SQL for applications

Call Level Interface (CLI)

Call Level Interface

- Requests are sent to the DBMS by means of functions offered by the guest language
  - solution based on predefined interfaces
    - API, Application Programming Interface
  - SQL instructions are passed as parameters of the functions of the guest language
  - there is no concept of precompilation
- The guest program directly contains the calls made to the functions offered by the API

Call Level Interface

- There exist many different solutions of type Call Level Interface (CLI)
  - standard SQL/CLI
  - ODBC (Open DataBase Connectivity)
    - proprietary Microsoft solution for SQL/CLI
  - JDBC (Java Database Connectivity)
    - Solution for the Java environment
  - OLE DB
  - ADO
  - ADO.NET

Usage

- Regardless of the CLI solution adopted, there is a common structure in the way they interact with the DBMS
  - opening the connection to the DBMS
  - executing SQL instructions
  - closing the connection

DBMS interaction

1. Call an API primitive to create a connection to the DBMS

DBMS interaction

1. Call an API primitive to create a connection to the DBMS
2. Send an SQL instruction on the connection

DBMS interaction

1. Call an API primitive to create a connection to the DBMS
2. Send an SQL instruction on the connection
3. Receive a result in response of the sent instruction
   - when using SELECT, result is a set ot tuples

DBMS interaction

1. Call an API primitive to create a connection to the DBMS
2. Send an SQL instruction on the connection
3. Receive a result in response of the sent instruction
   - when using SELECT, result is a set ot tuples
4. Process the obtained result
   - there are specific functions to read the result

DBMS interaction

1. Call an API primitive to create a connection to the DBMS
2. Send an SQL instruction on the connection
3. Receive a result in response of the sent instruction
   - when using SELECT, result is a set ot tuples
4. Process the obtained result
   - there are specific functions to read the result
5. Close the connection once the working session is over

DBMS interaction

- ODBC (Open DataBase Connectivity)
  - Standard method to access a database
  - Goal: make the access protocol independent of the kind of database used
  - Python offers the developer a library to access a database through ODBC
- Access methods tailored for a specific DBMS
  - MySQL, Postgres, Microsoft SQL server, …
  - Python offers the developer specific libraries for most DBMS

SQL for applications

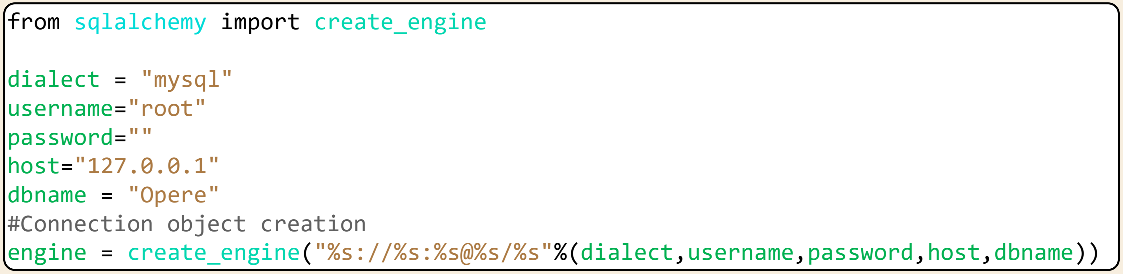
SQLAlchemy functions for Flask

MySQLi extension

- SQLAlchemy is a Python library that allows to interact with databases in an efficient way
- Supported functionalities
  - DB connection
  - Data reading and acquisition
  - Support for stored procedures, multiple queries and transactions

Creating a connection

- Call the create_engine() function
  - Starting point of applications using SQLAlchemy, it allows to specify the connection details
- It requires five parameters
  - dialect: name of the language that will be used in the connection
  - username: name of the user in the database
  - password: password of the user
  - host: name of the machine that hosts the DBMS
  - dbname: name of the DB
- It returns a connection identifier



```
from sqlalchemy import create_engine

dialect = "mysql"
username="root"
password=""
host="127.0.0.1"
dbname = "Opere"
#Connection object creation
engine = create_engine("%s://%s:%s@%s/%s"%(dialect,username,password,host,dbname))
```

