Homework3 보고서

“HY Delivery” App Development

|  |  |
| --- | --- |
| **과목명** | 데이터베이스시스템및응용 |
| **담당 교수님** | 차재혁 교수 |
| **제출일** | 2021년 12월 04일(토요일) |
| **소속** | 한양대학교 공과대학 |
| 컴퓨터소프트웨어학부 |
| **학번** | **이름** |
| 2019009261 | 최가온(CHOI GA ON) |

목차

Ⅰ. Overview

Ⅱ. New Relations / Attributes

1. Relation “address”

2. Relation “cart”

3. Relation “orders”

4. Relation “customer”

Ⅲ. Parsing

1. seller

2. store

3. customer

4. delivery

Ⅳ. SQL Query Design & Execution

Ⅳ-1. Seller(판매자)

1. seller 검색하기

2. seller 정보 변경하기

Ⅳ-2. Store(가게)

1. store 검색하기

2. store 메뉴 보기

3. store 메뉴 추가하기

4. store order 표시하기

5. store 배달중인 주문만 표시하기 / 배달완료된 주문만 표시하기

6. store 주문을 ‘delivering’ 상태로 변경하기

7. store 날짜별 총 주문건수 보이기(일정기간)

8. store 모든 메뉴를 주문한 고객 검색하기

Ⅳ-3. Customer(고객)

1. customer 검색하기

2. customer 주소록 표시하기

3. customer 주소록 추가하기

4. customer 주소록 변경하기

5. customer 주소록 삭제하기

6. customer 결제정보 표시하기

7. customer 결제정보 추가하기(카드)

8. customer 결제정보 추가하기(계좌)

9. customer 결제정보 삭제하기

10. customer 주변의 가게 검색하기

11. customer 주문 중 가게로 선택하기

12. customer 선택중인 가게 메뉴 모두 표시하기

13. customer 카트에 메뉴 넣기(1개)

14. customer 카트에 메뉴 넣기(여러 개)

15. customer 카트 보기

16. customer 카트 비우기(제거)

17. customer 카트 기반으로 결제하기

18. customer 모든 주문 표시하기

19. customer 배달중인 주문만 표시하기

Ⅳ-4. Delivery(배달 대행자)

1. delivery 현재 배달중인 주문 검색하기

2. delivery 현재 배달중인 주문 완료 처리하기

3. delivery 모든 배달 주문 표시하기

(부록)

가. cart 테이블

나. orders 테이블

Ⅰ. Overview

이번 프로젝트의 목적은 판매자, 구매자, 배달대행자 사이를 연결해주는 앱과 서비스를 개발하는 것이다. 주어진 DB를 기반으로 제시되는 시나리오를 만족시키는 앱을 개발하는 것이 가장 큰 목표이다.

이를 위해 python 기반에서 SQL문을 실행시키고 실제 데이터베이스에 이를 반영하는 것들이 수반된다.

구현해야 할 단위들은 아래와 같다.

- 판매자 (seller)

- 가게 (store)

- 구매자 (customer)

- 배달부 (delivery)

프로젝트에서 수반된 주요한 작업은 총 3가지로 아래와 같다.

1) CLI 명령어 인식을 위한 파싱(Parsing) 처리

2) 각각의 명령어에 대응되는 SQL문 작성 및 설계

3) 주어진 스키마 이외에 필요한 Relation, Attribution을 새롭게 정의

결과 출력 시 일정한 형태로 출력하기 위해 아래의 라이브러리를 사용하였다. 모든 DB의 내용들은 SQL문을 통해서만 수정되었다. 외부 라이브러리를 통한 수정은 일체 가하지 않았다.

|  |
| --- |
| pip install tabulate  pip install tabulate[widechars] |

Ⅱ. New Relations / Attributes

이 장에서는 이번 프로젝트를 진행함에 있어 필요에 의해 새롭게 정의한 Relation과 Attribute에 대해 서술한다.

1. Relation “address”

**SQL Sentence**

|  |
| --- |
| CREATE SEQUENCE seq\_address START 1; |
|  |
| CREATE TABLE address ( |
| id INTEGER DEFAULT nextval('seq\_address'), |
| addr VARCHAR(100) NOT NULL, |
| cid INTEGER, |
|  |
| PRIMARY KEY (id), |
| CONSTRAINT fk\_address FOREIGN KEY (cid) REFERENCES customer(id) ON DELETE CASCADE |
| ); |

**Explanation**

Customer에 대해 주소록 기능을 구현해야 한다. 그러나, 기존의 Customer 테이블에는 주소와 관련한 부분이 없다. 이때, 각각의 Customer는 여러 개여 주소를 가질 수 있는 일대다 관계에 해당한다. 그리고, 주소는 추가/변경/삭제 연산이 가능해야 한다.

따라서, 각각의 주소 정보는 독립적인 Entity로 구성하는 것이 적합하다는 생각을 하게 되었고, 별도로 address 테이블을 만들게 되었다.

1. Relation “cart”

**SQL Sentence**

|  |
| --- |
| CREATE SEQUENCE seq\_cart START 1; |
| CREATE TABLE cart ( |
| id INTEGER DEFAULT nextval('seq\_cart'), |
| cid INTEGER, |
| menu\_id INTEGER, |
| menu VARCHAR(100), |
| pcs INTEGER, |
| ordered BOOLEAN DEFAULT FALSE, |
|  |
| PRIMARY KEY (id), |
| CONSTRAINT fk\_cus\_id FOREIGN KEY (cid) REFERENCES customer(id) ON DELETE CASCADE, |
| CONSTRAINT fk\_menu\_id FOREIGN KEY (menu\_id) REFERENCES menu(id) ON DELETE CASCADE |
| ); |

**Explanation**

Customer의 기능 중 한 store를 선택하고, 그 가게에서 제공하는 메뉴를 카트에 담는 기능이 있었다. 또한, 주문의 기능은 Cart에 메뉴를 담는 행위로부터 파생된다는 점에서 테이블을 새롭게 만들어 관리할 필요성이 제기되었다.

customer id, menu id, 메뉴명, 개수(수량), ordered Boolean 형태로 구성하였다. 왜냐하면 cart -c 명령어를 통해 (메뉴명, 개수)의 튜플을 받기 때문이다. 한 메뉴에 한 cart 레코드가 대응되도록 디자인하였다. 맨 마지막의 ordered는 order 테이블과 관련이 있다. 결국, 이 테이블에 있는 정보는 order 테이블의 레코드를 만드는데 사용된다. 다만, “카트에 담기만 한 상황”과 “order가 실제로 이루어진 상황”은 엄연히 구분되어야 하는 사항이다. 그 이유는 아래의 두 가지이다.

1) 어떤 가게에서 “모든 메뉴를 주문한 Customer 찾기” 기능을 수행함에 있어, 주문 상태에 관계없이 모든 주문을 탐색하며 해당 Customer를 찾아야 한다.

2) cart -c를 통해 카트에 메뉴를 넣는 과정과 cart -p를 통해 실제 결제가 이루어져 최종적으로 order가 만들어지는 시점 사이에는 시간 차이가 존재한다. 따라서 그 사이동안 어떤 것들을 넣었는지, 그 사이에 어떤 메뉴가 주문 처리되었는지는 세밀히 기록되어야 할 필요가 있다.

1. Relation “orders”

**SQL Sentence**

|  |
| --- |
| CREATE SEQUENCE seq\_order START 1; |
|  |
| CREATE TABLE orders ( |
| id INTEGER DEFAULT nextval('seq\_order'), |
| sid INTEGER, |
| cid INTEGER, |
| did INTEGER DEFAULT NULL, |
| menu\_info jsonb, |
| payment VARCHAR(100), |
| otime TIMESTAMP WITHOUT TIME ZONE DEFAULT (now() at time zone 'Asia/Seoul'), |
| dtime TIMESTAMP WITHOUT TIME ZONE DEFAULT null, |
| cphone VARCHAR(20), |
| status VARCHAR(15) DEFAULT 'pending', |
|  |
| PRIMARY KEY (id), |
| FOREIGN KEY (sid) REFERENCES store(id) ON DELETE CASCADE, |
| FOREIGN KEY (cid) REFERENCES customer(id) ON DELETE CASCADE, |
| FOREIGN KEY (did) REFERENCES delivery(id) ON DELETE CASCADE |
| ); |

**Explanation**

Customer에 대해 주소록 기능을 구현해야 한다. 그러나, 기존의 Customer 테이블에는 주소와 관련한 부분이 없다. 이때, 각각의 Customer는 여러 개여 주소를 가질 수 있는 일대다 관계에 해당한다. 그리고, 주소는 추가/변경/삭제 연산이 가능해야 한다.

따라서, 각각의 주소 정보는 독립적인 Entity로 구성하는 것이 적합하다는 생각을 하게 되었고, 별도로 address 테이블을 만들게 되었다.

otime과 dtime은 각각 고객이 결제한 시점, 배달이 완료되어 ‘delivered’ 상태가 된 시점을 말한다. orders 테이블의 레코드는 기본적으로 고객이 결제한 시점에 생성된다. 따라서, otime의 기본값은 현재의 시간으로 설정하게 되었다. 마찬가지로, 실시간 정보를 반영하기 위해 dtime의 경우에도 배달원이 ‘배달완료’ 처리를 하는 시점의 시간으로 설정하게 된다.

주문의 상태는 아래와 같은 과정을 거친다.

pending 🡪 delivering 🡪 delivered

즉, 초기 단계에서는 주문이 ‘pending’ 상태이므로 status의 기본값을 ‘pending’으로 설정하였다.

1. Relation “customer”

**SQL Sentence**

|  |
| --- |
| ALTER TABLE customer |
| ADD COLUMN searching\_store INT |
| DEFAULT NULL; |

**Explanation**

customer테이블의 경우 기존의 상태에서 스키마 확장을 진행하였다. store 명령어를 통해 카트에 담을 대상에 해당하는 가게를 설정해야 하는 부분이 있다. 즉, 각각의 customer는 자신이 현재 선택하고 있는 store의 정보가 기록이 되어야 한다. 왜냐하면 customer가 cart -c를 통해 물건을 담는 시점과 cart -p를 이용하여 실제 결제가 이루어지고 order 레코드가 생성되기까지의 시점은 서로 차이가 있기 때문에 그 사이에서 해당 정보를 저장하여야 한다. searching\_store는 해당 customer가 선택한 store의 id이다.

Ⅲ. Parsing

이 장에서는 seller, customer, store, delivery 등 각각의 파일에서 명령어를 받을 때 파싱하는 방식에 대해 서술한다.

각각의 .py 파일에서는 argparse.ArgumentParser 객체를 이용하여 파싱을 진행한다. 이 객체는 main 함수 내에 생성되어 있으며, 이 객체를 파라미터로 넘겨 파싱 트리를 구성할 함수를 각각 구현하였다. 기본적으로 “parsing\_XX” 형태의 이름으로 구성하였다.

1. seller

**Parsing Strategy**

|  |
| --- |
| def parsing\_seller(parser:argparse.ArgumentParser):  sub\_parsers = parser.add\_subparsers(dest='function')   *# info* parser\_info = sub\_parsers.add\_parser('info')  parser\_info.add\_argument('id', type=int)   *# update* parser\_update = sub\_parsers.add\_parser('update')  parser\_update.add\_argument('id', type=int)  parser\_update.add\_argument('attr', type=str, choices=['name', 'phone', 'local', 'domain', 'passwd'])  parser\_update.add\_argument('data', type=str) |

**Explanation**

* ‘info’ 명령어의 경우, id값 하나만 받는다.
* ‘update’ 명령어의 경우 id값, 바꿀 attr 대상, 바꿀 attr 값의 3가지 인자를 받는다. 이때 attr은 ‘name’, ‘phone’, ‘local’, ‘domain’, ‘passwd’의 5가지 값 중 하나만을 받도록 구성하였다. 그리고, 어떤 attr의 값을 바꿀지는 사용자에게 정해져 있으므로, data의 타입은 str로 정하였다.

