Lab-3-sql-msarker000

March 2, 2019

1 Introduction to SQL

Due 5 PM, Tuesday, March 3rd, 2019

Our goal today is to begin learning SQL (for Structured Query Language). We'll make our web services API from last week persistent. To do this, we'll use Sqlite for the relational database.

I chose Sqlite for this exercise because it's ubiquitous--it is the world's "Most Widely Deployed and Used Database Engine". It's already on your development machine even if you use Windows 10. So we can avoid installation complications and get going immediately.

Sqlite is really a *library* that allows any application to be extended with database functionality. Although Sqlite is extremely powerful it isn't appropriate for all applications, specifically those with massive datasets and many concurrent writers. For such applications *client-server* databases such as PostGres or MySQL are a better fit.

This lab is organized into two sections. The first part is a tutorial. It shows with examples how to install sqlite, connect to a database, create tables, run queries, and update the database. In the second part, the problem set, you will use the concepts from the tutorial to add a database layer to the web services API.

1.1 Setup and Installation

To check whether sqlite3 is already installed on your system, run the magic shell command below:

```
In [1]: # determine if sqlite3 is installed
        ! which sqlite3
```

```
/Users/ayub/anaconda3/bin/sqlite3
```

The above command will print the location of the `sqlite3` executable. If you see some output, you're good to go. If not, follow the instructions here.

We'll use the SQLAlchemy library to access Sqlite (and other databases) from Python. SQLAlchemy is an abstraction and adaptation layer that runs above most relational databases. (A Sqlite-specific library for Python also exists but it does not support any other database; SQLAlchemy works with most popular SQL databases.)

SQLAlchemy provides two main ways to access databases: **core** and **ORM** (for Object Relational Mapping). We'll use the Core interface in this lab. With the Core interface, you write queries in *raw* SQL. The ORM interface is a higher-level abstraction. We'll explore the ORM interface next week.

You can install SQLAlchemy with the following commands:

```
In [2]: # install sqlalchemy with conda
        import sys

        !conda install --yes --prefix {sys.prefix} sqlalchemy
```

```
Collecting package metadata: done
Solving environment: done

# All requested packages already installed.
```

1.2 Connect to a Sqlite Database

With SQLAlchemy installed, you can connect to a database with the `create_engine` function. An `engine` is the central entry point for communicating with a specific database.

The first argument to `create_engine` specifies the kind of database to open. Here we'll open an *in-memory* sqlite database. The database won't be stored to disk. Instead it will disappear everytime the calling process (i.e., this Jupyter notebook) exits. A non-persistent database can be useful while learning: you get a blank database everytime you run the application.

The second argument `echo=True` tells SQLAlchemy to verbosely print to the console all the SQL statements it generates.

For more about the `create_engine` function see here.

```
In [14]: import sqlalchemy
         from sqlalchemy import create_engine
         from sqlalchemy import inspect

In [15]: # Type of database to connect to. The following specifies an in-memory sqlite.
         db_name = 'sqlite:///:memory:'

         # use a URL like the following to open (and create if necessary)
         # a file-backed sqlite database:

         # db_name = 'sqlite:///my_sqlite_db.sqlite3'
         # the above will create a database relative to current working directory. See
         # the documentation for how to create a database in a different location.


         # create an engine
         engine = create_engine(db_name, echo=True)
         print(sqlalchemy.__version__)
         print(engine)
```

```
1.2.7
Engine(sqlite:///:memory:)
```

1.3 Create Your First Table

At this point you'll have an empty database created in memory (not persistent storage). To do anything useful with a relational database, you have to first create a table (or tables) and insert some rows into those tables.

Creating a table defines the names and types of the attributes (columns) of the database.

