SQL with django ORM

기본 준비 사항

- django app
  - 가상환경 세팅
  - 패키지 설치
  - migrate

```
$ python manage.py migrate
```

- 제공 받은 `users.csv` 파일은 db.sqlite3와 같은 곳에 위치하도록 이동
- `db.sqlite3` 활용
  - `sqlite3` 실행

```
$ sqlite3 db.sqlite3
```

  - 테이블 확인

```
sqlite > .tables
auth_group                  django_admin_log
auth_group_permissions      django_content_type
auth_permission             django_migrations
auth_user                   django_session
auth_user_groups            auth_user_user_permissions
users_user
```

  - csv 파일 data 로드 및 확인

```
sqlite > .mode csv
sqlite > .import users.csv users_user

sqlite > SELECT COUNT(*) FROM users_user;
100
```

문제

> ORM은 django extensions의 shell_plus에서,
>
> SQL은 과거 수업에서 진행한 SQLite 확장프로그램 사용 방식으로 진행

1. 기본 CRUD 로직

1. 모든 user 레코드 조회

```
# orm

User.objects.all()
```

```
-- sql

SELECT * FROM users_user;
```

2. user 레코드 생성

```
# orm

>>> User.objects.create(first_name='길동',
... last_name='홍',
... age=100,
... country='제주도',
... phone='010-1234-4567',
... balance=10000,
... )

<User: User object (101)>
```

```
-- sql
-- 직접 id 컬럼을 만들었기 때문에 컬럼명을 명시하지 않으려면 직접 id 값도 넣어줘야한다.
-- id 값을 명시하고 싶지 않으면 컬럼명을 모두 명시 후 데이터를 넣는다.

-- 기존 레코드 확인
sqlite> SELECT * FROM users_user;
...
101,"길동","홍",100,"제주도",010-1234-4567,10000

-- ORM에서 101번 데이터가 들어갔기 때문에 102로 id 를 설정해준다.
sqlite> INSERT INTO users_user
VALUES (102, '길동', '김', 100, '경상북도', '010-1234-1234', 100);

-- 레코드 확인
sqlite> SELECT * FROM users_user;
id,first_name,last_name,age,country,phone,balance
....
102,"길동","김",100,"경상북도",010-1234-1234,100

-- 안되는 상황
-- 1. id를 안 넣은 상황
sqlite> INSERT INTO users_user
```

```
VALUES ('길동', '홍', 100, '제주도', '010-1234-4567', 100000);
Error: table users_user has 7 columns but 6 values were supplied

-- 2. orm 에서 넣은 101에 넣으려는 상황
-- 이미 존재하는 id에 새롭게 insert 하려니 에러 발생!
sqlite> INSERT INTO users_user VALUES (101, '길동',
'홍', 100, '제주도', '010-1234-5678', 10000);
Error: UNIQUE constraint failed: users_user.id
```

- 하나의 레코드를 빼고 작성 후 `NOT NULL` constraint 오류를 orm과 sql에서 모두 확인 해보세요.

3. 해당 user 레코드 조회
   - `102` 번 id의 전체 레코드 조회

```
# orm

>>> User.objects.get(pk=101)

<User: User object (101)>
```

```
-- sql

sqlite> SELECT * FROM users_user WHERE id=101;

101,"길동","홍",100,"제주도",010-1234-4567,10000
```

4. 해당 user 레코드 수정
   - ORM: `102` 번 글의 `last_name` 을 '김' 으로 수정
   - SQL: `102` 번 글의 `first_name` 을 '철수' 로 수정

```
# orm
>>> user = User.objects.get(pk=101)
>>> user
<User: User object (101)>

>>> user.last_name
>>> '홍'

>>> user.last_name = '김'
>>> user.save()

# 확인
>>> user.last_name
'김'
```

```
-- sql
sqlite> UPDATE users_user
...> SET first_name='철수'
...> WHERE id=101;

-- 변경 확인
sqlite> SELECT first_name FROM users_user WHERE id=101;
"철수"
```