2. store

**Parsing Strategy**

|  |
| --- |
| def parsing\_store(parser:argparse.ArgumentParser):  sub\_parsers = parser.add\_subparsers(dest='function')   *# info* parser\_info = sub\_parsers.add\_parser('info')  parser\_info.add\_argument('id', type=int)   *# menu* parser\_menu = sub\_parsers.add\_parser('menu')  parser\_menu.add\_argument('id', type=int)   *# add\_menu* parser\_amenu = sub\_parsers.add\_parser('add\_menu')  parser\_amenu.add\_argument('id', type=int)  parser\_amenu.add\_argument('menu', type=str)   *# order* parser\_order = sub\_parsers.add\_parser('order')  parser\_order.add\_argument('id', type=int)  parser\_order.add\_argument('status', type=str.lower, choices=['pending', 'delivering', 'delivered', '0', '1', '2'], nargs='?')   *# update\_order* parser\_update\_order = sub\_parsers.add\_parser('update\_order')  parser\_update\_order.add\_argument('id', type=int)  parser\_update\_order.add\_argument('order\_idx', type=int)   *# stat* parser\_stat = sub\_parsers.add\_parser('stat')  parser\_stat.add\_argument('id', type=int)  parser\_stat.add\_argument('start\_date', type=str)  parser\_stat.add\_argument('days', type=int)   *# search* parser\_search = sub\_parsers.add\_parser('search')  parser\_search.add\_argument('id', type=int) |

**Explanation**

* ‘info’의 경우, store의 id값 하나만 받는다.
* ‘menu’의 경우에도 위와 마찬가지로 store의 id값 하나만 받는다.
* ‘add\_menu’의 경우, store의 id값과 메뉴명에 해당하는 menu를 받는다.
* ‘order’의 경우, store의 id값과 status값을 받는다. 이때, 위에서 서술하였듯 status는 세 가지의 값이 있는데 이를 숫자로 받을 수도 있고 정해진 문자열로 받을 수도 있다. 다만, status값은 꼭 받아야하는 부분은 아니다. 주문 조회 시 status값에 따라 필터링을 하는 것이기에 status값이 명시될 때도 있고 되지 않을 때도 있다. 따라서, 이 부분은 nargs=’?’ 를 설정하여 구현하였다.
* ‘update\_order’의 경우, store의 id값과 ‘delivering’ 상태로 바꿀 order 레코드의 id값에 해당하는 order\_idx를 받는다.
* ‘stat’의 경우, store의 id값과 검색에서 시작일에 해당하는 start\_date와 시작일부터 탐색할 일수에 해당하는 days를 인자로 받는다.
* ‘search’의 경우, store의 id값 하나만 인자로 받는다.

3. customer

**Parsing Strategy**

|  |
| --- |
| def parsing\_customer(parser:argparse.ArgumentParser):  sub\_parsers = parser.add\_subparsers(dest='function')   *# info* parser\_info = sub\_parsers.add\_parser('info')  parser\_info.add\_argument('id', type=int)   *# address* parser\_address = sub\_parsers.add\_parser('address')  parser\_address.add\_argument('id', type=int)  parser\_address\_mode = parser\_address.add\_mutually\_exclusive\_group()  parser\_address\_mode.add\_argument('-c', '--create')  parser\_address\_mode.add\_argument('-e', '--edit', nargs=2)  parser\_address\_mode.add\_argument('-r', '--remove')   *# pay* parser\_pay = sub\_parsers.add\_parser('pay')  parser\_pay.add\_argument('id', type=int)  parser\_pay\_mode = parser\_pay.add\_mutually\_exclusive\_group()  parser\_pay\_mode.add\_argument('--add-card', type=str)  parser\_pay\_mode.add\_argument('--add-account', nargs=2)  parser\_pay\_mode.add\_argument('-r', '--remove', type=int)   *# search* parser\_search = sub\_parsers.add\_parser('search')  parser\_search.add\_argument('id', type=int)  parser\_search.add\_argument('-a', action='store\_true')  parser\_search.add\_argument('-o', type=int, default=0)  parser\_search.add\_argument('-l', type=int, default=10)   *# store* parser\_store = sub\_parsers.add\_parser('store')  parser\_store.add\_argument('id', type=int)  parser\_store.add\_argument('sid', type=int)   *# cart* parser\_cart = sub\_parsers.add\_parser('cart')  parser\_cart.add\_argument('id', type=int)  parser\_cart\_mode = parser\_cart.add\_mutually\_exclusive\_group()  parser\_cart\_mode.add\_argument('-c', type=int, nargs='+')  parser\_cart\_mode.add\_argument('-p', type=int)  parser\_cart\_mode.add\_argument('-r', action='store\_true')  parser\_cart\_mode.add\_argument('-l', action='store\_true')   *# list* parser\_list = sub\_parsers.add\_parser('list')  parser\_list.add\_argument('id', type=int)  parser\_list.add\_argument('-w', '--waiting', action='store\_true') |

**Explanation**

* ‘info’의 경우, customer의 id값 하나만 받는다.
* ‘address’의 경우, customer의 id값과, 아래의 3가지 flag 중 하나를 인자로 받는다. 3개의 옵션 중 하나만 받아야 하기에 exclusive group으로 선언하여 구성하였다.
  + -c, --create

추가할 address값 하나만 받는다.

* + -e, --edit

수정할 address의 인덱스 값과 수정할 address값을 받는다.

인자의 개수가 2개인데 타입을 각각 설정할 수 없어, 해당 인자가 사용되는 시점에서 적절히 체크할 수 있는 기능을 제공하였다.

* + -r, --remove

삭제할 address의 인덱스 값을 받는다.

* ‘pay’의 경우, customer의 id값과, 아래의 3가지 flag 중 하나를 인자로 받는다. 마찬가지로 3개의 옵션 중 하나만 받아야 하기에 exclusive group으로 선언하여 구성하였다.
  + --add\_card

추가할 카드번호 1개를 인자로 받는다.

* + --add-account
    - 1부터 19사이의 숫자에 해당하는 bid와 추가할 계좌번호를 인자로 받는다.
  + -r, --remove
    - 삭제할 결제 정보의 인덱스 값을 인자로 받는다.
* ‘search’의 경우, customer의 id값과, 아래의 3가지 flag를 받을 수 있다. 3개 모두 필수 옵션은 아니기 때문에 각각의 argument로 만들었다.
  + -a: 현재 주문 가능한 가게만 나타낼 것인지의 여부를 나타내므로 True/False에 해당하는 인자로 만들었다.
  + -o: 0, 1, 2 중 하나의 값을 가지며 정렬의 기준을 결정한다.
  + -l: 나타낼 가게의 수를 나타내며, 기본값은 10이다.
* ‘store’의 경우, customer의 id값과 store의 id값을 각각 받는다.
* ‘cart’의 경우, customer의 id값과 아래의 4가지 중 하나의 값을 받는다. 4개 중 하나만 받아야 하므로 exclusive group으로 구성하였다.
  + -c: 카트에 추가할 (menu, pcs)의 튜플을 계속하여 받는다. 몇 개를 받을지는 가변적이기 때문에 nargs=’?’을 넣었다. 각각의 값들이 유효한지, 그리고 전체적으로 짝수개의 입력을 받았는지는 이 값을 쓰는 함수 내에서 검사하도록 구현하였다.
  + -p: 결제를 할 때에 쓰는 인자이다. 이때에는 결제정보 인덱스 값 하나를 받는다.
  + -r: 카트의 모든 정보를 삭제하고 초기화한다.
  + -l: 현재 카트에 있는 정보를 출력할 때 쓰인다. 위와 마찬가지로 ‘store\_true’ 옵션을 사용하였다.
* ‘list’의 경우, customer의 id값과, 아래 한 가지를 선택적으로 받는다.
  + -w, --waiting: 주문 중 그것의 상태가 ‘delivering’인 것만 받는다. 이는 True/False 중 하나의 값에 해당되므로 ‘store\_true’ 옵션을 사용하였다.

4. delivery

**Parsing Strategy**

|  |
| --- |
| def parsing\_delivery(parser:argparse.ArgumentParser):  sub\_parsers = parser.add\_subparsers(dest='function')   *# status* parser\_status = sub\_parsers.add\_parser('status')  parser\_status.add\_argument('id', type=int)   parser\_status\_mode = parser\_status.add\_mutually\_exclusive\_group()  parser\_status\_mode.add\_argument('-e', type=int)  parser\_status\_mode.add\_argument('-a', action='store\_true') |

**Explanation**

* ‘status’만 첫 인자로 받는다.
  + 위의 경우에 대해 첫 번째 인자로는 delivery의 id값을 받는다.
  + 위의 경우에 대해 두 번째 인자로는 아래의 2개 중 하나의 값을 인자로 받는다. 2개 중 하나만을 받아야 하므로, exclusive group으로 구성하였다.
    - -e: 주문의 status를 ‘delivered’로 바꿀 order의 id값을 받는다.
    - -a: 모든 배달 주문을 표시할 것인지의 여부를 나타내므로 True/False에 해당하는 ‘store\_true’를 옵션으로 사용하였다.

Ⅳ. SQL Query Design & Execution

**IV – 1. Seller (판매자)**

**1. seller 검색하기**

**Format**

python seller.py info ID

**Function**

ㆍshow\_seller\_info

ㆍshow\_seller\_from\_table

**Implementation**

ㆍshow\_seller\_info

패스워드를 제외한 seller의 정보를 보이는 것이 이 기능의 핵심이다. 따라서, 특정 id에 해당하는 seller를 추출하기 위한 SQL문을 이 함수에서 수행한다. 해당하는 row를 인자로 넘겨 show\_seller\_from\_table 함수를 호출한다.

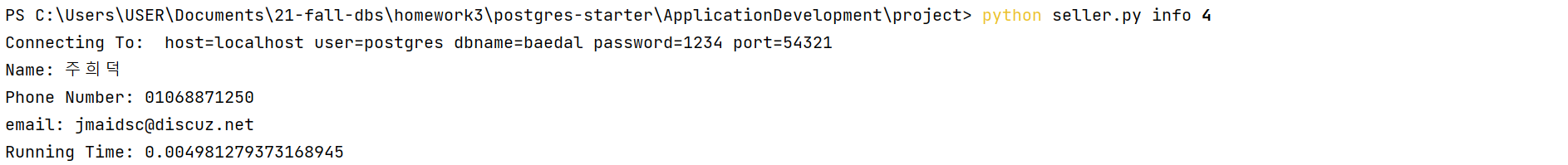
|  |
| --- |
| def show\_seller\_info(args):  *# TODO* try:  cur = conn.cursor()  sql = "SELECT \* FROM seller WHERE id={id};".format(id=args.id)  cur.execute(sql)  rows = cur.fetchall()  if not rows:  print("Given seller ID doesn't exist.")  return  for row in rows:  show\_seller\_from\_table(row)   except Exception as err:  print(err) |

ㆍshow\_seller\_from\_table

이 함수는 row를 입력으로 받아, 받은 인자를 정해진 형식에 맞추어 출력하는 함수이다. 따라서 실제 터미널 창에 나오는 결과를 출력하는 함수는 show\_seller\_from\_table 함수이다.

|  |
| --- |
| def show\_seller\_from\_table(row):  print("Name: {name}".format(name = row[1]))  print("Phone Number: {phone}".format(phone = row[2]))  print("email: {local}@{domain}".format(local=row[3], domain = row[4])) |

**Result**



**2. seller 정보 변경하기**

**Format**

python seller.py update ID [property] [VALUE]

**Function**

ㆍmodify\_seller\_info

**Implementation**

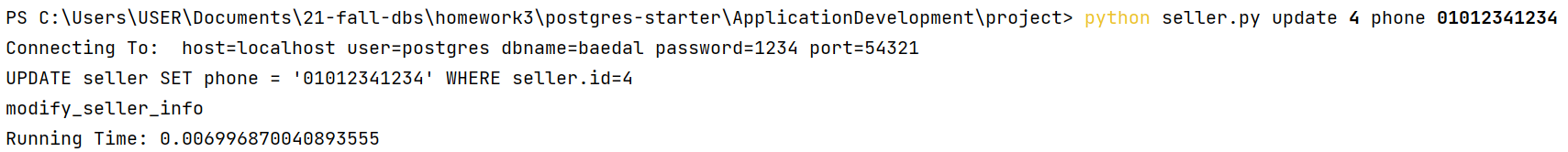
ㆍmodify\_seller\_info

UPDATE문을 통하여 seller의 정보를 수정한다. 바꾸고자 하는 대상은 args.attr에, 바꾸고자 하는 목표값은 args.data에 저장되어 있다.

|  |
| --- |
| def modify\_seller\_info(args):  *# TODO* try:  cur = conn.cursor()  sql = "UPDATE seller " \  "SET {attr} = \'{data}\' " \  "WHERE seller.id={id}".format(attr=args.attr, data=args.data, id=int(args.id))  print(sql)  cur.execute(sql)  conn.commit()   except Exception as err:  print(err)  conn.rollback()  print("modify\_seller\_info") |

**Result**

아래와 같이 id=4인 seller의 phone을 ‘01012341234’로 변경한다.