We'll create a simple table called `cities`. It will have the following columns:

```
{
    "id": 'the primary key for the table',
    "name": 'the city name, a text string',
    "lat": 'latitude, a floating point number',
    "lng": 'longitude, a floating point number',
    "country": 'country, a text string',
    "population": 'city population, an integer'
}
```

To create the `cities` table, use the `create table` SQL statement as shown below. The `engine.execute()` method is used to send the raw SQL to the connected database.

```
In [16]: # drop a table cities in case it existed already
         drop_table_statement = """drop table cities"""
         #engine.execute(drop_table_statement)

         # sql statement
         create_table_stmt = """create table cities(
           id integer primary key,
           name text not null,
           lat float not null,
           lng float not null,
           state text not null,
           country text not null,
           population integer not null
         );
         """
         engine.execute(create_table_stmt)
```

```
2019-03-02 12:45:45,853 INFO sqlalchemy.engine.base.Engine SELECT CAST('test plain returns' AS
2019-03-02 12:45:45,854 INFO sqlalchemy.engine.base.Engine ()
2019-03-02 12:45:45,857 INFO sqlalchemy.engine.base.Engine SELECT CAST('test unicode returns' A
2019-03-02 12:45:45,858 INFO sqlalchemy.engine.base.Engine ()
2019-03-02 12:45:45,860 INFO sqlalchemy.engine.base.Engine create table cities(
  id integer primary key,
  name text not null,
  lat float not null,
  lng float not null,
  state text not null,
  country text not null,
  population integer not null
```

```
);

2019-03-02 12:45:45,861 INFO sqlalchemy.engine.base.Engine ()
2019-03-02 12:45:45,863 INFO sqlalchemy.engine.base.Engine COMMIT


Out[16]: <sqlalchemy.engine.result.ResultProxy at 0x108e8ec18>
```

You now have an empty database created in memory (not persistent storage). The following code block shows how to retrieve information about the `users` table you just created:

```
In [17]: engine.table_names()

2019-03-02 12:45:47,251 INFO sqlalchemy.engine.base.Engine SELECT name FROM sqlite_master WHERE
2019-03-02 12:45:47,253 INFO sqlalchemy.engine.base.Engine ()


Out[17]: ['cities']

In [18]: # inspect the database
         inspector = inspect(engine)

         # Get table information
         print(inspector.get_table_names())

         # Get column information
         for col in inspector.get_columns('cities'):
             print(col)

2019-03-02 12:45:48,004 INFO sqlalchemy.engine.base.Engine SELECT name FROM sqlite_master WHERE
2019-03-02 12:45:48,005 INFO sqlalchemy.engine.base.Engine ()
['cities']
2019-03-02 12:45:48,007 INFO sqlalchemy.engine.base.Engine PRAGMA table_info("cities")
2019-03-02 12:45:48,008 INFO sqlalchemy.engine.base.Engine ()
{'name': 'id', 'type': INTEGER(), 'nullable': True, 'default': None, 'autoincrement': 'auto',
{'name': 'name', 'type': TEXT(), 'nullable': False, 'default': None, 'autoincrement': 'auto',
{'name': 'lat', 'type': FLOAT(), 'nullable': False, 'default': None, 'autoincrement': 'auto',
{'name': 'lng', 'type': FLOAT(), 'nullable': False, 'default': None, 'autoincrement': 'auto',
{'name': 'state', 'type': TEXT(), 'nullable': False, 'default': None, 'autoincrement': 'auto',
{'name': 'country', 'type': TEXT(), 'nullable': False, 'default': None, 'autoincrement': 'auto
{'name': 'population', 'type': INTEGER(), 'nullable': False, 'default': None, 'autoincrement':
```

1.4 Insert City Data Into the Database

Next, we will use a SQL `insert` statement to add some cities to the database. (The data for this example came from SimpleMaps.)

