Lab-4-msarker000

March 5, 2019

0.1 Preliminaries: Setup and Imports

```
In [40]: import sqlalchemy
         from sqlalchemy import create_engine
         from sqlalchemy import inspect

         import pandas as pd
         import unittest
         import math
         from IPython.display import display, HTML
```

0.1.1 Open Database and Create Tables

In the following snippet we create the database in the system temporary directory `/tmp`. You can move the database elsewhere if you like.

I highly recommend opening the database from the `sqlite3` console tool like so:

```
$ sqlite3 /tmp/customers.db
```

Once inside the database, you can issue SQL commands directly. For example:

```
sqlite> select * from customers;
id          name
----------  ----------
1           john
2           sam
3           sally
4           paul
5           liza
```

It may be easier to learn the various forms by playing around with SQL statements like I do above. Try out various commands. See if it matches your expectations for the statement.

```
In [41]: db_name = 'sqlite:////tmp/customers.db'


         engine = create_engine(db_name, echo=False)
         print(sqlalchemy.__version__)
         print(engine)
```

```
1.2.7
Engine(sqlite:////tmp/customers.db)
```

0.1.2 Create the Customer Table

```
In [42]: drop_table_statement = """drop table if exists customers"""
         engine.execute(drop_table_statement)

         # sql statement
         create_table_stmt = """create table customers(
           id integer primary key,
           name text not null
         );
         """
         engine.execute(create_table_stmt)
```

```
Out[42]: <sqlalchemy.engine.result.ResultProxy at 0x10cad8400>
```

0.1.3 Populate the Customer Table

```
In [43]: customer_list = [
             "john", "sam", "sally", "paul", "liza"
         ]

         insert_statement = """
         insert into customers (name)
           values(?)
         """

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

```
inserting john
inserting sam
inserting sally
inserting paul
inserting liza
```

0.1.4 Read the Customer Table

Here is what the `customers` table looks like.

Notice that we're using Pandas `read_sql` method to insert the result directly into a DataFrame. This will let us pretty print the table and also run assertions in the unit tests that go with each problem.

```
In [44]: d = pd.read_sql("select * from customers", engine)
         d

Out[44]:    id   name
         0   1   john
         1   2    sam
         2   3  sally
         3   4   paul
         4   5   liza
```

0.1.5 Create the Orders Table

```
In [45]: drop_orders_statement = """drop table if exists orders"""
         engine.execute(drop_orders_statement)

         # sql statement
         create_orders_table_stmt = """create table orders(
           id integer primary key,
           customer_id integer,
           amount float not null
         );
         """
         engine.execute(create_orders_table_stmt)

Out[45]: <sqlalchemy.engine.result.ResultProxy at 0x10cb9ed30>
```

0.1.6 Populate the Orders Table

```
In [46]: orders_list = [
             (20.99, 1),
             (55.00, 1),
             (33.99, 66),
             (190.72, 5),
             (12.33, 4)
         ]

         insert_orders_statement = """
         insert into orders (amount, customer_id)
           values(?,?)
         """

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

inserting 20.99
inserting 55.0
inserting 33.99
inserting 190.72
inserting 12.33
```

0.1.7 Read the Orders Table

```
In [47]: d = pd.read_sql("select * from orders", engine)

         d
```

```
Out[47]:    id  customer_id  amount
         0   1            1   20.99
         1   2            1   55.00
         2   3           66   33.99
         3   4            5  190.72
         4   5            4   12.33
```

1 Homework

60 Points Total

1.1 Problem 1: List Orders For Each Customer

10 Points

List all orders for each customer. Return a relation with the following columns:

customer_id	name	order_id	Amount
Customer ID	Customer Name	Order ID	Order Amount

The above relation will **not** include rows for customers that don't have associated orders.

In this and all subsequent problemsyou should fill in the `query` variable in the `setUpClass` method of the unit test. The test code is written for you and will ascertain whether your query meets the specification.

1.1.1 Aliasing

You will almost certainly use table and attribute **aliasing**. Aliasing can help shorten SQL statements. But more importantly, they *disambiguate* field/attribute names in joins where the joined tables have attributes with the same name. Often, we want to disambiguate IDs, the primary key.

Consider two examples:

```
select c.id, c.name from customers as c;
```

In the above `customers d` creates an alias `c` for `customers`. This alias can then be used to dereference column names (`c.id`, `c.name`).

In the above context an alias is hardly useful. You could have done the same with `select id, name from customers`. But aliases come into their own when multiple tables are involved and you need to disambiguate or rename common attributes. So, for example when joining `customers` and `orders` both tables have an `ID` key. We can use aliases to rename the `ID` attributes so they don't clash:

```
select c.id as customer_id, c.name as customer_name, o.id as order_id
    from customers as c
    join orders as o
    on o.customer_id = c.id
```

```
In [48]: class Problem1Test(unittest.TestCase):

