Project Tracker - Python

In this exercise, we'll connect the "project tracker" we created earlier (a.k.a. our PostgreSQL database) to Python code. This connection provides an interactive system where we can read, write, and delete data in our database.

> **Note: Getting a Database**
>
> If you were not able to finish yesterday's exercise, feel free to use the dump file provided in this directory (***hackbright.sql***) to create a working version. (The instructions for how to do dump/restore PostgreSQL databases are in Part 4 of yesterday's exercise, and they are also included in the "Setup" section for today's exercise below.)

At this point you should have ended up with a database called ***hackbright*** containing three tables:

- ***students***
- ***grades***
- ***projects***

Today's exercise is split into two sections. The first part is a code walkthrough of the ***hackbright.py*** file to help you understand what each part does. The second part is where you will update this code to incorporate new features.

Code Walkthrough

Use Sublime Text to look over ***hackbright.py***. Inside, you'll find a partial implementation of a database-backed application built off the database we constructed earlier.

Using SQL through Python is not too different from doing it directly through the ***psql*** app, but there are some different steps to follow.

Setup

First, you'll want to install the libraries this lab requires and make sure your database is set up so you can work with it.

You've already seen how to invoke a virtual environment and why it's necessary to compartmentalize your installations for a given application build, but here is a quick reminder of the process:

1. Create a new, empty virtual environment.

   ```
   $ virtualenv env
   ```

   And activate it:

```
$ source env/bin/activate
```

Keep your environment activated for the duration of this exercise, but in general, remember to ***deactivate*** your environment when you're done working on your app.

2. Install libraries with ***pip3***. For this exercise, you'll need **Flask**, **Flask-SQLAlchemy**, and **psycopg2**.

```
(env) $ pip3 install flask flask-sqlalchemy psycopg2-binary
```

3. Use the command ***pip3 freeze > requirements.txt*** to make a permanent record of the exact versions of libraries ***pip3*** installed, so you can always make another virtual environment just like this one.

```
(env) $ pip3 freeze > requirements.txt
```

4. Add your ***env/*** directory to a ***.gitignore*** file so that you don't accidentally check it in. Click the Sublime Text icon, and then in your terminal, type:

```
(env) $ subl .gitignore
```

Then just type ***env/*** into the file and save it. You can check the contents at the command line after saving with the ***cat*** command:

```
(env) $ cat .gitignore
env/
```

5. Create a git repository by using the ***git init*** command.

```
(env) $ git init
```

6. Add your ***.gitignore*** and ***requirements.txt*** file to git; you'll want these files in your repository.

```
(env) $ git add .gitignore requirements.txt
```

If you want a more in-depth review of ***pip3*** and ***virtualenv***, here's a full tutorial <http://www.dabapps.com/blog/introduction-to-pip-and-virtualenv-python/>.

Once your environment is set up, recreate your database by running the following commands:

```
(env) $ dropdb hackbright
(env) $ createdb hackbright
(env) $ psql hackbright < hackbright.sql
```

Database Connection

The next step is to connect to your database. Once we connect to the database, we can *execute* SQL commands and get results from that database.

Take a closer look at the function ***connect_to_db*** in the ***hackbright.py*** file, which creates a connection to the database:

hackbright.py

```
def connect_to_db(app):
    """Connect the database to our Flask app."""

    app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql:///hackbright'
    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
    db.app = app
    db.init_app(app)
```

This code is straight from the documentation of Flask-SQLAlchemy, and shows how to make a connection via SQLAlchemy.

> **Note: Connecting Python to other Databases**
>
> We're using PostgreSQL, but Python can also connect to other SQL databases, such as MySQL <http://mysql.org> and Sqlite <https://www.sqlite.org>. Since we're using SQLAlchemy, all you'd have to do to use a different database software is change the database URI that you provide.

Executing a Query

Now that you know how to connect to your database, let's find out how to use it to execute queries.

In your ***hackbright.py*** file, look at the first few lines of the ***get_student_by_github*** function:

hackbright.py

```
def get_student_by_github(github):
    """Given a GitHub account name, print info about the matching student."""

    QUERY = """
        SELECT first_name, last_name, github
        FROM students
        WHERE github = :github
        """
```

There are a few points to notice about this code.