Note the use of `?` placeholders in the `insert` statement. These are positional parameters that are filled in with actual values at runtime.

```
In [19]: cities = [
             ("Ammon", 43.4748, -111.9559, "Idaho", "USA", 15252),
             ("Idaho Falls", 43.4878, -112.0359, "Idaho", "USA", 96166),
             ("Iona", 43.5252, -111.931, "Idaho", "USA",        2213),
             ("Island Park",        44.5251, -111.3581, "Idaho", "USA", 272),
             ("Ririe", 43.6326, -111.7716,"Idaho", "USA",643),
             ("Sugar City", 43.8757,        -111.7518, "Idaho" ,"USA", 1361),
             ("Teton", 43.8872, -111.6726, "Idaho", "USA", 714),
         ]

         insert_statement = """
         insert into cities (name, lat, lng, state, country, population)
           values(?, ?, ?, ?, ?, ?)
         """

         for c in cities:
             print(f"inserting {c[0]}")
             # insert into db; note unpacking of tuple (*c)
             engine.execute(insert_statement, *c)


inserting Ammon
2019-03-02 12:45:49,423 INFO sqlalchemy.engine.base.Engine
insert into cities (name, lat, lng, state, country, population)
  values(?, ?, ?, ?, ?, ?)

2019-03-02 12:45:49,424 INFO sqlalchemy.engine.base.Engine ('Ammon', 43.4748, -111.9559, 'Idaho
2019-03-02 12:45:49,427 INFO sqlalchemy.engine.base.Engine COMMIT
inserting Idaho Falls
2019-03-02 12:45:49,428 INFO sqlalchemy.engine.base.Engine
insert into cities (name, lat, lng, state, country, population)
  values(?, ?, ?, ?, ?, ?)

2019-03-02 12:45:49,429 INFO sqlalchemy.engine.base.Engine ('Idaho Falls', 43.4878, -112.0359,
2019-03-02 12:45:49,431 INFO sqlalchemy.engine.base.Engine COMMIT
inserting Iona
2019-03-02 12:45:49,431 INFO sqlalchemy.engine.base.Engine
insert into cities (name, lat, lng, state, country, population)
  values(?, ?, ?, ?, ?, ?)

2019-03-02 12:45:49,432 INFO sqlalchemy.engine.base.Engine ('Iona', 43.5252, -111.931, 'Idaho'
2019-03-02 12:45:49,433 INFO sqlalchemy.engine.base.Engine COMMIT
inserting Island Park
2019-03-02 12:45:49,434 INFO sqlalchemy.engine.base.Engine
insert into cities (name, lat, lng, state, country, population)
  values(?, ?, ?, ?, ?, ?)

2019-03-02 12:45:49,435 INFO sqlalchemy.engine.base.Engine ('Island Park', 44.5251, -111.3581,
```

```
2019-03-02 12:45:49,436 INFO sqlalchemy.engine.base.Engine COMMIT
inserting Ririe
2019-03-02 12:45:49,437 INFO sqlalchemy.engine.base.Engine
insert into cities (name, lat, lng, state, country, population)
  values(?, ?, ?, ?, ?, ?)

2019-03-02 12:45:49,438 INFO sqlalchemy.engine.base.Engine ('Ririe', 43.6326, -111.7716, 'Idaho
2019-03-02 12:45:49,439 INFO sqlalchemy.engine.base.Engine COMMIT
inserting Sugar City
2019-03-02 12:45:49,440 INFO sqlalchemy.engine.base.Engine
insert into cities (name, lat, lng, state, country, population)
  values(?, ?, ?, ?, ?, ?)

2019-03-02 12:45:49,441 INFO sqlalchemy.engine.base.Engine ('Sugar City', 43.8757, -111.7518,
2019-03-02 12:45:49,443 INFO sqlalchemy.engine.base.Engine COMMIT
inserting Teton
2019-03-02 12:45:49,444 INFO sqlalchemy.engine.base.Engine
insert into cities (name, lat, lng, state, country, population)
  values(?, ?, ?, ?, ?, ?)

2019-03-02 12:45:49,445 INFO sqlalchemy.engine.base.Engine ('Teton', 43.8872, -111.6726, 'Idaho
2019-03-02 12:45:49,446 INFO sqlalchemy.engine.base.Engine COMMIT
```

1.5 Retrieve Selected Rows

We can now retrieve a subset of the cities with the SQL `select` statement. Here we use a `where` retrieve cities with a `population` less than 1000.

