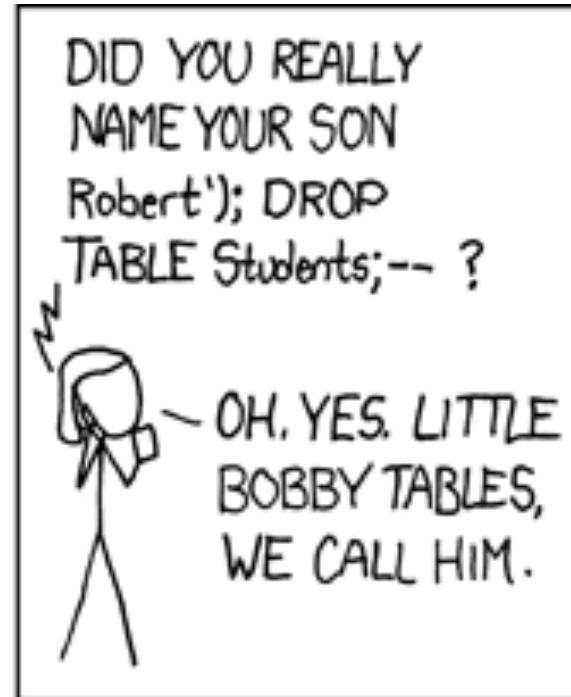
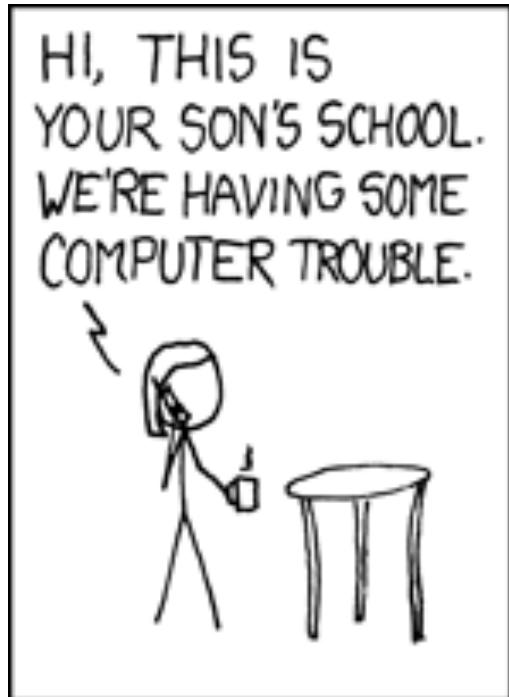
```
select * from users where  
username=''; delete * from  
users ; -- '
```

- **Must** detect or escape all dangerous characters!
  - Will **never** be perfect...
- **Never** trust user-entered data. Never. Not once. Really.

# SQL injection attempt 😊



# 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**) ↗ Uso dei **placeholders** al posto degli effettivi valori della query e poi quando faccio l'**execute** della query, mando gli effettivi valori da utilizzare al posto dei **placeholders** all'interno della query!
  - 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)

Utilizzando i **placeholders** ho tutti questi vantaggi!

# 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()
```

• La query la inserisco tra `""" """`. Al posto di indicare i parametri che ricevo in input nella query, utizzo, A parametro, il placeholder `%s`.

↳ Uso un placeholder A dato che dipende dalle input dell'utente!

- 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)  
↳ Quando faccio l'`execute`, passerò in una tupla (fra ()!), tutti i parametri da utilizzare al posto dei placeholders!
- 3. Commit the changes to the database
- 4. Close the cursor

↳ Quando faccio `INSERT, UPDATE, DELETE` devo sempre fare il `commit` sul cursor, altrimenti le modifiche non verranno salvate!

# 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()

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

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

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

```
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()
```

↳ Se passo un unico valore, devo comunque metterci una "," dopo il parametro, perché devo sempre passare una tupla

# Query Data

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

```
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

→ Quando la query è un **SELECT**, non dovrò fare **.commit()**

Per leggere dati di una tabella, dopo aver eseguito la query sul cursore, potrò utilizzare l'oggetto **cursor** come iteratore x ciclare sulle righe risultanti dalla query!

# Query Data: process the results

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

The diagram illustrates the state of a cursor across three rows of data:

- Cursor default position (before first record):** An arrow points to the first row (Employee ID 100). To its right, the table shows three rows: 1<sup>st</sup> Record (100), 2<sup>nd</sup> Record (101), and 3<sup>rd</sup> Record (102).
- Cursor on first record:** An arrow points to the first row (Employee ID 100). The first row is highlighted with a blue border, indicating it is the current record being processed. The table shows three rows: 1<sup>st</sup> Record (100), 2<sup>nd</sup> Record (101), and 3<sup>rd</sup> Record (102).
- Cursor position after last record:** An arrow points to the first row (Employee ID 100). The table shows three rows: 1<sup>st</sup> Record (100), 2<sup>nd</sup> Record (101), and 3<sup>rd</sup> Record (102).

|     |         |         |
|-----|---------|---------|
| 100 | S N Rao | 5500.50 |
| 101 | Jyostna | 6500.50 |
| 102 | Jyothi  | 7550.50 |

|     |         |         |
|-----|---------|---------|
| 100 | S N Rao | 5500.50 |
| 101 | Jyostna | 6500.50 |
| 102 | Jyothi  | 7550.50 |

|     |         |         |
|-----|---------|---------|
| 100 | S N Rao | 5500.50 |
| 101 | Jyostna | 6500.50 |
| 102 | Jyothi  | 7550.50 |

Python, di base, legge righe come fossero tuple.  
↗ Se non aggiungo nulla nella definizione del cursore!