- The function includes the query you'd execute to get a student record based on their GitHub account in a string assigned to the ***QUERY*** variable. (We chose to write ***QUERY*** in UPPER CASE because it's a constant that won't ever need to change.)
- You don't need a semicolon when you execute a query string from Python with SQLAlchemy.

- Finally, in place of a filter string for the ***WHERE*** clause, this query string has a field name (***github***) that follows the format ***:parameter***.

The ***:github*** piece is a placeholder. You'll use the same query over and over, each time to get the details of different students. The only part that changes between each query is the `WHERE github =` line.

The next line executes the query, like this:

hackbright.py

```
    db_cursor = db.session.execute(QUERY, {'github': github})
```

This gives that ***:github*** placeholder an actual value: the value that was passed to the ***get_student_by_github*** function. This is very much like the ***{}*** and ***%*** substitution operators for Python strings.

The result of executing a query like this from Python is called a *cursor*, so this code binds that result to a variable called ***db_cursor***. A cursor is very similar to a file handle. For now, know that it is simply the mechanism that you'll use to look at the rows contained in the result from our query.

Fetching Rows

Now, head back to ***hackbright.py*** in Sublime Text, and look further in the ***get_student_by_github*** function. Recall that the query is executed as shown in the previous section.

After the query executes, we need to fetch rows out of the database table to use them. In this case, we only want to display one student for that particular GitHub username, so we'll use ***fetchone***, a method on ***db_cursor*** that returns just one row:

hackbright.py

```
    row = db_cursor.fetchone()
```

This method returns a database row as a tuple-like object. The object contains values for each of the columns we selected:

```
('Jane', 'Hacker', 'jhacks')
```

In ***hackbright.py***, the print statement in ***get_student_by_github*** then displays this information by indexing into the tuple.

The print statement will display in the Python interpreter as:

```
>>> get_student_by_github('jhacks')
Student: Jane Hacker
GitHub account: jhacks
```

Try it Yourself!

Launch the Python file for this exercise in the interpreter to try out the completed functions in ***hackbright.py***.

From your command line, go ahead and type:

```
(env) $ python3 -i hackbright.py
```

Then call an existing function to query the database:

```
>>> get_student_by_github('jhacks')
Student: Jane Hacker
GitHub account: jhacks
```

The result you see contains data returned from the database, and should match the output you saw at the end of the previous section.

Adding Data to the Database

Now we can query for data, but we still need to be able to add more data to the system. The function ***make_new_student*** is incomplete.

Part of your future tasks will be to write an insert SQL command to be executed. But for now, let's move on.

Cleanup

Like files, you should always close your connection to your database when you're done. At the end of the ***hackbright.py*** file, you will see the following:

```
db.session.close()
```

This will call the ***close*** method on the connection when you quit the program.

Please pause here and ask for a code review to discuss what you've learned so far.

Now it's Your Turn!

Before we start changing any of the functions to get or insert data into the database, we need to make a few changes to allow for the handling of input commands.

Handling Input

So far, you've called these functions directly from the Python interpreter by running ***hackbright.py*** in interactive mode and entering the functions there. Instead, let's make a few changes to ***hackbright.py*** so you can use those functions from outside the Python console.

You can use the ***handle_input*** function to call functions using our custom syntax. Look at the following code at the bottom of the ***hackbright.py*** file:

hackbright.py

```
if __name__ == "__main__":
    connect_to_db(app)

    # handle_input()

    # To be tidy, we close our database connection -- though,
    # since this is where our program ends, we'd quit anyway.

    db.session.close()
```

Uncomment the call to ***handle_input***:

```
if __name__ == "__main__":
    connect_to_db(app)

    handle_input()

    # To be tidy, we close our database connection -- though,
    # since this is where our program ends, we'd quit anyway.

    db.session.close()
```

Now whenever you run this Python file from the command line, the ***handle_input*** function will be called, resulting in the following prompt:

```
HBA Database>
```

To exit the ***HBA Database>*** prompt, you can just type ***quit***.