In the second `select` We also use a `count` function to compute the number cities with a population greater than 1000.

```
In [20]: print(engine.table_names())

         c = engine.execute('select * from cities where population < 1000')

         for row in c:
             print(dict(row))

         cnt = engine.execute('select count(name) from cities where population > 10000')
         for row in cnt:
             print(dict(row))
```

```
2019-03-02 12:45:50,972 INFO sqlalchemy.engine.base.Engine SELECT name FROM sqlite_master WHERE
2019-03-02 12:45:50,973 INFO sqlalchemy.engine.base.Engine ()
['cities']
2019-03-02 12:45:50,976 INFO sqlalchemy.engine.base.Engine select * from cities where populatio
2019-03-02 12:45:50,978 INFO sqlalchemy.engine.base.Engine ()
{'id': 4, 'name': 'Island Park', 'lat': 44.5251, 'lng': -111.3581, 'state': 'Idaho', 'country'
```

```
{'id': 5, 'name': 'Ririe', 'lat': 43.6326, 'lng': -111.7716, 'state': 'Idaho', 'country': 'USA
{'id': 7, 'name': 'Teton', 'lat': 43.8872, 'lng': -111.6726, 'state': 'Idaho', 'country': 'USA
2019-03-02 12:45:50,980 INFO sqlalchemy.engine.base.Engine select count(name) from cities where
2019-03-02 12:45:50,982 INFO sqlalchemy.engine.base.Engine ()
{'count(name)': 2}
```

1.6 Update Selected Rows

In this example we shall change the 'State' from 'Idaho' to 'ID' for all cities with a `population` greater than 10000. To do this we will utilize a SQL `update` command:

```
In [21]: update_statement = """
         update cities
         set state = ?
         where population > ?
         """

         engine.execute(update_statement, 'ID', 10000)

         # read updated rows to see that the state attribute was changed
         cs = engine.execute('select id, population, state from cities where population > 10000

         # print out each row
         for row in cs:
             print(row)
```

```
2019-03-02 12:45:52,507 INFO sqlalchemy.engine.base.Engine
update cities
set state = ?
where population > ?

2019-03-02 12:45:52,510 INFO sqlalchemy.engine.base.Engine ('ID', 10000)
2019-03-02 12:45:52,512 INFO sqlalchemy.engine.base.Engine COMMIT
2019-03-02 12:45:52,513 INFO sqlalchemy.engine.base.Engine select id, population, state from ci
2019-03-02 12:45:52,514 INFO sqlalchemy.engine.base.Engine ()
(1, 15252, 'ID')
(2, 96166, 'ID')
```

2 Problem Set

50 Points

The above tutorial should have provided you with enough background to get started with the homework, which is to migrate your web app `users` resource to sqlite. The problems below are identical to those from last week except you will be retrieving, creating, and updating a database table instead of using an in-memory list of users.

2.1 Setup: Migrate Web Service Users to Sqlite

Before proceeding to the problem set, update `api.py` to initialize and connect to the database. Please follow these steps:

1. Install Flask-SQLAlchemy, a library that simplifies access to SQLAlchemy from within Flask: `>bash conda install Flask-SQLAlchemy`
2. Import `flask_sqlalchemy` into `api.py`. `>python from flask_sqlalchemy import SQLAlchemy`
3. Update `flask` startup code by replacing it with this: "'python if **name** == '**main**':

```
# save database handle in module-level global
global __db__

# use in-memory database for debugging
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///:memory:'

# app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///db.sqlite'

__db__ = SQLAlchemy(app)
engine = __db__.engine

# put your database initialization statements here
# create the users table

# insert each item from USERS list into the users table
app.run(debug=True)

'''
```

2.2 Problem 1: List Users

10 Points

Modify `api.py` to retrieve the collection of users. Essentially, you will convert your existing handler that returns data from the variable `USERS` to read from the database.

