SQLite3 - Python ORM 설명서

SQLite는 경량 데이터베이스로, Python의 ORM 라이브러리인 **SQLAlchemy**를 사용하여 쉽게 관리할 수 있습니다. `db.sqlite3` 파일은 SQLite 데이터베이스 파일을 의미하며, 이를 ORM을 통해 핸들링하는 방법을 설명하겠습니다.

SQLite와 SQLAlchemy ORM을 이용한 데이터베이스 핸들링 상세 설명서

1. 환경 설정 및 설치

먼저 필요한 라이브러리를 설치해야 합니다. SQLAlchemy와 SQLite는 Python 표준 라이브러리이므로 별도의 설치 없이 사용할 수 있지만, SQLAlchemy를 설치해야 ORM 기능을 사용할 수 있습니다.

```
pip install sqlalchemy
```

2. 데이터베이스 엔진 설정

SQLAlchemy에서 데이터베이스와 상호작용하기 위해 **엔진(engine)**을 설정합니다. 엔진은 데이터베이스에 연결하기 위한 객체입니다.

```
from sqlalchemy import create_engine

# SQLite 데이터베이스 엔진 생성 (db.sqlite3 파일로 연결)
engine = create_engine('sqlite:///db.sqlite3', echo=True)
```

- `sqlite:///db.sqlite3`: `sqlite` 스키마를 사용하여 로컬의 `db.sqlite3` 파일과 연결합니다.
- `echo=True`: SQLAlchemy가 생성하는 SQL 쿼리를 출력하도록 설정하여 디버깅을 돕습니다.

3. 베이스 클래스 및 ORM 모델 정의

SQLAlchemy에서는 데이터베이스 테이블을 Python 클래스와 매핑하여 ORM 방식으로 다룰 수 있습니다. 이때 각 클래스를 테이블에 매핑하기 위해 `Base` **클래스**에서 상속받습니다.

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, Float

# 베이스 클래스 생성
Base = declarative_base()

# 테이블 정의 (클래스 정의와 컬럼 매핑)
class User(Base):
    __tablename__ = 'users'  # 테이블 이름

    id = Column(Integer, primary_key=True)  # 기본 키
    name = Column(String, nullable=False)  # 이름
```

```
    age = Column(Integer)  # 나이
    height = Column(Float)  # 키

    def __repr__(self):
        return f"<User(name={self.name}, age={self.age}, height={self.heigh
t})>"
```

- `__tablename__`: 클래스와 데이터베이스 테이블을 매핑하기 위한 테이블 이름입니다.
- **컬럼 정의**: 각 클래스 속성을 테이블의 컬럼과 매핑하며, 타입을 명시합니다 (`Integer`, `String`, `Float` 등).

4. 데이터베이스 생성 및 테이블 만들기

모델을 정의한 후, `Base.metadata.create_all()`을 사용하여 정의된 테이블을 실제 데이터베이스에 생성할 수 있습니다.

```
# 데이터베이스 테이블 생성
Base.metadata.create_all(engine)
```

이 코드는 데이터베이스에 연결된 엔진을 사용하여 모델에 정의된 테이블을 만듭니다.

5. 세션 생성 및 데이터 조작

데이터베이스에 CRUD(Create, Read, Update, Delete) 작업을 하기 위해 **세션(Session)** 객체를 사용합니다. 세션은 데이터베이스와의 상호작용을 관리합니다.

```
from sqlalchemy.orm import sessionmaker

# 세션 생성
Session = sessionmaker(bind=engine)
session = Session(
```

6. 데이터 삽입(Create)

ORM을 사용하면 SQL 없이 Python 객체를 생성하고 데이터를 삽입할 수 있습니다.

```
# 새로운 사용자 객체 생성
new_user = User(name="John Doe", age=28, height=180.5)

# 세션에 추가하고 커밋
session.add(new_user)
session.commit()
```

7. 데이터 조회(Read)

SQLAlchemy ORM을 사용하여 데이터베이스에서 데이터를 조회할 수 있습니다. `session.query()`를 사용해 테이블의 모든 데이터를 가져오거나 특정 조건을 적용할 수 있습니다.

```
# 모든 사용자 조회
users = session.query(User).all()
for user in users:
```

```
    print(user)

# 특정 조건으로 데이터 조회
john = session.query(User).filter_by(name="John Doe").first()
print(john)
```

- `query()`: 데이터베이스에서 조회할 테이블 또는 객체를 지정합니다.
- `filter_by()`: 특정 조건으로 데이터를 필터링합니다.

8. 데이터 업데이트(Update)

특정 데이터를 업데이트하려면 객체를 조회한 후 속성을 수정하고 `commit()`을 호출하여 데이터베이스에 반영합니다.

```
# 특정 사용자 조회
user_to_update = session.query(User).filter_by(name="John Doe").first()

# 데이터 수정
user_to_update.age = 29
session.commit()
```

9. 데이터 삭제(Delete)

객체를 세션에서 삭제한 후 `commit()`으로 데이터베이스에서 삭제할 수 있습니다.

```
# 특정 사용자 조회
user_to_delete = session.query(User).filter_by(name="John Doe").first()

# 데이터 삭제
session.delete(user_to_delete)
session.commit()
```

10. 종료

세션을 사용한 후에는 반드시 세션을 종료해주는 것이 좋습니다.

```
session.close()
```

SQLite와 SQLAlchemy를 사용할 때의 장점

1. **데이터베이스 독립성**: SQLAlchemy는 다양한 데이터베이스를 지원하므로, SQLite에서 MySQL, PostgreSQL 등으로 쉽게 전환할 수 있습니다.
2. **객체 지향적 데이터 조작**: 데이터베이스 작업을 객체 지향적으로 수행할 수 있어 코드 가독성과 유지보수성이 높아집니다.
3. **자동 테이블 생성**: ORM 모델 정의에 따라 자동으로 테이블을 생성하고 매핑할 수 있어 SQL을 직접 작성할 필요가 없습니다.

이 설명서를 따라 SQLite와 SQLAlchemy를 사용하면 Python 코드에서 객체 지향적으로 데이터를 관리할 수 있습니다. ORM을 사용하면 SQL 작성의 복잡성을 줄이고 코드의 일관성을 높일 수 있습니다.