5. 해당 user 레코드 삭제
    - ORM: `102` 번 글 삭제
    - SQL: `101` 번 글 삭제

```
# orm
>>> User.objects.get(pk=101).delete()
(1, {'users.User': 1})

# 해당 글을 조회하려고 하면 에러 발생
>>> User.objects.get(pk=101)
...
users.models.User.DoesNotExist: User matching query does not exist.
```

```
-- sql
sqlite> DELETE FROM users_user
   ...> WHERE id=101;

-- 삭제 확인 --> 아무 것도 안나옴
sqlite> SELECT * FROM users_user WHERE id=101;
```

---

2. 조건에 따른 쿼리문

1. 전체 인원 수
    - `User` 의 전체 인원수

```
# orm

User.objects.count()

len(User.objects.all())
```

```
-- sql

sqlite> SELECT COUNT(*) FROM users_user;
```

> .count() : Returns the total number of entries in the database. (don't need `.all()`)

2. 나이가 30인 사람의 이름

- `ORM` : `.values` 활용
  - 예시: `User.objects.filter(조건).values(컬럼이름)`

```
# orm

User.objects.filter(age=30).values('first_name')
<QuerySet [{'first_name': '영환'}, {'first_name': '보람'}, {'first_name': '은
영'}]>

# 쿼리문 확인
print(User.objects.filter(age=30).values('first_name').query)
SELECT "users_user"."first_name" FROM "users_user"
WHERE "users_user"."age" = 30

########################

# 참고
user = User.objects.filter(age=30).values('first_name')

user
<QuerySet [{'first_name': '영환'}, {'first_name': '보람'}, {'first_name': '은
영'}]>

type(user)
<class 'django.db.models.query.QuerySet'>

user.first().get('first_name')
'영환'

user[0].get('first_name')
'영환'
```

```
-- sql

sqlite> SELECT first_name FROM users_user WHERE age=30;

"영환"
"보람"
"은영"
```

3. 나이가 30살 이상인 사람의 인원 수
   - ORM: `__gte` , `__lte` , `__gt`, `__lt` -> 대소관계 활용

```
# orm

User.objects.filter(age__gte=30).count()

print(User.objects.filter(age__gte=30).query)
SELECT "users_user"."id", "users_user"."first_name",
"users_user"."last_name",
"users_user"."age", "users_user"."country", "users_user"."phone",
"users_user"."balance"
FROM "users_user"
WHERE "users_user"."age" >= 30
```

```
-- sql

SELECT count(*) FROM users_user WHERE age >= 30;
```

4. 나이가 20살 이하인 사람의 인원 수

```
# orm
>>> User.objects.filter(age__lte=20).count()
23
```

```
-- sql
sqlite> SELECT COUNT(*) FROM users_user WHERE age<=20;
COUNT(*)
23
```

5. 나이가 30이면서 성이 김씨인 사람의 인원 수

```
# orm

>>> User.objects.filter(age=30, last_name='김').count()
1

>>> print(User.objects.filter(age=30, last_name='김').query)
SELECT "users_user"."id", "users_user"."first_name",
"users_user"."last_name",
"users_user"."age", "users_user"."country", "users_user"."phone",
"users_user"."balance"
FROM "users_user"
WHERE ("users_user"."age" = 30 AND "users_user"."last_name" = 김)

# 동일 - 단, 리턴되는게 쿼리셋 객체
>>> User.objects.filter(age=30).filter(last_name='김')
<QuerySet [<User: User object (60)>]>

>>> print(User.objects.filter(age=30).filter(last_name='김').query)
SELECT "users_user"."id", "users_user"."first_name",
"users_user"."last_name",
"users_user"."age", "users_user"."country", "users_user"."phone",
"users_user"."balance" FROM "users_user"
WHERE ("users_user"."age" = 30 AND "users_user"."last_name" = 김)
```

```
-- sql

SELECT COUNT(*) FROM users_user
WHERE age = 30 AND last_name = '김';
```

> - `OR` 을 활용하고 싶다면, `Q` **object 를 활용하여야 한다.**
>   - A `Q` object (django.db.models.Q) is an object used to encapsulate a collection of keyword arguments.
>   - They make it possible to define and reuse conditions, and combine them using operators such as `|` (OR) and `&` (AND).