Run the test below to show that your code is correct.

```
In [6]: import unittest
        import requests
        import json

        # The base URL for all HTTP requests
        BASE = 'http://localhost:5000/users'

        # set Content-Type to application/json for all HTTP requests
        headers={'Content-Type': 'application/json'}
```

```
        class Problem1Test(unittest.TestCase):

            # test
            def test_users_get_collection(self):
                r = requests.get(BASE, headers = headers)
                self.assertEqual(r.status_code, 200)

                j = r.json()
                self.assertEqual(type(j), list)
                self.assertGreater(len(j), 0)

                # extract the first element of the list
                first = j[0]

                # check all attributes exist
                self.assertIn('id', first)
                self.assertIn('first', first)
                self.assertIn('last', first)
                self.assertIn('email', first)
                self.assertIn('role', first)
                self.assertIn('active', first)



        # Run the unit tests
        unittest.main(defaultTest="Problem1Test", argv=['ignored', '-v'], exit=False)
```

```
test_users_get_collection (__main__.Problem1Test) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.010s

OK
```

```
Out[6]: <unittest.main.TestProgram at 0x10be767f0>
```

2.3 Problem 2: Retrieve a Single User

10 Points

Modify the method `GET /users/{id}` to retrieve a specific user by ID to use the database instead of the `USERS` list.

This method shall return an HTTP status code of `200` on success and `404` (not found) if the user with the specified ID does not exist. See the unit tests below.

```
In [7]: class Problem2Test(unittest.TestCase):

            def test_users_get_member(self):
```

```
                r = requests.get(BASE + '/1')
                self.assertEqual(r.status_code, 200)
                print(r.headers)
                j = r.json()

                self.assertIs(type(j), dict)
                self.assertEqual(j['id'], 1)
                self.assertIn('first', j)
                self.assertIn('last', j)
                self.assertIn('email', j)
                self.assertIn('role', j)

            def test_users_wont_get_nonexistent_member(self):

                r = requests.get(BASE + '/1000')
                self.assertEqual(r.status_code, 404)

        # Run the unit tests
        unittest.main(defaultTest="Problem2Test", argv=['ignored', '-v'], exit=False)
```

```
test_users_get_member (__main__.Problem2Test) ... ok
test_users_wont_get_nonexistent_member (__main__.Problem2Test) ...

{'Content-Type': 'application/json', 'Content-Length': '125', 'Server': 'Werkzeug/0.14.1 Pytho


ok

----------------------------------------------------------------------
Ran 2 tests in 0.018s

OK


Out[7]: <unittest.main.TestProgram at 0x10ce57a90>
```

2.4 Problem 3: Create a User

10 Points

Modify the `POST /users` method to save the user to the database.

All of these parameters are required and your code should enforce this. If validation succeeds, add the new user to the `USERS` list and give it a unique ID.

Return HTTP status code `201` (created) if the operation succeeds and `422` (Unprocessable Entity) if validation fails.

The created user will be returned as JSON if the operation succeeds.

```
In [8]: class Problem3Test(unittest.TestCase):
```

```
            def test_users_create(self):
                data = json.dumps({'first': 'Sammy', 'last': 'Davis', 'email': 'sammy@cuny.edu

                r = requests.post(BASE, headers = headers, data = data)
                self.assertEqual(r.status_code, 201)

            def test_wont_create_user_without_first_name(self):
                # simple validation (missing parameters)
                data = json.dumps({'last': 'Davis', 'email': 'sammy@cuny.edu'})

                r = requests.post(BASE, headers = headers, data = data)
                self.assertEqual(r.status_code, 422)


        # Run the unit tests
        unittest.main(defaultTest="Problem3Test", argv=['ignored', '-v'], exit=False)
```

```
test_users_create (__main__.Problem3Test) ... ok
test_wont_create_user_without_first_name (__main__.Problem3Test) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.019s

OK
```

```
Out[8]: <unittest.main.TestProgram at 0x10ce5fd68>
```

