Python - Object-relational mapping

Python Object-Relational Mapping (ORM) is a technique used to bridge the gap between the object-oriented programming paradigm and relational databases. It allows developers to work with databases using high-level, object-oriented constructs rather than dealing with the low-level details of SQL queries and database schema directly.

There are several popular libraries for ORM in Python, two of which are MySQLdb and SQLAlchemy:

1. MySQLdb: MySQLdb is a Python interface to MySQL. It provides a low-level API for interacting with MySQL databases directly from Python code. While it's powerful, it doesn't provide the same level of abstraction and convenience as higher-level ORM libraries.
2. SQLAlchemy: SQLAlchemy is a powerful and flexible ORM library for Python. It provides a high-level API for working with databases, allowing developers to interact with databases using Python objects and SQL expressions. SQLAlchemy supports multiple database backends, including MySQL, PostgreSQL, SQLite, and more.

MySQLdb

Main functions and concepts you should know when working with MySQLdb:

- connect: This function is used to establish a connection to a MySQL database. It takes connection parameters such as host, user, password, and database name, and returns a connection object that can be used to execute SQL queries and manage transactions.

Example:

```
import MySQLdb

# Establish a connection to the MySQL database
connection = MySQLdb.connect(host="localhost", user="username",
password="password", database="dbname")
```

- Cursor: This method is used to create a cursor object, which is used to execute SQL queries and fetch results from the database. The cursor object provides methods for

executing queries, fetching rows, and managing transactions.

```
cursor = connection.cursor()
```

- Execute: This method is used to execute SQL queries. It takes an SQL statement as an argument and executes it on the database server.

```
cursor.execute("SELECT * FROM tablename")
```

The execute method can accept a second argument. This 2d argument must be a sequence like a list or a tuple,
even if there's only one item. If only one item you need to provide it as a singleton tuple (tuple with only one item). To define a singleton tuple, you need to include a trailing comma after the item otherwise Python will interpret it as a string, not a tuple.

Ex:

```
cursor.execute(
    "SELECT states WHERE states.name = %s ORDER BY states.id", (sys.argv[4],))
```

%s: In Python, the %s symbol is used as a placeholder for string substitutions. It's often used in SQL queries to substitute values into the query string.

When you're using the cursor.execute() method in the MySQLdb module, you can use %s as a placeholder in the SQL query, and then provide a tuple of values as the second argument to the execute() method. The values in this tuple will replace the %s placeholders in the SQL query.

(sys.argv[4],): This is a tuple containing the parameter values to be substituted into the SQL query. sys.argv[4] retrieves the fifth command-line argument passed to the Python script and it is passed as the parameter value for the %s placeholder in the SQL query. Note the comma , at the end of the tuple; it's required to create a single-element tuple in Python.

- Fetchone: This method is used to fetch the next row of a query result set. It returns a single row as a tuple or None if there are no more rows.

```
row = cursor.fetchone()
```

- Fetchall: This method is used to fetch all rows of a query result set. It returns a list of tuples, where each tuple represents a row of the result set.

```
rows = cursor.fetchall()
```

- Commit: This method is used to commit the current transaction to the database. It saves any changes made within the current transaction to the database.

```
connection.commit()
```

- Rollback: This method is used to rollback the current transaction. It discards any changes made within the current transaction and restores the database to its state before the transaction began.

```
connection.rollback()
```

- Close: This method is used to close the connection to the database. It releases any resources associated with the connection and ends the database connection.

```
connection.close()
```

- LIKE, LIKE BINARY: The LIKE operator is used in a WHERE clause to search for a specified pattern in a column. It's commonly used to perform partial string matching. The pattern can include wildcard characters like % (matches any string of any length) and _ (matches any single character).

```
SELECT * FROM users WHERE name LIKE 'A%'; # Finds all users whose name starts with 'A'
```

To make the search case-sensitive, use the binary operator:

```
# lists all states with a name starting with N (upper N) using the % wildcard character
cursor = connection.cursor()
    cursor.execute("SELECT * FROM states WHERE states.name LIKE BINARY 'N%' ORDER BY states.id")
    rows = cursor.fetchall()
    for row in rows:
```

```
        print(row)
    connection.close()
```