변경된 결과가 반영되었음을 알 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**IV – 2. Store (가게)**

**1. store 검색하기**

**Format**

python store.py info ID

**Function**

ㆍshow\_store\_info

ㆍshow\_store\_from\_table

**Implementation**

ㆍshow\_store\_info

인자로 받은 id값에 해당하는 store를 SELECT문을 이용하여 찾는다. 이후 해당하는 튜플을 인자로 넘겨 show\_store\_from\_table 함수를 호출한다.

|  |
| --- |
| def show\_store\_info(args):  *# TODO* try:  cur = conn.cursor()  sql = "SELECT \* FROM store WHERE id=%(id)s;"  cur.execute(sql, {"id": args.id})  rows = cur.fetchall()  for row in rows:  show\_store\_from\_table(row)  except Exception as err:  print(err) |

ㆍshow\_seller\_from\_table

이 함수는 row를 입력으로 받아, store에 대한 정보를 일정한 형식을 갖추어 출력하는 함수이다. 아래의 예시 결과에서 볼 수 있듯이, 스케줄에 대한 정보를 표시하는 것을 포함한다.

|  |
| --- |
| def show\_store\_from\_table(row):  print("Name: {id}".format(id = row[2]))  print("Location: lat {lat} | lng {lng}".format(lat = row[3], lng = row[4]))  print("Address: {addr}".format(addr = row[1]))  print("Phone Number: {phone}".format(phone = row[5]))  print("Schedules: ")  print("\t|day| |open| |closed|")  for i in range(len(row[6])):  if(row[6][i]['holiday'] == False):  print("\t%-8s %-9s %-9s" % (row[6][i]['day'], time\_form(row[6][i]['open']), time\_form(row[6][i]['closed'])))  else:  print("\t%-8s H O L I D A Y" % (row[6][i]['day']))   print("Seller (id): {sid}".format(sid = row[7])) |

**Result**

텍스트이(가) 표시된 사진

자동 생성된 설명

**2. store 메뉴 보기**

**Format**

python store.py menu ID

**Function**

ㆍshow\_menu\_info\_store

**Implementation**

ㆍshow\_menu\_info\_store

인자로 받은 id값에 해당하는 store의 메뉴 정보를 출력하는 함수이다. WHERE절에 id값 조건을 명시하고 menu 테이블로부터 해당 정보를 가져왔다. 이후, tabulate 라이브러리를 사용하여 일정한 형식을 갖추어 메뉴를 표시하도록 구현하였다.

|  |
| --- |
| def show\_menu\_info\_store(args):  *# TODO* try:  cur = conn.cursor()  sql = "SELECT id, menu FROM menu WHERE sid=%(id)s;"  cur.execute(sql, {"id": args.id})  rows = cur.fetchall()  print("Menu of Store {sid}".format(sid=args.id))  print(tb.tabulate(rows, headers=['Menu ID', 'Name']))  except Exception as err:  print(err) |

**Result**

텍스트이(가) 표시된 사진

자동 생성된 설명

**3. store 메뉴 추가하기**

**Format**

python store.py add\_menu ID [NAME]

**Function**

ㆍadd\_menu\_into\_store

**Implementation**

ㆍadd\_menu\_into\_store

인자로 받은 id값에 대응되는 store에 menu를 추가하는 함수이다. INSERT문을 이용하여 menu 테이블에 새로운 메뉴 정보 레코드를 삽입한다. 이 부분은 DB가 수정되는 부분에 해당하므로 commit과 rollback을 이용하여 에러 발생 시 복구처리가 되도록 구현하였다.

|  |
| --- |
| def add\_menu\_into\_store(args):  *# TODO* try:  cur = conn.cursor()  sql = "INSERT INTO menu (menu, sid) " \  "VALUES (\’{menu}\’, {sid})".format(menu=args.menu, sid=args.id)  print(sql)  cur.execute(sql)  conn.commit()  except Exception as err:  print(err)  conn.rollback()  else:  print("adding menu success!") |

**Result**

아래와 같이 id=2인 store에 ‘pasta’라는 메뉴를 추가하였다.

텍스트, 안테나, 스크린샷이(가) 표시된 사진

자동 생성된 설명

메뉴가 추가된 것이 제대로 반영되었음을 확인할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**4. store order 표시하기**

**Format**

python store.py order ID

**Function**

ㆍshow\_order\_info\_store

**Implementation**

ㆍshow\_order\_info\_store

인자로 받은 id값에 해당하는 store가 받은 주문을 orders 테이블에서 가져온다. 가져오는 정보는 id, cid, otime, status이다. 아무 인자가 없는 경우에는 if args.status is None 부분으로 흐름이 진행된다. 여기에서는 모든 주문을 출력한다.

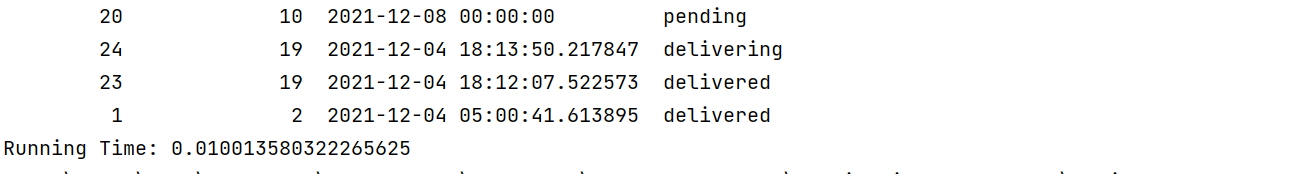
|  |
| --- |
| def show\_order\_info\_store(args):  *# TODO* try:  cur = conn.cursor()  sql=str()  if args.status is None:  sql = "SELECT id, cid, otime, status FROM orders WHERE sid={sid};".format(sid=args.id)  print("ALL orders for Store {sid}".format(sid=args.id))  cur.execute(sql)  rows = cur.fetchall()  print(tb.tabulate(rows, headers=['Order ID', 'Customer ID', 'OTime', 'Status']))  return  elif args.status == '0' or args.status == 'pending':  sql = "SELECT id, cid, otime FROM orders WHERE sid={sid} and status=\'pending\';".format(sid=args.id)  print("Pending orders for Store {sid}".format(sid=args.id))  elif args.status == '1' or args.status == 'delivering':  sql = "SELECT id, cid, otime FROM orders WHERE sid={sid} and status=\'delivering\';".format(sid=args.id)  print("Delivering orders for Store {sid}".format(sid=args.id))  else:  sql = "SELECT id, cid, otime FROM orders WHERE sid={sid} and status=\'delivered\';".format(sid=args.id)  print("Delivered orders for Store {sid}".format(sid=args.id))  cur.execute(sql)  rows = cur.fetchall()  print(tb.tabulate(rows, headers=['Order ID', 'Customer ID', 'OTime']))   except Exception as err:  print(err)   print("show\_order\_info\_store") |

**Result**

아래는 id=5인 store의 주문을 검색한 결과이다. 주문 상태에 관계없이 모든 주문 정보가 나옴을 알 수 있다.

테이블이(가) 표시된 사진

자동 생성된 설명



**5. store 배달중인 주문만 표시하기 / 배달완료된 주문만 표시하기**

**Format**

python store.py order ID [FILTER]

**Function**

ㆍshow\_order\_info\_store

**Implementation**

ㆍshow\_order\_info\_store

* 0 또는 ‘pending’을 입력받은 경우 🡪 준비중인 주문만 출력
* 1 또는 ‘delivering’을 입력받은 경우 🡪 배달중인 주문만 출력
* 2 또는 ‘delivered’을 입력받은 경우 🡪 배달 완료된 주문만 출력

**Result**

1) ‘pending’(주문 준비중) 상태인 주문만 출력하는 경우

테이블이(가) 표시된 사진

자동 생성된 설명



2) ‘delivering’(주문 배달중) 상태인 주문만 출력하는 경우

텍스트이(가) 표시된 사진

자동 생성된 설명

3) ‘delivered’(주문 배달완료) 상태인 주문만 출력하는 경우

텍스트이(가) 표시된 사진

자동 생성된 설명

**6. store 주문을 ‘delivering’ 상태로 변경하기**

**Format**

python store.py update\_order ID ORDER\_ID

**Function**

ㆍupdate\_order\_store

**Implementation**

ㆍupdate\_order\_store

sql0에서는 order의 인덱스에 해당하는 주문의 가게 id를 확인한다. 만약 그 id 자체가 존재하지 않을 경우 아이디가 유효하지 않다는 경고 메시지를 출력하고 종료한다. 만약 해당 id가 존재는 하는데, 명시된 가게의 주문이 아닐 경우 접근이 불가능하다는 경고 메시지를 출력하고 종료한다.

sql1에서는 가게의 위치 정보를 받아와서 각각 latitude, longitude에 저장한다.

sql2에서는 위에서 받아온 위치 정보를 기반으로 가게로부터 가장 가까운 배달대행자 1명을 뽑는다. 그리고, 그 배달원의 id값을 delivery\_id에 저장한다.

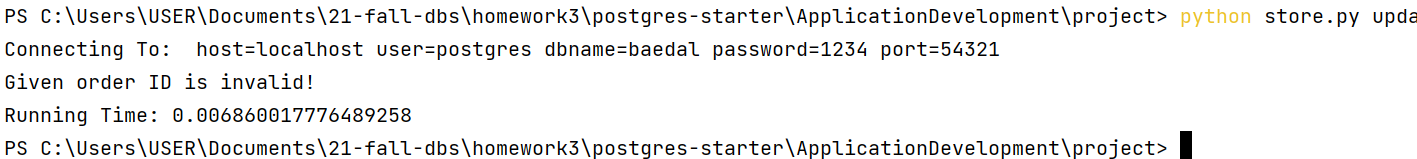
sql3에서는 해당하는 주문 레코드에 배달대행자의 id값을 넣고, 주문 상태를 기존의 ‘pending’에서 ‘delivering’으로 갱신한다.

sql4에서는 주문을 배정받은 배달대행자의 stock을 1만큼 증가시킨다.

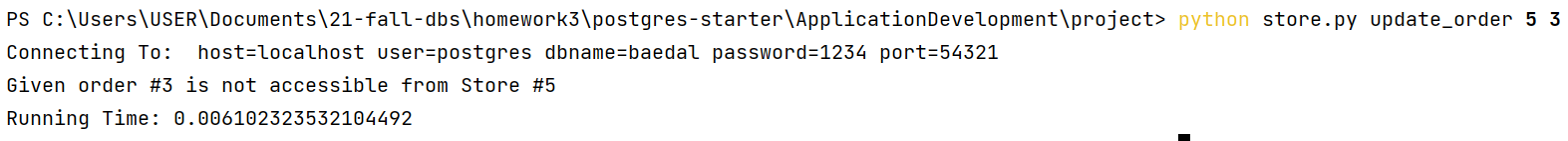
|  |
| --- |
| def update\_order\_store(args):  *# TODO* try:  cur = conn.cursor()  *# check validity of order-store relationship* sql0 = "SELECT sid FROM orders WHERE id={oid}".format(oid=args.order\_idx)  cur.execute(sql0); tmp\_id = cur.fetchone()  if tmp\_id is None:  print("Given order ID is invalid!")  return  if tmp\_id[0] != args.id:  print("Given order #{oid} is not accessible from Store #{sid}".format(oid=args.order\_idx, sid=args.id))  return   *# fetch store's location info* sql1 = "SELECT lat, lng " \  "FROM customer WHERE id = {sid};".format(sid=args.id)  cur.execute(sql1)  info\_ = cur.fetchone()  latitude, longitude = info\_[0], info\_[1]   *# get the closest deliver from the given store* sql2 = "SELECT d.id " \  "FROM delivery d " \  "WHERE d.stock <= 4 " \  "ORDER BY power(({lat}-d.lat), 2) + power(({lng}-d.lng), 2) " \  "LIMIT 1;".format(lat=latitude, lng=longitude)  cur.execute(sql2)  delivery\_id = (cur.fetchone())[0] *# delivery id* print("Closest deliver is found: Deliver #{did}".format(did=delivery\_id))   *# update order record, status: pending->delivering* sql3 = "UPDATE orders SET did = {did}, status = \'delivering\' " \  "WHERE id={order\_id}".format(did=delivery\_id, order\_id=args.order\_idx)  cur.execute(sql3)  conn.commit()   *# increment stock of corresponding deliver* sql4 = "UPDATE delivery SET stock = stock + 1 WHERE id={did}".format(did=delivery\_id)  cur.execute(sql4)  conn.commit()   except Exception as err:  print(err)  conn.rollback()  else:  print("update\_order\_store") |