2.5 Problem 4: Update a User

10 Points

Change the method that handles user updates (`PATCH/PUT /users/<id>`) so that it writes the update to the database.

```
In [9]: class Problem4Test(unittest.TestCase):

            def test_users_update_member(self):
                data = json.dumps({'first': 'testing'})
                r = requests.patch(BASE + '/1', headers = headers, data = data)
                self.assertEqual(r.status_code, 200)

                j = r.json()
                self.assertIs(type(j), dict)
                self.assertEqual(j['id'], 1)
                self.assertEqual(j['first'], 'testing')

                # now retrieve the same object to ensure that it was really updated
                r = requests.get(BASE + '/1', headers = headers, data = data)
                self.assertEqual(r.status_code, 200)
```

```
            j = r.json()
            self.assertEqual(j['first'], 'testing')


        def test_users_update_member_not_found(self):
            data = json.dumps({'first': 'testing'})
            r = requests.patch(BASE + '/1000', headers = headers, data = data)
            self.assertEqual(r.status_code, 404)



    # Run the unit tests
    unittest.main(defaultTest="Problem4Test", argv=['ignored', '-v'], exit=False)
```

```
test_users_update_member (__main__.Problem4Test) ... ok
test_users_update_member_not_found (__main__.Problem4Test) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.033s

OK
```

```
Out[9]: <unittest.main.TestProgram at 0x10ce57550>
```

