Embedded SQL demo

This is the terminal output from the embedded SQL demo we did in class.

For this demo, I had two terminal windows open, both connected to dbsrv1:

- Python window: where I run the python scripts.
- Psql window: where I run SQL commands in the psql shell - in the same way we have been doing so far.

Note that both the SQL commands I run in the "Psql window" as well as the Python programs I run in the "Python window" are accessing the same database. A change made from the psql shell will be visible via a Python program and vice versa.

While completing part 3 on A2, you might find it helpful to also have two windows open. One where you can run your A2 Python program and the other where you can check the effects of running your A2 Python program.

Psycopg2 Basics

We used `psycopg2_basics.py` to introduce the basic components of the `psycopg2` library.

Python window

First, I commented out all the code in the `try` block except for the section titled `SECTION A`. This is the state in which the `psycopg2_basics.py` file is when you first download it from Marina's sections page.

Here is the output:

```
dbsrv1:~/343-26s/afternoon$ python3 psycopg2_basics.py
(99132, 'Avery', 'Marchmount', 'StG', 'avery@cs', Decimal('3.13')) <class 'tuple'>
Student number was 99132 and cgpa was 3.13.
(98000, 'William', 'Fairgrieve', 'StG', 'will@cs', Decimal('4.00')) <class 'tuple'>
Student number was 98000 and cgpa was 4.00.
(99999, 'Afsaneh', 'Ali', 'UTSC', 'aali@cs', Decimal('2.98')) <class 'tuple'>
Student number was 99999 and cgpa was 2.98.
(157, 'Leilani', 'Lakemeyer', 'UTM', 'lani@cs', Decimal('3.42')) <class 'tuple'>
Student number was 157 and cgpa was 3.42.
(11111, 'Homer', 'Simpson', 'StG', 'doh@gmail', Decimal('0.40')) <class 'tuple'>
Student number was 11111 and cgpa was 0.40.
```

Psql window

In psql, I verified that these are indeed the students in my instance:

```
csc343h-marinat=> SET SEARCH_PATH TO University;
SET
csc343h-marinat=> select * from student;
  sid  | firstname |  surname   | campus |   email   | cgpa
-------+-----------+------------+--------+-----------+------
 99132 | Avery     | Marchmount | StG    | avery@cs  | 3.13
 98000 | William   | Fairgrieve | StG    | will@cs   | 4.00
 99999 | Afsaneh   | Ali        | UTSC   | aali@cs   | 2.98
   157 | Leilani   | Lakemeyer  | UTM    | lani@cs   | 3.42
 11111 | Homer     | Simpson    | StG    | doh@gmail | 0.40
(5 rows)
```

We talked about how after issuing a command through `cursor.execute`, our cursor "points" to the first record in our result set. Through the for loop, that cursor keeps advancing, thus consuming one record for each loop iteration.

Python window

I then uncommented out the section titled `SECTION B` as well.

```
dbsrv1:~/343-26s/afternoon$ python3 psycopg2_basics.py
(99132, 'Avery', 'Marchmount', 'StG', 'avery@cs', Decimal('3.13')) <class 'tuple'>
Student number was 99132 and cgpa was 3.13.
(98000, 'William', 'Fairgrieve', 'StG', 'will@cs', Decimal('4.00')) <class 'tuple'>
Student number was 98000 and cgpa was 4.00.
(99999, 'Afsaneh', 'Ali', 'UTSC', 'aali@cs', Decimal('2.98')) <class 'tuple'>
Student number was 99999 and cgpa was 2.98.
(157, 'Leilani', 'Lakemeyer', 'UTM', 'lani@cs', Decimal('3.42')) <class 'tuple'>
Student number was 157 and cgpa was 3.42.
(11111, 'Homer', 'Simpson', 'StG', 'doh@gmail', Decimal('0.40')) <class 'tuple'>
Student number was 11111 and cgpa was 0.40.
[CUR2] Student 99132 from StG has a cgpa of 3.13.
[CUR2] Student 98000 from StG has a cgpa of 4.00.
[CUR2] Student 99999 from UTSC has a cgpa of 2.98.
[CUR2] Student 157 from UTM has a cgpa of 3.42.
[CUR2] Student 11111 from StG has a cgpa of 0.40.
```

This showed us that:

- We can define multiple cursors using the same connection object.
- We were also introduced to a different way of defining cursors through which each record is returned as a dictionary.