- Usage of the % wildcard character (in combination with the LIKE operator):

%A: This pattern matches any string that ends with 'A'. The % wildcard character matches zero or more characters, so %A means any string followed by 'A'
A%: This pattern matches any string that starts with 'A'. Here, A% means 'A' followed by any number of characters.
%A%: This pattern matches any string that contains 'A' anywhere in it. The % wildcard characters before and after 'A' allow for any characters before and after 'A'

Examples:

```
import MySQLdb  # Importing the MySQLdb module for interacting with MySQL
databases
import sys      # Importing the sys module to access command-line arguments

if __name__ == "__main__":
    """Including the guard clause"""

    """Establishing a connection to the MySQL database"""
    connection = MySQLdb.connect(
        user=sys.argv[1],       # MySQL username obtained from the command-line
argument
        password=sys.argv[2],   # MySQL password obtained from the command-line
argument
        database=sys.argv[3])   # MySQL database name obtained from the
command-line argument

    cursor = connection.cursor()  # Creating a cursor object to execute SQL
queries

    # Constructing the SQL query to select rows from the 'states' table based
on a given name,
    # and ordering the results by the 'id' column
    cmd = "SELECT * FROM states WHERE states.name = %s ORDER BY states.id"

    # Executing the SQL query with a parameterized value (sys.argv[4]) for the
name condition
```

```
    cursor.execute(cmd, (sys.argv[4],))

    # Fetching all rows returned by the query
    rows = cursor.fetchall()

    # Iterating over the fetched rows and printing each row
    for row in rows:
        print(row)

    # Closing the database connection
    connection.close()
```

Making this code safe from SQL injections:

SQL injection attacks occur when an attacker inserts malicious SQL code into a query via user input fields or URL parameters, exploiting vulnerabilities in the application to execute unintended SQL commands.

In the provided code, the SQL query is constructed using placeholders ("%s") for parameters, and the actual parameter values are passed separately when executing the query. This is done using the cursor.execute() method, where the second argument is a tuple containing the parameter values (sys.argv[4],).

Here's why this approach makes the code safe from SQL injection:

Parameterization: By using parameterized queries, the input values (in this case, sys.argv[4]) are treated as data rather than part of the SQL command. The database driver automatically handles escaping special characters within the parameter values, preventing them from being interpreted as SQL commands.

Automatic Escaping: The MySQLdb library automatically escapes special characters in the parameter values before substituting them into the query. This ensures that even if the parameter value contains characters with special meaning in SQL (such as quotes), they will be treated as literals rather than part of the SQL syntax.

No String Concatenation: The SQL query is not constructed by concatenating strings, which is a common practice in vulnerable code. Concatenating strings with user input directly into SQL queries leaves the application open to SQL injection attacks. In this code, the SQL query is constructed separately from the parameter values, eliminating the risk of injection.

Overall, by using parameterized queries and passing parameter values separately, this code ensures that user input is properly sanitized and prevents SQL injection attacks.

SQLAlchemy

Main functions and concepts you should know when working with SQLAlchemy:

- create_engine: This function is used to create an engine, which represents the core interface to a particular database server. It's typically the first step in setting up SQLAlchemy and connecting to a database. It takes a connection string as an argument, which specifies the details of the database to connect to.

Syntax:

```
engine = create_engine('sqlite:///example.db', echo=True)
```

- declarative_base: This function is used to create a base class for declarative class definitions. Declarative base allows you to define database tables and their mappings to Python classes in a concise and declarative manner.

Syntax:

```
Base = declarative_base()
```

- Column: This class is used to define columns within a table. Columns are defined as class attributes within the declarative class definitions. You specify the data type and other properties of the column using arguments to the Column constructor.

Syntax:

```
name = Column(String)
age = Column(Integer)
```

- relationship: This function is used to define relationships between tables. It establishes a link between two classes, indicating that instances of one class are related to instances of another class. It takes arguments that specify the target class, as well as any additional properties of the relationship.

