

Python과 데이터베이스

Python DB API

SQLite and MySQL

Python DB API

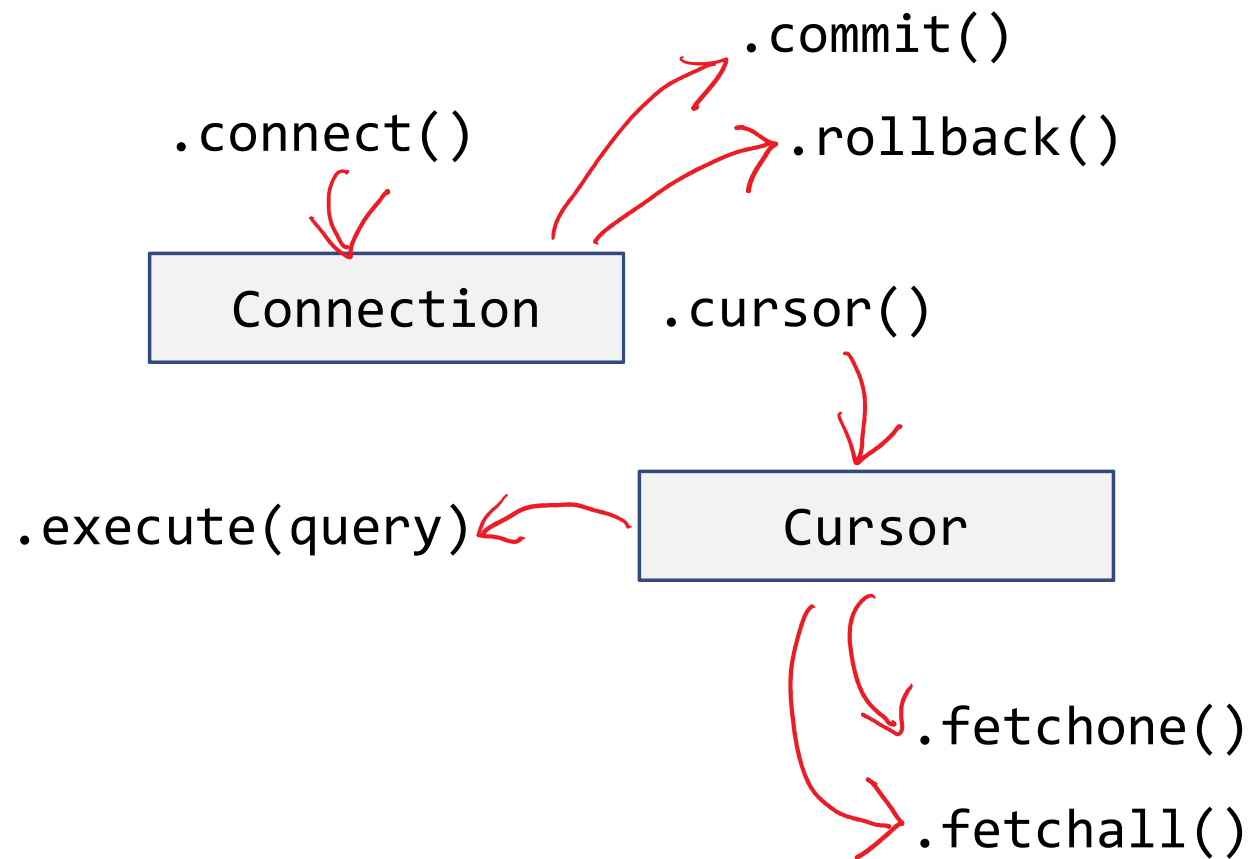
▶ Python DB API

- ▶ 여러 데이터베이스를 액세스하는 표준 **API**
 - ▶ 여러 데이터베이스 액세스 모듈에서 따르는 최소한의 **API** 인터페이스 표준
 - ▶ 현재 버전 **2.0**을 사용하며, 스펙은 **PEP 249**에 소개