> https://docs.djangoproject.com/en/3.2/topics/db/queries/#complex-lookups-with-q-objects

6. 나이가 30이거나 성이 김씨인 사람?

```
# ORM

# 기존의 방법대로 filter만 사용하면
>>> (User.objects.filter(age=30) |
User.objects.filter(last_name='김')).count()

# 이때, Q를 이용하면 조금 더 간단하다.
>>> from django.db.models import Q
>>> User.objects.filter(Q(age=30) | Q(last_name='김'))

# 총 인원수
In [30]: User.objects.filter(Q(age=30) | Q(last_name='김')).count()
Out[30]: 26

# 쿼리문 확인
print(User.objects.filter(Q(age=30) | Q(last_name='김')).query)
SELECT "users_user"."id", "users_user"."first_name",
"users_user"."last_name",
 "users_user"."age", "users_user"."country", "users_user"."phone",
"users_user"."balance"
FROM "users_user"
WHERE ("users_user"."age" = 30 OR "users_user"."last_name" = 김)
```

```
-- sql

SELECT * FROM users_user WHERE age=30 OR last_name='김';

id,first_name,last_name,age,country,phone,balance
5,"영환","차",30,"충청북도",011-2921-4284,220
8,"예진","김",33,"충청북도",010-5123-9107,3700
9,"서현","김",23,"제주특별자치도",016-6839-1106,43000
11,"서영","김",15,"제주특별자치도",016-3046-9822,640000
14,"영일","김",35,"전라남도",011-4448-6198,720
...
```

7. 지역번호가 02인 사람의 인원 수

- `ORM`: `__startswith`
- 주의! `%` 와일드카드이다. `_` 는 정규표현식 필요

```
# orm

>>> User.objects.filter(phone__startswith='02-').count()
24
```

```sql
-- sql

SELECT COUNT(*) FROM users_user
WHERE phone LIKE '02-%';

24
```

8. 거주 지역이 강원도이면서 성이 황씨인 사람의 이름
    - `.values()` 는 값을 뽑아 내고 싶은 데이터의 필드와 값을 가져온다.
    - `filter` 는 값의 개수를 보장할 수 없기 때문에 무조건 QuerySet으로 return 된다.

```
# .values 확인

>>> User.objects.filter(age__gte=30).values()

>>> User.objects.filter(age__gte=30).values('first_name')
```

```
# orm

>>> User.objects.filter(country='강원도',
last_name='황').values('first_name')
<QuerySet [{'first_name': '은정'}]>

# 조건에 부합하는 첫번째 사람의 이름
In [32]: User.objects.filter(country='강원도',
last_name='황').values('first_name').first().get('first_name')
Out[32]: '은정'

# 조건에 부합하는 첫번째 사람의 폰번호
>>> user = User.objects.filter(country='강원도',
last_name='황').values('first_name').first()

>>> type(user)
user.models.User

>>> user.phone
'016-5956-2725'
```

```sql
-- sql
sqlite> SELECT first_name FROM users_user
   ...> WHERE country = '강원도' and last_name = '황';
"은정"
```