- 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

# Query Data

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

```
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

Se uso `dictionary=True` nella definizione del cursore, e vado a ciclare sul cursore, ogni riga sarà un dizionario con chiavi il nome della colonna e valore il "valore" assunto da quell'istanza x la colonna corrispondente!

↳ Conviene sempre lavorare con i dizionari!

# Query Data: fetchone, fetchmany, fetchall

- The cursor object also has other methods to fetch the results retrieved by executing 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(N)`, `.fetchall()` sono metodi che posso chiamare sul cursore x potermi salvare in una variabile una, o + righe risultanti dall'esecuzione di una query `SELECT`.*
- ↳ `.fetchone()` restituisce una tupla/dizionario a seconda di come è stato definito il cursore; `.fetchmany()`, `.fetchall()` restituiscono una LISTA di tuple/dizionari!*

# 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 | 1 <sup>st</sup> Record |
| 101 | Jyostna | 6500.50  | 2 <sup>nd</sup> Record |
| 102 | Jyothi  | 7550.50  | 3 <sup>rd</sup> Record |

Cursor at the first row



|     |         |          |                        |
|-----|---------|----------|------------------------|
| 100 | S N Rao | 5500. 50 | 1 <sup>st</sup> Record |
| 101 | Jyostna | 6500.50  | 2 <sup>nd</sup> Record |
| 102 | Jyothi  | 7550.50  | 3 <sup>rd</sup> Record |

Cursor after the last row



|     |         |          |                        |
|-----|---------|----------|------------------------|
| 100 | S N Rao | 5500. 50 | 1 <sup>st</sup> Record |
| 101 | Jyostna | 6500.50  | 2 <sup>nd</sup> Record |
| 102 | Jyothi  | 7550.50  | 3 <sup>rd</sup> Record |

# Query Data: fetchone, fetchmany, fetchall

## Example

```
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()
```

In questo caso, la variabile  
**rows** sarà una lista di dizionari!

# Query Data: fetchone, fetchmany, fetchall

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

• Dovrò leggere **TUTTE** le righe di una query SELECT, altrimenti avrò un'eccezione!

```
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/mysql
    self._cnx.handle_unread_result()
File "/Users/carlo/TdP2024/test_db/pythonProject/.venv/lib/python3.12/site-packages/mysql
    raise InternalError("Unread result found")
mysql.connector.errors.InternalError: Unread result found
(1, 'prova carico', 99.9)
```

# Query Data: fetchone, fetchmany, fetchall

```
43 query = """SELECT * from test"""
44 cursor.execute(query)
45
46 row = cursor.fetchone()
47 while row is not None:
48     print(row)
49     row = cursor.fetchone()
50
51 cursor.close()
```

while row is not None

Run test\_db ×

```
/Users/carlo/TdP2024/test_db/python
(1, 'prova carico', 99.9)
(2, 'prova latenza', 1.0)
(3, 'test supporto', 3.4)
(4, 'test campo', 3.4)
```

Risultati rispettivamente di un fetchone() e di una fetchall()!

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

row = cursor.fetchone()
print(row)
rows = cursor.fetchmany()
print(rows)

cursor.close()
```

Run test\_db ×

```
:
/Users/carlo/TdP2024/test_db/pythonProject/.venv/bin/python /Users/carlo/TdP2024/test_db/python
(1, 'prova carico', 99.9)
[(2, 'prova latenza', 1.0), (3, 'test supporto', 3.4), (4, 'test campo', 3.4)]
```

# 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)` ↗  
La conversione dei tipi di dato è automatica da MySQL a Python;  
Se non volessi questa conversione, dovrò mettere `(raw=True)` nella  
definizione del cursore!
- You can check the read type using the Python `type()` function

# DATA ACCESS OBJECT (DAO)

Data  
Access  
Object  
Design Pattern

# 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?

• Non bisogna mischiare la gestione del database con la logica dell'applicazione!

DAO ≈ MVC

# 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
  - Avrò dei package con delle classi opposte che utilizzerò x gestire l'interfacciamento col database.