**Result**

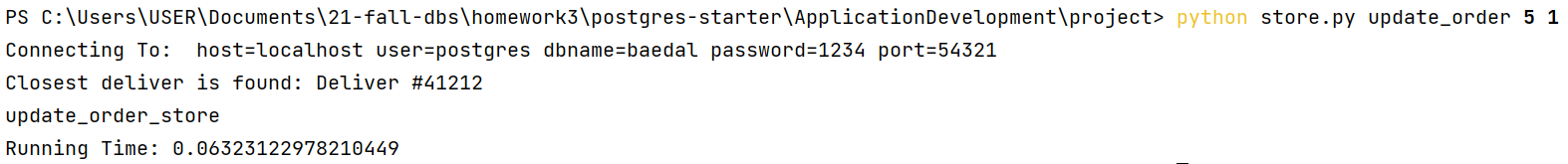
1) 해당 주문의 id값이 유효하지 않은 경우



2) 해당 주문이 가게의 주문이 아닌 경우



3) 위의 두 조건을 제대로 만족하는 경우



성공적으로 배달이 배달대행자에게 접수되었고, 배정된 배달대행자의 id값을 확인할 수 있다. 이 예시에서는 id=41212인 배달대행자에게 주문이 배정되었음을 알 수 있다.**7. store 날짜별 총 주문건수 보이기(일정기간)**

**Format**

python store.py stat ID START\_DATE DAYS

**Function**

ㆍstat\_info\_store

**Implementation**

ㆍstat\_info\_store

인자로 받은 날짜값을 파싱하여 각각 y, m, d 즉 년, 월, 일의 값을 받는다. 이것을 기반으로 orders 테이블에서 주문의 개수를 표시를 하는데 GROUP BY를 이용하여 1일 단위로 표시되게끔 구성하였다.

|  |
| --- |
| def stat\_info\_store(args):  *# TODO* try:  cur = conn.cursor()  y, m, d = (args.start\_date).split('/')  sql = "SELECT otime::date as Date, COUNT(\*) as Orders " \  "FROM orders " \  "WHERE sid={sid} and otime::date >= \'{year}/{month}/{day}\'::date and otime::date < \'{year}/{month}/{day}\'::date + interval \'{interval} day\' " \  "GROUP BY otime::date;".format(sid=args.id, year=y, month=m, day=d, interval=args.days)   cur.execute(sql)  rows = cur.fetchall()  print("STAT info of Store {sid}".format(sid=args.id))  print(tb.tabulate(rows, headers=['Date', 'Orders']))  except Exception as err:  print(err) |

**Result**

아래와 같이 id=5인 store의 2021-12-04부터 총 5일간의 주문기록을 열람할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

\* 날짜 범위 중 주문이 들어오지 않은 날에 대해서는 출력을 하지 않는다. 위의 예시에서는 2021년 12월 7일에는 아무런 주문이 들어오지 않은 상태인 것이다.

**8. store 모든 메뉴를 주문한 고객 검색하기**

**Format**

python store.py search StoreID

**Function**

ㆍsearch\_info\_store

**Implementation**

ㆍsearch\_info\_store

cart 테이블에서 cid값과 menu\_id을 뽑은 결과를 Table 1이라 하고, menu 테이블에서 해당 가게의 모든 menu\_id을 뽑은 결과를 Table 2라 하자. 그렇다면, “모든 메뉴를 주문한 사람 검색”이라는 문제는 Table 1과 Table 2 사이의 Relational Division으로 생각해볼 수 있다.

Nested Query 방식으로 해당 부분을 구하였고, end\_query에는 모든 메뉴를 주문한 customer의 id값이 들어있다. 해당 id값에 대응되는 고객의 이름까지 출력을 하기 위해 해당 부분은 Cartesian product를 이용하여 해결하였다.

|  |
| --- |
| def search\_info\_store(args):  *# TODO* try:  cur = conn.cursor()  sql = \  "SELECT DISTINCT cid, name " \  "FROM ( " \  "SELECT \* FROM " \  "( " \  "SELECT DISTINCT cid, menu\_id FROM cart WHERE menu\_id IN (SELECT id FROM menu WHERE sid={sid}) " \  ")sx " \  "WHERE NOT EXISTS ( " \  "(SELECT p.id FROM (SELECT id FROM menu WHERE sid={sid})p) " \  "EXCEPT " \  "(SELECT sp.menu\_id FROM ( " \  "SELECT DISTINCT cid, menu\_id FROM cart WHERE menu\_id IN (SELECT id FROM menu WHERE sid={sid}) " \  ")sp " \  "WHERE sp.cid = sx.cid " \  ") " \  ") " \  ")end\_query, customer " \  "WHERE customer.id = end\_query.cid".format(sid=args.id)  cur.execute(sql)  vip\_customer = cur.fetchall()  print("VIP LIST of STORE {sid}".format(sid=args.id))  print(tb.tabulate(vip\_customer, headers=['Customer ID', 'Customer Name']))   except Exception as err:  print(err) |

**Result**

아래는 id=2인 store의 모든 메뉴를 주문한 고객을 검색한 결과이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

참고) orders 테이블에는 아래와 같이 id=2, id=10인 customer가 모든 메뉴를 주문했음을 알 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**IV – 3. Customer (고객)**

**1. customer 검색하기**

**Format**

python customer.py info ID

**Function**

ㆍshow\_customer\_info

ㆍshow\_customer\_from\_table

**Implementation**

ㆍshow\_customer\_info

id에 해당하는 customer의 레코드를 SELECT문으로 불러와서 그것을 show\_customer\_from\_table 함수의 인자로 넣어 그 함수를 호출한다.

|  |
| --- |
| def show\_customer\_info(args):  *# TODO* try:  cur = conn.cursor()  sql = "SELECT \* FROM customer WHERE id=%(id)s;"  cur.execute(sql, {"id": args.id})  rows = cur.fetchall()  for row in rows:  show\_customer\_from\_table(row)  except Exception as err:  print(err) |

ㆍshow\_customer\_from\_table

customer 레코드 하나를 받아 일정한 형태를 갖추어 customer의 정보를 출력하는 함수이다.

|  |
| --- |
| def show\_customer\_from\_table(row):  print("Info of Customer {id}".format(id = row[0]))  print()  print("Name: {name}".format(name = row[1]))  print("Phone Number: {phone}".format(phone = row[2]))  print("email: {local}@{domain}".format(local=row[3], domain = row[4])) |

**Result**

id=2인 customer를 검색한 결과이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**2. customer 주소록 표시하기**

**Format**

python customer.py address ID

**Function**

ㆍshow\_address\_info\_customer

**Implementation**

ㆍshow\_address\_info\_customer

id에 해당하는 customer의 주소 정보를 address 테이블에서 가져온다. 만약, address 정보가 하나도 없는 store가 있다면 그 부분에 대해서는 관련한 메시지를 출력하고 종료한다.

주소 정보가 최소 1개 이상 있을 경우, 그것을 모두 출력한다.

|  |
| --- |
| def show\_address\_info\_customer(args):  *# TODO* try:  cur = conn.cursor()  sql = "SELECT addr FROM address WHERE cid={id};".format(id=args.id)  cur.execute(sql)  rows = cur.fetchall(); rows = sorted(rows)  i = 1  if (len(rows) == 0):  print("Customer id={id} has no address yet.".format(id=args.id))  return   print("Address of Customer {id}".format(id=args.id))  print(tb.tabulate(rows, headers=['index', 'Address'], showindex=True))  except Exception as err:  print(err)  else:  print("show\_address\_info\_customer") |

**Result**

아래는 id=2인 customer의 주소 정보를 조회한 결과이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

(\* 테스트를 진행하기 전, 2가지의 주소를 미리 삽입해두었다.)

**3. customer 주소록 추가하기**

**Format**

python customer.py address ID -c [TEXT]

**Function**

ㆍadd\_address\_customer

**Implementation**

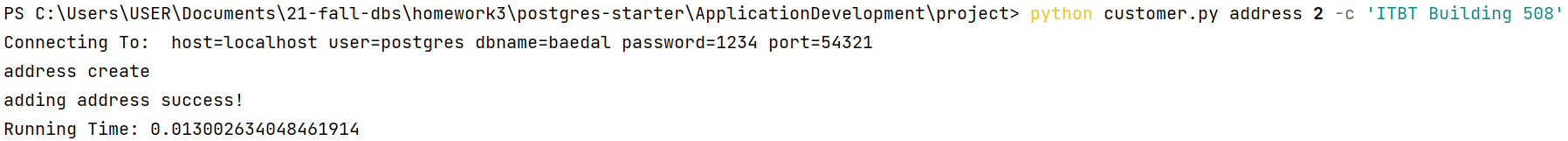
ㆍadd\_address\_customer

INSERT문을 이용하여 새로운 address 레코드를 만들어 그것을 address 테이블에 삽입한다. 이 부분은 DB에 변화를 가져오므로 commit과 rollback을 통해 에러 발생 시 복구될 수 있도록 구성하였다.

|  |
| --- |
| def add\_address\_customer(args):  *# TODO* try:  cur = conn.cursor()  sql = "INSERT INTO address " \  "(addr, cid) " \  "VALUES (\'{addr}\', {cid});".format(addr=args.create, cid=args.id)  cur.execute(sql)  conn.commit()   except Exception as err:  print(err)  conn.rollback()  else:  print("adding address success!") |

**Result**

아래의 예시에서는 id=2인 customer에 대해 ‘ITBT Building 508’이라는 새로운 주소 정보를 추가한 결과이다.



추가한 정보가 제대로 삽입되었음을 확인할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**4. customer 주소록 변경하기**

**Format**

python customer.py address ID -c IDX [TEXT]

**Function**

ㆍedit\_address\_customer

**Implementation**

ㆍedit\_address\_customer

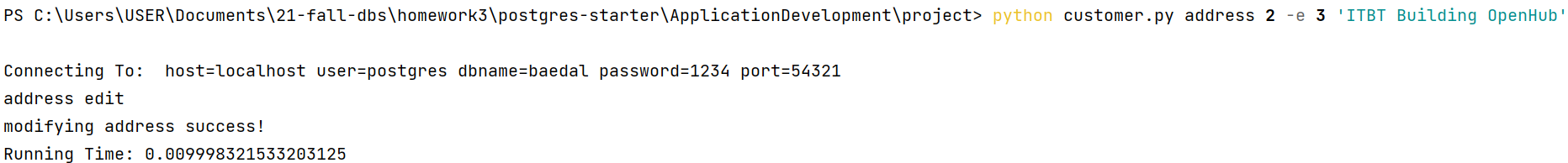
sql1에서는 입력받은 id에 해당하는 customer가 가지고 있는 모든 주소 정보를 address 테이블에서 가져온다. 입력을 받은 인덱스 숫자가 범위를 넘어가면 관련한 경고 메시지를 출력하고 종료한다. 범위에 대한 검사가 끝난 후, 입력받은 새로운 주소로 해당 주소를 변경하고 그것을 실제 DB에 반영한다.

마찬가지로 이 부분은 DB에 수정이 이뤄지는 부분이기에, commit과 rollback을 통해 문제 발생 시 복구가 가능하도록 구성하였다.

|  |
| --- |
| def edit\_address\_customer(args):  *# TODO* try:  cur = conn.cursor()  sql1 = "SELECT \* " \  "FROM address " \  "WHERE cid={cid};".format(cid=args.id)  cur.execute(sql1)  rows = cur.fetchall(); rows = sorted(rows)   if int(args.edit[0]) > len(rows) or int(args.edit[0]) - 1 < 0:  print("Index out of Range!")  return   addr\_id = rows[int(args.edit[0]) - 1][0]  sql2 = "UPDATE address " \  "SET addr = \'{addr}\' " \  "WHERE id={addr\_id};".format(addr=args.edit[1], addr\_id=addr\_id)  cur.execute(sql2)  conn.commit()   except Exception as err:  print(err)  conn.rollback()  else:  print("modifying address success!") |

**Result**

아래의 예시에서는 id=2에 추가한 3번째 주소 ‘ITBT Building 508’을 ‘ITBT Building OpenHub’로 변경하는 예시이다.



실제로 변경된 사항이 반영되었음을 확인할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

\* 이때 인덱스 값을 넘겨줄 때에는 1부터 세는 방식으로 넘겨야 한다.

그림에서 나온 index는 tabulate에서 지정한 숫자이다.

**5. customer 주소록 삭제하기**

**Format**

python customer.py address ID -r IDX

**Function**

ㆍdelete\_address\_customer

**Implementation**

ㆍdelete\_address\_customer

sql1에서는 주소를 변경할 때와 마찬가지로 해당 id을 가진 store의 모든 주소 정보를 탐색한 후 인덱스의 범위 체크를 수행한다. 범위를 넘을 경우 관련한 경고 메시지를 출력한 후 종료한다.