▶ DBMS 시스템을 이용하기 위해서는 각각 **DB**에 상응하는 모듈을 다운 받아야 함

- ▶ 하지만 대부분 **Python DB API** 표준을 따르고 있어 동일한 방식으로 **DBMS**를 사용할 수 있음
- ▶ **MySQL**, **PostgreSQL**, **MSSQL**, **Oracle**, **Sybase**, **Informix**, **mSQL** 등 대표적 **DB** 모두 지원
- ▶ **SQLite**는 **Embedded SQL DB** 엔진으로 **Python 2.5** 이상에 기본 내장

Python DB API



SQLite

▶ SQLite

- ▶ Embedded SQL DB 엔진으로 별도의 DB 서버가 필요 없는 파일 기반 엔진
- ▶ 별도의 설치가 필요 없고, 쉽고 편리하게 사용할 수 있다는 점에서 널리 사용
- ▶ Mac OS X, Linux, Android 등에 기본으로 내장
- ▶ Windows 등, 기본 내장되어 있지 않은 OS의 경우 다음 경로에서 다운로드 받아 사용
 - ▶ <https://sqlite.org/download.html>
 - ▶ sqlite-tools-win32-x86-xxxxxxx.zip
 - ▶ SQLite Viewer for Windows : <https://sqlitebrowser.org/>

SQLite

: 데이터베이스 접속

▶ SQLite 접속과 해제

- ▶ `sqlite3 import`
- ▶ 접속 : `.connect(데이터베이스명)`
- ▶ 해제 : `.close()`

▶ SQLite3는 `connect`시 데이터베이스가 없으면 자동으로 생성

```
import sqlite3
from sqlite3 import Error

# SQLite DB 연결
conn = sqlite3.connect("test.db")

# Sqlite DB 연결 해제
conn.close()
```

SQLite

: 쿼리의 수행

▶ 쿼리의 수행

- ▶ Connection 객체의 `execute()` 메서드로 실행
- ▶ 쿼리 수행 결과로 `Cursor`를 반환

▶ 반환 받은 결과는 다음 메서드로 확인

- ▶ `fetchone` : 결과 하나를 반환
- ▶ `fetchmany` : 결과 여러 개를 반환
- ▶ `fetchall` : 전체 결과를 반환

```
conn = sqlite3.connect("./database/mysqlite.db")  
  
cursor = conn.execute("SELECT * FROM customer")  
  
for customer in cursor.fetchall():  
    print(customer)
```

▶ 수행 결과 영향을 받은 레코드의 개수는 커서의 `.rowcount` 필드로 확인

SQLite

: commit과 rollback

- ▶ 데이터가 추가(**INSERT**)되거나, 변경(**UPDATE**)되거나, 삭제(**DELETE**)되어도 실제 데이터베이스에 반영되지는 않음
 - ▶ 해당 변화 반영을 위해서는 `.commit()`
 - ▶ 해당 변화 취소를 위해서는 `.rollback()`

SQLite

: Parameterized Query

- ▶ 실제 개발에서는 **SQL** 문장을 **String** 연결로 하기보다 포맷을 만들어 두고 동적으로 컬럼 데이터 값을 집어 넣는 경우가 많음
 - ▶ Parameterized Query를 이용하면 편리
 - ▶ Anonymous Placeholder : tuple로 바인딩

```
conn.execute("SELECT * FROM customer WHERE category=?", (1,))
```

- ▶ Named Placeholder : dict로 바인딩

```
conn.execute("SELECT * FROM customer WHERE category=:cat", {"cat": 2})
```

MySQL

:pymysql

- ▶ pymysql 모듈은 MySQL 데이터베이스를 파이썬에서 사용하기 위한 모듈 중 하나
 - ▶ Python DB API 표준을 따르므로 다른 데이터베이스 모듈과 사용 방식이 거의 같다
- ▶ pymysql 모듈의 설치

```
pip install pymysql
```

- ▶ pymysql 연결

```
conn = pymysql.connect(host='localhost',  
                        user='root',  
                        password='*****',  
                        db='sakila',  
                        charset='utf8')
```