# 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

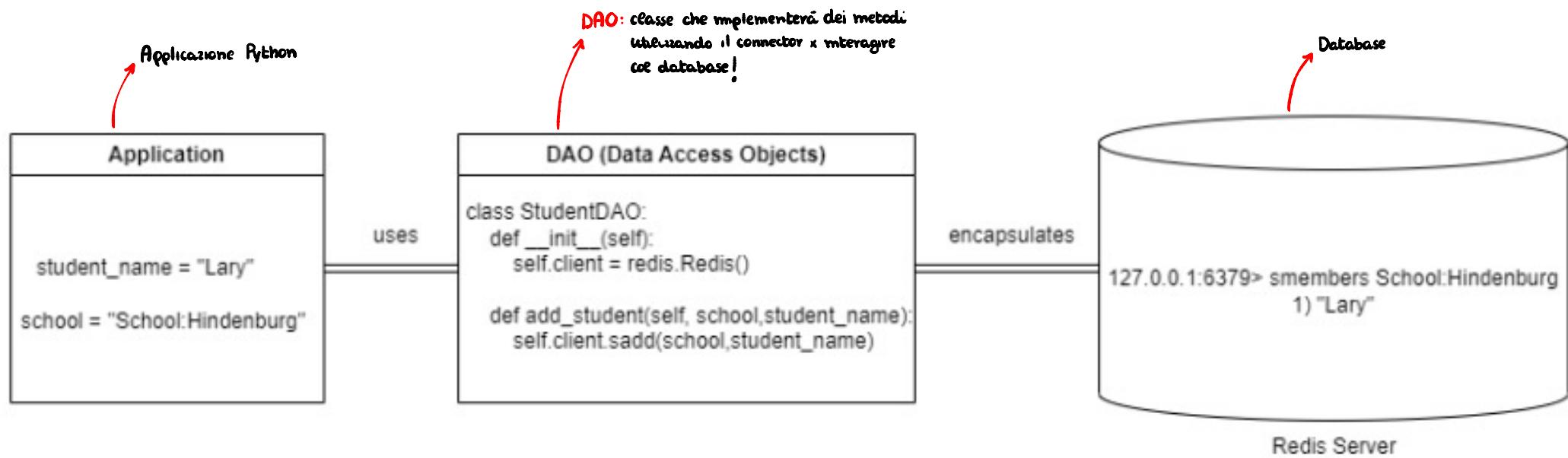


Image source <https://www.analyticsvidhya.com/blog/2023/02/what-are-data-access-object-and-data-transfer-object-in-python/>

# Data Access Object (DAO) – 1/2

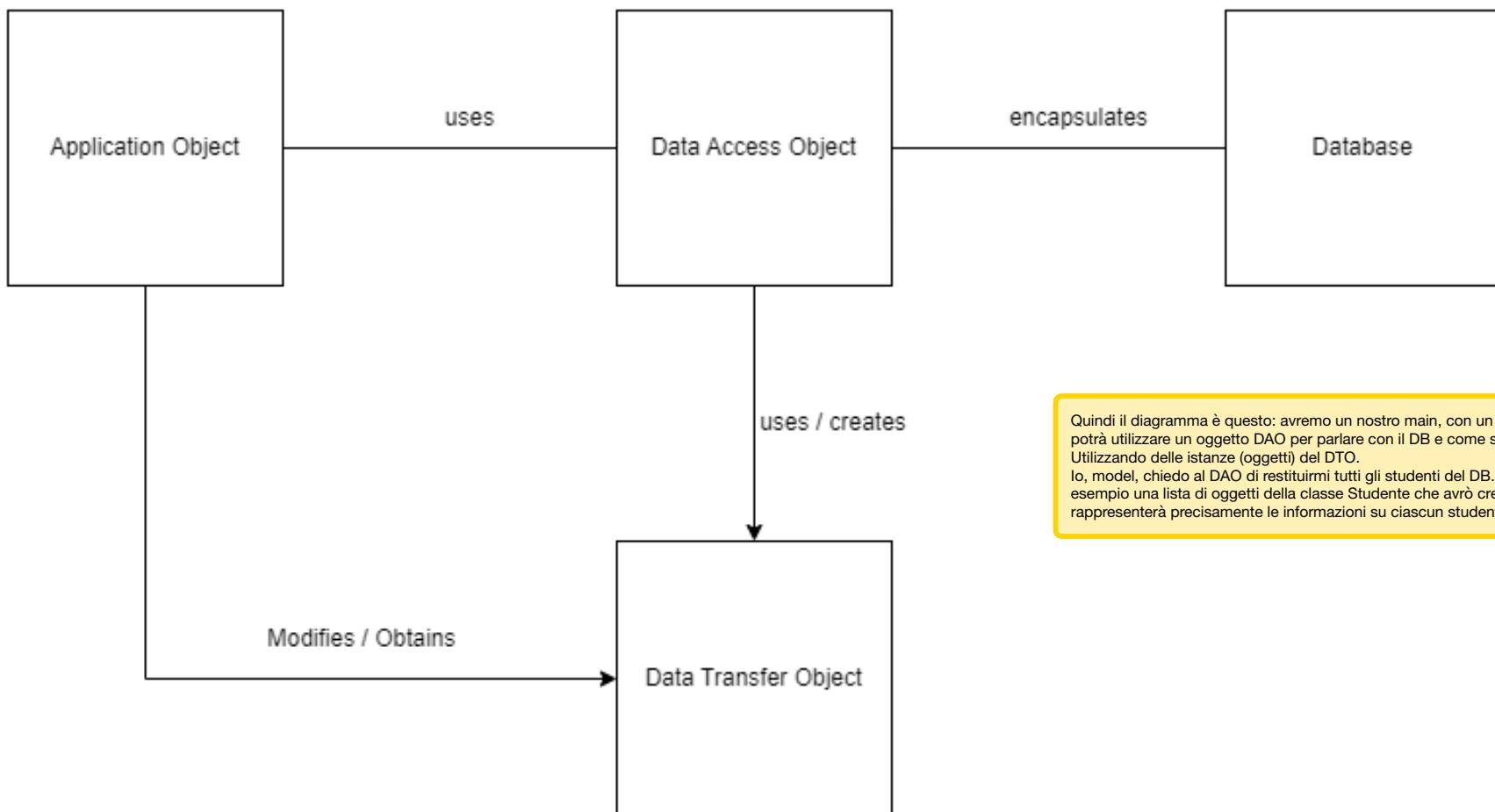
- «**Client**» classes:
    - Application code that needs to access the database
    - Ignorant of database details (connection, queries, schema, ...)
  - «**DAO**» classes:
    - Encapsulate all database access code ([mysql-connector-python](#))
    - The only ones that will ever contact the database
    - Ignorant of the goal of the Client
- Il client non vede mai le DB!*
- Il client richiede dei dati... sarà poi il DAO a effettuare le query al DB e ottenere quei dati e fornirli al client!*
- Solo le classi dei DAO interagiscono col DB!*

