Using Your SQL Database in Python

How to Connect to and Protect Your SQL Database

galvanize

Morning Objectives

- Install, import, and connect to your PostgreSQL with psycopg2
- Run queries to retrieve data from a database
- Recognize and use techniques that avoid SQL Injection Attacks

First Thing’s First! Installing Psycopg2

In your terminal (and not in python or ipython shells):

```
conda install psycopg2
```

Motivation!

- We want to be able to retrieve data and add data to our database while in python!
- We want to automate some of our processes like writing repetitive queries or populating tables with new data.
- We want to perform dynamic queries (queries with inputs from users for example)
- We want to create a “pipeline” to handle that process!

Basic Psycopg2 Flow

1. Create a connection object in python to connect to your PostgreSQL Database
2. Create a cursor from the connection object
   a. The cursor handles database operations through your connection
3. Use the cursor to execute SQL queries
   a. Returns a generator for fetching data rows represented as tuples.
      i. “Fetching” gives you the results from your executed queries
4. Commit SQL actions
   a. Makes sure changes to tables persist! Permanent changes to tables won’t actually be made until commit is performed, which can be a very good thing.
5. *Close* the cursor and connection

Basic Psycopg2 Flow

Why close cursor? Connection?   http://initd.org/psycopg/docs/faq.html#best-practices

"… Note that cursors used to fetch result sets will cache the data and use memory in proportion to the result set size." Open connections may block new attempts to access DB.

Explain to your neighbor:

What does psycopg2 help us do that postgresql/psql alone can’t?

To the iPython notebook!

galvanize

SQL Injection Attacks!
(or “How to avoid getting fired when using SQL”)

galvanize

Psycopg2’s Subtle Solution, String Sanitizing

Some Bad Stuff Prevented

```
>>> SQL = "INSERT INTO authors (name)
VALUES ('%s');" # NEVER DO THIS
>>> data = ("O'Reilly", )
>>> cur.execute(SQL % data) # THIS
WILL FAIL MISERABLY
ProgrammingError: syntax error at or
near "Reilly"
LINE 1: INSERT INTO authors (name)
VALUES ('O'Reilly')
```

Sanitizing in Progress

```
>>> SQL = "INSERT INTO authors (name)
VALUES (%s);" # Note: no quotes
>>> data = ("O'Reilly", )
>>> cur.execute(SQL, data) # Note: no
% operator
```

http://initd.org/psycopg/docs/usage.html

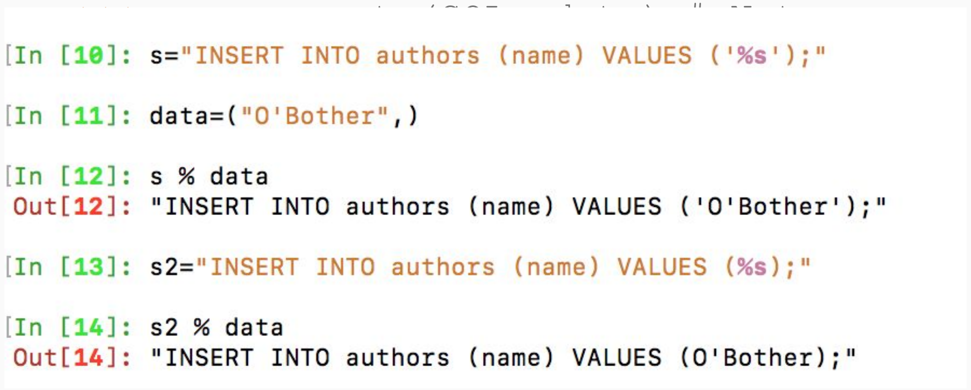
Psycopg2's Subtle Solution, String Sanitizing

Some Bad Stuff Prevented

```
>>> SQL = "INSERT INTO authors (name)
VALUES ('%s');" # NEVER DO THIS
>>> data = ("O'Reilly", )
>>> cur.execute(SQL % data) # THIS
WILL FAIL MISERABLY
ProgrammingError: syntax error at or
near "Reilly"
LINE 1: INSERT INTO authors (name)
VALUES ('O'Reilly')
```