             @classmethod
             def setUpClass(cls):
                 query = """
                 SELECT c.id, c.name, o.id as order_id , o.amount
                 from customers as c inner join orders as o
                 on c.id = o.customer_id
                 """
                 cls.df = pd.read_sql(query, engine)
                 display(cls.df)

             def test_query(self):

                 df = self.df
                 self.assertTrue(type(df))
                 self.assertEqual(len(df), 4)

                 keys = df.keys()
                 self.assertIn('id', keys)
                 self.assertIn('name', keys)
                 self.assertIn('order_id', keys)
                 self.assertIn('amount', keys)

                 # john should have two orders
                 self.assertEqual(len(df[df['name'] == 'john']), 2)

                 # sally won't be in the results
                 x = df['name'] == 'sally'
                 self.assertNotIn(True, enumerate(x))

                 # sam won't be in the results
                 x = df['name'] == 'sam'
                 self.assertNotIn(True, enumerate(x))


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

```
   id  name  order_id  amount
```

```
0    1  john          1    20.99
1    1  john          2    55.00
2    5  liza          4   190.72
3    4  paul          5    12.33


test_query (__main__.Problem1Test) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.011s

OK


Out[48]: <unittest.main.TestProgram at 0x10cbd9198>
```

1.2 Problem 2: List Customers With No Orders

10 Points

List all customers for which no orders exist. The resulting relation will have the following format:

id	name
Customer ID	Customer Name

```
In [49]: class Problem2Test(unittest.TestCase):

             @classmethod
             def setUpClass(cls):
                 query = """
                     SELECT c.id, c.name from customers as c
                     left join orders as o
                     on c.id = o.customer_id
                     where o.customer_id is NULL
                 """
                 cls.df = pd.read_sql(query, engine)
                 display(cls.df)

             def test_query(self):

                 df = self.df

                 self.assertEqual(len(df), 2)

                 keys = df.keys()
                 self.assertIn('id', keys)
                 self.assertIn('name', keys)
```

```
                self.assertTrue((df['name'] == 'sally').any)
                self.assertTrue((df['name'] == 'sam').any)


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

```
   id   name
0   2    sam
1   3  sally


test_query (__main__.Problem2Test) ... ok

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

OK
```

```
Out[49]: <unittest.main.TestProgram at 0x10cc21d30>
```

1.3 Problem 3: Associate Customer Name with Orders

10 Points

For each `order` list the customer name associated with the order. If no customer exists for an order omit the row.

The resulting relation will have the following attributes:

order_id	customer_name	amount
Order ID	Customer Name	Order Amount

```
In [50]: class Problem3Test(unittest.TestCase):

             @classmethod
             def setUpClass(cls):
                 query = """
                 SELECT o.id as order_id, c.name as customer_name, o.amount
                 from orders as o
                 inner join customers as c
                 on o.customer_id = c.id
                 """
                 cls.df = pd.read_sql(query, engine)
                 display(cls.df)

             def test_query(self):
                 df = self.df

                 self.assertEqual(len(df), 4)

                 keys = df.keys()
         #        self.assertIn('id', keys)
                 self.assertIn('order_id', keys)
                 self.assertIn('customer_name', keys)
                 self.assertIn('amount', keys)

                 for name in ['john', 'liza', 'paul']:
                     self.assertTrue((df['customer_name'] == name).any)


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

```
   order_id customer_name  amount
0         1          john   20.99
1         2          john   55.00
2         4          liza  190.72
3         5          paul   12.33


test_query (__main__.Problem3Test) ... ok

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

OK
```

```
Out[50]: <unittest.main.TestProgram at 0x10cc24630>
```

1.4 Problem 4: List Orders Per Customers, Include Customers Without Orders

10 Points

For each customer, list the orders associated with the customer. However, in the case where a customer does not have any orders include the customer in the output relation.

customer_id	customer_name	order_id	amount
Customer ID	Customer Name	Order ID	Order Amount

Some customers (e.g., John) have multiple orders. Others have none. In contrast with the problem above, also include the customers that don't have any rows in the result. These rows will have NULL values for their respective `order_id` and `amount` attributes.

```
In [51]: class Problem4Test(unittest.TestCase):
        @classmethod
        def setUpClass(cls):
            query = """
            SELECT c.id as customer_id, c.name as customer_name,
            o.id as order_id, o.amount
            from customers as c left join orders as o
            on c.id = o.customer_id
            """
            cls.df = pd.read_sql(query, engine)
            display(cls.df)

        def test_query(self):

            df = self.df
            self.assertTrue(type(df))
            self.assertEqual(len(df), 6)

            keys = df.keys()
            self.assertIn('customer_id', keys)
            self.assertIn('customer_name', keys)
            self.assertIn('order_id', keys)
            self.assertIn('amount', keys)

            # john should have two orders
            self.assertEqual(len(df[df['customer_name'] == 'john']), 2)

            for name in ['sam', 'sally']:
                r = df[df['customer_name'] == name].iloc[0]
                self.assertTrue(math.isnan(r['order_id']))
                self.assertTrue(math.isnan(r['amount']))


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