# Data Access Object (DAO) – 2/2

Le classi DAO encapsulano tutto il codice che abbiamo visto con mysql-connector per parlare con il database. Quindi connettersi, eseguire query, leggere i risultati e distribuire i risultati. Questo nel nostro codice deve essere l'unico punto di contatto con il database. Oltre alla connessione al database ci servirà però un'altra cosa... perchè noi con il DB vogliamo prendere dei dati. Come li modelliamo questi dati? Questa parte prevede di utilizzare i Transfer Objects o Data Transfer Object (DTO) che sono delle classi modellate per rappresentare i dati contenuti all'interno delle tabelle. Quindi, se noi abbiamo una tabella Users con colonne "id" e "nome", quello che vorremo fare è quindi creare una classe User, ad esempio, che avrà due attributi, un id e un nome, proprio per rispecchiare la struttura della tabella. Queste DTO classes sono delle classi normalissime, e quindi per creare possiamo utilizzare tranquillamente il decoratore `@dataclass`, mettendo gli attributi che ci servono, i metodi che ci servono (`__str__`, `__eq__`, `__hash__` ecc.). Quindi il DTO è semplicemente una classe di dato che utilizzeremo per memorizzare le informazioni degli oggetti presi dalla tabella.

- 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



Quindi il diagramma è questo: avremo un nostro main, con un model e quant'altro. Il model potrà utilizzare un oggetto DAO per parlare con il DB e come si comunicano i dati? Utilizzando delle istanze (oggetti) del DTO. Io, model, chiedo al DAO di restituirmi tutti gli studenti del DB. Ok, il DAO mi restituirà ad esempio una lista di oggetti della classe Studente che avrò creato ad hoc nel DTO e che rappresenterà precisamente le informazioni su ciascun studente contenute nella tabella.

# DAO: application example

```
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)
```

La classe DTO si occupa di creare gli oggetti Studente con gli stessi attributi presenti poi nella tabella!

il DAO si occupa di interagire col DB aggiungendo gli oggetti Studente DTO al DB!

For example a @dataclass

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

# 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

Il DAO non ha uno stato: non deve avere dei dati/informazioni al suo interno, ma si occupa solamente di fornire i metodi per interagire con il DB.  
Tipicamente, se il DB è semplice, possiamo avere un unico DAO che “parla” con tutte le tabelle; se i DB sono più complessi, può aver senso avere un DAO per ogni tabella. Ogni DAO dovrà poi utilizzare una o più classi di dati DTO, perchè? Perchè esempio, dopo aver letto gli studenti dalla tabella studenti, a quel punto dovrò utilizzare la classe DTO per restituire una lista di oggetti Studente.  
Quello che faremo noi con il DAO al 99% delle volte sono metodi che restituiscono al modello collezioni di oggetti DTO.