Connection to the DB

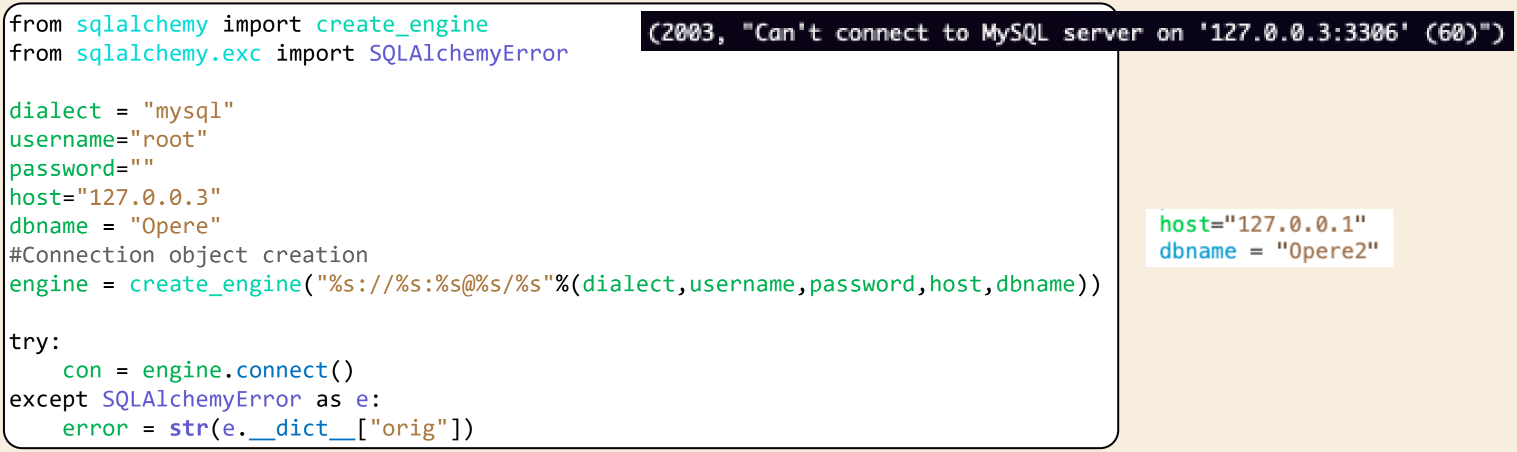
- Call the connect() function
  - When invoked, SQLAlchemy creates the connection to the DB
  - It uses the connection identifier returned by create_engine()

- It returns a connection identifier
  - If successful, it returns an active connection
  - If unsuccessful, it raises an **exception**

```
#Connection object creation
con = engine.connect()
```

Errors handling

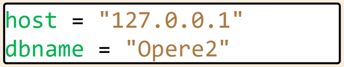
- Example including the handling of possible connection errors
  - Try: instructions to be always executed
  - Except: instructions to be executed only in case of **exceptions** raised during the execution of instructions inside the try
  - SQLAlchemyError: allows to obtain a string containing the error to be visualized



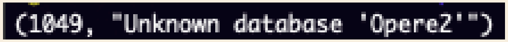
```
from sqlalchemy import create_engine
from sqlalchemy.exc import SQLAlchemyError

dialect = "mysql"
username="root"
password=""
host="127.0.0.3"
dbname = "Opere"
#Connection object creation
engine = create_engine("%s://%s:%s@%s/%s"%(dialect,username,password,host,dbname))

try:
    con = engine.connect()
except SQLAlchemyError as e:
    error = str(e.__dict__["orig"])
```

```
(2003, "Can't connect to MySQL server on '127.0.0.3:3306' (60)")
```

```
host="127.0.0.1"
dbname = "Opere2"
```



```
host = "127.0.0.1"
dbname = "Opere2"
```



```
(1049, "Unknown database 'Opere2'")
```

Closing a connection

- Must be executed when it’s not needed to interact with the DBMS anymore
  - It closes the connection to the DBMS and releases the corresponding resources
- Call the close() function
  - It uses the connection identifier returned by the connect() function

```
#Close the DB connection
con.close()
```

Execution of SQL instructions

- Immediate execution
  - The server compiles and immediately execute the received SQL instruction
- "Prepared" execution – **[Not easy with SQLAlchemy]**
  - The SQL instruction
    - Is compiled (prepared) once, and its execution plan is memorized by the DBMS
    - Is executed many times during the session
  - Useful when the same SQL instruction has to be executed many times in the same working session
    - only the value of some parameters changes

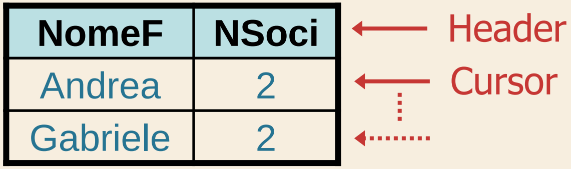
Immediate execution

- Call the execute() function
  - It uses the connection identifier returned by the connect() function
  - It requires as parameter the SQL query to be executed, in string format
  - If successful, it returns the result of the query, else it raises an exception
- Example:

```
#QUERY SQL
query = "SELECT autore.cognome, opera.nome\
         FROM autore, opera\
         WHERE autore.coda = opera.autore"

result = con.execute(query)
```