3. 정렬 및 LIMIT, OFFSET

1. 나이가 많은 사람순으로 10명

```
# orm
>>> User.objects.order_by('-age')[:10]
<QuerySet [<User: User object (102)>, <User: User object (1)>, <User: User
object (4)>, <User: User object (28)>, <User: User object (53)>, <User: User
object (65)>, <User: User
object (26)>, <User: User object (55)>, <User: User object (58)>, <User:
User object (74)>]>

# 쿼리 확인
>>> print(User.objects.order_by('-age')[:10].query)
SELECT "users_user"."id", "users_user"."first_name",
"users_user"."last_name",
"users_user"."age", "users_user"."country", "users_user"."phone",
"users_user"."balance" FROM "users_user" ORDER BY "users_user"."age" DESC
 LIMIT 10
```

```
-- sql

SELECT * FROM users_user
ORDER BY age DESC LIMIT 10;

102,"길동","홍",100,"제주도",010-1234-5678,10000
1,"정호","유",40,"전라북도",016-7280-2855,370
4,"미경","장",40,"충청남도",011-9079-4419,250000
28,"성현","박",40,"경상남도",011-2884-6546,580000
53,"상훈","홍",40,"전라북도",016-7698-6684,550
65,"민서","송",40,"경기도",011-9812-5681,51000
26,"영식","이",39,"경상북도",016-2645-6128,400000
55,"미경","이",39,"경기도",02-6697-3997,890000
58,"영일","배",39,"전라남도",010-3486-8085,280000
74,"승민","배",39,"강원도",010-4833-9657,840
```

2. 잔액이 적은 사람순으로 10명

```
# orm

User.objects.order_by('balance')[:10]

<QuerySet [<User: User object (102)>, <User: User object (99)>,
<User: User object (48)>, <User: User object (100)>, <User: User object
(5)>,
<User: User object (24)>, <User: User object (61)>, <User: User object
(92)>,
<User: User object (46)>, <User: User object (38)>]>

# id=102 사람의 잔고 확인 - 최저액
>>> User.objects.get(id=102).balance
100
```

```
-- sql
```

```
SELECT * FROM users_user
ORDER BY balance ASC LIMIT 10;

102,"길동","김",100,"경상북도",010-1234-1234,100
99,"우진","성",32,"전라북도",010-7636-4368,150
48,"보람","이",28,"강원도",02-2055-4138,210
100,"재현","김",25,"경상북도",016-1252-2316,210
5,"영환","차",30,"충청북도",011-2921-4284,220
24,"숙자","권",33,"경상남도",016-4610-3200,230
61,"우진","고",15,"경상북도",011-3124-1126,300
92,"미경","박",35,"경상북도",010-5203-5705,300
46,"명자","김",23,"전라남도",011-3545-5608,330
38,"준호","심",28,"충청북도",016-6703-7656,340
```

3. 잔고는 오름차순, 나이는 내림차순으로 10명?

```
# orm

User.objects.order_by('balance', '-age')[:10]
```

```
-- sql

sqlite> SELECT * FROM users_user ORDER BY balance, age DESC LIMIT 10;

-- 정렬의 우선순위는 앞에 나온 것
-- 만약에 정렬 기준이 충돌하면 먼저 정렬 기준으로 잡아 놓은 (balance)것이 우선순위를 갖는다.
```

4. 성, 이름 내림차순 순으로 5번째 있는 사람

```
# orm
>>> User.objects.order_by('-last_name', '-first_name')[4]
<User: User object (67)>
```

```
-- sql
sqlite>
SELECT * FROM users_user
ORDER BY last_name DESC, first_name DESC
LIMIT 1 OFFSET 4;

id,first_name,last_name,age,country,phone,balance
67,"보람","허",28,"충청북도",016-4392-9432,82000
```

---

4. 표현식

4.1 Aggregate

> - https://docs.djangoproject.com/en/3.2/topics/db/aggregation/#aggregation
> - '종합', '집합', '합계' 등의 사전적 의미
> - 특정 필드 전체의 합, 평균 등을 계산할 때 사용
> - `Django_aggregation.md` 문서 참고

1. 전체 평균 나이

```
# orm
# age 필드의 평균 값을 구하자!

>>> from django.db.models import Avg
>>> User.objects.aggregate(Avg('age'))
{'age__avg': 28.940594059405942}

# 이름은 필드와 함수를 조합해 자동 생성되지만 다음과 같이 수동으로 이름을 지정할 수 있다.
>>> User.objects.aggregate(avg_value=Avg('age'))
{'avg_value': 28.940594059405942}
```

```
-- sql

sqlite> SELECT AVG(age) FROM users_user;
28.9405940594059
```

