

Bases de Datos

Clase 6: Programación y SQL

Programación y SQL

- Hasta ahora hemos visto a la Base de Datos como un componente aislado
- Pero una Base de Datos no tiene sentido si no podemos conectarla a una aplicación

Programación y SQL

Contamos con:

- Un DBMS
- Un entorno de programación (Java, Python, PHP...)

Obs: La mayoría de esta clase se orienta a conectar Python con el Sistema PostgreSQL

SQL en Python

Preliminares

IIC2413 / Syllabus-2020-2

<> Code

! Issues 40

🔗 Pull requests

🎬 Actions

📁 Projects

📖 Wiki

🛡 Security

📈 Insights

⚙ Settings

Home

arpincheira edited this page 22 days ago · 4 revisions

Bienvenido al Wiki del curso, aquí subiremos tutoriales que pueden servir para el desarrollo de tu proyecto!

Para la entrega 2

- [Conexión PHP a Postgres](#)

Para la entrega 1

- [Tutorial Servidor y BD PostgreSQL](#)(Servidor, Postgres, Filezilla, etc.)

Otros

- [Tutorial para subir archivos desde editor de texto](#)(Facilitar desarrollo, archivo de configuración listo para usar, etc.)

A continuación, tutoriales que podrían ser útiles para el desarrollo de las clases:

- [Instalación y Ejecución Jupyter Notebooks](#)
- [Instalación Local de SQLite3](#)
- [Instalación Local de PostgreSQL y psycopg2](#)



Conexión a la Base de Datos

<https://www.psycopg.org/docs/>

Psycopg – PostgreSQL database adapter for Python

Psycopg is the most popular PostgreSQL database adapter for the Python programming language. Its main features are the complete implementation of the Python DB API 2.0 specification and the thread safety (several threads can share the same connection). It was designed for heavily multi-threaded applications that create and destroy lots of cursors and make a large number of concurrent INSERTS OR UPDATES.

Psycopg 2 is mostly implemented in C as a libpq wrapper, resulting in being both efficient and secure. It features client-side and server-side cursors, asynchronous communication and notifications, COPY support. Many Python types are supported out-of-the-box and adapted to matching PostgreSQL data types; adaptation can be extended and customized thanks to a flexible objects adaptation system.

Psycopg 2 is both Unicode and Python 3 friendly.

Conexión a la Base de Datos

Python

Primero debemos importar la librería (esta sirve sólo con Postgres):

```
import psycopg2
```

Conexión a la Base de Datos

Python

Objeto para conectar:

```
import psycopg2

try:
    conn = psycopg2.connect(
        database='database',
        user='user',
        host='localhost',
        port=5432,
        password='pass'
    )

    conn.close()

except Exception as e:
    print('Hubo un problema :c')
    print(e)
```


Cursores

Para ejecutar comandos SQL desde Python, debemos usar **cursores**.

Un objeto de la clase cursor nos permite ejecutar un comando SQL en Python.