범위 조건이 만족할 경우, 입력 받은 인덱스 값에 해당하는 주소 정보를 삭제하고 commit을 통해 실제 DB에 삭제된 내용을 반영한다.

|  |
| --- |
| def delete\_address\_customer(args):  *# TODO* try:  cur = conn.cursor()  sql1 = "SELECT \* FROM address WHERE cid={cid}".format(cid=args.id)  cur.execute(sql1)  rows = cur.fetchall(); rows = sorted(rows)   idx = int(args.remove)  if idx > len(rows) or idx - 1 < 0:  print("Index out of Range!")  return   addr\_id = rows[idx - 1][0]  sql2 = "DELETE FROM address " \  "WHERE id={addr\_id}".format(addr\_id=addr\_id)  cur.execute(sql2)  conn.commit()   except Exception as err:  print(err)  conn.rollback()  else:  print("deleting address success!") |

**Result**

아래의 예시는 이전에 추가한 3번째 주소인 ‘ITBT Building OpenHub’을 삭제하는 예시이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**6. customer 결제정보 표시하기**

**Format**

python customer.py pay ID

**Function**

ㆍshow\_pay\_info\_customer

**Implementation**

ㆍshow\_pay\_info\_customer

패스워드를 제외한 seller의 정보를 보이는 것이 이 기능의 핵심이다. 따라서, 특정 id에 해당하는 seller를 추출하기 위한 SQL문을 이 함수에서 수행한다. 해당하는 row를 인자로 넘겨 show\_seller\_from\_table 함수를 호출한다.

|  |
| --- |
| def show\_pay\_info\_customer(args):  *# TODO* try:  cur = conn.cursor()  sql1 = "SELECT payments FROM customer WHERE id={cid}".format(cid=args.id)  cur.execute(sql1)  rows = cur.fetchall();  rows = sorted(rows)   print("Index | Customer ID | Payment")  print("-" \* 70)  i = 1  for pay in rows[0][0]:  if pay['type'] == 'account':  pay\_ = "bid={bid}, acc\_num={acc\_num}, type={type}".format(  bid=(pay['data'])['bid'], acc\_num=(pay['data'])['acc\_num'], type=pay['type']  )  print("%-10s%-16s%s" % (i, args.id, pay\_))  else:  pay\_ = "card\_num={card}, type={type}".format(  card=(pay['data'])['card\_num'], type=pay['type']  )  print("%-10s%-16s%s" % (i, args.id, pay\_))  i += 1   except Exception as err:  print(err)  else:  print("show\_pay\_info\_customer") |

**Result**

아래의 예시는 id=2인 customer의 결제정보를 표시한 결과이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**7. customer 결제정보 추가하기(카드)**

**Format**

python customer.py pay ID –add-card [CARD\_NUM]

**Function**

ㆍadd\_card\_pay\_customer

**Implementation**

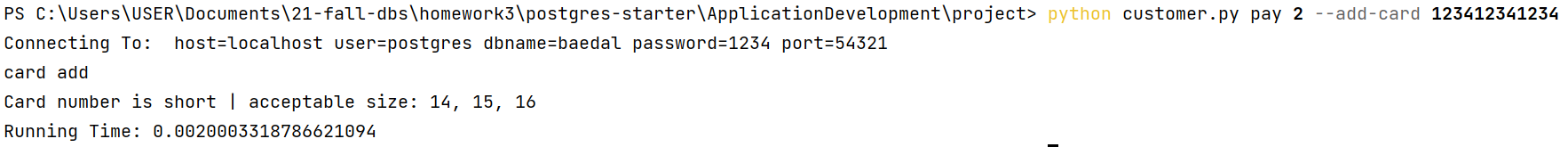
ㆍadd\_card\_pay\_customer

첫 번째로 길이가 14 이상, 16 이하인지를 체크하고, 이 조건을 만족하지 않으면 경고 메시지를 출력하고 종료한다. 두 번째로 입력받은 카드번호가 모두 숫자로 이루어져 있는지의 여부를 검사하고, 이 조건을 만족하지 않으면 경고 메시지를 출력하고 종료한다.

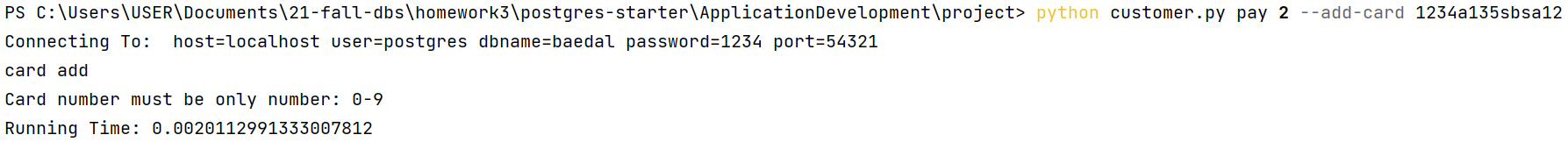
|  |
| --- |
| def add\_card\_pay\_customer(args):  *# TODO* try:  if len(str(args.add\_card)) < 14 or len(str(args.add\_card)) > 16:  print("Card number is short | acceptable size: 14, 15, 16")  return  if not (args.add\_card).isdigit():  print("Card number must be only number: 0-9")  return  card\_info = {"data": {"card\_num": None}, "type": "card"}  (card\_info["data"])["card\_num"] = args.add\_card  print("New payment : ", card\_info)   cur = conn.cursor()  sql1 = "SELECT payments FROM customer WHERE id={cid}".format(cid=args.id)  cur.execute(sql1)  rows = cur.fetchall()  rows = sorted(rows)   rows[0][0].append(card\_info)  new\_row = str(rows[0][0]).replace("\'", "\"")   sql2 = "UPDATE customer " \  "SET payments = \'{pay}\' " \  "WHERE id={cid}".format(pay=new\_row, cid=args.id)  cur.execute(sql2)  conn.commit()   except Exception as err:  print(err)  conn.rollback()   else:  print("add\_card\_pay\_customer") |

**Result**

아래는 카드번호가 ‘123412341234’일 때의 모습이다. 길이 조건을 만족하지 못하므로 경고 메시지를 출력한다.



아래는 카드번호가 ‘1234a135sbsa12’일 때의 모습이다. 길이 조건은 만족하였으나 형식 조건을 만족하지 못하므로 경고 메시지를 출력한다.



아래는 카드번호가 ‘123412341234123’일 때의 모습이다. 길이 조건과 형식 조건 모두 만족하므로 제대로 삽입됨을 알 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**8. customer 결제정보 추가하기(계좌)**

**Format**

python customer.py pay ID –add-account bid [ACC\_NUM]

**Function**

ㆍadd\_acc\_pay\_customer

**Implementation**

ㆍadd\_acc\_pay\_customer

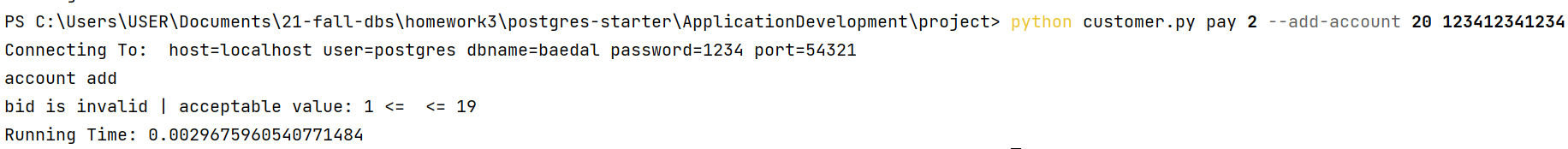
계좌정보를 추가하는 함수이다. 먼저 bid의 값이 1 이상 19 이하인지를 체크하고 해당 조건을 만족하지 못하는 경우 경고 메시지를 출력하고 종료한다.

dict 타입의 acc\_info를 생성하고 해당 내용을 모두 인자로 적절하게 채운 후 sql1에서는 입력 받은 id에 대응되는 customer에서 결제 정보를 추출한 후, 가장 끝부분에 acc\_info를 추가한다. 그러나, JSON에서는 double quote를 사용하므로 그 부분을 적절하게 바꿔준 후, 그것을 UPDATE문을 이용하여 결제정보를 갱신한다.

|  |
| --- |
| def add\_acc\_pay\_customer(args):  *# TODO* try:  if int(args.add\_account[0]) < 1 or int(args.add\_account[0]) > 19:  print("bid is invalid | acceptable value: 1 <= <= 19")  return   acc\_info = {'data': {'bid': None, 'acc\_num': None}, 'type': 'account'}  (acc\_info['data'])['bid'] = args.add\_account[0]  (acc\_info['data'])['acc\_num'] = args.add\_account[1]  print("New payment : ", acc\_info)   cur = conn.cursor()  sql1 = "SELECT payments " \  "FROM customer " \  "WHERE id={cid}".format(cid=args.id)  cur.execute(sql1)  rows = cur.fetchall(); rows = sorted(rows)   rows[0][0].append(acc\_info)   *# JSON uses double quote, not single quote* new\_row = str(rows[0][0]).replace("\'", "\"")   sql2 = "UPDATE customer " \  "SET payments = \'{pay}\' " \  "WHERE id={cid}".format(pay=new\_row, cid=args.id)  cur.execute(sql2)  conn.commit()   except Exception as err:  print(err)  conn.rollback()  else:  print("add\_acc\_pay\_customer") |

**Result**

아래는 bid가 유효하지 않은 경우의 예시이다.



bid가 유효하면 아래와 같은 결과를 출력한다. 해당 부분이 제대로 DB에 반영되었음을 확인할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**9. customer 결제정보 삭제하기**

**Format**

python customer.py pay ID -r IDX

**Function**

ㆍremove\_pay\_customer

**Implementation**

ㆍremove\_pay\_customer

sql1에서는 범위 체크를 수행하며, sql2에서는 인덱스 값과 대응되는 실제 결제 정보를 지우고 그것을 DB에 반영한다.

|  |
| --- |
| def remove\_pay\_customer(args):  *# TODO* try:  cur = conn.cursor()  sql1 = "SELECT payments FROM customer WHERE id={cid}".format(cid=args.id)  cur.execute(sql1)  rows = cur.fetchall();  rows = sorted(rows)  ROWS = rows[0][0]   if args.remove <= 0 or args.remove > len(ROWS):  print("Index Out of Range!")  return   del ROWS[args.remove - 1]   ROWS = str(ROWS).replace("\'", "\"")   sql2 = "UPDATE customer " \  "SET payments = \'{pay}\' " \  "WHERE id={cid}".format(pay=ROWS, cid=args.id)  cur.execute(sql2)  conn.commit()   except Exception as err:  print(err)  conn.rollback()  else:  print("remove\_pay\_customer") |

**Result**

텍스트이(가) 표시된 사진

자동 생성된 설명

**10. customer 주변의 가게 검색하기**

**Format**

python customer.py search ID [-a|-o|-l] [ARGS]

**Function**

ㆍsearch\_stores\_customer

**Implementation**

ㆍsearch\_stores\_customer

- sql\_에서는 선택된 customer의 위치 정보를 추출한다.

- -a, -o, -l의 입력에 따라 where\_clause와 order\_clause를 적절히 구성한다.