Python window

Next, I uncommented the section titled `SECTION C`:

```
dbsrv1:~/343-26s/afternoon$ python3 psycopg2_basics.py
(99132, 'Avery', 'Marchmount', 'StG', 'avery@cs', Decimal('3.13')) <class 'tuple'>
Student number was 99132 and cgpa was 3.13.
(98000, 'William', 'Fairgrieve', 'StG', 'will@cs', Decimal('4.00')) <class 'tuple'>
Student number was 98000 and cgpa was 4.00.
(99999, 'Afsaneh', 'Ali', 'UTSC', 'aali@cs', Decimal('2.98')) <class 'tuple'>
Student number was 99999 and cgpa was 2.98.
(157, 'Leilani', 'Lakemeyer', 'UTM', 'lani@cs', Decimal('3.42')) <class 'tuple'>
Student number was 157 and cgpa was 3.42.
(11111, 'Homer', 'Simpson', 'StG', 'doh@gmail', Decimal('0.40')) <class 'tuple'>
Student number was 11111 and cgpa was 0.40.
[CUR2] Student 99132 from StG has a cgpa of 3.13.
[CUR2] Student 98000 from StG has a cgpa of 4.00.
[CUR2] Student 99999 from UTSC has a cgpa of 2.98.
[CUR2] Student 157 from UTM has a cgpa of 3.42.
[CUR2] Student 11111 from StG has a cgpa of 0.40.
[CUR2 - fetchone result] [99132, 'Avery', 'Marchmount', 'StG', 'avery@cs', Decimal('3.13')]
[CUR2 - fetchone result] [98000, 'William', 'Fairgrieve', 'StG', 'will@cs', Decimal('4.00')]
[CUR2] 5 rows were returned from the query.
```

This was an example where we reused an existing cursor. This also introduced us to `cursor.fetchone()` and `cursor.rowcount`.

Python window

Any statement that we can execute in the psql shell can be executed within a Python program. This includes updates, defining tables, and even postgreSQL commands like SET SEARCH_PATH.

In my Python script, I showed how you can issue an UPDATE (in the section titled `SECTION D`). The update intends to give every student 20 bonus marks, capping the grade at most `100`.

Psql window

Let's first check the contents of our `Took` table before the update.

```
csc343h-marinat=> SELECT * FROM Took;

-- Took had too many rows so let's just focus on one student.

csc343h-marinat=> -- We decided to focus on one specific student.
csc343h-marinat=> -- Took before
csc343h-marinat=> SELECT * FROM Took where sid = 99132;
  sid  | oid | grade
-------+-----+-------
 99132 |   1 |    79
 99132 |  16 |    98
 99132 |  31 |    82
 99132 |  11 |    99
 99132 |  14 |    39
 99132 |  15 |    62
 99132 |  34 |    75
(7 rows)

csc343h-marinat=>
```

These are the student's grades before the update.

Python window

I then uncommented the section titled `SECTION D` - This is the section that updates all students' grades by adding 20 points and capping the grade at 100.

We then re-ran the code:

```
dbsrv1:~/343-26s/afternoon$ python3 psycopg2_basics.py
(99132, 'Avery', 'Marchmount', 'StG', 'avery@cs', Decimal('3.13')) <class 'tuple'>
Student number was 99132 and cgpa was 3.13.
(98000, 'William', 'Fairgrieve', 'StG', 'will@cs', Decimal('4.00')) <class 'tuple'>
Student number was 98000 and cgpa was 4.00.
(99999, 'Afsaneh', 'Ali', 'UTSC', 'aali@cs', Decimal('2.98')) <class 'tuple'>
Student number was 99999 and cgpa was 2.98.
(157, 'Leilani', 'Lakemeyer', 'UTM', 'lani@cs', Decimal('3.42')) <class 'tuple'>
Student number was 157 and cgpa was 3.42.
(11111, 'Homer', 'Simpson', 'StG', 'doh@gmail', Decimal('0.40')) <class 'tuple'>
Student number was 11111 and cgpa was 0.40.
[CUR2] Student 99132 from StG has a cgpa of 3.13.
[CUR2] Student 98000 from StG has a cgpa of 4.00.
[CUR2] Student 99999 from UTSC has a cgpa of 2.98.
[CUR2] Student 157 from UTM has a cgpa of 3.42.
[CUR2] Student 11111 from StG has a cgpa of 0.40.
[CUR2 - fetchone result] [99132, 'Avery', 'Marchmount', 'StG', 'avery@cs', Decimal('3.13')]
[CUR2 - fetchone result] [98000, 'William', 'Fairgrieve', 'StG', 'will@cs', Decimal('4.00')]
[CUR2] 5 rows were returned from the query.
[CUR2] 54 rows were affected by the update.
```

Note how the last printed line mentions that 54 rows were affected by the update. This is the same number of rows in Took. Great! If I switch to the psql window now, all grades should be increased by 20 points.