2.6 Problem 5: Deactivate a User

Modify the handler for `POST /users/<id>/deactivate` so that it persists the deactivation to the database.

```
In [10]: class Problem5Test(unittest.TestCase):

             def test_users_deactivate_member(self):

                 r = requests.post(BASE + '/1/deactivate', headers = headers)
                 self.assertEqual(r.status_code, 200)

                 j = r.json()
                 self.assertIs(type(j), dict)
                 self.assertEqual(j['active'], False)


         # Run the unit tests
         unittest.main(defaultTest="Problem5Test", argv=['ignored', '-v'], exit=False)
```

```
test_users_deactivate_member (__main__.Problem5Test) ... ok
```

```
----------------------------------------------------------------------
Ran 1 test in 0.009s

OK


Out[10]: <unittest.main.TestProgram at 0x10ce79fd0>
```

```
In [ ]: import os
        from flask import Flask, request, Response, jsonify
        from functools import wraps
        import json
        from flask_sqlalchemy import SQLAlchemy


        print(__name__)
        app = Flask(__name__)
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///:memory:'
        app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = True
        db = SQLAlchemy(app)
        __db__ = SQLAlchemy(app)


        # Custom error handler. Raise this exception
        # to return a status_code, message, and body
        class InvalidUsage(Exception):
            status_code = 400

            def __init__(self, message, status_code=None, payload=None):
                Exception.__init__(self)
                self.message = message
                if status_code is not None:
                    self.status_code = status_code
                self.payload = payload

            def to_dict(self):
                rv = dict(self.payload or ())
                rv['message'] = self.message
                return rv


        # set the default error handler
        @app.errorhandler(InvalidUsage)
        def handle_invalid_usage(error):
            response = jsonify(error.to_dict())
            response.status_code = error.status_code
            return response
```

```
# dummy users
USERS = [
    {'id': 1, 'first': 'Joe', 'last': 'Bloggs',
        'email': 'joe@bloggs.com', 'role': 'student', 'active': 1},
    {'id': 2, 'first': 'Ben', 'last': 'Bitdiddle',
        'email': 'ben@cuny.edu', 'role': 'student', 'active': 1},
    {'id': 3, 'first': 'Alissa P', 'last': 'Hacker',
        'email': 'missalissa@cuny.edu', 'role': 'professor', 'active': 1},
]


def initFlask():

     # use in-memory database for debugging
    global  __db__
    engine = __db__.engine

    drop_table_statement = """drop table users"""
    # engine.execute(drop_table_statement)

    # create the users table
    # sql statement
    create_table_stmt = """create table users(
        id integer primary key,
        first text not null,
        last text not null,
        email text not null,
        role text default null,
        active integer default 0
        );
    """
    engine.execute(create_table_stmt)
    insert_statement = """
         insert into users(first, last, email, role, active)
         values(?, ?, ?, ?, ?)
         """
    for u in USERS:
        print(f"inserting {u}")
         # insert into db; note unpacking of tuple (*c)
        engine.execute(insert_statement, u['first'],  u['last'], u['email'],u['role'],

    print('******* This is end of initialization *******')
    return


if __name__ == '__main__':
     app.run(debug=True)
```

```
initFlask()

# Your code here...
# E.g.,
# @app.route("/users", methods=["GET"])

# Problem 1
def convertToUserObject(row):
    user = dict(row)
    if user['active'] == 1:
        user['active'] = True
    else:
        user['active'] = False
    return user

@app.route("/users", methods=["GET"])
def get_users():
    global __db__
    engine = __db__.engine
    users = engine.execute('select * from users')
    selected_users = []
    for row in users:
        user = convertToUserObject(row)
        selected_users.append(user)
    return jsonify(selected_users)


# Problem 2
@app.route("/users/<int:id>", methods=["GET"])
def get_user(id):
    global __db__
    engine = __db__.engine
    selected_user = engine.execute('select * from users where id  =  ?',id).fetchone()
    if selected_user == None:
        raise InvalidUsage("user not found" , 404)
    else:
        return jsonify(convertToUserObject(selected_user))

# Problem 3
@app.route("/users", methods=["POST"])
def create_user():
    if request.headers['Content-Type'] == 'application/json':
        pdata = request.get_json()
        first = pdata.get('first')
        last = pdata.get('last')
        email = pdata.get('email')
```

```
        if first == None or last == None or email == None:
            raise InvalidUsage("nprocessable Entity", 422)

        # defult role is student
        if pdata.get('role') is None:
            pdata['role'] = 'Student'

        if pdata.get('active') is None:
            pdata['active'] = False

        global __db__
        engine = __db__.engine

        insert_statement = """
                 insert into users(first, last, email, role, active)
                 values(?, ?, ?, ?, ?)
                 """

        result = engine.execute(insert_statement, pdata['first'], pdata['last'], pdata
        new_user = engine.execute('select * from users where id  =  ?',  result.lastrow
        response = jsonify(convertToUserObject(new_user))
        response.status_code = 201
        return response

    else:
        raise InvalidUsage("Unsupported Media Type", 415)


# Problem 4
@app.route("/users/<int:id>", methods=["PATCH", "PUT"])
def updated_user(id):
    if request.headers['Content-Type'] == 'application/json':
        global __db__
        engine = __db__.engine

        selected_user = engine.execute('select * from users where id  =  ?', id).fetch
        if selected_user == None:
            raise InvalidUsage("user not found" , 404)

        pdata = request.get_json()
        selected_user = dict(selected_user)
        update_statement = """
        update users
        set first = ?, last=?, email=?, role=?, active=?
        where id = ?
        """
        if pdata.get('first') != None:
            selected_user['first'] = pdata.get('first')
```

```
        if pdata.get('last') != None:
            selected_user['last'] = pdata.get('last')
        if pdata.get('email') != None:
            selected_user['email'] = pdata.get('email')
        if pdata.get('role') != None:
            selected_user['role'] = pdata.get('role')
        if pdata.get('role') != None:
            selected_user['active'] = pdata.get('active')

        result = engine.execute(update_statement, selected_user['first'], selected_user
                        selected_user['role'], selected_user['active'], selected_user['

        updated_user = engine.execute('select * from users where id  =  ?', id).fetchon
        response = jsonify(convertToUserObject(updated_user))
        response.status_code = 200
        return response
    else:
        raise InvalidUsage("Unsupported Media Type", 415)


# Problem 5
@app.route("/users/<int:id>/deactivate", methods=["POST"])
def delete_user(id):
    global __db__
    engine = __db__.engine

    selected_user = engine.execute('select * from users where id  =  ?', id).fetchone()
    if selected_user == None:
        raise InvalidUsage("user not found", 404)

    update_statement = """
            update users
            set active=?
            where id = ?
            """
    engine.execute(update_statement, False, id)
    deleted_user = engine.execute('select * from users where id  =  ?', id).fetchone()
    response = jsonify(convertToUserObject(deleted_user))
    response.status_code = 200
    return response
```