- 3개의 서브쿼리를 UNION하는 방식으로 구성하였다.

i) holiday = false이면서 현재 근무시간에 해당하는 가게

ii) holiday = true인 가게는 일괄적으로 휴무로 표시

iii) 24:30 등 이상치의 경우 24시간 영업으로 간주

|  |
| --- |
| def search\_stores\_customer(args):  *# TODO* try:  cur = conn.cursor()   *# fetch location info of customer* sql\_ = "SELECT lat, lng FROM customer WHERE id={cid}".format(cid=args.id)  cur.execute(sql\_)  data = cur.fetchone(); lat, lng = data[0], data[1]   where\_clause = str()  order\_clause = str()  if args.a is True:  where\_clause = " "  else:  where\_clause = "WHERE open\_closed = true"   if args.o == 0:  order\_clause = "ORDER BY sname"  elif args.o == 1:  order\_clause = "ORDER BY address collate \"C\""  elif args.o == 2:  order\_clause = "ORDER BY distance"   sql = \  "SELECT id, sname, address, distance, open\_time, close\_time, open\_closed " \  "FROM ( " \  "SELECT sq1.id, sname, address, distance, open\_time::time as open\_time, close\_time::time as close\_time, ((holiday::bool=false) and ((open\_time::time < (now() at time zone \'Asia/Seoul\')::time) and (close\_time::time > (now() at time zone \'Asia/Seoul\')::time))) or open\_time::time >= close\_time::time as open\_closed " \  "FROM ( " \  "SELECT id, sname, address, (power({latitude}-lat, 2)+power({longitude}-lng, 2)) as distance, datas->>\'holiday\' AS holiday, datas->>\'open\' AS open\_time, datas->>\'closed\' AS close\_time " \  "FROM ( " \  "SELECT jsonb\_array\_elements(schedules) AS datas, id, sname, address, lat, lng " \  "FROM store " \  ")s " \  "WHERE (datas->>\'day\')::int={weekday} " \  ")sq1 " \  "WHERE not open\_time::int > 2400 and not close\_time::int > 2400 " \  "UNION " \  "SELECT id, sname, address, distance, open\_time::time as open\_time, close\_time::time as close\_time, false as open\_closed " \  "FROM ( " \  "SELECT id, sname, address, (power({latitude}-lat, 2)+power({longitude}-lng, 2)) as distance, datas->>\'open\' AS open\_time, datas->>\'closed\' AS close\_time " \  "FROM ( " \  "SELECT jsonb\_array\_elements(schedules) AS datas, id, sname, address, lat, lng " \  "FROM store " \  ")s " \  "WHERE (datas->>\'day\')::int={weekday} and (datas->>\'holiday\')::bool = true " \  ")sq2 " \  "UNION " \  "SELECT id, sname, address, distance, null as open\_time, null as close\_time, true as open\_closed " \  "FROM ( " \  "SELECT id, sname, address, (power({latitude}-lat, 2)+power({longitude}-lng, 2)) as distance, datas->>\'open\' AS open\_time, datas->>\'closed\' AS close\_time " \  "FROM ( " \  "SELECT jsonb\_array\_elements(schedules) AS datas, id, sname, address, lat, lng " \  "FROM store " \  ")s " \  ")sq3 " \  "WHERE open\_time::int > 2400 or close\_time::int > 2400 " \  ")big\_query " \  "{where} " \  "{orderby} " \  "LIMIT {limit};".format(  where=where\_clause, orderby=order\_clause, limit=args.l, latitude=lat, longitude=lng,  weekday=datetime.datetime.today().weekday()  )   cur.execute(sql)  row = cur.fetchall()  print("-----\* SEARCH RESULT \*-----")  print(tb.tabulate(row, headers=['id', 'sname', 'address', 'distance', 'opening', 'closing', 'open/closed'], showindex=True))   except Exception as err:  print(err)  else:  print("search\_stores\_customer") |

**Result**

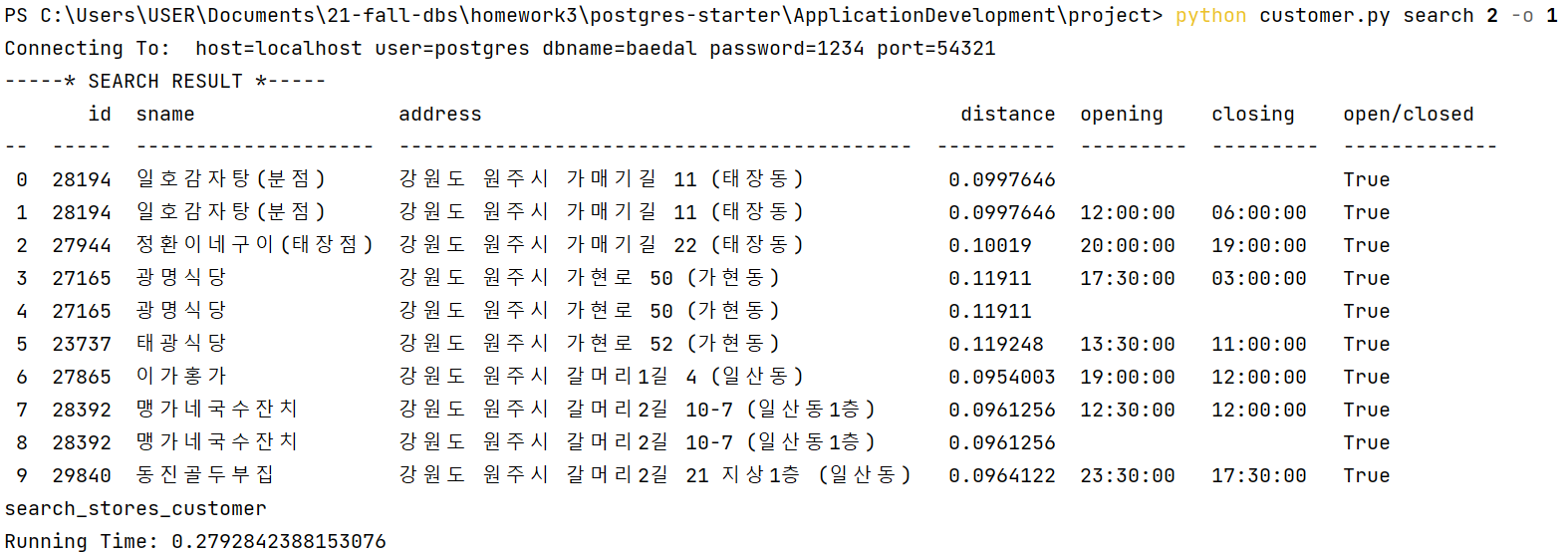
\* 해당 테스트는 2021년 12월 04일 토요일 오후 3시 50분 ~ 4시 00분 사이에 실행한 결과입니다.

1) 아무런 인자를 넘겨주지 않았을 때에는 상위 10개, 이름기준으로 정렬된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

2) -o 1을 명시하면 주소 기준으로 정렬된다.

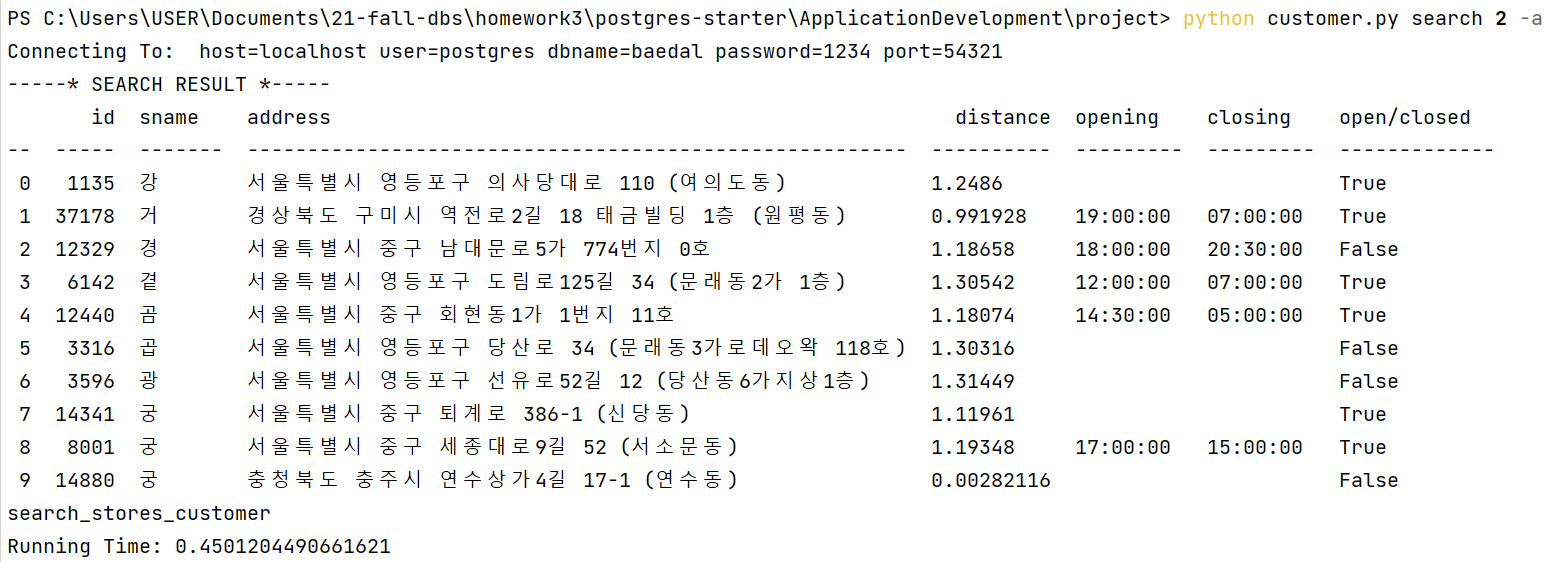


3) -o 2을 명시하면 거리 기준으로 정렬된다.

테이블이(가) 표시된 사진

자동 생성된 설명

4) -a를 명시하면 현재 영업중이지 않은 가게도 표시된다.



\* opening과 closing이 공백인 가게의 경우 해당 영업일이 휴일인 경우에 해당한다.

5) -l을 이용하면 표시할 가게의 개수를 지정할 수 있다.

테이블이(가) 표시된 사진

자동 생성된 설명

6) 과제 명세의 예시를 그대로 입력하여 3개의 조건을 모두 명시했을 때의 결과는 아래와 같다.

테이블이(가) 표시된 사진

자동 생성된 설명

**11. customer 주문 중 가게로 선택하기**

**Format**

python customer.py store ID SID

**Function**

ㆍset\_store\_customer

**Implementation**

ㆍset\_store\_customer

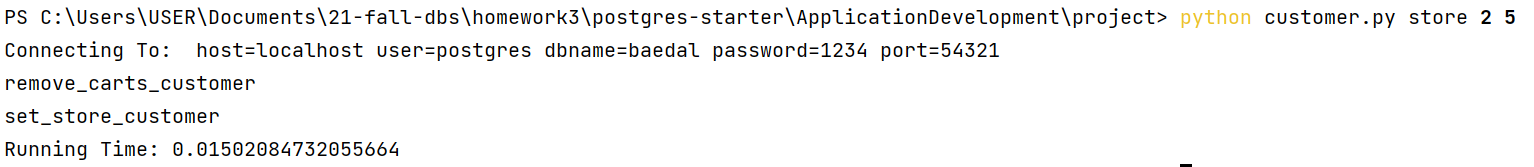
기본적으로 customer가 설정한 가게에 대한 정보는 customer 테이블의 searching\_store에 저장되어 있다. 먼저 sql\_에서는 이 정보를 뽑아서 그것이 None이 아닌 어떤 값이 쓰였다면, remove\_carts\_customer 함수를 호출하여 카트 내부를 완전히 비우고 시작한다. (searching\_store가 그전에 어떤 값으로 쓰였다면 cart에 다른 store의 메뉴가 들어있을 가능성이 있으므로.)

이후 받은 sid 값을 해당하는 customer의 searching\_store에 UPDATE문을 이용하여 갱신한다. 이 부분은 DB에 직접적으로 수정을 가하는 부분이기 때문에 commit과 rollback을 이용하여 문제가 발생할 경우 복구할 수 있도록 구현하였다.

|  |
| --- |
| def set\_store\_customer(args):  *# TODO* try:  cur = conn.cursor()  sql\_ = "SELECT searching\_store FROM customer WHERE id = {id}".format(id=args.id)  cur.execute(sql\_)   searching\_store = cur.fetchone()[0]  if (searching\_store != None):  remove\_carts\_customer(args)   sql = "UPDATE customer " \  "SET searching\_store = {sid} " \  "WHERE id = {id};".format(sid=args.sid, id=args.id)  cur.execute(sql)  conn.commit()  except Exception as err:  print(err)  conn.rollback()  else:  print("set\_store\_customer") |

**Result**

id=2인 customer가 id=5인 가게를 선택하는 것을 나타낸 것이다.



**12. customer 선택중인 가게 메뉴 모두 표시하기**

**Format**

python customer.py cart ID

**Function**

ㆍshow\_menu\_info\_store

**Implementation**

ㆍshow\_menu\_info\_store

sql\_에서는 해당하는 customer가 선택한 store의 id를 불러온다. 그리고 그 가게에서 제공하는 모든 메뉴 정보를 불러와서 출력한다. 이때, 메뉴가 아예 없거나 store id가 유효하지 않은 경우 관련한 경고 메시지를 출력한다.

|  |
| --- |
| def show\_menu\_info\_store(args):  *# TODO* try:  cur = conn.cursor()  sql\_ = "SELECT searching\_store FROM customer WHERE id={id};".format(id=args.id)  cur.execute(sql\_)   sid = int((cur.fetchall())[0][0])  sql = "SELECT \* FROM menu WHERE sid=%(id)s;"  cur.execute(sql, {"id": sid})  rows = cur.fetchall()  if rows:  print("Menu of store {id}".format(id= sid))  print("---------------------------------------------")  else: print("NOOOOOO~")   idx = 1  for row in rows:  print("{index}. Menu ID: {id}, Name: {name}".format(index=idx, id=row[0], name=row[1])); idx+=1  print("---------------------------------------------")  except Exception as err:  print(err) |

**Result**

텍스트이(가) 표시된 사진

자동 생성된 설명

**13. customer 카트에 메뉴 넣기(1개)**

**Format**

python customer.py cart 2 -c p1 c1

**Function**

ㆍadd\_carts\_customer

ㆍadd\_cart\_customer

**Implementation**

이 부분을 구현함에 있어서, 한 개의 메뉴를 장바구니에 넣는 것과 여러 개의 메뉴를 한꺼번에 장바구니에 넣는 것은 서로 같은 경로로 동작하도록 구현했다. 그러나, 엄연히 과제 명세에서는 둘을 나누고 있으므로 따로 서술하였다.