2. 김씨의 평균 나이

```
# orm

from django.db.models import Avg

User.objects.filter(last_name='김').aggregate(Avg('age'))
Out[52]: {'age__avg': 31.75}

# 둘 이상의 aggregate 생성
>>> User.objects.filter(last_name='김').aggregate(Avg('age'), Max('age'), Min('age'))
```

```
-- sql

SELECT AVG(age) FROM users_user WHERE last_name = '김';
31.75
```

3. 강원도에 사는 사람의 평균 계좌 잔고

```
# orm

User.objects.filter(country='강원도').aggregate(Avg('balance'))
{'balance__avg': 157895.0}
```

```
-- sql

SELECT AVG(balance) FROM users_user
WHERE country = '강원도';
AVG(balance)
157895.0
```

4. 계좌 잔액 중 가장 높은 값

```
# orm

>>> from django.db.models import Max
>>> User.objects.aggregate(Max('balance'))
{'balance__max': 1000000}
```

```
-- sql

SELECT MAX(balance) FROM users_user;
1000000
```

5. 계좌 잔액 총액

```
# orm

>>> from django.db.models import Sum
>>> User.objects.aggregate(Sum('balance'))
{'balance__sum': 14425140}
```

```
-- sql

sqlite> SELECT SUM(balance) FROM users_user;

14425140
```

4.2 Annotate

1. 지역별 인원 수

```
# orm

from django.db.models import Count

# 1
User.objects.values('country').annotate(Count('country'))
# <QuerySet [{'country': '강원도', 'country__count': 14}, {'country': '경기
도', 'country__count': 9}, ...

# 2
User.objects.values('country').annotate(num_countries=Count('country'))
# <QuerySet [{'country': '강원도', 'num_countries': 14}, {'country': '경기도',
'num_countries': 9}, ...
```

```
print(User.objects.values('country').annotate(Count('country')).query)
# SELECT "users_user"."country", COUNT("users_user"."country") AS
"country__count" FROM "users_user" GROUP BY "users_user"."country"

# 3
User.objects.values('country').annotate(Count('country'),
avg_balance=Avg('balance'))
# <QuerySet [{'country': '강원도', 'country__count': 14, 'avg_balance':
157895.0}, {'country': '경기도', 'country__count': 9, 'avg_balance':
182852.22222222222}, {'country': '경상남도', 'country__count': 9,
'avg_balance': 73870.0}, {'country': '경상북도', 'country__count': 16,
'avg_balance': 70909.375}, {'country': '전라남도', 'country__count': 10,
'avg_balance': 66265.0}, {'country': '전라북도', 'country__count': 10,
'avg_balance': 177215.0}, {'country': '제주도', 'country__count': 1,
'avg_balance': 10000.0}, {'country': '제주특별자치도', 'country__count': 9,
'avg_balance': 351233.3333333333}, {'country': '충청남도', 'country__count':
9, 'avg_balance': 104304.44444444444}, {'country': '충청북도',
'country__count': 14, 'avg_balance': 159610.7142857143}]>
```

```
-- sql

SELECT country, COUNT(country) FROM users_user GROUP BY country;

강원도|14
경기도|9
경상남도|9
경상북도|15
전라남도|10
전라북도|11
제주특별자치도|9
충청남도|9
충청북도|14
```

2. 1:N 예시
    - Article - Comment 관계가 1:N 인 경우

```
Article.objects.annotate(
    total_count=Count('comment'),
    pub_date=Count('comment', filter=Q(comment__created_at__lte='2000-01-
01'))
)
```

    - `Article.objects.all()` 에 더해서 annotate로 우리가 필요로 인해 만든 2개의 컬럼 (`total_count`, `pub_date`)를 붙여서 가져오는 것.