aiffel / Week5 / Fundamental20.ipynb

jiyeoon Create Week5/Fundamental20.ipynb 5367f6c · 4 years ago

2691 lines (2691 loc) · 84.3 KB

Python에서 SQL을 이용해 DB와 대화해 보자!

파이썬에서 DB에 접근하기 위한 방법 중 가장 많이 사용되는 것은 Python Database API(DB-API) 입니다. 이는 여러 DB에 접근하는 표준 API입니다. 표준 API는 크게 3가지 작업을 합니다.

1. DB를 연결한다
2. SQL문을 실행한다.
3. DB 연결을 닫는다.

파이썬 DB-API는 기본적으로 PEP249 인터페이스를 따르도록 권장됩니다. 여기서 기본적으로 선언되어 있는 함수들로는 `connect()`, `close()`, `commit()`, `rollback()` 등 여러가지가 있습니다. 이 함수들을 이용하면 거의 모든 DB에 대해 동일한 함수를 사용해 조작할 수 있습니다.

- PEP 249 -- Python Database API Specification v2.0

파이썬에서 지원하는 DB는 매우 다양합니다. 각 DB에 상응하는 별도의 DB 모듈을 다운받아야 합니다. 수많은 DB 모듈이 존재하나, 대부분이 Python DB-API 표준을 따르고 있으므로 동일한 API로 여러 DB를 사용할 수 있습니다.

파이썬에서는 MySQL, PostgreSQL, Oracle 등 대표적인 DB들을 모두 지원합니다. 이번 시간에는 **SQLite**를 사용할 거에요.

그럼 시작해봅시다!

> SQLite

SQLite는 별도의 *서버 필요 없이* DB 파일에 기초하여 DB 처리를 구현한 임베디드 SQL DB 엔진입니다. SQLite는 별도의 설치 없이 쉽고 편리하게 사용될 수 있다는 점에서 많이 사용되고 있습니다.

파이썬은 버전 2.5 이상일 경우 설치가 되어있습니다.

Python DB API

그럼 파이썬과 DB를 연결해봅시다.

일단 파이썬에서 DB를 연결하기 위해 **sqlite3** 모듈을 import 해줍시다.

In [1]:
```
import sqlite3
```

sqlite3 모듈은 파이썬 표준 라이브러리로 SQLite에 대한 인터페이스를 기본적으로 제공합니다.

작업 디렉토리를 구성해봅시다! 현재 폴더에서 `data` 라는 폴더를 따로 만들어 여기에 DB를 만들어봅시다.

```
$ mkdir -p ./data
```

이제 파이썬과 DB를 연결해봅시다. conn 에 DB이름을 정하여 입력합니다. 여기서 저는 mydb 로 지었는데, [이름].[확장자명] 의 형태로 저장하면 됩니다.

In [2]:
```
import os
db_path = os.path.dirname(os.path.abspath('__file__'))
db_path += r'/data/mydb.db'

conn = sqlite3.connect(db_path)
print(conn)
```

```
<sqlite3.Connection object at 0x7f0a4c0213b0>
```

conn 객체에는 SQL 연결과 관련된 셋팅이 포함되어 있습니다. 이번에는 Connect() 함수의 연결을 사용하여 새로운 Cursor 객체를 만들어봅시다.

In [3]:
```
c = conn.cursor()
print(c)
```

```
<sqlite3.Cursor object at 0x7f0a3e7ee8f0>
```

Cursor는 SQL의 쿼리를 수행하고 결과를 얻는데 사용하는 객체입니다. INSERT 처럼 DB에만 적용되는 명령어를 사용한다면 Cursor를 사용하지 않을 수 있지만, SELECT 와 같이 데이터를 불러올 때에는 SQL 질의 수행 결과에 접근하기 위한 Cursor가 반드시 필요합니다.

이러한 이유로 습관적으로라도 conn.cursor() 를 사용하는 것을 권장합니다.

이제 SQL문을 실행해봅시다! 질의의 수행은 **execute()** 를 이용하게 됩니다.

In [4]:
```
# stocks라는 테이블을 하나 생성합니다. 혹시 생성되어 있다면 생략합니다.
c.execute("₩
          CREATE TABLE IF NOT EXISTS stocks(₩
              date text,₩
              trans text,₩
              symbol text,₩
              qty real,₩
              price real)"
)
```

Out[4]:
```
<sqlite3.Cursor at 0x7f0a3e7ee8f0>
```

In [5]:
```
# stocks 테이블에 데이터를 하나 삽입합니다.
c.execute("""
    INSERT INTO stocks
    VALUES('20200701', 'TEST', 'AIFFEL', 1, 10000)
""")
```

Out[5]:
```
<sqlite3.Cursor at 0x7f0a3e7ee8f0>
```

In [7]:
```
# 방금 넣은 데이터를 조회해봅시다.
c.execute("SELECT * FROM stocks")
```

Out[7]:
```
<sqlite3.Cursor at 0x7f0a3e7ee8f0>
```

In [8]:
```
# 조회된 내역을 커서를 통해 가져와 출력해봅시다.
print(c.fetchone())
```

```
('20200701', 'TEST', 'AIFFEL', 1.0, 10000.0)
```

이렇게 sqlite3 모듈을 사용해 데이터베이스에 테이블을 하나 만들고 데이터를 삽입해보았습니다!