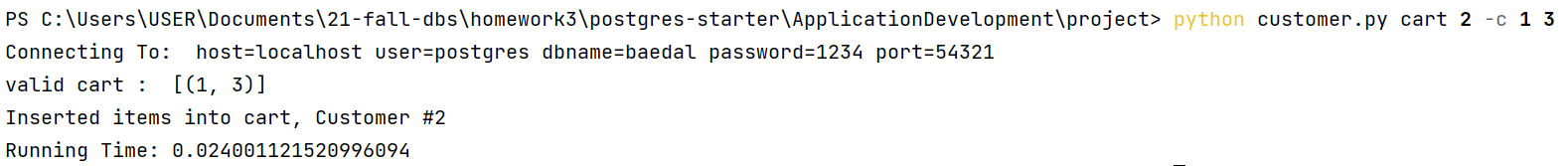
ㆍadd\_carts\_customer

사용자로부터 받은 (메뉴명, 개수)의 튜플을 모두 받아 유효성을 검사하는 함수이다. 14번 카트에 메뉴 넣기(여러 개)에서 세부 사항을 서술하였다.

ㆍadd\_cart\_customer

각각의 유효한 튜플에 대해 cart 레코드를 만드는 함수이다. 14번 카트에 메뉴 넣기(여러 개)에서 세부 사항을 서술하였다.

**Result**



**14. customer 카트에 메뉴 넣기(여러 개)**

**Format**

python customer.py cart 2 -c p1 c1 p2 c2 …

**Function**

ㆍadd\_carts\_customer

ㆍadd\_cart\_customer

**Implementation**

ㆍadd\_carts\_customer

사용자로부터 받은 (메뉴명, 개수)의 튜플을 모두 받아 유효성을 검사하는 함수이다. 사용자의 입력에는 오류가 있을 가능성이 있다고 전제하고 진행한다. valid\_cart는 그중에서 유효하다고 판단된 (메뉴명, 개수) 튜플을 저장한다. 먼저 들어온 인자의 개수가 아예 없거나 짝수가 아닐 경우에는 관련한 경고 메시지를 출력하고 종료한다.

sql\_에서는 대응되는 customer가 선택하고 있는 가게의 id를 탐색하고 그 가게의 id를 통해 그 가게에서 제공하는 모든 메뉴 정보를 sql에서 탐색한다. 각각의 튜플들에 대해 “개수조건, 인덱스조건 불충분”, “개수조건만 불충분”, “인덱스조건 불충분”의 3가지를 검사하고 조건이 맞지 않으면 그대로 버리고, 이 3개 테스트를 모두 통과한 경우에 대해서만 valid\_cart에 append한다.

마지막으로는 남은 valid\_cart의 튜플들에 대해 add\_cart\_customer 함수를 호출한다.

|  |
| --- |
| def add\_carts\_customer(args):  menus = list()  valid\_cart = list()   *# check the validity (format)* if len(args.c) == 0 or len(args.c) % 2 != 0:  print("the number of argument: must be even!")  return *# terminate all things   # TODO* try:  cur = conn.cursor()  sql\_ = "SELECT searching\_store FROM customer WHERE id={id};".format(id=args.id)  cur.execute(sql\_)  sid\_ = (cur.fetchall())[0][0]  if sid\_ is None:  print("No store is selected.")  return  sid = int(sid\_) *# searching store* sql = "SELECT \* FROM menu WHERE sid=%(id)s;"  cur.execute(sql, {"id": sid})  menus = cur.fetchall()  except Exception as err:  print(err)  else:  for i in range(len(args.c) // 2):  idx = args.c[2 \* i];  pcs = args.c[2 \* i + 1]  A = idx <= 0 or idx > len(menus); B = pcs <= 0  if A and B: *# if invalid* print("INVALID - idx={i} | pcs={j} (idx|pcs)".format(i=idx, j=pcs))  elif A:  print("INVALID - idx={i} | pcs={j} (idx)".format(i=idx, j=pcs))  elif B:  print("INVALID - idx={i} | pcs={j} (pcs)".format(i=idx, j=pcs))  else: valid\_cart.append((idx, pcs))  print("valid cart : ", valid\_cart)   for \_ in range(len(valid\_cart)):  add\_cart\_customer(args, menus, valid\_cart[\_]) |

ㆍadd\_cart\_customer

이 함수에서는 cart 테이블에 전달받은 (메뉴명, 개수) 정보를 레코드 하나 규모로 삽입하는 과정이 이뤄진다.

|  |
| --- |
| def add\_cart\_customer(args, menus, valid\_cart):  idx, pcs = valid\_cart[0], valid\_cart[1]   *# TODO* try:  cur = conn.cursor()  sql = "INSERT INTO cart " \  "(cid, menu\_id, menu, pcs) " \  "VALUES ({cid}, {menuid}, \'{menu}\', {pcs});".format(  cid=args.id, menuid=menus[idx - 1][0], menu=menus[idx - 1][1], pcs=pcs  )  cur.execute(sql)  conn.commit()  except Exception as err:  print(err)  conn.rollback()  else:  print("Inserted items into cart, Customer #{cid}".format(cid=args.id)) |

**Result**

아래의 예시에서는 id=2인 customer가 1번 메뉴 0개, 2번 메뉴 8개, 3번 메뉴 4개, 4번 메뉴 1개, 5번 메뉴 10개, 6번 메뉴 -3개를 넣는 과정이다.

상식에 의거할 때, 1번 메뉴는 개수조건에 맞지 않고, 5번 메뉴는 인덱스조건에 맞지 않고, 6번 메뉴는 인덱스조건과 개수조건 모두 맞지 않다. 이 세 가지의 모순 케이스에 대해 각각 경고 메시지를 다르게 출력하는 방식으로 구현하여 사용자가 잘못된 부분을 인지할 수 있도록 하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**15. customer 카트 보기**

**Format**

python customer.py cart 2 -l

**Function**

ㆍshow\_carts\_customer

**Implementation**

ㆍshow\_carts\_customer

이 함수에서는 입력받은 id에 대응되는 customer의 cart를 탐색하는 기능을 수행한다. cart 테이블에서 각각의 (메뉴명, 개수) 튜플을 일정한 형태로 한줄씩 출력한다.

|  |
| --- |
| def show\_carts\_customer(args):  *# TODO* try:  cur = conn.cursor()   sql = "SELECT menu, pcs " \  "FROM cart " \  "WHERE cid={cid};".format(cid=args.id)  cur.execute(sql)  rows = cur.fetchall()   print("CART of Customer #{cid}".format(cid=args.id))  print(tb.tabulate(rows, headers=['Menu', 'PCS']))   except Exception as err:  print(err)   else:  print("show\_carts\_customer") |

**Result**

id=2인 customer의 장바구니를 탐색하는 모습이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**16. customer 카트 비우기(제거)**

**Format**

python customer.py cart ID -r

**Function**

ㆍremove\_carts\_customer

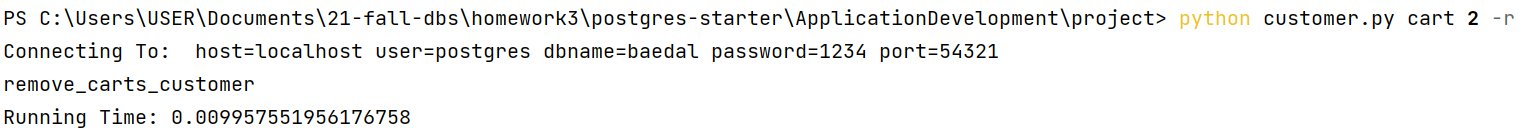
**Implementation**

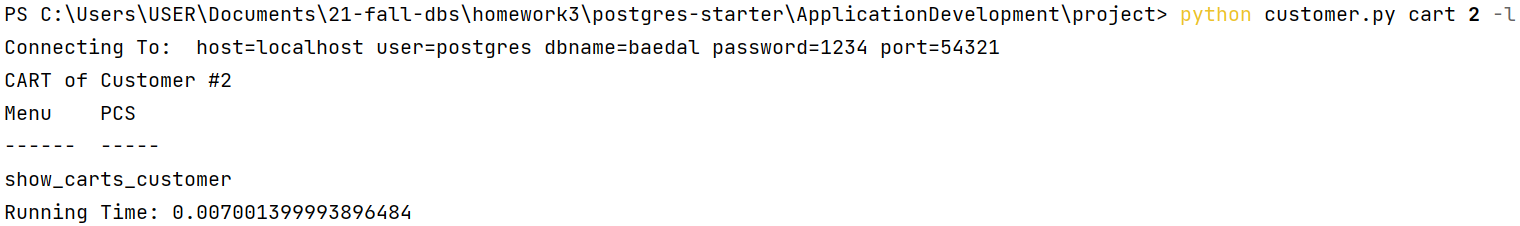
ㆍremove\_carts\_customer

입력받은 id값에 해당하는 customer의 장바구니를 모두 비우는 함수이다. cart 테이블에서 cid 정보를 이용하여 해당 customer가 가지고 있는 모든 장바구니 메뉴를 지운다. 다만, “모든 메뉴를 주문한 customer를 찾는 기능”을 구현할 때에는 cart 정보를 모두 탐색을 하기 때문에 ordered가 false인 것을 대상으로 삭제를 진행한다. 이후, 그 customer의 searching\_store의 값을 NULL로 설정함으로써, 보고 있는 가게 정보 또한 지운다.

|  |
| --- |
| def remove\_carts\_customer(args):  *# TODO* try:  cur = conn.cursor()  sql = "DELETE FROM cart WHERE cid={cid} and ordered=false;".format(cid=args.id)  cur.execute(sql)   sql\_ = "UPDATE customer " \  "SET searching\_store = NULL " \  "WHERE id = {cid};".format(cid=args.id)  cur.execute(sql\_)  conn.commit()   except Exception as err:  print(err)  conn.rollback()  else:  print("remove\_carts\_customer") |

**Result**





**17. customer 카트 기반으로 결제하기**

**Format**

python customer.py cart ID -p IDX

**Function**

ㆍapprove\_payment\_customer

**Implementation**

ㆍapprove\_payment\_customer

sql1에서는 입력받은 id값에 해당하는 customer의 결제정보, 현재 선택중인 가게정보, 휴대폰 번호를 추출한다.

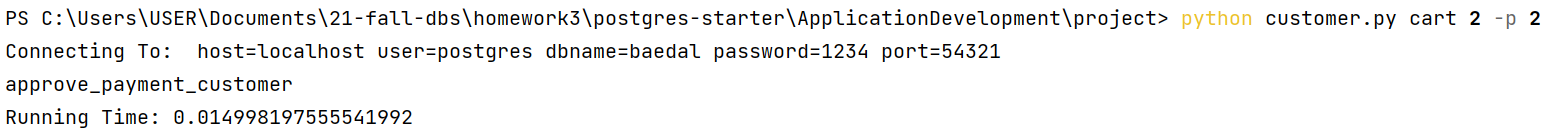
sql2에서는 각각의 메뉴들은 dict 타입으로 관리를 하고 그 dict 타입들 여러 개를 하나의 리스트로 묶어 menu로 관리하는 방식으로 구현하였다. 더불어 결제정보도 불러온다.

sql3에서는 위에서 모은 정보들을 바탕으로 order 테이블에 새로운 주문 정보를 만들어 INSERT하는 역할을 수행한다.