Syntax:

```
addresses = relationship("Address", back_populates="user")
```

- sessionmaker: This function is used to create a session class. Sessions are used to interact with the database and perform CRUD operations (Create, Read, Update, Delete). The session class provides methods for adding, querying, updating, and deleting database objects.

Syntax:

```
Session = sessionmaker(bind=engine)
session = Session()
```

- add: This method is used to add objects to the session. It queues up objects to be persisted in the database during the next flush operation.

Syntax:

```
session.add(user1)
```

- update: The update() function is used to update records in a database table. It allows you to modify the values of one or more columns in existing rows that meet specific criteria.

Syntax:

```
session.query(Table).filter(<filter_criteria>).update({<column>: <new_value>})

# Example:
session.query(State).filter(State.id == 2).update({"name": "New Mexico"})
session.commit()
session.close()
```

- delete: The delete() function is used to delete records from a database table. It allows you to remove one or more rows that meet specific criteria.

Syntax:

```
session.query(Table).filter(<filter_criteria>).delete()
```

```
# Example:
# deleting state objects containing the letter a
session.query(State).filter(State.name.like('%a%')).delete()
session.commit()
session.close()
```

- commit: This method is used to commit the current transaction to the database. It persists all changes made within the session to the database.

Syntax:

```
session.commit()
```

- query: This method is used to construct and execute queries against the database. It returns a Query object that can be further refined with filter, join, and other methods.

Syntax:

```
users = session.query(User).all()
```

- close: This method is used to close the session. It releases any resources associated with the session and ends the database connection.

Syntax:

```
session.close()
```

- first: The first() function is used to retrieve the first result from a query. It's often used in combination with queries that are expected to return multiple results, but where you're only interested in the first one.

Examples:

```
# Query the database by selecting the first row
    from the State table
state = session.query(State).first()
    if state:
        print(f"{state.id}: {state.name}")
    else:
        print("Nothing")
```

```
# Query the database to print the state id if there is a match with the name
state declared on the command line
state = session.query(State).filter(
        State.name == sys.argv[4]).first()
    if state:
        print(state.id)
    else:
        print("Not found")
```

Here's a basic example of structured Python code using SQLAlchemy for object-relational mapping:

```
# Import necessary modules from SQLAlchemy
from sqlalchemy import create_engine, Column, Integer, String, ForeignKey
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, sessionmaker

# Create an engine to connect to the database
engine = create_engine('sqlite:///example.db', echo=True)

# Create a base class for declarative class definitions
Base = declarative_base()

# Define a class representing a User table
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    age = Column(Integer)

    # Relationship to another table
    addresses = relationship("Address", back_populates="user")

# Define a class representing an Address table
class Address(Base):
    __tablename__ = 'addresses'

    id = Column(Integer, primary_key=True)
```

```
    email = Column(String)
    user_id = Column(Integer, ForeignKey('users.id'))

    # Relationship to the User table
    user = relationship("User", back_populates="addresses")

# Create the tables in the database
Base.metadata.create_all(engine)

# Create a session to interact with the database
Session = sessionmaker(bind=engine)
session = Session()

# Create some user objects
user1 = User(name='Alice', age=30)
user2 = User(name='Bob', age=35)

# Add users to the session
session.add(user1)
session.add(user2)

# Commit changes to the database
session.commit()

# Query the database
users = session.query(User).all()
for user in users:
    print(user.name, user.age)

# Close the session
session.close()
```

In this example:

We define two classes, User and Address, which represent tables in the database.
We establish relationships between these tables using SQLAlchemy's relationship function.
We create an engine to connect to the database (example.db in this case, which will be a SQLite database).
We define the database schema by subclassing Base and defining the structure of the tables using SQLAlchemy's Column class.
We create tables in the database using Base.metadata.create_all(engine).

We create a session to interact with the database and perform CRUD (Create, Read, Update, Delete) operations.
We add some user objects to the session and commit changes to the database.
Finally, we query the database and print the results.