Inserción

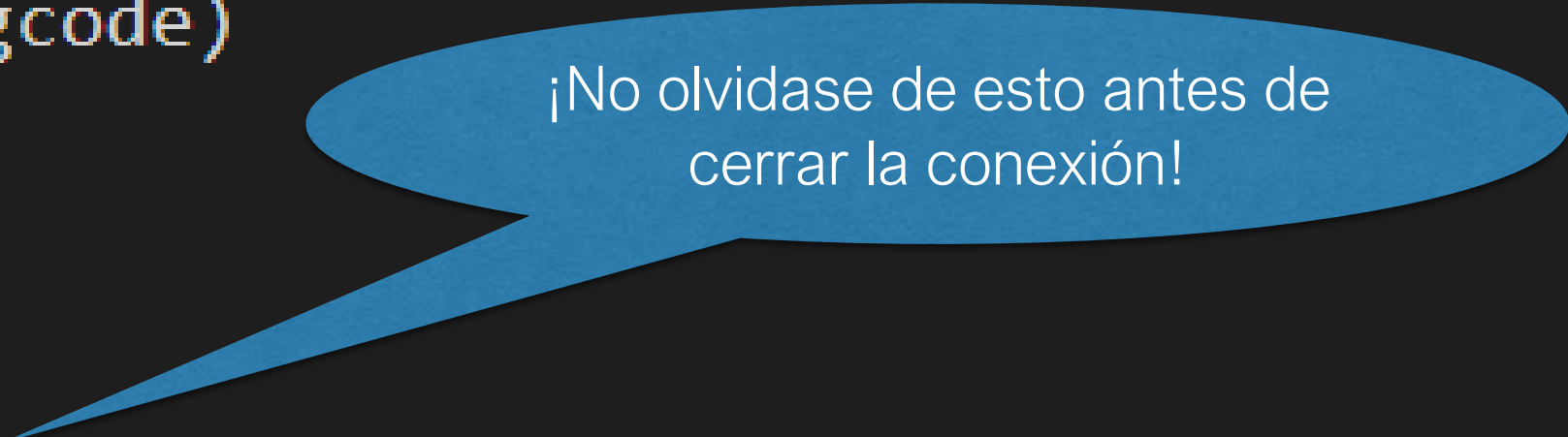
Python

```
query = "INSERT INTO Peliculas VALUES (" + \  
|      | titulo + ", " + año + ", " + director + ")"  
cur = conn.cursor()  
try:  
|      cur.execute(query)  
except psycopg2.Error as e:  
|      print(e.pgcode)  
  
...  
  
conn.commit()
```

Inserción

Python

```
query = "INSERT INTO Peliculas VALUES (" + \  
|      | titulo + ", " + año + ", " + director + ")"  
cur = conn.cursor()  
try:  
|      cur.execute(query)  
except psycopg2.Error as e:  
|      print(e.pgcode)  
  
...  
  
conn.commit()
```



¡No olvidase de esto antes de
cerrar la conexión!

Conexión a la Base de Datos

Python

Objeto para con

Ojo: user tiene que poder hacer la operación

Recordar:

GRANT ALL PRIVILEGES ON DATABASE database TO user;
GRANT ALL PRIVILEGES ON TABLE table TO user

```
import psycopg2

try:
    conn = psycopg2.connect(
        database='database',
        user='user',
        host='localhost',
        port=5432,
        password='pass'
    )

    conn.close()

except Exception as e:
    print('Hubo un problema :c')
    print(e)
```

Consultas básicas

Python

```
query = "SELECT * FROM Peliculas WHERE director = 'Eastwood' "  
cur = conn.cursor()  
try:  
    cur.execute(query)  
except psycopg2.Error as e:  
    print(e.pgcode)
```

Cursores

Sabemos hacer consultas, ¿pero cómo le entregamos los resultados al lenguaje de programación?

Cursores nos entregan los resultados de una fila a la vez

¿Por qué no nos conviene entregar todos los resultados de una sola vez?

Cursores

Python

```
import psycopg2

try:
    conn = psycopg2.connect(database="dbname",
                            user="dbuser", host="localhost", password="dbpass")
    cur = conn.cursor()
    cur.execute("SELECT * FROM R")
    row = cur.fetchone()
    while row:
        print(row)
        row = cur.fetchone()
except:
    print("Hubo algún problema")
```

Cursores

Lógica de cursores:

- 1) Ejecutar la consulta y posicionarse **antes** de la primera tupla en la respuesta

```
import psycopg2

try:
    conn = psycopg2.connect(database="dbname",
                            user="dbuser", host="localhost", password="dbpass")
    cur = conn.cursor()
    cur.execute("SELECT * FROM R")
    row = cur.fetchone()
    while row:
        print(row)
        row = cur.fetchone()
except:
    print("Hubo algún problema")
```


Cursors

Lógica de cursores:

- 1) Ejecutar la consulta y posicionarse **antes** de la primera tupla en la respuesta
- 2) Conseguir la primera tupla

```
import psycopg2

try:
    conn = psycopg2.connect(database="dbname",
                            user="dbuser", host="localhost", password="dbpass")
    cur = conn.cursor()
    cur.execute("SELECT * FROM R")
    row = cur.fetchone()
    while row:
        print(row)
        row = cur.fetchone()
except:
    print("Hubo algún problema")
```

Cursores

Lógica de cursores:

- 1) Ejecutar la consulta y posicionarse **antes** de la primera tupla en la respuesta
- 2) Conseguir la primera tupla
- 3) Seguir si la tupla existe

```
import psycopg2

try:
    conn = psycopg2.connect(database=dbname,
                            user="dbuser", host="localhost", password="dbpass")
    cur = conn.cursor()
    cur.execute("SELECT * FROM R")
    row = cur.fetchone()
    while row:
        print(row)
        row = cur.fetchone()
except:
    print("Hubo algún problema")
```

Cursors

Lógica de cursores:

- 1) Ejecutar la consulta y posicionarse **antes** de la primera tupla en la respuesta
- 2) Conseguir la primera tupla
- 3) Seguir si la tupla existe
- 4) Conseguir la siguiente tupla

```
import psycopg2

try:
    conn = psycopg2.connect(database="db",
                            user="dbuser", host="localhost", password="dbpass")
    cur = conn.cursor()
    cur.execute("SELECT * FROM TABLE")
    row = cur.fetchone()
    while row:
        print(row)
        row = cur.fetchone()
except:
    print("Hubo algún problema")
```

Cursores

Python - Fetchall

```
try:
    conn = psycopg2.connect(database="dbname",
                            user="dbuser", host="localhost",
                            password="dbpass")
    cur = conn.cursor()
    cur.execute("SELECT * FROM R")
    rows = cur.fetchall()
    for row in rows:
        print(row)
except:
    print("Hubo algún problema")
```

Cursores

Python - Fetchall

```
try:
    conn = psycopg2.connect(database="dbname",
                             user="dbuser", host="localhost",
                             password="dbpass")
    cur = conn.cursor()
    cur.execute("SELECT * FROM R")
    rows = cur.fetchall()
    for row in rows:
        print(row)
except:
    print("Hubo algún problema")
```

Caso de uso más relevante

- Tengo un servicio Web
- Usuarios ingresan datos
- Basado en los datos que ingresan realizo una consulta a mi base de datos, y retorno algo

SQL Parametrizado

- Se trata de preparar una consulta con variables
- Cuando conozco el valor de las variables se instancia la nueva consulta concreta
- A su debido tiempo la ejecuto

SQL Parametrizado

Python



Si van a conseguir parámetros para su consulta desde un form web, hay que hacerlo así:

```
query = "SELECT * FROM Students \
        WHERE edad =%(edad)d  AND nombre = %(nombre)s"

cursor.execute(query, {"nombre": "Rivera", "edad": 11 })
```


SQL Parametrizado

Python

 <https://www.psycopg.org/docs/usage.html> 

Psycopg can automatically convert Python objects to and from SQL literals: using this feature your code will be more robust and reliable. We must stress this point:

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

The correct way to pass variables in a SQL command is using the second argument of the `execute()` method:

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

Pero profesor,
¿qué tiene de malo concatenar
strings para generar consultas?