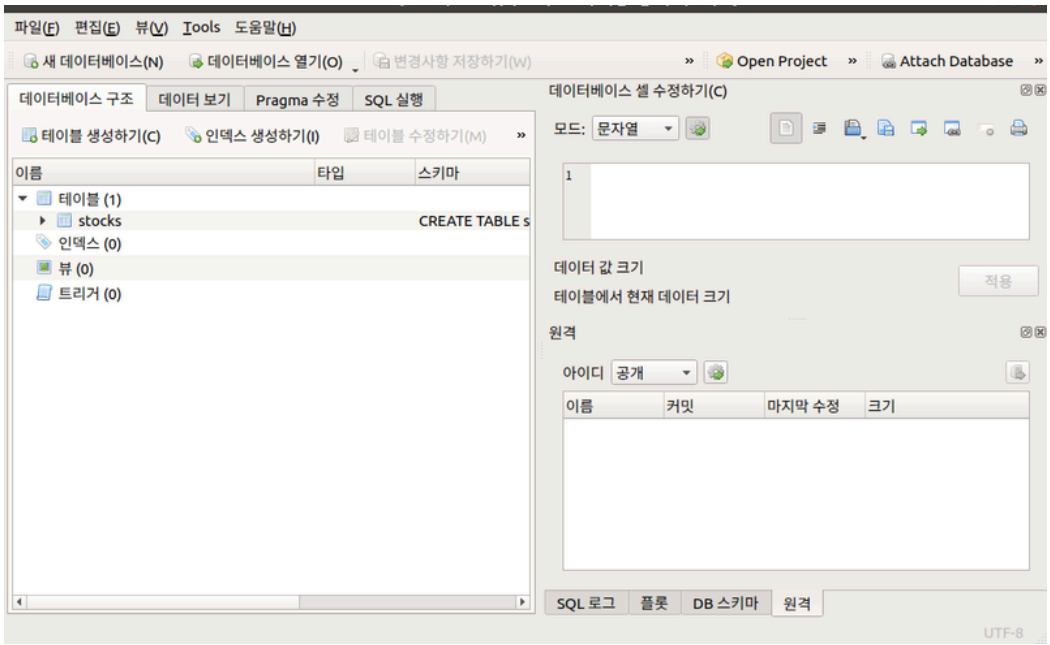
sqlite DB Browser

데이터베이스를 다루는 방법에는 sqlite3과 같은 Python DB API만 존재하는 것은 아닙니다. DB 브라우저 어플리케이션을 통해 데이터베이스에 직접 접근해서 질의를 수행할 수 있습니다.

우분투 환경이라면 아래와 같이 터미널을 열어 설치해주세요.

```
$ sudo add-apt-repository -y ppa:linuxgndu/sqlitebrowser
$ sudo apt-get update
$ sudo apt-get install sqlitebrowser
```

설치가 완료되었다면 어플리케이션이 설치되었을겁니다. 검색해서 나오는 DB Browser를 실행해주세요!



정상적으로 설치가 되었다면 위와 같은 화면이 뜹니다. `데이터베이스 열기` 버튼을 클릭후 위에서 생성했던 `./data/mydb.db` 파일을 찾아서 열어봅시다.

아래 그림과 같이 `데이터베이스 구조` 탭에 `stocks` 라는 이름의 테이블이 생성되었음을 확인할 수 있습니다. 이것으로 우리는 DB Browser에서 sqlite 데이터베이스를 관리할 수 있습니다.

그런데 우리가 stocks에 하나 입력했던 정보를 보려고 `데이터 보기` 탭을 눌러보면 아무런 데이터가 조회되지 않습니다. 무슨 일일까요?

삽입, 갱신, 삭제 등의 SQL 질의가 끝났다면 `conn.commit()` 를 호출해야 DB가 실제로 업데이트를 합니다. `commit()` 를 하기 전에는 DB에 데이터가 업데이트된것

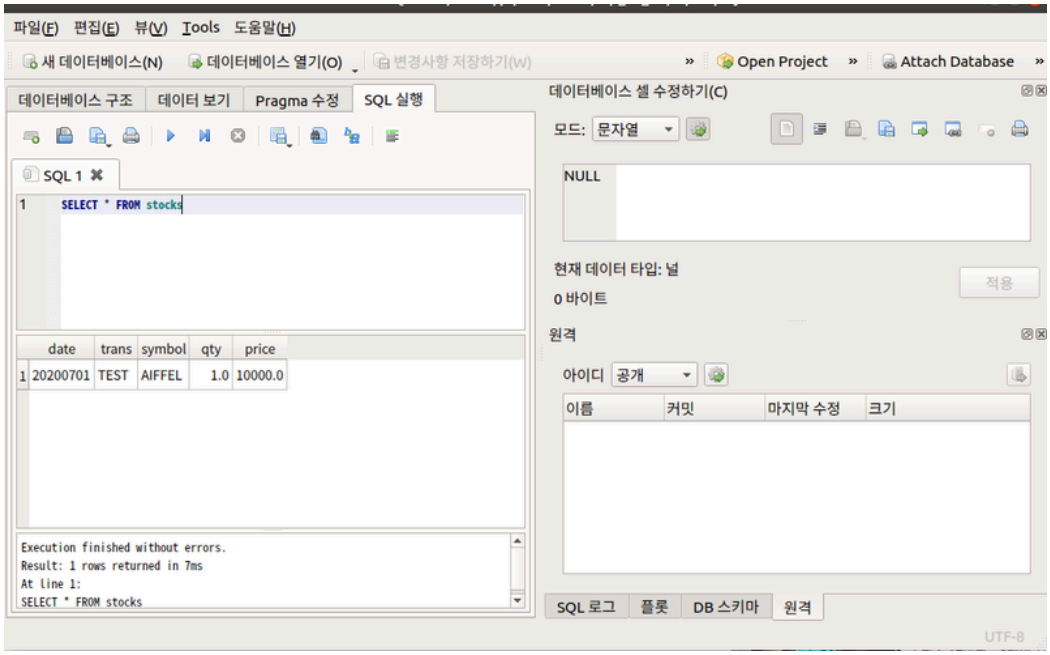
같아 보여도 임시로만 바뀐 것이니 주의해야 합니다. 하지만 sqlite3를 이용해 데이터가 잘 인서트 되었음을 `SELECT` 문의 결과를 통해서 이미 확인했었습니다. 그럼 이것은 무엇인가요?

