COMP3311 21T1 Database Systems

Week 08
Python/Psycopg2/SQLite3

Note: Q1-2 make use of the Psycopg2 library, while Q3-5 make use of the sqlite3 library.

1. What is the difference between a *connection* and a *cursor* in Psycopg2? How do you create each?

   Answer:

   A *connection* provides authenticated access to a database. You create a *connection* as follows

   ```
   connection = psycopg2.connect(ConnectionParameters)
   ```

   Typical connection parameters are: `dbname` or `database` (required), `user` and `password` for authentication, and `host` and `port` to specify the "location" of the PostgreSQL server.

   Some examples:

   ```
   conn = psycopg2.connect("dbname=mydb")
   conn = psycopg2.connect(database="mydb")
   conn = psycopg2.connect("user=jas password=abc123 dbname=mymydb)
   conn = psycopg2.connect("host=db.server.org port=5432 dbname=mymydb)
   ```

   A *cursor* provides a pipeline between the Python program and the PostgreSQL database. You create a *cursor* as follows:

   ```
   cursor = connection.cursor()
   ```

   Cursors can be used to send queries to the database and read back rsults as in:

   ```
   cursor.execute("SQL Query")
   results = cursor.fetchall()
   ```

   where `results` is a list of tuples.

2. [Question courtesy of Clifford Sesel] The following Python script (in a executable file called `opendb`) aims to open a connection to a database whose name is specified on the command line:

   ```
   1. #!/usr/bin/python3
   2. import sys
   3. import psycopg2
   4. if len(sys.argv) < 2:
   5.     print("Usage: opendb DBname")
   6.     exit(1)
   7. db = sys.argv[1]
   8. try:
   9.     conn = psycopg2.connect(f"dbname={db}")
  10.     print(conn)
  11.     cur = conn.cursor()
  12. except psycopg2.Error as err:
  13.     print("database error: ", err)
  14. finally:
  15.     if conn is not None:
   ```

```
16.         conn.close()
17.     print("finished with the database")
```

When invoked with an existing database, it behaves as follows

```
$ ./opendb beers2
<connection object at 0x7fac401799f0; dsn: 'dbname=beers2', closed: 0>
finished with the database
```

but when invoked with a non-existent database it produces

```
$ ./opendb nonexistent
database error:  FATAL:  database "nonexistent" does not exist

Traceback (most recent call last):
  File "./opendb", line 16, in
    if conn :
NameError: name 'conn' is not defined
```

rather than

```
$ ./opendb nonexistent
database error:  FATAL:  database "nonexistent" does not exist

finished with the database
```

What is the problem? And how can we fix it?

Answer:

The scope of the `conn`object does not extend into the `finally` clause if `conn` is only initialised in the `try` clause. This is essentially what the error message is telling you:

```
NameError: name 'conn' is not defined
```

Ask the Python developers why variable/object scope works this way.

To fix the problem, you need to initialise `conn` in the outer scope. You can't simply move the `connect()` call there, because it would then be outside the `try` clause and the exception would be handled by the standard Python exception handler rather than our exception handler, e.g.

```
Traceback (most recent call last):
  File "./opendb", line 9, in
    conn = psycopg2.connect(f"dbname={db}")
  File "/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages/psycopg2/__init__.py", line 127, in connect
    conn = _connect(dsn, connection_factory=connection_factory, **kwasync)
psycopg2.OperationalError: FATAL:  database "xyzzyyyy" does not exist
```

The solution is to initialise `conn` outside the `try` clause:

```
db = sys.argv[1]
conn = None
try:
   ...
```

3. Using the `beers` database from Prac 05, write a Python script called `cheapest` that takes one command-line argument (beer name) and outputs the name of the bar which sells that beer at the cheapest price, as well as the price to two decimal places in parentheses. Each line should be in the format: barName ($price). If there are multiple bars selling at the same lowest price, output all of them sorted by alphabetical order.