Psql window

Let's now see the change to Took:

```
csc343h-marinat=> SELECT * FROM Took where sid = 99132;
  sid  | oid | grade
-------+-----+-------
 99132 |   1 |    79
 99132 |  16 |    98
 99132 |  31 |    82
 99132 |  11 |    99
 99132 |  14 |    39
 99132 |  15 |    62
 99132 |  34 |    75
(7 rows)
```

Hmmmm! It doesn't look like anything changed - What gives? If you ever encounter this issue, you most probably didn't commit your changes.

Python window

I then uncommented the one line under the scetion titled `SECTION E` - this line commits our changes so that they are actually reflected in the database.

Without this line, the moment you close the connection, all the changes will be undone (we say that they are "rolled back"). No changes will actually be committed to the database until you call `commit`.

`commit` will apply all the changes you made since you opened the connection, or since the previous commit if you have already called that method. Notice that I use `conn.commit()` and not `cur2.commit()`, since you are committing all changes made over this connection from any cursor.

Let's run our code again now.

```
dbsrv1:~/343-26s/afternoon$ python3 psycopg2_basics.py
(99132, 'Avery', 'Marchmount', 'StG', 'avery@cs', Decimal('3.13')) <class 'tuple'>
Student number was 99132 and cgpa was 3.13.
(98000, 'William', 'Fairgrieve', 'StG', 'will@cs', Decimal('4.00')) <class 'tuple'>
Student number was 98000 and cgpa was 4.00.
(99999, 'Afsaneh', 'Ali', 'UTSC', 'aali@cs', Decimal('2.98')) <class 'tuple'>
Student number was 99999 and cgpa was 2.98.
(157, 'Leilani', 'Lakemeyer', 'UTM', 'lani@cs', Decimal('3.42')) <class 'tuple'>
Student number was 157 and cgpa was 3.42.
(11111, 'Homer', 'Simpson', 'StG', 'doh@gmail', Decimal('0.40')) <class 'tuple'>
Student number was 11111 and cgpa was 0.40.
[CUR2] Student 99132 from StG has a cgpa of 3.13.
[CUR2] Student 98000 from StG has a cgpa of 4.00.
[CUR2] Student 99999 from UTSC has a cgpa of 2.98.
[CUR2] Student 157 from UTM has a cgpa of 3.42.
[CUR2] Student 11111 from StG has a cgpa of 0.40.
[CUR2 - fetchone result] [99132, 'Avery', 'Marchmount', 'StG', 'avery@cs', Decimal('3.13')]
[CUR2 - fetchone result] [98000, 'William', 'Fairgrieve', 'StG', 'will@cs', Decimal('4.00')]
[CUR2] 5 rows were returned from the query.
[CUR2] 54 rows were affected by the update.
```

Psql window

Now, we can see the changes reflected in our database. Yeay!

```
csc343h-marinat=> SELECT * FROM Took where sid = 99132;
  sid  | oid | grade
-------+-----+-------
 99132 |   1 |    99
 99132 |  16 |   100
 99132 |  31 |   100
 99132 |  11 |   100
 99132 |  14 |    59
 99132 |  15 |    82
 99132 |  34 |    95
(7 rows)
```

Python window

Next, we considered the code in the except block. `rollback` and `commit` do pretty much the opposite thing. While `commit` commits all changes that have happened so far. `rollback` undoes them.

To motivate the need for `rollback`, I went over the following brief demo.

Psql window

To motivate rollback, let's consider a problemmatic insert statement:

```
csc343h-marinat=> select * from took limit 1;
  sid  | oid | grade
-------+-----+-------
 99132 |   1 |    79
(1 row)

csc343h-marinat=> insert into took values (99132, 1, 95);
ERROR:  duplicate key value violates unique constraint "took_pkey"
DETAIL:  Key (sid, oid)=(99132, 1) already exists.
```

It is probably not surprising that this error occurred - inserting this row violates our key constraint, but how does this look like from the Python code?

Python window

You can imagine that if I am receiving input from users, I might run into these issues, and if an error occurs, I won't be able to use the connection until I rollback.