Por qué no concatenar

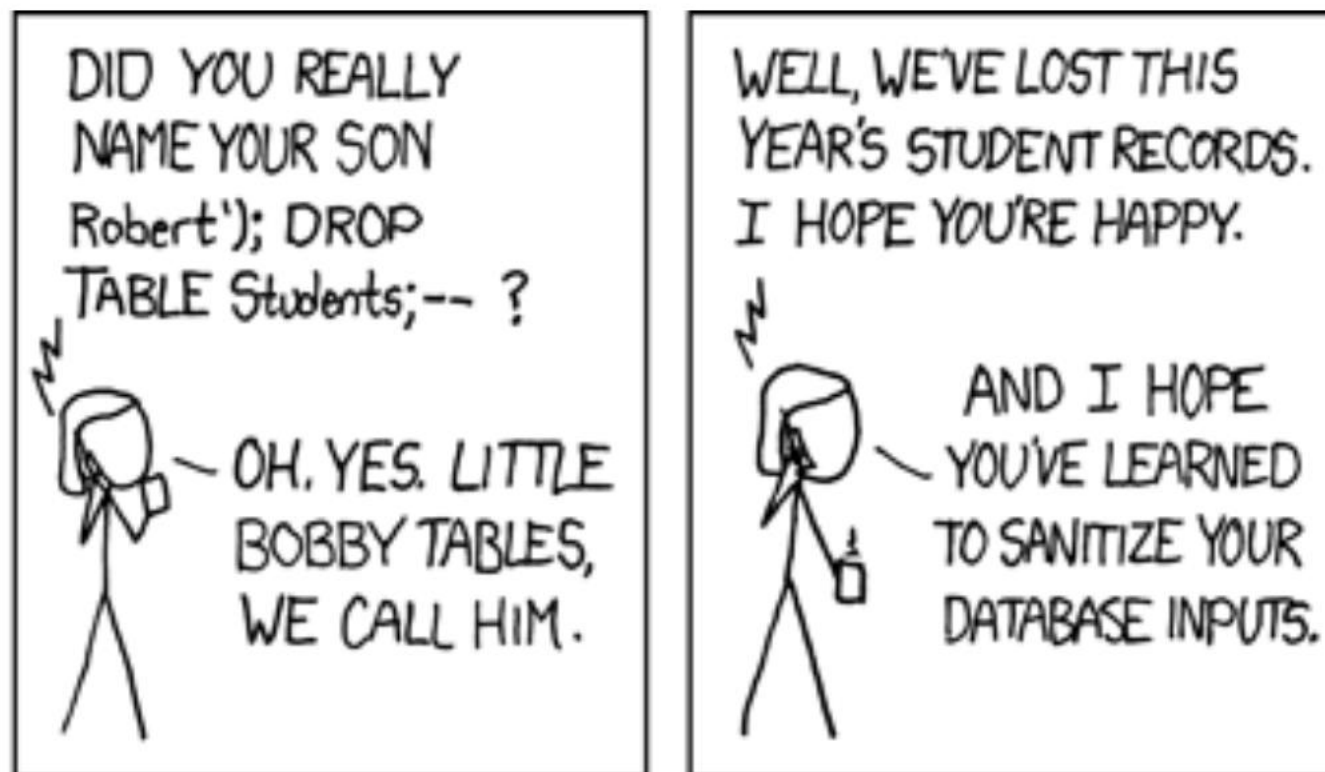
SQL inyección (injection en Inglés)

```
query = "SELECT * FROM Students WHERE name = " + nombre  
  
cursor.execute(query)
```

Por qué no concatenar

<http://xkcd.com/327/>

```
query = "SELECT * FROM Students WHERE name = " + nombre
cursor.execute(query)
```



Por qué no concatenar

SQL injection

Idea: poder inyectar un comando SQL no deseado

Veamos esto en un ejemplo

SQL en Java

Conexión a la Base de Datos

Java

Primero debemos importar la librería:

```
import java.sql.*;
```

Conexión a la Base de Datos

Java

Cargar el driver (en este caso el de PostgreSQL):

```
Class.forName("org.postgresql.Driver");
```


Conexión a la Base de Datos

Java

Objeto para conectar:

```
Connection db = DriverManager.getConnection(url, username, password);
```

Donde la URL es:

```
jdbc:postgresql://host:puerto/nombre_basededatos
```

Inserción

Java

```
String query = "INSERT INTO Peliculas VALUES (" +  
                titulo + ", " + año + ", " + director + ")";
```

```
try {  
    stmt = con.createStatement();  
    stmt.executeQuery(query);  
} catch (SQLException e ) {  
    error = e.getSQLState();  
    System.out.println(error);  
}
```

SQLSTATE y SQLException

Java

```
catch (SQLException e ) {  
    error = e.getSQLState();  
    System.out.println(error);  
}
```

SQLSTATE y SQLException

Java

SQLSTATE es una variable especial con la que SQL se comunica al programa

Convenciones:

- SQLSTATE = 00000 - No hay error
- SQLSTATE = 02000 - No encontré tupla

Consideraciones

- Los tipos de datos de los lenguajes pueden no ser soportados por SQL o vice versa
- SQL funciona en base a tuplas, los lenguajes de programación en general no
- Para esto, de nuevo ocuparemos **cursores**

Cursores

Java

```
String query = "SELECT * FROM Café";  
try {  
    stmt = con.createStatement();  
    ResultSet rs = stmt.executeQuery(query);  
    while (rs.next()) {  
        String nombreCafe = rs.getString("nombre");  
        int proveedor = rs.getInt("id_proveedor");  
        float precio = rs.getFloat("precio");  
        System.out.println(nombre + "\t" + proveedor + "\t" + precio);  
    }  
}
```