이게 바로 데이터베이스를 사용할 때의 *주의 사항*입니다. 우리는 sqlite3를 이용해 데이터베이스에 `connection` 을 하나 맺었습니다. 이후 해당 `connection` 을 통해 인서트 된 데이터는 `conn.commit()` 를 호출하기 전까지는 그 `connection` 안에서만 유효합니다. 그래서 sqlite3에서는 `SELECT` 가 되었지만, DB Browser안에서는 조회되지 않습니다.

이러한 이유로 원본 데이터에 실제로 적용하려면 `commit()` 명령어를 이용해야 합니다. 참고로 select처럼 데이터를 가져오기만 하는 질의문의 경우에는 `commit()` 가 필요 없습니다. 데이터에 아무런 변경 사항이 없으니까요!

In [9]:
```
# commit은 connection의 메소드입니다.
conn.commit()
```

그럼 이제 다시 확인해봅시다.



이렇게 `commit()` 를 통해 원본 데이터베이스에 변경이 실제적으로 반영되었습니다.

이런 것을 데이터베이스에서는 **트랜잭션(transaction)**이라고 합니다.

- 트랜잭션이란

위 글에서는 ATM 계좌이체시 출금 계좌와 입금 계좌 양쪽에서 금액이 빠지고 더해지는 과정이 쪼개져서는 안된다고 예를 들고있습니다.

`commit()` 을 완료했다면 DB와 대화하는 것을 마무리 지어야합니다. 이때에는 `close()` 메소드를 이용하면 됩니다.

In [10]:
```
c.close() # 먼저 커서를 닫은 후
conn.close() # DB 연결을 닫아줍니다.
```

DDL 문으로 테이블 생성하기

다양한 테이블을 조회해보기 앞서 실제 예제 테이블들을 한번 생성해 보도록 하겠습니다.

In [11]:

```
import sqlite3
import os
db_path = os.getenv('HOME')+'/aiffel/sql_to_db/sqlite/mydb.db'

conn = sqlite3.connect(db_path)
c = conn.cursor()

#- ! 재실행 시 테이블이 존재할 수 있으므로 아래처럼 해당 테이블들을 모두 지우
c.execute("DROP TABLE IF EXISTS 도서대출내역")
c.execute("DROP TABLE IF EXISTS 도서대출내역2")
c.execute("DROP TABLE IF EXISTS 대출내역")
c.execute("DROP TABLE IF EXISTS 도서명")


#----- 1st table : 도서대출내역 -----#
c.execute("CREATE TABLE IF NOT EXISTS 도서대출내역 (ID varchar, 이름 varchar
#- 생성(create)문 : 테이블명, 변수명, 변수타입을 지정

data = [('101','문강태','aaa','2020-06-01','2020-06-05'),
            ('101','문강태','ccc','2020-06-20','2020-06-25'),
            ('102','고문영','bbb','2020-06-01',None),
            ('102','고문영','ddd','2020-06-08',None),
            ('103','문상태','ccc','2020-06-01','2020-06-05'),
            ('104','강기둥',None,None,None)]
#- 입력할 데이터를 그대로 입력 (변수명 순서 기준대로)

c.executemany('INSERT INTO 도서대출내역 VALUES (?,?,?,?,?)', data)
#- 입력할 데이터를 실제 테이블에 insert하기
#-------------------------------------------#


#----- 2nd table : 도서대출내역2 -----#
c.execute("CREATE TABLE IF NOT EXISTS 도서대출내역2 (ID varchar, 이름 varcha

data = [('101','문강태','2020-06','20일'),
            ('102','고문영','2020-06','10일'),
            ('103','문상태','2020-06','8일'),
            ('104','강기둥','2020-06','3일')]
c.executemany('INSERT INTO 도서대출내역2 VALUES (?,?,?,?)', data)
#-------------------------------------------------#


#----- 3rd table : 대출내역 -----#
c.execute("CREATE TABLE IF NOT EXISTS 대출내역 (ID varchar, 이름 varchar, 도

data = [('101','문강태','aaa'),
            ('102','고문영','bbb'),
            ('102','고문영','fff'),
            ('103','문상태','ccc'),
            ('104','강기둥',None)]
c.executemany('INSERT INTO 대출내역 VALUES (?,?,?)', data)
#---------------------------------------#


#----- 4th table : 도서명 -----#
c.execute("CREATE TABLE IF NOT EXISTS 도서명 (도서ID varchar, 도서명 varchar
```

```
c.execute('CREATE TABLE IF NOT EXISTS 도서명 (도서ID varchar, 도서명 varchar')

data = [('aaa','악몽을 먹고 자란 소년'),
            ('bbb','좀비아이'),
            ('ccc','공룡백과사전'),
            ('ddd','빨간구두'),
            ('eee','잠자는 숲속의 미녀')]

c.executemany('INSERT INTO 도서명 VALUES (?,?)', data)
#-------------------------------------#

conn.commit()
conn.close()
```

하나하나 살펴봅시다.