Notice that we can do everything that is in our Python program step by step in the Python shell. This might be helpful if you want to quickly test a feature of `psycopg2`.

```
dbsrv1:~/343-26s/afternoon$ python3
Python 3.13.7 (main, Sep  2 2025, 16:35:01) [GCC 13.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> # Add your import(s)
>>> import psycopg2 as pg
>>>
>>> # Make a connection
>>> conn = pg.connect(
...         dbname="csc343h-marinat", user="marinat", password="",
...         options="-c search_path=university,public"
```

```
...     )
>>> # conn now refers to a psycopg2 connection object.
>>> conn
<connection object at 0x785d973f8b80; dsn: 'user=marinat password=xxx dbname=csc343h-marinat options='-
>>>
>>> # Let's create a cursor and issue a simple command.
>>> cur1 = conn.cursor()
>>> cur1.execute("SELECT * FROM Student;")
>>>
>>> # I was asked in lecture what happens if I issue another command without
>>> # consuming the previous result set.
>>> cur1.execute("select * from took;")
>>> cur1.rowcount
54
>>> # Our cursor just refer to the first record in our new result set
>>> # (the rows in Took).
>>> cur1.fetchone()  # Gets us a row in Took
(99132, 1, 79)
>>>
>>>
>>> # Back to causing an error - I am using the same command I used in the psql shell.
>>> cur1.execute("insert into took values (99132, 1, 95);")
Traceback (most recent call last):
  File "<python-input-21>", line 1, in <module>
    cur1.execute("insert into took values (99132, 1, 95);")
    ~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
psycopg2.errors.UniqueViolation: duplicate key value violates unique constraint "took_pkey"
DETAIL:  Key (sid, oid)=(99132, 1) already exists.

>>> # Again, not that surprising
>>> # But now, we can't use the cursor to issue a simple command.
>>> cur1.execute("select * from student;")
Traceback (most recent call last):
  File "<python-input-22>", line 1, in <module>
    cur1.execute("select * from student;")
    ~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^
psycopg2.errors.InFailedSqlTransaction: current transaction is aborted, commands ignored until end of t

>>> # No problem, let's try a new cursor - that ought to work, right?
>>> cur2 = conn.cursor()
>>> cur2.execute("select * from student;")
Traceback (most recent call last):
  File "<python-input-24>", line 1, in <module>
    cur2.execute("select * from student;")
    ~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^
psycopg2.errors.InFailedSqlTransaction: current transaction is aborted, commands ignored until end of t

>>> # Nope! the connection is unusable until we rollback (undo our changes)
>>> # so any cursor we create using this connection object will have this same issue.
>>>
>>>
>>> # The solution? rollback your connection.
>>> # Note that I call rollback on the connection object and not the cursor.
>>> conn.rollback()
```

```
>>> # Now, I can use the cursors.
>>> cur2.execute("select * from student;")
>>> quit()
```

NOTE: I didn't close the connection in the demo, but you definietly should in your programs (We do that for you in A2).

How does this look like in a Python program?

Go back and look at the code in our `psycopg2_basics.py` file - note how we catch any `pg.Error` exceptions and if one occurs, we rollback our connection.

In the above demo, I was not using a try-except-finally clause. When an error is raised, it is printed at the shell in its raw form. With a try-except, we have the opportunity to shield the user from that and provide nicer messaging, as well as to take stepts to recover from the problem.

Dynamic SQL and preventing injection attacks

So far, we have executed only "static" SQL statements in our Python programs. These are commands that are completely specified at the time of writing the Python program, and that do exactly the same thing every time we run the program. But what if our SQL command needs to be different depending on input to the program? This happens all the time in real code. For example, when you type things into a search box in the interface to an online shopping site, that input may go into a SQL query to find what you're looking for.

In this next demo, we will explore writing Python programs that run so-called "dynamic" SQL statements. Take a look at `dynamic_danger.py`. This is a very simple script that prompts the user to provide information for a new course. Using this information, it issues an `INSERT` command to our database to add this new row.

This is an example of a "dynamic statement" i.e., a statement that use the user's input.

Psql window

We looked at our course data before we run the script:

```
csc343h-marinat=> select * from course;
 cnum |           name            | dept | breadth
------+---------------------------+------+---------
  343 | Intro to Databases        | CSC  | f
  207 | Software Design           | CSC  | f
  148 | Intro to Comp Sci         | CSC  | f
  263 | Data Struct & Anal        | CSC  | f
  320 | Intro to Visual Computing | CSC  | f
  200 | Intro Archaeology         | ANT  | t
  203 | Human Biol & Evol         | ANT  | f
  150 | Organisms in Environ      | EEB  | f
  216 | Marine Mammal Bio         | EEB  | f
  263 | Compar Vert Anatomy       | EEB  | f
  110 | Narrative                 | ENG  | t
  205 | Rhetoric                  | ENG  | t
  235 | The Graphic Novel         | ENG  | t
  200 | Environmental Change      | ENV  | f
  320 | Natl & Intl Env Policy    | ENV  | f
  220 | Mediaeval Society         | HIS  | t
  296 | Black Freedom             | HIS  | t
  222 | COBOL programming         | CSC  | f
(18 rows)
```