마지막으로, 주문이 완료된 메뉴들에 대해 cart 테이블에서 ordered의 값을 true로 갱신한다.

|  |
| --- |
| def approve\_payment\_customer(args):  *# TODO* try:  cur = conn.cursor()   *# fetching payment info of given customer's id* sql1 = "SELECT payments, searching\_store, phone FROM customer WHERE id={id};".format(id=args.id)  cur.execute(sql1)  rows = (cur.fetchall())[0]  payments = rows[0]; store\_id = rows[1]; cphone = rows[2]  payment\_ = payments[args.p-1]   *# fetching menus from customer's cart* menu = list()  menu\_elem = {'menu': None, 'pcs': None}  sql2 = "SELECT \* FROM cart WHERE cid={cid} and ordered=false".format(cid=args.id)  cur.execute(sql2)  menus\_ = cur.fetchall()  for elems in menus\_:  menu\_elem\_ = menu\_elem.copy()  menu\_elem\_['menu']=elems[3]; menu\_elem\_['pcs']=elems[4]  menu.append(menu\_elem\_)   if payment\_['type'] == 'card':  payment\_ = "cardnum={a}, type={b}".format(a=payment\_['data']['card\_num'], b=payment\_['type'])  else:  payment\_ = "accnum={a}, type={b}".format(a=payment\_['data']['acc\_num'], b=payment\_['type'])   *# insert order record* sql3 = "INSERT INTO orders (sid, cid, menu\_info, payment, cphone) " \  "VALUES ({sid}, {cid}, \'{menu\_info}\', \'{payment}\', \'{cphone}\');".format(  sid=store\_id, cid=args.id, menu\_info=str(json.dumps(menu)),  payment = payment\_, cphone=cphone  )  cur.execute(sql3)   *# update the cart table, mark as 'ordered'* sql4 = "UPDATE cart SET ordered = true " \  "WHERE cid={cid} and ordered = false".format(cid=args.id)  cur.execute(sql4)   conn.commit()  except Exception as err:  print(err)  conn.rollback()  else:  print("approve\_payment\_customer") |

**Result**



**18. customer 모든 주문 표시하기**

**Format**

python customer.py list ID

**Function**

ㆍprint\_all\_orders\_customer

**Implementation**

ㆍprint\_all\_orders\_customer

orders 테이블에서 해당 customer가 주문한 모든 주문정보에 대해 가게의 이름, 주문한 시각, 주문 상태를 뽑아온다. 그리고, 그것들을 일정한 형태를 갖추어 출력한다.

|  |
| --- |
| def print\_all\_orders\_customer(args):  *# TODO* try:  cur = conn.cursor()   sql = "SELECT sname, otime::timestamp(0), status " \  "FROM orders, store " \  "WHERE store.id=orders.sid and cid={cid}".format(cid=args.id)  cur.execute(sql)  rows = cur.fetchall()  print("Orders for Store #{sid}".format(sid=args.id))  print(tb.tabulate(rows, headers=['Store name', 'Order time', 'Delivery Status']))   except Exception as err:  print(err)  else:  print("print\_all\_orders\_customer") |

**Result**

텍스트이(가) 표시된 사진

자동 생성된 설명

**19. customer 배달중인 주문만 표시하기**

**Format**

python customer.py list ID -w

python customer.py list ID --waiting

**Function**

ㆍprint\_delivering\_orders\_customer

**Implementation**

ㆍprint\_delivering\_orders\_customer

주문상태가 ‘delivering’인 것만 출력하는 함수이다. 위에서 서술한 print\_all\_orders\_customer 함수에서 WHERE절 부분에만 해당 조건이 추가된 것이고 이외의 차이점은 없다.

|  |
| --- |
| def print\_delivering\_orders\_customer(args):  *# TODO* try:  cur = conn.cursor()   sql = "SELECT sname, otime::timestamp(0), status " \  "FROM orders, store " \  "WHERE store.id=orders.sid and cid={cid} and status=\'delivering\'".format(cid=args.id)  cur.execute(sql)  rows = cur.fetchall()  print(tb.tabulate(rows, headers=['Store name', 'Order time', 'Delivery Status']))   except Exception as err:  print(err)  else:  print("print\_delivering\_orders\_customer(args)") |

**Result**

위에서 출력하여 관찰한 전체 주문 중 상태가 ‘delivering’인 것만 출력되는 것을 확인할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**IV – 4. Delivery (배달대행자)**

**1. delivery 현재 배달중인 주문 검색하기**

**Format**

python delivery.py status ID

**Function**

ㆍshow\_delivering\_delivery

**Implementation**

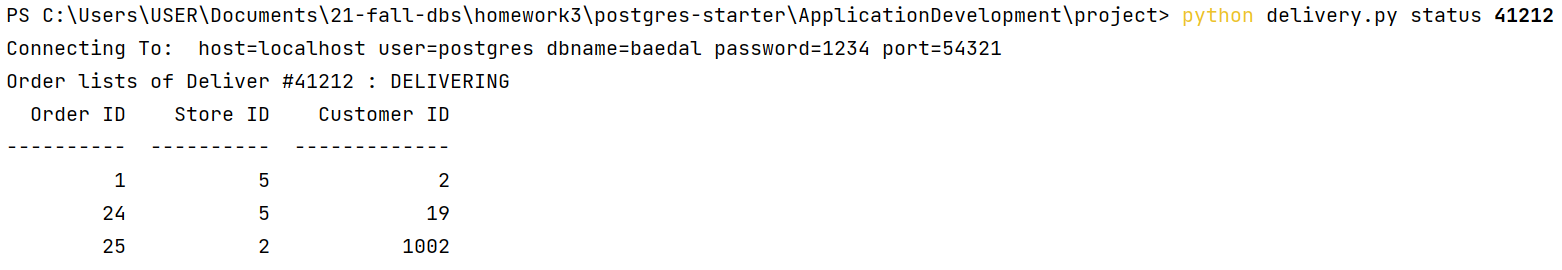
ㆍshow\_delivering\_delivery

입력받은 id값에 해당하는 배달대행자가 가지고 있는 주문 중 status=’delivering’, 즉 현재 배달중 상태에 있는 주문만을 SELECT문을 이용하여 받아온 것이다.

|  |
| --- |
| def show\_delivering\_delivery(args):  *# TODO* try:  cur = conn.cursor()  sql = "SELECT id, sid, cid FROM orders " \  "WHERE did={did} and status=\'delivering\' " \  "ORDER BY id;".format(did=args.id)  cur.execute(sql)  rows = cur.fetchall()  print("Order lists of Deliver #{did} : DELIVERING".format(did=args.id))  print(tb.tabulate(rows, headers=['Order ID', 'Store ID', 'Customer ID']))   except Exception as err:  print(err)   print("show\_delivering\_delivery") |

**Result**

id=41212인 배달대행자에게 배정된 주문 중 현재의 상태가 ‘delivering’인 것을 모두 나타낸 결과이다.



**2. delivery 현재 배달중인 주문 완료 처리하기**

**Format**

python delivery.py status ID -e ORDER\_ID

**Function**

ㆍmake\_delivered\_delivery

**Implementation**

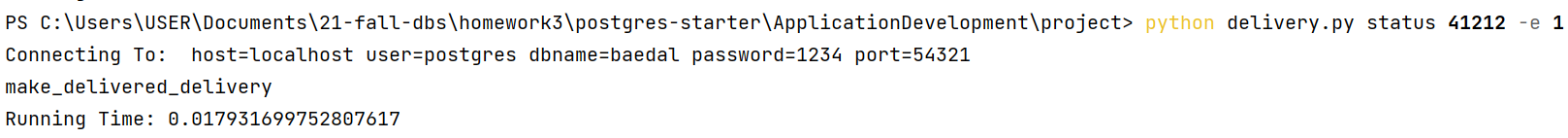
ㆍmake\_delivered\_delivery

* sql0에서는 입력받은 id값에 해당하는 주문 레코드를 담당하는 배달대행자의 id값을 검색한다.
  + 만약, 해당 id값 자체가 None이라면 주문 레코드 자체가 존재하지 않는 것이므로 “해당 주문이 존재하지 않는다.”라는 메시지를 출력하고 즉시 종료한다.
  + 만약, 해당 id값 자체가 존재는 하나, 주문 레코드 상의 did값이 비어있다면 이는 아직 pending 상태이기에 배달대행자가 배정되지 않은 것이다. 따라서, 해당 경우에는 “해당 주문은 pending 상태이다.”라는 메시지를 출력하고 즉시 종료한다.
  + 마지막으로 검색된 배달대행자의 id값과 현재의 프로그램을 실행중인 배달대행자 주체의 id값이 서로 다른 경우, “해당 주문은 접근이 불가능합니다.”라는 메시지를 출력하고 즉시 종료한다.  
    (다른 배달대행자의 주문 상태 변경을 막기 위함이다.)
* sql1에서는 해당 주문 정보의 status를 받아온다. 만약 status값이 이미 ‘delivered’라면 이미 처리된 주문에 해당한다. 따라서 이 경우에는 “이미 처리되었습니다.”라는 메시지를 출력하고 즉시 종료한다.
* 위의 테스트를 모두 통과했다면 sql2에서는 실제 DB에 해당 주문 레코드의 status 값을 ‘delivering’에서 ‘delivered’로 변경한다. 또한 이 시점에서는 dtime의 값이 채워져야 한다. 배달완료 처리를 하는 시점의 현재 시간을 dtime에 넣는다.
* 마지막으로 해당 주문의 배달대행자 정보 중 stock을 1만큼 다시 감소시킨다. 이는 남은 주문의 개수가 1만큼 줄었음을 의미한다.

|  |
| --- |
| def make\_delivered\_delivery(args):  *# TODO* try:  cur = conn.cursor()   *# check the validity: delivery-orders relationship* sql0 = "SELECT did FROM orders WHERE id={order\_id}".format(order\_id=args.e)  cur.execute(sql0)  rows = cur.fetchone()   if rows is None:  print("Given Order ID doesn't exist.")  return  if rows[0] is None:  print("Given Order is in pending status.")  return  did\_ = rows[0]  if did\_ != args.id:  print("Given Order is not accessible from Deliver #{did}".format(did=args.id))  return   *# check the order's status  # : if the given order is already finished, do nothing* sql1 = "SELECT status FROM orders WHERE id={order\_id}".format(order\_id=args.e)  cur.execute(sql1)  status\_ = (cur.fetchone())[0]  if status\_ == 'delivered':  print("Given order is already finished.")  return   *# make the given order's status 'delivered'* sql2 = "UPDATE orders SET status=\'delivered\', dtime = now() at time zone \'Asia/Seoul\' " \  "WHERE id = {order\_id}".format(order\_id=args.e)  cur.execute(sql2)  conn.commit()   *# decrement stock of corresponding deliver* sql3 = "UPDATE delivery SET stock = stock - 1 WHERE id={did}".format(did=did\_)  cur.execute(sql3)  conn.commit()   except Exception as err:  print(err)  conn.rollback()  print("make\_delivered\_delivery") |

**Result**

id=41212인 배달대행자의 주문 중 id=1인 주문이 현재에는 ‘delivering’ 상태이다. 이 주문을 주문 완료 처리해보자.



실제로 해당 주문의 상태가 ‘delivered’로 바뀌었음을 알 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**3. delivery 모든 배달 주문 표시하기**

**Format**

python delivery.py status ID -a

**Function**

ㆍshow\_all\_orders\_delivery

**Implementation**

ㆍshow\_all\_orders\_delivery

패스워드를 제외한 seller의 정보를 보이는 것이 이 기능의 핵심이다. 따라서, 특정 id에 해당하는 seller를 추출하기 위한 SQL문을 이 함수에서 수행한다. 해당하는 row를 인자로 넘겨 show\_seller\_from\_table 함수를 호출한다.

|  |
| --- |
| def show\_all\_orders\_delivery(args):  try:  cur = conn.cursor()  sql = "SELECT id, sid, cid, status FROM orders " \  "WHERE did={did} " \  "ORDER BY id".format(did=args.id)  cur.execute(sql)  rows = cur.fetchall()  print("Order lists of Deliver #{did} : ALL".format(did=args.id))  print(tb.tabulate(rows, headers=['Order ID', 'Store ID', 'Customer ID', 'Status']))  except Exception as err:  print(err)  print("show\_all\_orders\_delivery") |

**Result**

텍스트이(가) 표시된 사진

자동 생성된 설명

**(부록)**

테스트를 하며 인공적으로 데이터를 추가하였다. 테스트 진행 시 사용한 데이터는 아래와 같다.

가. cart 테이블

테이블이(가) 표시된 사진

자동 생성된 설명

나. orders 테이블

테이블이(가) 표시된 사진

자동 생성된 설명

\* 상위 5개의 데이터를 제외한 나머지의 데이터는 직접 INSERT문으로 데이터를 삽입한 것이며 그 용도는 아래와 같다.

- store의 주문 통계 정보 출력

- store에서 모든 메뉴를 출력한 customer 찾기

- delivery에서 각각의 주문 조회