- `CREATE TABLE IF NOT EXISTS 도서명 (도서ID varchar, 도서명 varchar)` : 테이블이 존재하지 않으면 테이블을 생성
- `data = [('aaa', '악몽을 먹고 자..'), ..., ]` : 각 변수명에 맞게 데이터를 실제로 생성
- `c.executemany()` : 한꺼번에 여러 데이터의 처리를 가능하도록.
  - `INSERT INTO 도서명 VALUES (?, ?)` : 각 테이블의 변수 (도서 ID, 도서명) 에 data를 넣겠다는 뜻

In [12]:

```
conn = sqlite3.connect(db_path)
c = conn.cursor()

for row in c.execute("SELECT * FROM 도서명"):
    print(row)
```

```
('aaa', '악몽을 먹고 자란 소년')
('bbb', '좀비아이')
('ccc', '공룡백과사전')
('ddd', '빨간구두')
('eee', '잠자는 숲속의 미녀')
```

제대로 생성되었네요!

그럼 이제 SQL문을 조작해보면서 만든 테이블을 조회해봅시다.

SQL의 기본

*SQL (Structured Query Language)

SQL은 데이터베이스에서 데이터를 조회하고자 할 때 필요한 컴퓨터 언어라고 할 수 있습니다.

DB라는 공간에 '정형화된' 데이터가 차곡차곡 저장되어 있는 것으로, 이러한 DB를 특정 언어로 조회해서 가져오는데 그때 사용하는 언어가 SQL 입니다.

이제 쿼리의 대표 기본 구조를 살펴봅시다.

- `SELECT ~` : 조회할 칼럼명 선택
- `FROM ~` : 조회할 테이블명을 지정
- `WHERE ~` : 질의할 때 필요한 조건을 설정

- GROUP BY ~ : 특정 칼럼을 기준으로 그룹지어 출력
- ORDER BY ~ : SELECT 다음에 오는 칼럼 중 정렬이 필요한 부분을 정렬
- LIMIT 숫자 : DISPLAY하고자 하는 행의 수를 결정

그럼 한번 직접 하면서 확인해봅시다!!

In [15]:
```
import pandas as pd

col = ['ID', '이름', '도서ID', '대출일', '반납일']

df = pd.DataFrame(c.execute("SELECT * FROM 도서대출내역"), columns = col)
df
```

Out[15]:

	ID	이름	도서ID	대출일	반납일
0	101	문강태	aaa	2020-06-01	2020-06-05
1	101	문강태	ccc	2020-06-20	2020-06-25
2	102	고문영	bbb	2020-06-01	None
3	102	고문영	ddd	2020-06-08	None
4	103	문상태	ccc	2020-06-01	2020-06-05
5	104	강기둥	None	None	None

전체 데이터를 조회할때는 아래와 같이 사용합니다.

```
SELECT * FROM 도서대출내역;
```

SELECT 와 FROM 사이에는 특정 칼럼을 넣어 출력하곤 합니다. 위의 쿼리처럼 별(*)을 입력하게 되면 테이블 전체를 다 가져오라는 명령어가 됩니다.

특정 칼럼만 지정할 수도 있습니다.

In [25]:
```
query = "SELECT ID FROM 도서대출내역"
pd.DataFrame(c.execute(query), columns = ['ID'])
```

Out[25]:

	ID
0	101
1	101
2	102
3	102
4	103
5	104

```
SELECT ID FROM 도서대출내역;
```

SELECT 와 FROM 사이에 ID 를 넣으면 전체 테이블중에 ID 칼럼만 가져오라는 명령어가 됩니다.

In [26]:
```
# 조건 입력하기
query = "SELECT * FROM 도서대출내역 WHERE 이름 = ₩"문강태₩""
pd.DataFrame(c.execute(query), columns=col)
```

Out[26]:

	ID	이름	도서ID	대출일	반납일
0	101	문강태	aaa	2020-06-01	2020-06-05
1	101	문강태	ccc	2020-06-20	2020-06-25

```
SELECT * FROM 도서대출내역
WHERE 이름 = "문강태";
```

WHERE 절을 추가하면 특정 조건을 입력할 수 있습니다. 여기서는 이름이 문강태인 사람을 출력하게 됩니다.

In [27]:
```
# group by로 중복 제거하기
query = "SELECT 이름 FROM 도서대출내역 GROUP BY 이름"
pd.DataFrame(c.execute(query), columns=['이름'])
```

Out[27]:

	이름
0	강기둥
1	고문영
2	문강태
3	문상태

```
SELECT 이름 FROM 도서대출내역
GROUP BY 이름;
```

GROUP BY 는 이름 그대로 데이터를 그룹화시키는 역할을 합니다. '가나다라' 순으로 재정렬해서 뱉어주는 것도 확인할 수 있습니다. GROUP BY 는 보통 집계성 함수와 함께 사용되고 있습니다.

