SQL-Python

Erich Wellinger

April 6, 2017

Objectives

Today's objectives:

- Learn how to connect and run `Postgres` queries from Python
- Understand `psycopg2`'s cursors, specifically executes and commits
- Learn how to generate dynamic queries from within Python

Agenda

- SQL-Python motivation
- `psycopg2` introduction and workflow
  - Creating databases
  - Executing queries (static and dynamic)

SQL-Python Motivation

- Why write SQL queries from within Python?
  - Allows for the combination of data sources all in one place (e.g. you can use Python to pull data from other databases as well)
  - Allows for use of all our Python tools when working with the data (dataframes, machine learning models, etc.)
  - Allows for more easy dynamic query generation and hence automations

psycopg2

- psycopg2 is the Python library that we'll use to interface to a Postgres database from within a Python script
  - Install this package by running the following command from the command line: `$ conda install psycopg2`
- There are a variety of Python libraries to connect to almost any kind of database that you might want to:
  - MySql: `mysql-connector-python`
  - SQLite: `sqlite`
  - MongoDB: `pymongo` (we'll work with this later in this class)

`psycopg2` Workflow

- There are five general steps to a `psycopg2` workflow:
  1. Open a connection
  2. Create a `cursor` object
  3. Use the `cursor` to execute SQL queries
  4. `Commit` SQL actions
  5. Close the cursor and connection

1. Open a Connection

```
1 import psycopg2
2
3 conn = psycopg2.connect(dbname='election_data',
4                         user='ewellinger', host='localhost')
```

- The `host` can be used to connect to a remote database as well
- Depending on the security measures in place, you may have to put in the `password` or `port` arguments as well

2. Create a `cursor` Object

```
1  cur = conn.cursor()
```

- A **cursor** is a control structure that enables traversal over the records in a database
  - Executes queries to fetch data
  - Handles transactions with our SQL database
- When results are returned from a cursor object, they are returned as a generator (e.g. it gives back the results lazily)
- Furthermore, each result in the result set can only be accessed **once** (if we want it again we have to re-run the query)

3. Use the `cursor` to `execute` SQL queries

```
1  query = '''SELECT headline
2             FROM articles
3             WHERE source = 'fox_news';'''
4  cur.execute(query)
```

- The `cursor` object (`cur`) now holds the results to this query

Getting results from the `cursor`

- There are a number of ways to grab results from the `cursor`:
  - `cur.fetchone()` - Returns the next result
  - `next(cur)` - Returns the next result
  - `cur.fetchmany(n)` - Returns the next `n` results
  - `cur.fetchall()` - Returns all results in the result set
  - `for res in cur:` - Iterates over all results in the cursor

4. Commit the Results

- Data changes are not actually stored until you **commit** them
  - This is only important if you are creating a database/datatable, or altering the data in an existing database/datatable
  - NOTE: You will very rarely be doing this...

```
1 query = '''ALTER TABLE articles
2            RENAME ted_cruz TO lucifer_in_the_flesh;'''
3 cur.execute(query)
4 conn.commit()
```

- The column's name is **not changed** in the db until we issue the `commit`
  - It is, however, changed on the `connection` and `cursor` as soon as we issue the `execute`

Rollback

- If you make a mistake on a query, you need to use the rollback function to restart the transaction

```
1 conn.rollback()
```

5. Close the Connection

- Don't forget to do this!!

```
1 cur.close()
2 conn.close()
```

- Closing the `cursor` is technically optional because closing the `connection` close all `cursor` objects associated with it, but it is good practice to close both

Getting Column Names

You will notice that iterating through the cursor object will just give you tuples of the raw data. psycopg2 does allow us to get information about those various fields like so:

```
1  import psycopg2
2
3  conn = psycopg2.connect(dbname='socialmedia', user='postgres',
4                          host='/tmp')
5  cur = conn.cursor()
6
7  query = '''SELECT * FROM messages;'''
8
9  cur.execute(query)
10 colnames = [desc[0] for desc in cur.description]
```

A Short Note

- Anything executed through the `query` method on the `cursor` is done so as a **temporary transaction**. Since Postgres doesn't have these at the database level, we have to specify an additional attribute on the connection before trying to perform database level operations (create/dropping databases)

```
1 conn.psycopg2.connect(dbname='election_data',
2                       user='ewellinger', host='localhost')
3 conn.autocommit = True
```

Dynamic Queries

- psycopg2 gives us the ability to create dynamic queries where we can insert certain values into our queries on the fly

Dynamic Queries - The Wrong Way

- **THE WRONG WAY** to write a dynamic query:

```
1 unsafe_query = '''SELECT * FROM users
2                   WHERE name = {0}'''.format(name_var)
```

- Does anyone know why this is unsafe?

Dynamic Queries - The Wrong Way

- **THE WRONG WAY** to write a dynamic query:

```
1 unsafe_query = '''SELECT * FROM users
2                   WHERE name = {0}'''.format(name_var)
```

- Does anyone know why this is unsafe?
- What happens if somebody inputs a `name_var` equal to `'Erich'; DROP TABLE users;`?

Dynamic Queries - The Wrong Way

- **THE WRONG WAY** to write a dynamic query:

```
1 unsafe_query = '''SELECT * FROM users
2                   WHERE name = {0}'''.format(name_var)
```

- Does anyone know why this is unsafe?
- What happens if somebody inputs a `name_var` equal to `'Erich'; DROP TABLE users;`?
- Something like this is referred to as a **SQL INJECTION**

Dynamic Queries - The Right Way

- The proper way to write a dynamic query is to use the `execute` method on our `cursor` object, passing the dynamic part as the second argument

```
1 cur.execute('''SELECT * FROM
2                users WHERE name = %s''', (name_var, ))
```

- This ensures that the variables you are inserting are kept as the same variable type. If we tried to perform SQL injection using the `execute` method, it would look for a `name` exactly equal to this string rather than potentially executing further commands.