MySQL

:pymysql

▶ 기본적인 SELECT 문의 수행

```
# Connection으로부터 Cursor 생성
cursor = conn.cursor()

# SQL문 실행
sql = "SELECT * FROM actor"
cursor.execute(sql)

# Data Fetch
rows = cursor.fetchall()

# Print
for actor in rows:
    print(actor)

# Connection Close
conn.close()
```

MySQL

:pymysql

▶ Parameter Placeholder

- ▶ 동적으로 컬럼 데이터 값을 집어넣고자 할 때 **Parameter Placeholder %s**를 사용
 - ▶ 일반 문자열 포매팅에 사용되는 %s 와는 다른 것이며, 매칭될 값은 튜플로 넣어준다
 - ▶ Placeholder 문자는 컬럼 값을 대치할 때만 사용할 수 있으며 SQL 문장 다른 부분에 Placeholder를 사용할 수는 없음

```
...
cursor = conn.cursor()
sql = "SELECT * FROM actor WHERE last_name = %s or last_name = %s"
cursor.execute(sql, ("PENN", "GRANT"))
...
```

MySQL

:pymysql

▶ Dictionary Cursor

- ▶ Cursor를 획득할 때, 파라미터로 DictCursor를 지정하면 결과 셋을 사전 형태로 반환 받을 수 있다

```
...  
cursor = conn.cursor(pymysql.cursors.DictCursor)  
sql = "SELECT * FROM actor WHERE last_name = %s or last_name = %s"  
cursor.execute(sql, ("PENN", "GRANT"))  
...
```

Object Relational Mapping

with PEEWEE

ORM

▶ 객체 관계 매핑

- ▶ "객체(Object)와 관계형 데이터베이스(RDBMS)를 매핑(Mapping) 한다"

▶ OOP의 관점과 RDBMS의 관점

▶ 객체 지향 프로그래밍(Object Oriented Programming)

- ▶ 애플리케이션이 발전하면 내부 복잡성(Complexity)이 증가, 복잡성을 제거할 수 있어야 지속적인 애플리케이션 개발이 가능하며 유지보수도 가능해진다.
- ▶ 프로그래밍 개발은 기존의 절차적 프로그래밍 방식에서 객체가 중심이 된 객체 지향 프로그래밍 방식으로 발전

▶ 관계형 데이터베이스 (Relational Database)

- ▶ 데이터 중심(테이블)으로 구조화에 목적
- ▶ 일련의 정형화된 테이블로 구성된 데이터 항목들의 집합체
- ▶ 현대 OOP 프로그래밍에 적용되는 OOP의 개념들은 포함하지 않음

ORM

▶ 패러다임의 불일치

- ▶ 객체 지향으로 작성된 애플리케이션의 객체를 영구(Persistence) 저장할 때 발생
- ▶ 단순 객체는 객체 인스턴스의 상태인 속성만 저장했다가 필요할 때 불러와 복구하면 됨
- ▶ 하지만, 다른 객체를 참조하는 경우 그 객체의 상태를 함께 저장하는 것은 쉽지 않다.
- ▶ Python에서 제공하는 객체 직렬화(Pickle) 등을 이용하면 객체를 저장할 수는 있으나 객체 검색이 용이하지 않다

- ▶ 객체의 저장 및 검색을 위해 주로 관계형 데이터베이스를 사용하게 되는데 이때, 객체와 관계형 데이터베이스는 지향하는 목적이 서로 다르기 때문에 둘의 기능과 표현 방법이 다르다 (패러다임 불일치)
- ▶ 패러다임의 불일치로 인해, 객체 지향 언어로 개발된 프로그램의 입장에서 보면 객체 구조와 관계를 테이블 구조에 저장하는데 한계와 제한을 가지게 됨
- ▶ 이 문제의 극복을 위해 개발자가 중간에 객체 <-> 테이블 변환 등에 많은 노력을 소모