그럼 이번에는 한번 정렬을 해봅시다.

In [28]:
```
# order by 로 정렬하기
query = "SELECT * FROM 도서대출내역 ORDER BY ID"
pd.DataFrame(c.execute(query), columns = col)
```

Out[28]:

	ID	이름	도서ID	대출일	반납일
0	101	문강태	aaa	2020-06-01	2020-06-05
1	101	문강태	ccc	2020-06-20	2020-06-25
2	102	고문영	bbb	2020-06-01	None
3	102	고문영	ddd	2020-06-08	None
4	103	문상태	ccc	2020-06-01	2020-06-05
5	104	강기둥	None	None	None

```
SELECT * FROM 도서대출내역
```

```
SELECT * FROM 도서대출내역
ORDER BY ID;
```

ORDER BY 뒤에 특정 칼럼명을 적으면 그 칼럼을 기준 값으로 정렬해서 보여줍니다.여기서는 ID를 기준으로 정렬했습니다.

일반적으로 ORDER BY 는 뒤에 ASC 가 생략되어있습니다. 오름차순이 기본적이지만, 내림차순을 하고싶다면 DESC 라고 입력하면 됩니다.

In [29]:
```
# 내림차순으로 정렬하기
query = "SELECT * FROM 도서대출내역 ORDER BY ID DESC"
pd.DataFrame(c.execute(query), columns = col)
```

Out[29]:

	ID	이름	도서ID	대출일	반납일
0	104	강기둥	None	None	None
1	103	문상태	ccc	2020-06-01	2020-06-05
2	102	고문영	bbb	2020-06-01	None
3	102	고문영	ddd	2020-06-08	None
4	101	문강태	aaa	2020-06-01	2020-06-05
5	101	문강태	ccc	2020-06-20	2020-06-25

차이가 보이죠?

이번엔 몇개만 선택해서 봐봅시다.

In [30]:
```
# 몇개의 row만 조회하기
query = "SELECT * FROM 도서대출내역 LIMIT 5"
pd.DataFrame(c.execute(query), columns=col)
```

Out[30]:

	ID	이름	도서ID	대출일	반납일
0	101	문강태	aaa	2020-06-01	2020-06-05
1	101	문강태	ccc	2020-06-20	2020-06-25
2	102	고문영	bbb	2020-06-01	None
3	102	고문영	ddd	2020-06-08	None
4	103	문상태	ccc	2020-06-01	2020-06-05

```
SELECT * FROM 도서대출내역 LIMIT 5;
```

LIMIT 구문 뒤에 숫자를 적으면 그 숫자만큼의 행만 출력됩니다. LIMIT 구문은 주로 처음 테이블을 조회할 때 그 테이블의 구조를 파악하고 어떤 값들이 존재하는지 샘플로 파악하고자 할 때 사용합니다.

한번 다 섞어서 확인해봅시다!

In [31]:
```
query = "SELECT 이름, 대출일, 반납일 FROM 도서대출내역 ORDER BY 대출일 DESC
pd.DataFrame(c.execute(query), columns = ['이름', '대출일', '반납일'])
```

Out[31]:

	이름	대출일	반납일
0	문강태	2020-06-20	2020-06-25
1	고문영	2020-06-08	None

DISTINCT와 GROUP BY

중복을 제거한다는 의미를 떠올렸을 때 `DISTINCT` 와 `GROUP BY` 의 차이는 무엇일까요?

아래 표를 통해 한번 살펴봅시다.

함수명	DISTINCT	GROUP BY
기능	특정 컬럼들이 갖고있는 값들의 중복을 제거할 때 사용	특정 기준으로 집계
주로 같이 사용하는 함수	COUNT 등	집계성 함수 (COUNT, MAX, MIN, AVG 등)
예시	SELECT DISTINCT A FROM 테이블명	SELECT A, MAX(B) FROM 테이블명 GROUP BY A DESC
	SELECT COUNT(DISTINCT A) FROM 테이블명	

한눈에 정리가 가능하죠?

추가로, 기준 별로 중복없이 집계를 하고싶을 때에는 집계함수와 더불어 `DISTINCT` , `GROUP BY` 를 모두 활용해야 합니다.

Data Type

Oracle, SQL Server, MySQL 등등 SQL의 종류가 다양한 만큼 데이터 타입도 다양하고 조회 및 조작어도 세밀하게 다릅니다.

한번 이번에는 데이터의 타입(형)을 바꾸는 실습을 해봅시다.

위의 4명의 사람의 평균 대출일수가 궁금하다면 어떻게 해야할까요? 평균은 `AVG()` 함수를 사용하면 됩니다. 평균을 구하기 위해서는 일단 `대출일수` 라는 칼럼이 숫자 타입을 지니고 있어야 합니다.

In [39]:

```
col = ['ID', '이름', '대출년월', '대출일수']
query = "SELECT * FROM 도서대출내역2"
pd.DataFrame(c.execute(query), columns = col)
```

Out[39]:

	ID	이름	대출년월	대출일수
0	101	문강태	2020-06	20일
1	102	고문영	2020-06	10일
2	103	문상태	2020-06	8일
3	104	강기둥	2020-06	3일

그런데 대출일수는 숫자 + '일' 이런 형태로 되어있네요. 흠.. 일단 각 데이터의 타입들을 확인해볼까요?