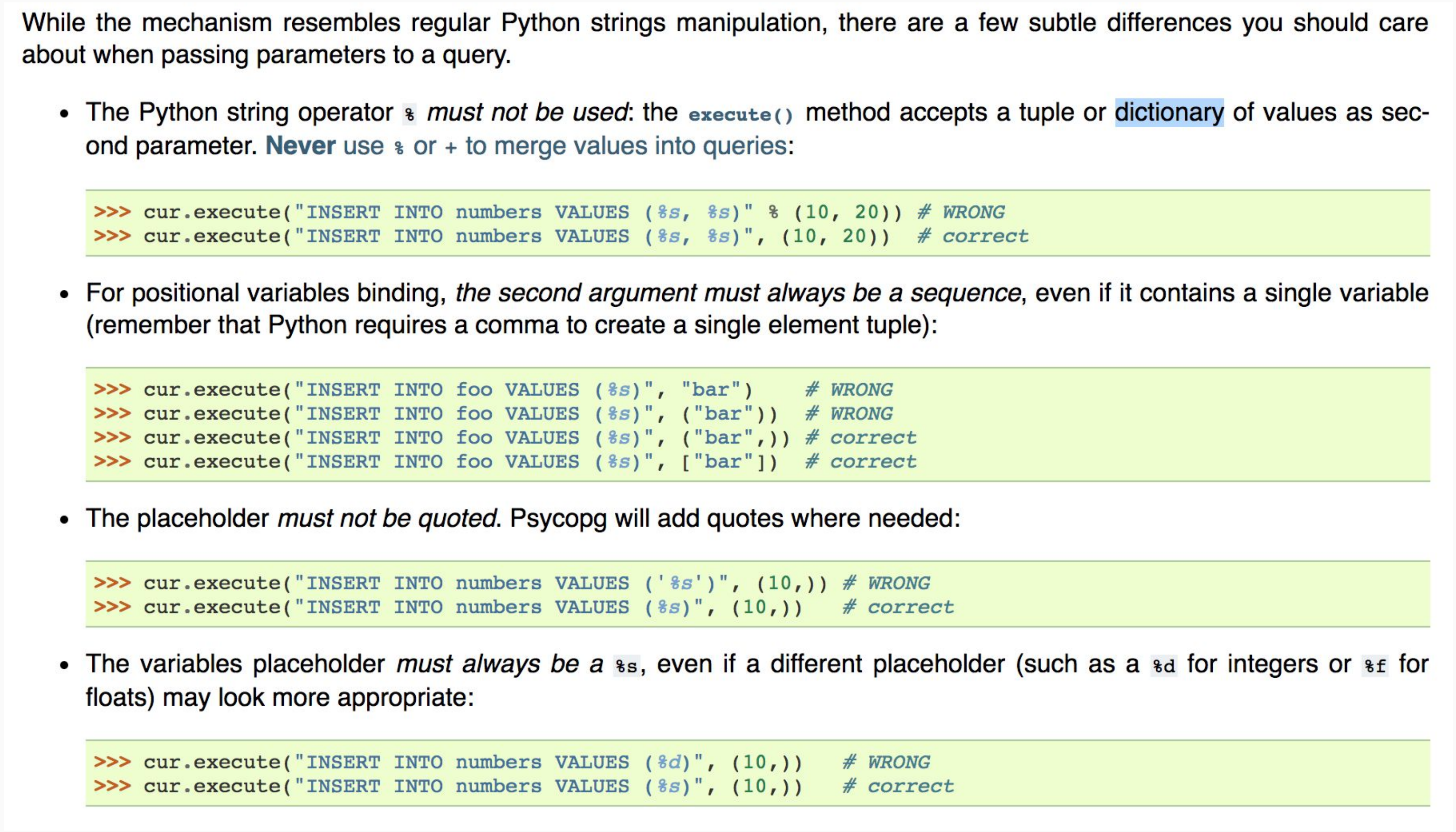
Sanitizing in Progress

```
>>> SQL = "INSERT INTO authors (name)
VALUES (%s);" # Note: no quotes
>>> data = ("O'Reilly", )
```



```
In [10]: s="INSERT INTO authors (name) VALUES ('%s');"

In [11]: data=("O'Bother",)

In [12]: s % data
Out[12]: "INSERT INTO authors (name) VALUES ('O'Bother');"

In [13]: s2="INSERT INTO authors (name) VALUES (%s);"

In [14]: s2 % data
Out[14]: "INSERT INTO authors (name) VALUES (O'Bother);"
```

Psycopg2’s Subtle Solution, String Sanitizing



While the mechanism resembles regular Python strings manipulation, there are a few subtle differences you should care about when passing parameters to a query.