ORM

▶ 패러다임 불일치 문제 해소를 위한 ORM

- ▶ 컬렉션 객체 등에 객체를 저장하는 것처럼 ORM 프레임워크에 저장하면 ORM 프레임워크가 적절한 SQL 문을 생성하고 데이터베이스에 저장
- ▶ ORM 프레임워크는 객체와 테이블을 매핑하여 패러다임 불일치 문제를 개발자 대신 해결: 개발자는 객체와 테이블의 매핑 방법만 ORM 프레임워크에 알려 주면 된다
- ▶ 개발자는 객체의 관점에서 객체 모델링을 할 수 있고 관계형 데이터베이스 관리자는 데이터베이스에 맞게 모델링 하면 된다
- ▶ 데이터 중심의 관계형 데이터베이스를 사용해도 개발자는 객체 지향 애플리케이션 개발에 좀 더 집중할 수 있다

- ▶ Python의 ORM Framework 들 : SQLAlchemy, PEEWEE 등

PEEWEE

: install and import

- ▶ PEEWEE ORM을 이용하려면 pip를 이용, peewee 모듈을 설치한다

```
pip install peewee
```

- ▶ PEEWEE 모듈 사용을 위한 임포트

```
pip install peewee
```

- ▶ 데이터베이스 객체의 생성

```
database = peewee.SqliteDatabase("mypeewee.db")
```

PEEWEE

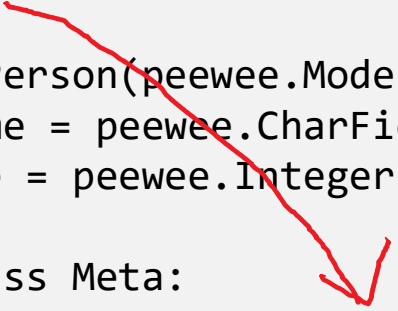
: Model의 생성

- ▶ PEEWEE 모델을 만들려면 `peewee.Model`을 상속받은 모델 클래스를 생성한다

```
database = peewee.SqliteDatabase("mypeewee.db")

class Person(peewee.Model):
    name = peewee.CharField()
    age = peewee.IntegerField()

    class Meta:
        database = database
```



- ▶ Inner Class `Meta`에는 데이터베이스 관련된 메타데이터를 지정한다

PEEWEE

: PEEWEE 필드형식과 데이터베이스

- ▶ PEEWEE는 다양한 필드형식을 지원하며 데이터베이스의 컬럼 타입과 매핑된다
- ▶ 주요 PEEWEE Field Type과 데이터베이스의 컬럼 타입

PEEWEE	Sqlite	Postgresql	MySQL
IntegerField	integer	integer	integer
FloatField	real	real	real
CharField	varchar	varchar	varchar
FixedCharField	char	char	char
TextField	text	text	text
DateTimeField	datetime	timestamp	datetime
.DateField	date	date	date
ForeignKeyField	integer	integer	integer

PEEWEE

: PEEWEE 필드형식과 데이터베이스

- ▶ 필드 형식은 기본값 등, 파라미터를 받아들일 수 있다
 - ▶ `null` : null 값을 허용함 (기본값 `False`)
 - ▶ `index` : 컬럼에 인덱스를 생성함 (기본값 `False`)
 - ▶ `unique` : 컬럼에 `Unique` 제약을 추가함 (기본값 `False`)
 - ▶ `default` : 기본 값을 설정함 (기본값 `None`)
 - ▶ `primary_key` : 기본 키 컬럼으로 설정함 (기본값 `False`)
 - ▶ 기타
- ▶ 특정 필드 형식에만 부여할 수 있는 파라미터들도 있음
 - ▶ 예) `CharField`, `FixedCharField` -> `max_length` 속성

PEEWEE

: 데이터베이스 생성과 테이블 생성

- ▶ 별도의 DML 쿼리를 수행하지 않고 모델을 이용, 테이블을 생성

```
database = peewee.SqliteDatabase("mypeewee.db")

class Person(peewee.Model):
    ...

class Pet(peewee.Model):
    ...

database.connect()
database.create_tables([Person, Pet])
database.close()
```