In [40]:
```
for row in c.execute('pragma table_info(도서대출내역2)'):
    print(row)
```

```
(0, 'ID', 'varchar', 0, None, 0)
(1, '이름', 'varchar', 0, None, 0)
(2, '대출년월', 'varchar', 0, None, 0)
(3, '대출일수', 'varchar', 0, None, 0)
```

sqlite에서는 테이블의 데이터 타입 정의를 확인할 수 있도록 pragma table_info('테이블명') 문을 제공합니다. 이것은 표준 SQL 쿼리가 아니라 데이터베이스마다 달라지는 유틸리티 쿼리라고 할 수 있습니다.

데이터 타입을 보니 모두 VARCHAR 로 되어있습니다. 문자형이네요! 대출일수 칼럼을 그럼 문자형에서 숫자형으로 변환해봅시다.

문자형의 특정 부분을 떼어내는 함수는 LEFT , RIGHT , SUBSTRING 를 사용하면 됩니다.

함수명	설명	사용법
LEFT	왼쪽부터 원하는 길이만큼 자르는 함수	LEFT(문자형 칼럼, 길이)
RIGHT	오른쪽부터 원하는 길이만큼 자르는 함수	RIGHT(문자형 칼럼, 길이)
SUBSTR , SUBSTRING	일정 영역만큼을 자르는 함수	SUBSTR(문자형 칼럼, 길이)

언어에 따라 SUBSTRING 함수는 약간씩 차이가 날 수 있습니다. sqlite에서는 SUBSTR() 함수입니다.

In [42]:
```
col = ['ID', '이름', '대출년월', '대출일수', '대출일수_수정']
query = "SELECT *, SUBSTR(대출일수, 1, (length(대출일수)-1)) AS 대출일수_수정
pd.DataFrame(c.execute(query), columns=col)
```

Out[42]:

	ID	이름	대출년월	대출일수	대출일수_수정
0	101	문강태	2020-06	20일	20
1	102	고문영	2020-06	10일	10
2	103	문상태	2020-06	8일	8
3	104	강기둥	2020-06	3일	3

```
SELECT *, SUBSTR(대출일수, 1, (length(대출일수)-1)) AS 대출일수_수정
FROM 도서대출내역2;
```

'대출일수'의 마지막 컬럼 값이 항상 '일'로 끝나므로 마지막 자리만 삭제하기 위해 위와 같은 쿼리를 작성하였습니다. 만약 무조건 'OO일'과 같이 3자리라면 앞에서부터 2개만 잘라내면 되겠지만, 20일/10일/8일과 같이 '일'의 앞 자리수가 다른 경우

위처럼 하거나 sqlite에서 제공하는 `split_part()` 함수를 사용하면 됩니다.

그리고 `AS` 를 사용해 수정한 값을 '대출일수_수정'이라는 컬럼으로 새롭게 저장합니다. 조회해보면 한 개의 컬럼이 맨 뒤에 추가된 것을 확인할 수 있을 거예요.

그 다음은 `CAST` 함수를 이용하여, 잘라낸 부분에 더하여 숫자로 변환해 보겠습니다.

In [43]:
```
query = """
    SELECT *, CAST(SUBSTR(대출일수, 1, (length(대출일수)-1)) AS INT) AS 대출
FROM 도서대출내역2
"""
pd.DataFrame(c.execute(query), columns=col)
```

Out[43]:

	ID	이름	대출년월	대출일수	대출일수_수정
0	101	문강태	2020-06	20일	20
1	102	고문영	2020-06	10일	10
2	103	문상태	2020-06	8일	8
3	104	강기둥	2020-06	3일	3

```
SELECT *, CAST(SUBSTR(대출일수, 1, (length(대출일수)-1)) AS INT) AS
대출일수_수정
FROM 도서대출내역2 ;
```

코드를 보면 `CAST` 라는 것이 나왔습니다. `CAST()` 는 형변환을 위한 함수입니다.

`CAST (형변환을 하고싶은 칼럼명 AS 변환하고 싶은 타입)` 과 같이 쓰면 간단히 형변환을 할 수 있습니다.

이렇게 `대출일수_수정` 이라는 칼럼을 만들었습니다. 한번 평균을 구해봅시다!

In [44]:
```
query = """
SELECT ID, 이름, 대출년월
, AVG(CAST(SUBSTR(대출일수, 1, (length(대출일수)-1)) AS INT)) AS 대출일수_평
FROM 도서대출내역2
GROUP BY 1,2,3;
"""
pd.DataFrame(c.execute(query), columns=['ID', '이름', '대출년월', '대출일수_
```

Out[44]:

	ID	이름	대출년월	대출일수_평균
0	101	문강태	2020-06	20.0
1	102	고문영	2020-06	10.0
2	103	문상태	2020-06	8.0
3	104	강기둥	2020-06	3.0

```
SELECT ID, 이름, 대출년월
, AVG(CAST(SUBSTR(대출일수, 1, (length(대출일수)-1)) AS INT)) AS 대
```

```
    출일수_평균
FROM 도서대출내역2
GROUP BY 1,2,3;
```

집계함수 등을 사용할 때는 `GROUP BY` 를 사용해 주는 것을 유념해주세요. 또한 `GROUP BY` 이후에는 칼럼명 또는 숫자를 적어주면 됩니다.