# DAO: example

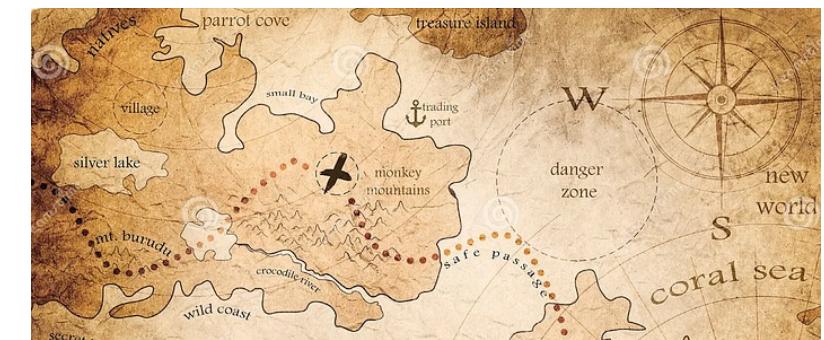
```
class studentDAO:  
    def __init__(self):  
        #possibly a state to keep information about a connection  
        # we will see this with pooling  
  
    def get_methods(self):  
        try:  
            cnx = mysql.connector.connect(database='student')  
        except mysql.connector.Error 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
```

• Nell'esempio a de viene mostrato come il **DAO** fa le query al database ottenendo come risultato una lista di dizionario (`dictionary = True`). Al model però non andrò a restituire una lista di dizionari...

Faccio un ciclo e, A diconario della lista (`@student`) mi vado a creare un oggetto **studentDTO** tramite appunto la classe **DTO**; questa classe avrà come attributi gli attributi della tabella del DB degli studenti, con cui cui il DAO sta interagendo!

Al Model restituirò dunque una lista di oggetti **studentDTO**!

# OBJECT-RELATIONAL MAPPING

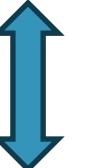


# Object Relational Mapping (ORM)

L'Object Relation Mapping è un pattern che cerca di stabilire e darci un modo per mappare le informazioni da un database a degli oggetti, che possiamo utilizzare nella programmazione a oggetti. Abbiamo visto che nei database le informazioni sono espresse come tabelle, e le tabelle possono avere relazione tra di loro. In Python, non gestiamo tabelle/entità, ma gestiamo oggetti. In parte abbiamo visto che per gestire questa cosa dovremo utilizzare le classi DTO. Con le classi DTO, gli attributi degli oggetti erano gli stessi delle colonne della tabella corrispondente. E usiamo questi oggetti per trasferire le informazioni dal database al programma Python e viceversa. I database però hanno un qualcosa in più, che sono le relazioni... per trattare le relazioni tra le tabelle del database utilizzeremo l'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

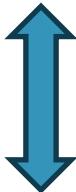
# Object Relational Mapping (ORM)

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

## REGOLE CHE DOBBIAMO SEGUIRE PER IL PATTERN ORM:

- Dobbiamo creare una classe (utilizzando il decoratore @dataclass) DTO per ogni tabella principale presente nel database. Gli attributi delle classi saranno le colonne della tabella principale corrispondente del database. Questa cosa vale per tutte le colonne, TRANNE per le colonne che sono Foreign Keys della tabella, che invece non andranno indicate come attributi della classe DTO corrispondente.

Nel database ci sono però anche tabelle "non principali" che nascono dalle relazioni tra due tabelle diverse (ad esempio in presenza di due relazione (1,N) tra due entità). Per queste tabelle che non sono la rappresentazione di dati, ma che derivano dalla relazione di tabelle diverse (per cui ho creato invece delle DTO), non dovrò creare alcuna classe DTO.

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

Dato il nome della tabella, il nome della classe DTO  
dovrà essere = al nome della classe, al singolare.  
Voti → Voto

Le foreign Keys della tabella principale  
non andrà indicata fra gli attributi  
deve classe DTO corrispondente!

# Tables → data class ORM rules

4. 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.
5. Define `__eq__()` and `__hash__()` using the **exact** set of fields that compose the primary key of the table

Dopo aver creato una classe DTO per ogni tabella principale del DB, potrò dotarla di getter e setter per gli attributi di cui ne ho bisogno, ed è importante dotarle dei metodi `__eq__()` e `__hash__()` -> nell'implementare questi due metodi, dovrò utilizzare SOLAMENTE la CHIAVE PRIMARIA!

# 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!

Per le **RELAZIONI** tra tabelle diverse cosa facciamo? —> NEXT SLIDES!

# 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

- Quando abbiamo relazioni fra tabelle di cardinalità **(1,1)**, non creo alcuna nuova classe, ma fra gli attributi delle classi DTO che ho creato potrò:
  - aggiungere come ulteriore attributo la **FK** che fa riferimento alla tabella con cui c'è la relazione;
  - aggiungere come ulteriore attributo direttamente un oggetto della classe DTO corrispondente alla tabella principale con cui ho la relazione, come nell'esempio.

## PERSONA

-----  
codice\_fiscale (PK)  
fk\_studente

@dataclass  
class Persone:

    studente: Studente  
    matricola: int

  
Metto come attributo della classe **Studente** un oggetto **Persone**, dato che ho la relazione **(1,1)** fra studente e persona!

# 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]

- Nelle relazioni di tipo **(1,N)**, per la tabella che ha cardinalità massima **1** (nell'esempio, uno Studente può abitare in un'unica città) posso usare come attributo di Studente l'oggetto della classe DTO **Citta**.

Per le tabelle che hanno relazione max **N** con un'altra tabella, (in una Città posso avere N studenti), dovrò utilizzare come loro attributo una **COLLEZIONE** di oggetti della classe DTO con cui **è** la relazione **N**!