PEEWEE

: 객체 관계의 연결

- ▶ ForeignKeyField를 이용하여 객체 관계를 매핑할 수 있음

```
database = peewee.SqliteDatabase("mypeewee.db")

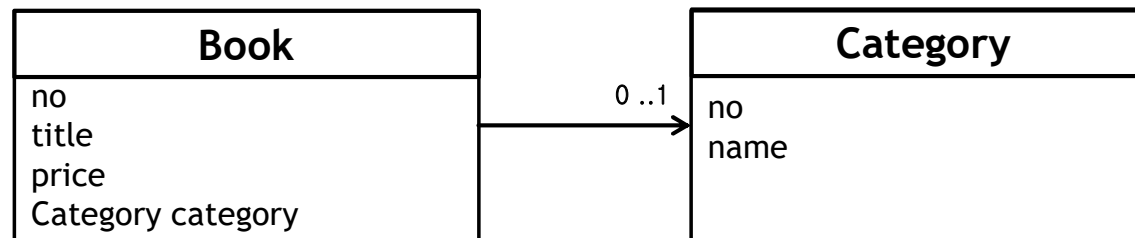
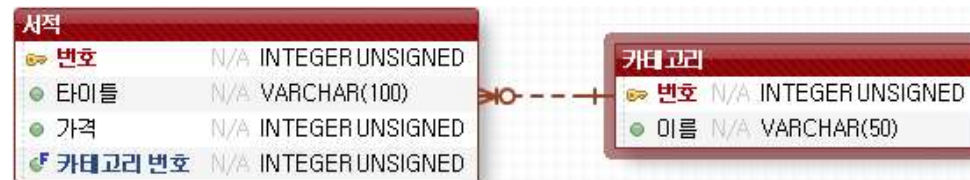
class Category(peewee.Model):
    no = peewee.IntegerField(primary_key=True)
    ...

class Book(peewee.Model):
    no = peewee.IntegerField(primary_key=True)
    ...
    category = peewee.ForeignKeyField(Category, backref= "no", null=True)
```

PEEWEE

: 객체 관계의 연결

- ▶ 다음 관계도를 보고 객체 관계를 연결해 봅시다



PEEWEE

: INSERT 쿼리

- ▶ 모델 객체의 `.create()` 메서드를 이용하면 **INSERT** 할 객체를 생성할 수 있음
 - ▶ `save()` 메서드를 이용하면 객체 내용을 매핑된 테이블에 **INSERT**

```
b1 = Book.create(no = 1,  
                 title = "Learning Python",  
                 price = 12000,  
                 pub_date = datetime.date(2015, 12, 11))  
b1.save()
```

PEEWEE

: UPDATE, DELETE 쿼리

- ▶ UPDATE : 모델 객체를 `.get()` 메서드 등으로 메모리에 적재한 후, 내용을 변경하고 `.save()` 메서드를 수행

```
book = Book.get(Book.no == 1)
book.price += book.price * 0.1
book.save()
```

- ▶ DELETE : 모델 객체를 `.get()` 메서드 등으로 메모리에 적재한 후, `.delete_instance()` 메서드를 수행

```
book = Book.get(no == 1)
book.delete_instance()
```

PEEWEE

: SELECT 쿼리

- ▶ 모든 필드를 SELECT : 모델 객체의 `.select()` 메서드를 수행하면 결과를 반환

```
books = Book.select()  
print(books.sql()) # 작성된 SQL문의 확인
```

- ▶ 특정 필드를 SELECT : 모델 객체를 `.select()` 메서드에 얻어올 필드를 지정

```
books = Book.select(Book.title, Book.price)
```

PEEWEE

: SELECT ~ WHERE ~ ORDER BY 쿼리

- ▶ WHERE 절을 이용한 filtering과 ORDER BY 절:
 - ▶ .where() 메서드와 .order_by 메서드의 이용
 - ▶ 정렬 순서를 내림자순(DESC)으로 지정하려면 order_by에 사용할 필드에 .desc() 함수 이용

```
books = Book.select() \
    .where((Book.price > 15000) and (Book.price < 20000)) \
    .order_by(Book.price.desc())
```