다양한 조건으로 조회하기

함수보다도 더 많이 쓰게 되는 것은 `WHERE` 절에서 쓰는 조건들입니다. 몇가지 조건들을 통해 다양하게 조회해봅시다.

```
SELECT * FROM 도서대출내역2
WHERE
    조건1
AND 조건2
AND 조건3
AND (조건 4 OR 조건5);
```

위와 같이 무수히 많은 조건들을 이어서 원하는 형태로 만들 수 있습니다. 조금 더 디테일하게 가봅시다.

1. 특정 문자를 포함하는 row를 가져오고 싶을 때

먼저 '특정 문자를 포함하는 row를 가져오고 싶을 때'를 확인해봅시다.

In [45]:
```
query = "SELECT * FROM 도서대출내역2 WHERE 이름 LIKE W"문%W""
pd.DataFrame(c.execute(query))
```

Out[45]:

	0	1	2	3
0	101	문강태	2020-06	20일
1	103	문상태	2020-06	8일

```
SELECT * FROM 도서대출내역2
WHERE 이름 LIKE "문%";
```

`LIKE` 는 문자열 칼럼에서 사용할 수 있는 것으로 해당 문자를 포함 또는 해당 문자로 시작 또는 종료하는 것을 불러올 수 있습니다.

2. 특정 기간 혹은 특정 날짜 이전 또는 이후의 row를 가져오고 싶을 때

특정 기간을 가져오고 싶을 때에는 어떻게 해야할까요? 두가지 방법이 있습니다.

1. 첫 번째 -> 그냥 괄호 사용하기

```
SELECT * FROM 도서대출내역
WHERE 대출일 >= "2020-06-01"
AND 대출일 <= "2020-06-07" ;
```

이렇게 간단한 부등호로 결과를 조회할 수 있습니다.

2. `BETWEEN` 사용하기

```
SELECT * FROM 도서대출내역
WHERE 대출일 BETWEEN "2020-06-01" AND "2020-06-07" ;
```

BETWEEN(시작일 AND 종료일) 로 범위를 정해 사용할 수 있습니다. 시작일과 종료일을 포함한다는 점을 기억하세요.

In [47]:
```
query = """
SELECT * FROM 도서대출내역
WHERE 대출일 BETWEEN "2020-06-01" AND "2020-06-07" ;
"""
pd.DataFrame(c.execute(query), columns=col)
```

Out[47]:

	ID	이름	대출년월	대출일수	대출일수_수정
0	101	문강태	aaa	2020-06-01	2020-06-05
1	102	고문영	bbb	2020-06-01	None
2	103	문상태	ccc	2020-06-01	2020-06-05

3. 특정 숫자 이상 또는 이하의 row를 조회하고 싶을 때

그 외에도 부등호를 통해 특정 숫자 이상 또는 이하의 row를 조회하고 싶을 때에도 사용할 수 있습니다.

```
SELECT *
, CAST(SUBSTR(대출일수, 1, (length(대출일수)-1)) AS INT) AS 대출일수
_수정
FROM 도서대출내역2
WHERE 대출일수_수정 > 5 ;
```

우선 대출일수_수정이라는 컬럼을 만들고, 그 밑 조건절에서 '대출일수'가 5일을 초과하는 사람들의 정보를 가져온다는 것을 확인하실 수 있습니다.

In [48]:
```
query = """
SELECT *
, CAST(SUBSTR(대출일수, 1, (length(대출일수)-1)) AS INT) AS 대출일수_수정
FROM 도서대출내역2
WHERE 대출일수_수정 > 5 ;
"""
pd.DataFrame(c.execute(query))
```

Out[48]:

	0	1	2	3	4
0	101	문강태	2020-06	20일	20
1	102	고문영	2020-06	10일	10
2	103	문상태	2020-06	8일	8

4. NULL 조건을 다루는 방법

NULL(None) 조건을 다룰 때에는 어떻게 해야할까요? 테이블을 조회하다보면 NULL만 가져오고 싶거나 NULL을 제외하고 가져오고 싶은 경우가 있습니다. 그럴 때에도 WHERE 조건절을 활용하면 됩니다.

```
SELECT * FROM 도서대출내역
WHERE 반납일 IS NOT NULL;
```

위의 코드에서 NOT 만 제외하면 NULL인 반납일이 NULL인 row를 출력합니다.

In [53]:
```
query = """
SELECT * FROM 도서대출내역
WHERE 반납일 IS NULL;
"""
pd.DataFrame(c.execute(query))
```

Out[53]:

	0	1	2	3	4
0	102	고문영	bbb	2020-06-01	None
1	102	고문영	ddd	2020-06-08	None
2	104	강기둥	None	None	None

JOIN 사용하기

이제 SQL에서 정말 많이 사용하는 join을 알아봅시다.

JOIN 은 집합과 연관지어 이해하면 좋습니다. A집합과 B집합을 나타내는 A테이블과 B테이블이 있다고 생각해봅시다. 어떻게 결합지어 조회할 수 있을까요?