↳ Ad esempio, una lista!

# 1:N relationship

STUDENTE

matricola (PK)

fk\_citta\_residenza

@dataclass

class **Studente**:

**cittaResidenza**: Citta

CITTA

cod\_citta (PK)

nome\_citta

In SQL, there is no «explicit»  
Citta->Studente foreign key.

The same FK is used to  
navigate the relationship in  
both directions.

@dataclass

class **Citta**:

**studentiResidenti**: list[Studente]

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

# N:M relationship

ARTICLE

-----  
id\_article (PK)  
Article data...

@dataclass

class Article:

creators: set[Creator]

AUTHORSHIP

-----  
id\_article (FK,PK\*)  
id\_creator (FK,PK\*)  
id\_authorship (PK#)

@dataclass  
class Creator:

articles: set[Article]

- Nelle relazioni **(N,N)**, in SQL avrò una tabella apposita in cui ogni istanza sarà una relazione fra le due tabelle; in Python, ∀ classe DTO che ha una relazione **N** con un'altra classe DTO, dovrò utilizzare come suo attributo una **COLLEZIONE** di oggetti dell'altra classe DTO con cui **È** la relazione!

↳ **N.B.**: Per popolare le collezioni che rappresentano la relazione a **N** della mia classe DTO con un'altra classe DTO, dovrò andare a leggere (con un **SELECT**) tutte le informazioni sulle relazioni esistenti della tabella del database che contiene queste relazioni, e x cui non dovrò creare una classe in Python!

||  
Nee! esempio, Autorship!

**N.B.**: Qualora la 3ª tabella che rappresenta la relazione fra le due tabelle principali avesse altri attributi oltre alle FK/PK, potrebbe essere sensato rappresentarla con una classe, come anche le tabelle principali, perché altrimenti potrei avere una perdita d'informazioni. Nel compilare la terza classe, seguirò le stesse regole del pattern ORML! Le FK andranno nuovamente indicate a seconda del tipo di relazione che sussiste fra le tabelle (**1,1**, **1,N**, **N,M**)!

# N:M relationship

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

ARTICLE

-----  
id\_article (PK)  
Article data...

@dataclass

class Article:

creators: set[Creator]

AUTHORSHIP

-----  
id\_article (FK,PK\*)  
id\_creator (FK,PK\*)  
id\_authorship (PK#)

CREATOR

-----  
id\_creator (PK)  
Creator data...

@dataclass

class Creator:

articles: set[Article]

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

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

Quando dobbiamo gestire le relazioni tra tabelle diverse abbiamo visto che ci sono due opzioni: utilizzare come attributo della nostra classe la PK (o la lista di PK) dell'oggetto della classe con cui sussiste la relazione, oppure posso usare come attributo della nostra classe direttamente l'oggetto (o lista di oggetti) della classe con cui sussiste la relazione. Ci sono vantaggi e svantaggi per i due metodi.

Lo storage della chiave è molto semplice da fare, ma se volessi usare l'oggetto dovrò fare una query aggiuntiva per ottenerlo dal DAO.  
Lo storage dell'oggetto è più difficile da inizializzare, ma poi quando devo utilizzarlo ce l'ho già e non devo più stare a fare query aggiuntive.

## `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

Quando dobbiamo gestire relazioni tra tabelle, utilizziamo una regola, chiamata **LAZY INITIALIZATION**: utilizziamo come attributo della nostra classe l'**OGGETTO** (o collezione di OGGETTI) della classe con cui abbiamo la relazione, però non lo inizializziamo subito quando creiamo la nostra classe. Lo definiamo inizialmente come un **None** o come una **lista vuota**. Praticamente andiamo a dare un valore di default all’oggetto nel costruttore. Se dovessi avere bisogno effettivamente dell’oggetto (o della lista di oggetti), lo andrò a creare! Come? Facendo una query con il DAO! Per fare ciò, nella mia classe rappresentante la tabella e avente come attributo l’oggetto, o la lista di oggetti della classe con cui ho la relazione, andrò a scrivere un metodo `get_oggetto`, che, se l’oggetto che devo restituire è pari a `None` (o a una lista vuota), allora crea l’oggetto (oppure popola la lista) e lo restituisce. Altrimenti, se è già stato creato l’oggetto (o popolata la lista), ritorno direttamente l’oggetto (o la lista stessa) che è attributo della mia classe!

# Identity problem

Un'altra soluzione che può essere utilizzata è la seguente: se ci sono degli oggetti che dobbiamo leggere MOLTE volte nel nostro programma, non ha senso che li leggo dal database tutte le volte. La cosa più sensata sarebbe che me li leggo una volta e li salvo in una mappa, dove la chiave della mappa è la chiave primaria dell'oggetto, e il valore è l'oggetto stesso! Questo si chiama **IDENTITY MAP**.