- The Python string operator `%` *must not be used*: the `execute()` method accepts a tuple or dictionary of values as second parameter. **Never** use `%` or `+` to merge values into queries:

```
>>> cur.execute("INSERT INTO numbers VALUES (%s, %s)" % (10, 20)) # WRONG
>>> cur.execute("INSERT INTO numbers VALUES (%s, %s)", (10, 20))  # correct
```

- For positional variables binding, *the second argument must always be a sequence*, even if it contains a single variable (remember that Python requires a comma to create a single element tuple):

```
>>> cur.execute("INSERT INTO foo VALUES (%s)", "bar")    # WRONG
>>> cur.execute("INSERT INTO foo VALUES (%s)", ("bar"))  # WRONG
>>> cur.execute("INSERT INTO foo VALUES (%s)", ("bar",)) # correct
>>> cur.execute("INSERT INTO foo VALUES (%s)", ["bar"])  # correct
```

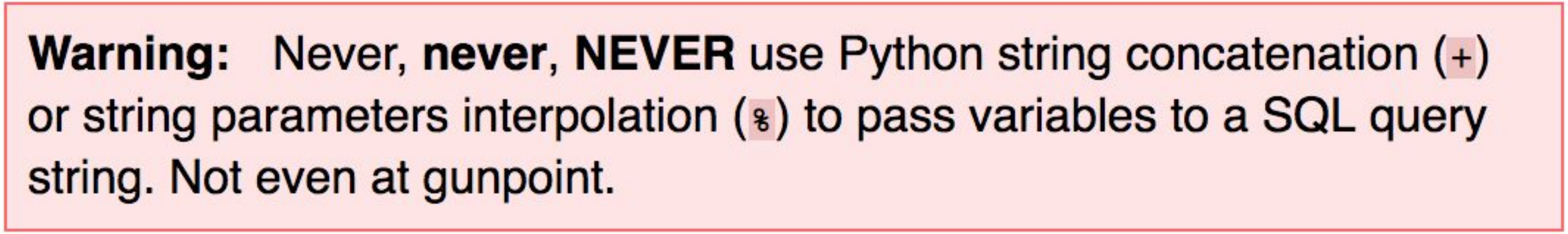
- The placeholder *must not be quoted*. Psycopg will add quotes where needed:

```
>>> cur.execute("INSERT INTO numbers VALUES ('%s')", (10,)) # WRONG
>>> cur.execute("INSERT INTO numbers VALUES (%s)", (10,))   # correct
```

- The variables placeholder *must always be a* `%s`, even if a different placeholder (such as a `%d` for integers or `%f` for floats) may look more appropriate:

```
>>> cur.execute("INSERT INTO numbers VALUES (%d)", (10,))   # WRONG
>>> cur.execute("INSERT INTO numbers VALUES (%s)", (10,))   # correct
```

String Sanitizing Can Only Prevent So Much!



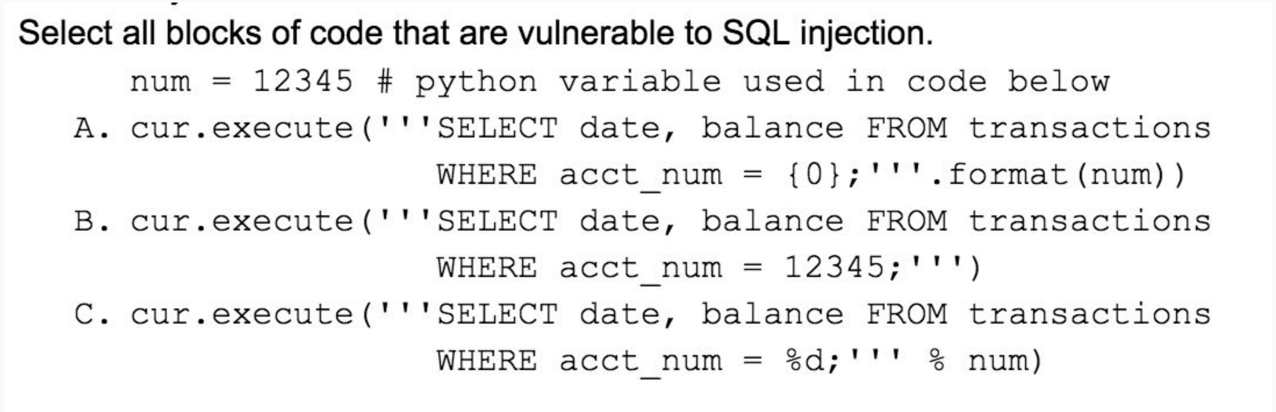
Warning: Never, **never**, **NEVER** use Python string concatenation (`+`) or string parameters interpolation (`%`) to pass variables to a SQL query string. Not even at gunpoint.

Let’s take a poll!:
PollEv.com/markllorente225

Pop Quiz

How would you rewrite any of the faulty blocks of code in the poll to help psycopg2 to sanitize the query?

Reminder:


Select all blocks of code that are vulnerable to SQL injection.

```
    num = 12345 # python variable used in code below
 A. cur.execute('''SELECT date, balance FROM transactions
                   WHERE acct_num = {0};'''.format(num))
 B. cur.execute('''SELECT date, balance FROM transactions
                   WHERE acct_num = 12345;''')
 C. cur.execute('''SELECT date, balance FROM transactions
                   WHERE acct_num = %d;''' % num)
```

Back to the Notebook

galvanize