- **INNER JOIN** : A 테이블과 B 테이블의 교집합을 조회
- **LEFT JOIN** : (기준은 A 테이블) A 테이블을 기준으로 해서 B 테이블은 공통되는 부분만 조회
- **RIGHT JOIN** : (기준은 B테이블) B 테이블을 기준으로 해서 A 테이블은 공통되는 부분만 조회
- **FULL JOIN** : A 테이블과 B 테이블 모두에서 빠트리는 부분 없이 모두 조회

그럼 하나하나 살펴보도록 합시다.

위에서 예제로 사용했던 테이블 2개를 활용해 알아봅시다.

In [54]:
```
query = "SELECT * FROM 대출내역"
pd.DataFrame(c.execute(query), columns=['ID', '이름', '도서ID'])
```

Out[54]:

	ID	이름	도서ID
0	101	문강태	aaa
1	102	고문영	bbb
2	102	고문영	fff
3	103	문상태	ccc
4	104	강기둥	None

In [55]:
```
"SELECT * FROM 도서명"
```

```
query = "SELECT * FROM 도서명"
pd.DataFrame(c.execute(query), columns=['도서ID', '도서명'])
```

Out[55]:

	도서ID	도서명
0	aaa	악몽을 먹고 자란 소년
1	bbb	좀비아이
2	ccc	공룡백과사전
3	ddd	빨간구두
4	eee	잠자는 숲속의 미녀

대출내역 테이블과 도서명 테이블입니다. 테이블만 봐도 어떤 도서를 빌렸는지 알 수 있죠?

JOIN 에서 가장 중요한 것은 KEY 를 포착하는 것입니다. **KEY**란 두 테이블을 연결할 수 있는 다리역할을 하는 칼럼입니다. 위에서 보면 도서 ID가 KEY가 되겠네요.

그럼 해당 칼럼을 기준으로 연결해봅시다.

JOIN 기본 구문

```
SELECT 컬럼1, 컬럼2, 컬럼3... FROM A테이블 AS A
{INNER/LEFT/RIGHT/FULL OUTER} JOIN B테이블 AS B
ON A.결합컬럼 = B.결합컬럼
```


INNER JOIN 은 두 테이블의 교집합을 붙이는 방법입니다. 즉, 두 테이블에 모두 있는 정보만 들고 오는 것이죠. 한 쪽이라도 해당 정보가 없다면 조회가 불가능합니다.

직접 조회해봅시다.

In [59]:

```
query = """
SELECT A.*, B.도서명
FROM 대출내역 AS A
INNER JOIN 도서명 AS B
ON A.도서ID = B.도서ID
"""
pd.DataFrame(c.execute(query), columns=['ID', '이름', '도서ID', '도서명'])
```

Out[59]:

	ID	이름	도서ID	도서명
0	101	문강태	aaa	악몽을 먹고 자란 소년
1	102	고문영	bbb	좀비아이
2	103	문상태	ccc	공룡백과사전

```
SELECT A.*, B.도서명
FROM 대출내역 AS A
INNER JOIN 도서명 AS B
```

```
ON A.도서ID = B.도서ID
```

머릿속으로 그려보면, 도서ID를 기준으로 합친다고 했을 때 공통되는 부분은 'aaa, bbb, ccc' 세 개입니다.

따라서 아래와 같이 결과가 나올 것입니다.

2. LEFT JOIN

위와 똑같이 하되, `LEFT JOIN` 으로 한번 해봅시다.

`LEFT JOIN` 은 왼쪽 A 테이블을 기준으로 오른쪽 테이블을 붙이는 것이라고 했었죠? 즉, A 테이블에 있는 데이터는 모두 가져오고 B 테이블과 공통되는 부분만 오른쪽에 붙이게 됩니다.

In [60]:
```
query = """
SELECT A.*, B.도서명
FROM 대출내역 AS A
LEFT JOIN 도서명 AS B
ON A.도서ID = B.도서ID
"""
pd.DataFrame(c.execute(query), columns=['ID', '이름', '도서ID', '도서명'])
```

Out[60]:

	ID	이름	도서ID	도서명
0	101	문강태	aaa	악몽을 먹고 자란 소년
1	102	고문영	bbb	좀비아이
2	102	고문영	fff	None
3	103	문상태	ccc	공룡백과사전
4	104	강기둥	None	None

```
SELECT A.*, B.도서명
FROM 대출내역 AS A
LEFT JOIN 도서명 AS B
ON A.도서ID = B.도서ID
```

기준이 된 대출내역 테이블은 바뀐 부분 없이, 도서명만 오른쪽에 붙은 것을 확인할 수 있습니다. 도서명에 도서ID 'fff'가 없으므로 세 번째 줄은 `NULL` 이 생성되었고, 강기둥의 도서ID는 `NULL` 이므로 `도서명` 도 `NULL` 이 됩니다.

아래의 `RIGHT JOIN` 과 `FULL OUTER JOIN` 은 아래와 같은 에러가 발생합니다.

```
OperationalError: RIGHT and FULL OUTER JOINs are not currently supported
```

현재 SQLite에서는 나머지 두 가지를 지원하지 않는가 봅니다. 하지만 MYSQL이나 다른 RDBMS에서는 위와 같은 `JOIN` 이 지원되므로, 다음의 개념도 같이 숙지해 두시면 좋을 것 같습니다.

3. RIGHT JOIN

```
SELECT B.*, A.ID, A.이름
```

```
FROM 내돌내식 AS A
```