Reading the result, SQLAlchemy

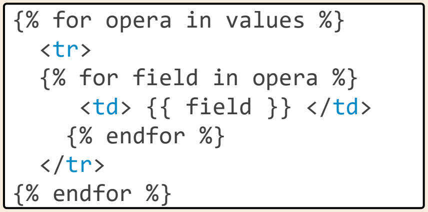
- The result of the execute() function is stored in a variable of type “cursor”
  - A special variable, that contain the result of the query
  - It’s possible to retrieve the header of a table using the keys() function on the result
- Reading the result is done row by row by means of the cursor



NomeF	NSoci
Andrea	2
Gabriele	2

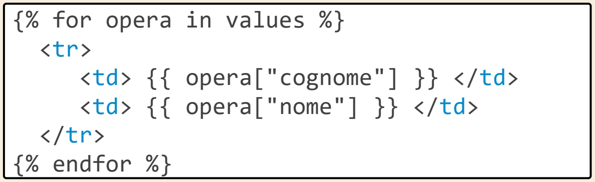
Reading the result, Jinja2

- The result is passed to Jinja2 for visualization as an array made of rows
  - It's possible to iterate on rows as if they were arrays
- Each row is coded as **a tuple** of values representing the attributes requested in the SELECT
  - It's possible to read tuple as
    - arrays



```
{% for opera in values %}
  <tr>
  {% for field in opera %}
     <td> {{ field }} </td>
    {% endfor %}
  </tr>
{% endfor %}
```

    - dictionaries



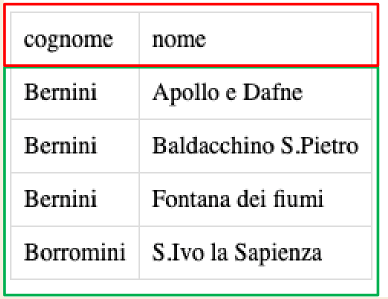
```
{% for opera in values %}
  <tr>
     <td> {{ opera["cognome"] }} </td>
     <td> {{ opera["nome"] }} </td>
  </tr>
{% endfor %}
```

Visualizing the result

- It’s possible to pass to Jinja2 different arrays to specify the **header** of the table and its **content**

```
try:
    con = engine.connect()
    query = "SELECT autore.cognome, opera.nome\
             FROM autore, opera\
             WHERE autore.coda = opera.autore"

     result = con.execute(query)
     header = result.keys()

     return render_template("opere.html", annoDa=annoDa, annoA=annoA,citta=citta, header= header, values=result)
except SQLAlchemyError as e:
    error = str(e.__dict__["orig"])
   return render_template("errore.html", error_message=error)
```

```
<table>
  <tr>
  {% for field in header %}
     <td> {{ field }} </td>
  {% endfor %}
</tr>
{% for opera in values %}
  <tr>
  {% for field in opera %}
     <td> {{ field }} </td>
    {% endfor %}
  </tr>
</table>
```



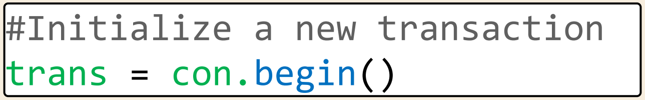
cognome	nome
Bernini	Apollo e Dafne
Bernini	Baldacchino S.Pietro
Bernini	Fontana dei fiumi
Borromini	S.Ivo la Sapienza

Transactions

- Connections are implicitly created in auto-commit mode
  - After the successful execution of each SQL instruction, a commit is automaticaly executed
- Whenever it's necessary to commit exclusively after having succesfully executed a sequence of SQL instructions
  - The commit has to be managed in a non-automated way
  - A single commit is executed once every instruction has been performed

Managing transactions

- Call the begin() function
  - When invoked, SQLAlchemy initializes a transaction and disables the auto-commit
  - If successful, it returns an active transaction
  - If unsuccessful, it raises an **exception**
  - It uses the connection identifier returned by the connect() function



```
#Initialize a new transaction
trans = con.begin()
```

Managing transactions

- If the auto-commit is disabled, commit and rollback operations must be explicitly requested
  - They use the transaction identifier returned by the begin() function
- commit ()
  - Executes the commit of the current transaction
  - If unsuccessful, it raises an exception

```
#Commits the operations
trans.commit()
```

- rollback ()
  - Executes the rollback of the current transaction
  - If unsuccessful, it raises an exception

```
#Rollback the operations
trans.rollback()
```

Managing transactions

- If the auto-commit is disabled, commit and rollback operations must be explicitly requested
  - They use the transaction identifier returned by the begin() function
- Using the **with** construct, SQLAlchemy automatically handles the commit and rollback
  - Executes the commit if successful
  - Executes the rollback if unsuccessful, and raises an exception

```
#Initialize a transaction and Commit or Rollback
with con.begin() as trans:
  #... SQL and SQLAlchemy code ...
```