```
   customer_id customer_name  order_id  amount
0            1          john       1.0   20.99
1            1          john       2.0   55.00
2            2           sam       NaN     NaN
3            3         sally       NaN     NaN
4            4          paul       5.0   12.33
5            5          liza       4.0  190.72


test_query (__main__.Problem4Test) ... ok

----------------------------------------------------------------------
```

```
Ran 1 test in 0.013s

OK
```

```
Out[51]: <unittest.main.TestProgram at 0x10cc21a20>
```

1.5 Problem 5: Compute Total Amount Spent Per Customer

For each customer, list the `total` spend for that customer. That is you will sum the totals for each order by a customer. If a customer does not have any associated orders, print `0`. The resulting relation will have a single row for each customer in the customers table.

The output table will have the following attributes:

customer_id	customer_name	order_count	total
Order ID	Customer Name	Number of orders per customer	Total Amount Spent or 0

You should use the SQL coalesce function to replace a NULL value for `total` with a zero. Use the count function to compute the number of orders per customer

```
In [52]: class Problem5Test(unittest.TestCase):

             @classmethod
             def setUpClass(cls):
                 query = """
                      SELECT c.id as customer_id, c.name as customer_name,
                      COUNT(o.id) as order_count,
                      SUM(CASE WHEN o.amount is NULL THEN 0 ELSE o.amount END) as total
                      from customers as c left join orders as o
                      on c.id = o.customer_id  group by c.id
                 """

                 #query = """
                 #     SELECT c.id as customer_id, c.name as customer_name, COUNT(o.id) as ord
                 #     SUM(COALESCE(o.amount, 0)) as total
                 #     from customers as c left join orders as o on c.id = o.customer_id  grou
                 #"""

                 cls.df = pd.read_sql(query, engine)
                 display(cls.df)

             def test_query(self):

                 df = self.df
                 self.assertTrue(type(df))
                 self.assertEqual(len(df), 5)
```

```
        keys = df.keys()
        self.assertIn('customer_id', keys)
        self.assertIn('customer_name', keys)
        self.assertIn('order_count', keys)
        self.assertIn('total', keys)

        expected = {
            'john': (2, 75.99),
            'sam': (0, 0.00),
            'sally': (0, 0.00),
            'paul': (1, 12.33),
            'liza': (1, 190.72),
        }

        for name, val in expected.items():
            cnt = val[0]
            total = val[1]
            r = df[df['customer_name'] == name].iloc[0]
            self.assertEqual(r['order_count'], cnt)
            self.assertEqual(r['total'], total)
```

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

```
   customer_id customer_name  order_count   total
0            1          john            2   75.99
1            2           sam            0    0.00
2            3         sally            0    0.00
3            4          paul            1   12.33
4            5          liza            1  190.72


test_query (__main__.Problem5Test) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.020s

OK
```

```
Out[52]: <unittest.main.TestProgram at 0x10cc30710>
```

1.6 Problem 6: Find Customers Who Spent More than $70

10 Points

This problem is identical to the one above, except that you will filter out customers who spent less in total than $70. As above, your result relation will have the following columns:

customer_id	customer_name	order_count	total
Order ID	Customer Name	Number of orders per customer	Total Amount Spent or 0

```
In [53]: class Problem6Test(unittest.TestCase):

             @classmethod
             def setUpClass(cls):
                 query = """
                      SELECT c.id as customer_id, c.name as customer_name,
                      COUNT(o.id) as order_count,
                      SUM(CASE WHEN o.amount is NULL THEN 0 ELSE o.amount END) as total
                      from customers as c left join orders as o
                      on c.id = o.customer_id group by c.id
                      having total > 70
                 """
                 cls.df = pd.read_sql(query, engine)
                 display(cls.df)

             def test_query(self):

                 df = self.df
                 self.assertTrue(type(df))
                 self.assertEqual(len(df), 2)

                 keys = df.keys()
                 self.assertIn('customer_id', keys)
                 self.assertIn('customer_name', keys)
                 self.assertIn('order_count', keys)
                 self.assertIn('total', keys)

                 expected = {
                     'john': (2, 75.99),
                     'liza': (1, 190.72),
                 }

                 for name, val in expected.items():
                     cnt = val[0]
                     total = val[1]
                     r = df[df['customer_name'] == name].iloc[0]
                     self.assertEqual(r['order_count'], cnt)
                     self.assertEqual(r['total'], total)



         # Run the unit tests
         unittest.main(defaultTest="Problem6Test", argv=['ignored', '-v'], exit=False)
```

```
   customer_id customer_name  order_count   total
0            1          john            2   75.99
1            5          liza            1  190.72


test_query (__main__.Problem6Test) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.013s

OK
```

```
Out[53]: <unittest.main.TestProgram at 0x10cc37d68>
```