Going forward, you will be entering commands at this prompt. These commands will be parsed in the ***handle_input*** loop which will then call each function.

For example, you can already call ***get_student_by_github*** by entering ***student jhacks*** at the prompt, like this:

```
HBA Database> student jhacks
```

When a user enters ***student jhacks***, the ***get_student_by_github*** function is called, and that function will print out information about the student.

We've implemented this feature for you, so go ahead and try it out now. Run ***python3 hackbright.py***. When you see the ***HBA Database>*** prompt, enter ***student jhacks*** and observe the output.

Read over the ***handle_input*** code in ***hackbright.py*** to see how the user interacts with the database.

A Second *handle_input* Command Example

We've provided you with another command example in the ***handle_input*** function; this command is called ***new_student***. The syntax for this command at the ***HBA Database>*** prompt will be as follows:

```
HBA Database> new_student Jasmine Debugger jdebugger
```

Now, look at the ***elif*** statements in the ***handle_input*** loop, and find one for ***new_student***:

hackbright.py

```
        elif command == "new_student":
            first_name, last_name, github = args  # unpack!
            make_new_student(first_name, last_name, github)
```

If a user tried to use the ***new_student*** command now, it would fail because ***make_new_student*** isn't implemented yet. You'll need to complete this function before you can insert data. Let's walk through the code.

Inserting Data

Just as ***get_student_by_github*** uses a ***SELECT*** statement, you'll need to execute an ***INSERT*** statement to create a new student record with ***make_new_student***.

First, type out your template for the ***INSERT***, then fill the values in for when you execute it:

solution/hackbright.py

```
def make_new_student(first_name, last_name, github):
    """Add a new student and print confirmation.

    Given a first name, last name, and GitHub account, add student to the
    database and print a confirmation message.
    """

    QUERY = """
        INSERT INTO students (first_name, last_name, github)
          VALUES (:first_name, :last_name, :github)
        """

    db.session.execute(QUERY, {'first_name': first_name,
                               'last_name': last_name,
                               'github': github})
    db.session.commit()

    print(f"Successfully added student: {first_name} {last_name}")
```

Note the ***db.session.commit()*** call. Inserting rows automatically puts you in a transaction, so for your changes to be committed, you have to ask for that.

Type the code above that isn't already in ***hackbright.py*** into the file to replace the ***pass*** statement in the ***make_new_student*** function now.

Please ask for a code review at this point, discuss the changes you've made.

Your Remaining Tasks

The next step is to add more features to ***hackbright.py*** using the scaffold provided. Before you begin, comment out the line to call the ***handle_input*** function from ***hackbright.py*** again, so you can interact with ***hackbright.py*** and test functions as you go.

```
if __name__ == "__main__":
    connect_to_db(app)

    # handle_input()
```

Now, each time you write a new function, you can enter the interactive Python interpreter:

```
(env) $ python3 -i hackbright.py
```

And from within the Python interpreter, you can call each function you complete to test it:

```
>>> get_student_by_github('jhacks')
Student: Jane Hacker
GitHub account: jhacks
```

Let's start adding features!

Query for Projects by Title

Complete the ***get_project_by_title*** function. Then, test it in the Python interpreter and compare the data you get with the contents of your database in ***psql***.

Query for a Grade by GitHub Username and Project Title

Complete the ***get_grade_by_github_title*** function. Test it in the Python interpreter and compare the output against the data in your database once more.

Give a Grade to a Student

Complete the ***assign_grade*** function. Test the function in the Python interpereter, and then check your ***grades*** table with ***psql*** to see that the new grade made it into the database.

Add New Commands to handle_input

Finally, when you know your new functions work, add code to the ***handle_input*** function to allow users to enter commands that call these functions.

Uncomment the call to ***handle_input*** at the end of ***hackbright.py*** so you can test your new commands when you run the file at the command line. Remember, don't use interactive mode to test ***handle_input***.

Finally, use ***pg_dump hackbright > hackbright.sql*** and check ***hackbright.sql*** into git along with your other files when you are done.

Please stop here and get a code review before proceeding.

If you've received a code review and there is time, you may wish to work on the Further Study <further-study.html>.