- 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)
```

Se uso questa query molte volte, quello che vado a fare è creare  
molte copie dello stesso oggetto varie volte. ↴ INEFFICIENTE!

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

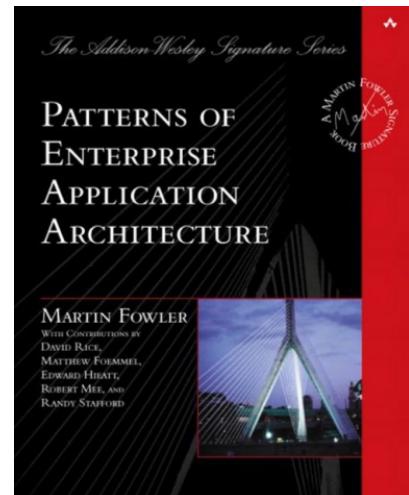
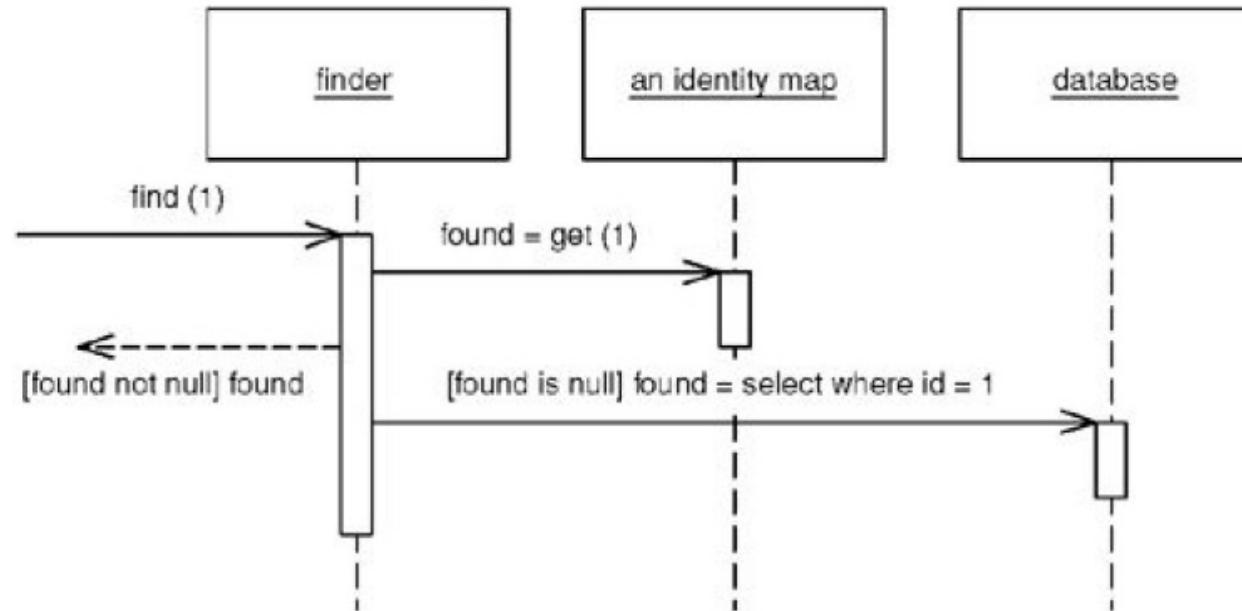
If the same Creator is author of many articles, a we would repeatedly query the database to create new objects (with identical information)

# 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

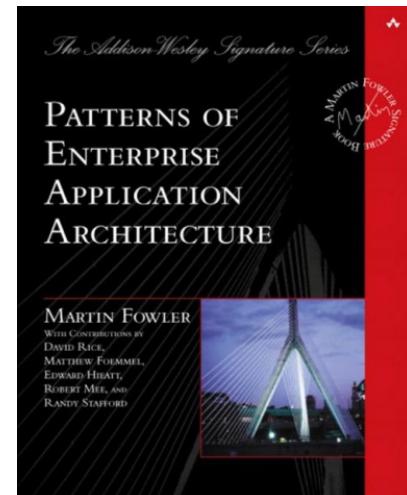
- Ensures that each object gets loaded only once, by keeping every loaded object in a map
- Looks up objects using the map when referring to them.



# 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

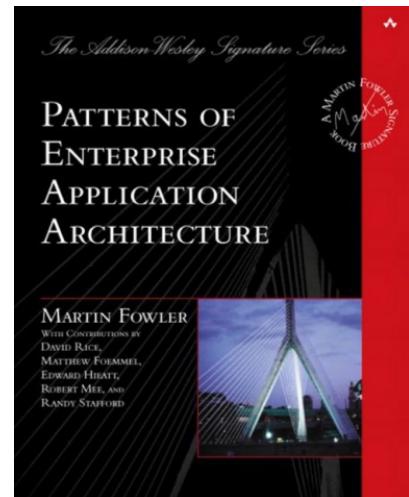
Quindi per evitare di creare TANTE volte lo stesso oggetto inutilmente, quello che posso fare è creare una IDENTITY MAP, che non è altro che un dizionario, dove la chiave è la chiave primaria della classe e il valore è l'oggetto stesso.  
Se mi è utile, dovrò creare una IDENTITY MAP per ogni tabella del database.



# Using the Identity Map

La IDENTITY MAP va creata per ogni tabella del database.  
Le identity map vanno salvate come attributi del Model, e ogni volta faccio una query per ottenere un oggetto, posso fare un check per vedere se l'oggetto è già presente nella identity map (posso farlo facilmente dato che le keys della identity map solo le PK della classe).  
Questa verifica posso farla prima di fare la query, oppure posso passare la identity map ai metodi della DAO ogni qualvolta effettuo una query e, prima di farlo, faccio un check sulla IM: se l'oggetto è già presente nella mappa, lo ritorno. Altrimenti lo creo prendendo le informazioni dal database, lo metto all'interno della identity map, e lo restituisco. Se in futuro avrò bisogno nuovamente dello stesso oggetto, non dovrò più crearlo da capo ma lo restituirò direttamente dalla 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)

**SQLAlchemy**

**django**

**PELLE**

**Pony** Object-Relational  
Mapper