Python window

We then ran our program.

```
dbsrv1:~/343-26s/afternoon$ python3 dynamic_danger.py
We are going to add a new course!
Course number: 108
Course name: Intro to CS
Department: CSC
```

Psql window

Let's check out our courses now.

```
csc343h-marinat=> select * from course;
 cnum |           name             | dept | breadth
------+----------------------------+------+---------
  343 | Intro to Databases         | CSC  | f
  207 | Software Design            | CSC  | f
  148 | Intro to Comp Sci          | CSC  | f
  263 | Data Struct & Anal         | CSC  | f
  320 | Intro to Visual Computing  | CSC  | f
  200 | Intro Archaeology          | ANT  | t
  203 | Human Biol & Evol          | ANT  | f
  150 | Organisms in Environ       | EEB  | f
  216 | Marine Mammal Bio          | EEB  | f
  263 | Compar Vert Anatomy        | EEB  | f
  110 | Narrative                  | ENG  | t
  205 | Rhetoric                   | ENG  | t
  235 | The Graphic Novel          | ENG  | t
  200 | Environmental Change       | ENV  | f
  320 | Natl & Intl Env Policy     | ENV  | f
  220 | Mediaeval Society          | HIS  | t
  296 | Black Freedom              | HIS  | t
  222 | COBOL programming          | CSC  | f
  108 | Intro to CS                | CSC  |
(19 rows)
```

You can see that information for our new course got added. Pretty cool!

But what can go wrong if our user is malicious? Most users are pretty harmless and won't try to cause any issues, but we might encounter malicious users. We used the file `devious_inputs.txt` to think about inputs a malicious user might provide which would have determintal effects.

Here is an example - a malicious user might try to inject an update that modifies all students grades and set them to 0.

Let's first remember the current student's grades:

```
csc343h-marinat=> SELECT * FROM Took where sid = 99132;
  sid  | oid | grade
-------+-----+-------
 99132 |   1 |    99
 99132 |  16 |   100
 99132 |  31 |   100
 99132 |  11 |   100
 99132 |  14 |    59
 99132 |  15 |    82
```

```
 99132 |  34 |    95
(7 rows)
```

Python window

This is the critical (and very problematic) line of code in `dynamic_danger.py`:

```
cur.execute(f"INSERT INTO COURSE VALUES ({cnum}, '{name}', '{dept}');")
```

We are going to inject the following SQL statement right after we enter the department name:

```
UPDATE Took set grade = 0;
```

As the user of this program, we shouldn't be able to give everyone a grade of 0 in all their courses! But we can make this work if we are clever. We must:

- enter a valid course number, course name, and department name
- complete the INSERT INTO command that is written in `dynamic_danger.py`, including closing the quote and bracket after entering the department name, and adding a semi-colon.
- enter our UPDATE statement after that
- add the comment symbol (`--`) so that the last bit of the string that the Python program is building, that is, `');` will be commented out. Otherwise, it would cause a syntax error.

Let's see what happens when we try providing our devious user input to the program

```
dbsrv1:~/343-26s/afternoon$ python3 dynamic_danger.py
We are going to add a new course!
Course number: 111
Course name: Malicious
Department: CSC'); UPDATE Took SET grade = 0; --
dbsrv1:~/343-26s/afternoon$
```

Psql window

Now let's check out the grades:

```
csc343h-marinat=> SELECT * FROM Took where sid = 99132;
  sid  | oid | grade
-------+-----+-------
 99132 |   1 |     0
 99132 |  16 |     0
 99132 |  31 |     0
 99132 |  11 |     0
 99132 |  14 |     0
 99132 |  15 |     0
 99132 |  34 |     0
(7 rows)
```

Well that is not good!

Python window

But it can be even worse:

```
dbsrv1:~/343-26s/afternoon$ python3 dynamic_danger.py
We are going to add a new course!
Course number: 112
Course name: Also malicious
Department: CSC'); DROP Table Took cascade;--
dbsrv1:~/343-26s/afternoon$
```

Psql window

Now our Took table is completely gone.

```
csc343h-marinat=> select * from took;
ERROR:  relation "took" does not exist
LINE 1: select * from took;
                      ^
csc343h-marinat=>
```

Imagine if that was a real table with real students' grades. This would be a disaster. Luckily, this is just a demo. Next time, I will recreate the schema and re-import the data and we will talk about the correct way to handle users' inputs.