   Some examples of use:

```
$ ./cheapest many args
Usage: ./cheapest beerName

$ ./cheapest abc
Invalid beerName: abc

$ ./cheapest New
Bar(s) where New is sold the cheapest:
Regent Hotel ($2.20)

$ ./cheapest Old
Bar(s) where Old is sold the cheapest:
Coogee Bay Hotel ($2.50)
```

Answer:

Possible script for `cheapest`:

```
#!/usr/bin/python3
import sys
import sqlite3

if len(sys.argv) != 2:
    print("Usage:",sys.argv[0],"beerName")
    exit(1)

beerName = sys.argv[1]
conn = None
beerCheck = "select * from Beers where name = ?"
query = """
select ba.name, s.price
from   bars ba
       join sells s on ba.id = s.bar
       join beers be on s.beer = be.id
where  be.name = ? and
       s.price = (select min(sells.price)
                  from sells join beers on sells.beer = beers.id
                  where beers.name = ?)
order  by ba.name
"""

try:
    conn = sqlite3.connect('beers.db')
    cur = conn.cursor()
    cur.execute(beerCheck, (beerName,))
    res = cur.fetchone()
    if res is None:
        print("Invalid beerName:", beerName)
```

```
        conn.close()
        exit(1)
    cur.execute(query, (beerName, beerName))
    bars = cur.fetchall()
    print(f"Bar(s) where {beerName} is sold the cheapest:")
    if len(bars) == 0:
        print("Not sold at any bars.")
    else:
        for bar in bars:
            price = str(bar[1])
            if len(price) == 3:
                price += '0'
            print(bar[0], '($' + price + ')')
except Exception as err:
    print("error: ", err)
finally:
    if conn:
        conn.close()
```

4. Still using the beers database, write a Python script called `similar-bars1` that takes two command-line arguments (bar name and N) and outputs the name of the bars which sell at least N same beers as the input bar. Each line should be in the format: barName (numSameBeers). Order the output in descending order of number of same beers sold, then by ascending order of the bar names. Try to do this in the 'less efficient' approach, that is, keeping most of the logic in Python.

   Some examples of use:

```
$ ./similar-bars1 abc
Usage: ./similar-bars1 barName N

$ ./similar-bars1 fakeBar 0
Invalid barName: fakeBar

$ ./similar-bars1 'Marble Bar' 1
Bar(s) which sell at least 1 same beers as Marble Bar:
Coogee Bay Hotel (3)
Royal Hotel (3)
Regent Hotel (2)
Australia Hotel (1)
Lord Nelson (1)

$ ./similar-bars1 'Royal Hotel' 3
Bar(s) which sell at least 3 same beers as Royal Hotel:
Coogee Bay Hotel (3)
Marble Bar (3)

$ ./similar-bars1 'Royal Hotel' 5
No bars sell at least 5 same beers as Royal Hotel.
```

   Answer:

   Possible script for `similar-bars1`:

```
#!/usr/bin/python3
import sys
import sqlite3
```

```
 4.
 5. if len(sys.argv) != 3:
 6.     print("Usage:",sys.argv[0],"barName N")
 7.     exit(1)
 8.
 9. barName = sys.argv[1]
10. nSame = int(sys.argv[2]) # Assumes integer, you can improve the error handling
11. conn = None
12. getBarId = "select id from Bars where name = ?"
13. getBarsExcept = "select id from Bars where id != ?"
14. getBeerIds = "select beer from Sells where bar = ?"
15. getBarName = "select name from Bars where id = ?"
16.
17. try:
18.     conn = sqlite3.connect('beers.db')
19.     cur = conn.cursor()
20.     cur.execute(getBarId, (barName,))
21.     barId = cur.fetchone()
22.     if barId is None:
23.         print("Invalid barName:", barName)
24.         conn.close()
25.         exit(1)
26.
27.     # Get all beers sold by the input bar
28.     cur.execute(getBeerIds, (barId[0],))
29.     res = cur.fetchall()
30.     barBeers = set()
31.     for beer in res:
32.         barBeers.add(beer[0])
33.
34.     # Get all bar ids except the input bar
35.     cur.execute(getBarsExcept, (barId[0],))
36.     res = cur.fetchall()
37.     otherBars = []
38.     for bar in res:
39.         otherBars.append(bar[0])
40.
41.     bars = []
42.     for bar in otherBars:
43.         # For each other bar, get their beer ids
44.         cur.execute(getBeerIds, (bar,))
45.         res = cur.fetchall()
46.         beers = set()
47.         for beer in res:
48.             beers.add(beer[0])
49.
50.         # Count how many are same beers as input bar's beers, add to results
51.         # if passes nSame threshold
52.         same = len(barBeers.intersection(beers))
53.         if same >= nSame:
54.             cur.execute(getBarName, (bar,))
55.             res = cur.fetchone()
56.             bars.append((res[0], same))
57.
```

```
    # Adhere to ordering restrictions
    temp = sorted(bars, key=lambda x: x[0])
    outputBars = sorted(temp, key=lambda x: x[1], reverse=True)

    if len(outputBars) == 0:
        print(f"No bars sell at least {nSame} same beers as {barName}.")
    else:
        print(f"Bar(
            EXAMPLE OUTPUTS HERE
            s) which sell at least {nSame} same beers as {barName}:")
        for bar in outputBars:
            print(bar[0], '(' + str(bar[1]) + ')')
except Exception as err:
    print("error: ", err)
finally:
    if conn:
        conn.close()
```

5. The approach to the previous question can certainly be improved by pushing the logic down to the DB level. Re-write the Python script for `similar-bars1` as a new 'improved' script named `similar-bars2`.

   Some examples of use:

```
$ ./similar-bars2 abc
Usage: ./similar-bars2 barName N

$ ./similar-bars2 fakeBar 0
Invalid barName: fakeBar

$ ./similar-bars2 'Marble Bar' 1
Bar(s) which sell at least 1 same beers as Marble Bar:
Coogee Bay Hotel (3)
Royal Hotel (3)
Regent Hotel (2)
Australia Hotel (1)
Lord Nelson (1)

$ ./similar-bars2 'Royal Hotel' 3
Bar(s) which sell at least 3 same beers as Royal Hotel:
Coogee Bay Hotel (3)
Marble Bar (3)

$ ./similar-bars2 'Royal Hotel' 5
No bars sell at least 5 same beers as Royal Hotel.
```

   Answer:

   Possible script for `similar-bars2`:

```
#!/usr/bin/python3
import sys
import sqlite3

if len(sys.argv) != 3:
```

```
    print("Usage:",sys.argv[0],"barName N")
    exit(1)

barName = sys.argv[1]
nSame = int(sys.argv[2]) # Assumes integer, you can improve the error handling
conn = None
barCheck = "select * from Bars where name = ?"
query = """
select b2.name as name, count(s1.beer) as numSameBeers
from   bars b1
       join sells s1 on b1.id = s1.bar
       join sells s2 on s1.beer = s2.beer
       join bars b2 on b2.id = s2.bar
where  s1.bar != s2.bar and b1.name = ?
group  by s2.bar
having numSameBeers >= ?
order  by numSameBeers desc, name
"""

try:
    conn = sqlite3.connect('beers.db')
    cur = conn.cursor()
    cur.execute(barCheck, (barName,))
    res = cur.fetchone()
    if res is None:
        print("Invalid barName:", barName)
        conn.close()
        exit(1)
    cur.execute(query, (barName, nSame))
    bars = cur.fetchall()
    if len(bars) == 0:
        print(f"No bars sell at least {nSame} same beers as {barName}.")
    else:
        print(f"Bar(s) which sell at least {nSame} same beers as {barName}:")
        for bar in bars:
            print(bar[0], '(' + str(bar[1]) + ')')
except Exception as err:
    print("error: ", err)
finally:
    if conn:
        conn.close()
```