# CONNECTION POOLING



# Connection pooling

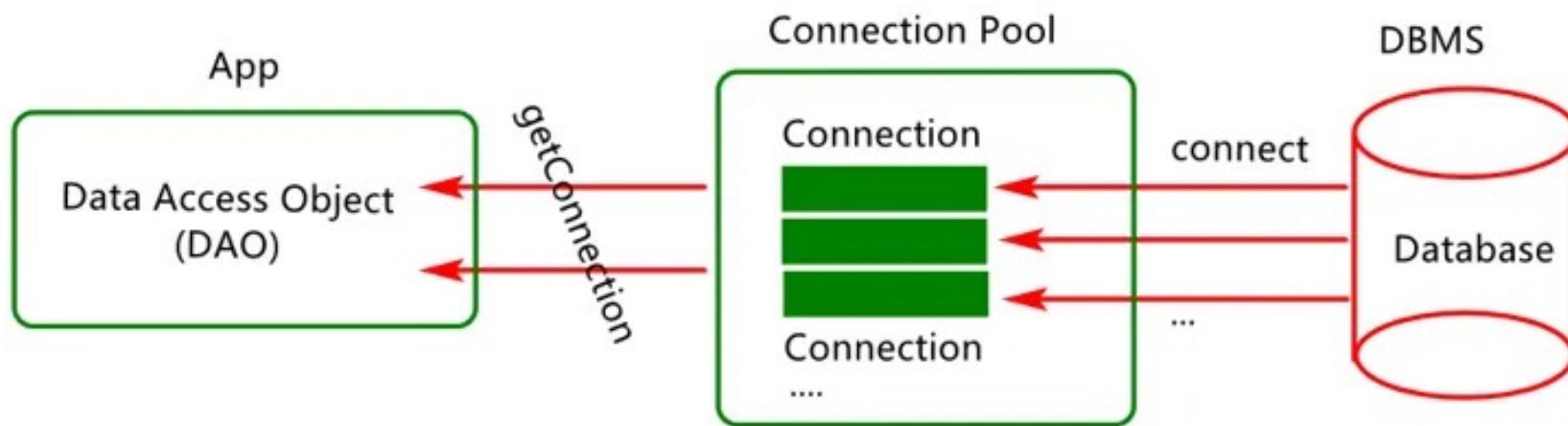
• La **connection pool** serve principalmente x risparmiare sul tempo di connessione al database!

Richiede molto + tempo la connessione al database piuttosto che l'esecuzione delle queries!

- 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

Una connection pool è un insieme di connessioni che vengono aperte contemporaneamente e vengono mantenute nell'oggetto Connection Pool. Quando voglio connettermi al database, i metodi DAO “prendono in prestito” un oggetto Connection dalla Connection Pool, che sarà già una connessione aperta e quindi non perderò ulteriore tempo nell'aprire una nuova connessione. Quando termino di utilizzare la connessione nel metodo DAO, la restituisco alla Pool (e non viene chiusa).

# Connection Pool conceptual model



# Connection pooling with mysql-connector

La libreria `my-sql-connector` implementa direttamente il **pooling** di connessione!

- 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.

• `size`: numero di connessioni che voglio aprire contemporaneamente nella pool  
• `name`: nome della Connection Pool

Di default, noi utilizzeremo una size pari a 3!

# Creating a pool

Metodo che dobbiamo usare x creare un pool di connessioni → all'esame ci verrà già fornito nel DB\_connect()!

- A livello di codice, quello che cambia nella connessione è l'aggiunta di `.pooling.MySQLConnectionPool` con il passaggio di due ulteriori parametri: `• pool-name = "mypool"`  
`• pool-size = 3`

```
cnxpool = mysql.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

• Dopo aver creato il pool di connessioni, quando voglio fare una query prenderò in prestito una connessione dalla pool con `.get_connection()` chiamato sull'oggetto Connection Pool; al termine delle operazioni, chiamerò `.close()` sulla connessione, che NON chiuderà la connessione ma la **restituirà** al pool di connessioni!

```
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 |
| <del>Non</del> -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://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.](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/>