SQL Parametrizado

- Se trata de preparar una consulta con variables
- Cuando conozco el valor de las variables se instancia la nueva consulta concreta
- A su debido tiempo la ejecuto

SQL Parametrizado

Preparación

```
Statement stmt;
```

```
try {
```

```
    stmt = db.createStatement( );
```

```
}
```

```
catch (SQLException e) {
```

```
}
```


SQL Parametrizado

En nuestro sistema, el usuario ingresa la película que desea consultar y se almacena en la variable `pel`

```
String sql;  
sql = "SELECT cine, hora  
      FROM Programacion  
      WHERE pelicula LIKE '%" + pel + "%'";  
ResultSet rs = stmt.executeQuery(sql);
```

Luego navegamos por el `ResultSet` como si fuese un iterador

SQL Parametrizado

En nuestro sistema, el usuario ingresa la película que desea eliminar y se almacena en la variable `pel`

```
String sql;  
sql = "DELETE FROM Programacion  
      WHERE pelicula LIKE '%" + pel + "%'";  
int tuplasmodificadas = stmt.executeUpdate(sql);
```

Si mi consulta inserta, elimina o actualiza, `executeUpdate()` retorna el número de tuplas afectadas

SQL Parametrizado

¿Qué pasa si modifico varias películas?

```
String sql;
```

```
sql = "SELECT cine, hora
```

```
    FROM Programacion
```

```
    WHERE pelicula LIKE '%' + pel + '%';
```

```
ResultSet rs = stmt.executeQuery(sql);
```

Peor solución: ir modificando el string

SQL Parametrizado

```
String sql;  
sql = "SELECT DISTINCT titulo  
      FROM Peliculas  
      WHERE director LIKE ?";  
stmt = db.prepareStatement(sql);  
stmt.setString(1, "%Tarantino%")  
stmt.executeQuery();
```

SQL Parametrizado

```
stmt.setString(1, "%Tarantino%")  
stmt.executeQuery();
```

- Voy actualizando los directores con `setString()`
- Existen “`setXYZ()`” para los distintos tipos de datos

SQL Parametrizado

Agregando más variables

```
String sql = "SELECT rut FROM Empleados  
            WHERE apellido = ? AND sueldo =?";
```

```
stmt = db.prepareStatement(sql);
```

```
apellido = leer("Ingrese el apellido");
```

```
stmt.setString(1, apellido);
```

```
sueldo = leer("Ingrese el sueldo");
```

```
stmt.setString(2, sueldo);
```

```
ResultSet rs = stmt.executeQuery();
```

```
while (rs.next()){
```

```
    System.out.println("rut: " + rs.getString(1));
```

```
}
```

SQL en *frameworks* web

ORM

Los *frameworks* web tienen librerías para abstraerse de la base de datos

Un ORM (Object-Relational Mapping) es una técnica para tratar a los datos de un sistema como objetos de un lenguaje de programación

ORM

Ejemplo - Modelos

```
from django.db import models
```

```
class Musician(models.Model):
```

```
    first_name = models.CharField(max_length=50)
```

```
    last_name = models.CharField(max_length=50)
```

```
    instrument = models.CharField(max_length=100)
```

```
class Album(models.Model):
```

```
    artist = models.ForeignKey(Musician, on_delete=models.CASCADE)
```

```
    name = models.CharField(max_length=100)
```

```
    release_date = models.DateField()
```

```
    num_stars = models.IntegerField()
```

ORM

Ejemplo - Consultas

Obtener todos los músicos:

```
>>> Musician.objects.all()
```

Obtener todos los músicos con nombre 'James':

```
>>> Musician.objects.filter(first_name='James')
```

Obtener todos los álbumes del artista con id 1:

```
>>> Musician.objects.get(id=1).album_set.all()
```

ORM

Un ORM permite abstraerse de un sistema de bases de datos en particular

No es tan flexible como utilizar SQL, y no depende del desarrollador cómo se traducen las consultas

Nosotros instalamos la base de datos, pero el ORM se encarga de utilizarla