PECL3

BBDD 2022/2023

Adrián Rodríguez Hurtado – 09064004A

Víctor Sanavia Valdeolivas – 03202543T

ÍNDICE

Contenido

Librerías necesarias

Todas las librerías utilizadas son propias de Python a excepción de una, "*dotenv*", la cual es necesario descargarse para que el programa en Python ***index.py***, pueda leer las contraseñas y datos de conexión de los distintos usuarios. Para ello, hay que escribir el siguiente comando en la terminal "*pip install python-dotenv*". Decidimos implementar esta librería para que las contraseñas de los distintos usuarios no estuviesen visibles y así lograr cierta privacidad y limpieza en el código del programa.

Creación de los disparadores que se necesiten para completar la lógica de negocio

Triggers de auditoría

En primer lugar, para poder almacenar las auditorias, tenemos que crear una **tabla de auditoría**, la cual contiene información relacionada al evento que ha tenido lugar en alguna de las tablas

```
CREATE TABLE peliculas.auditoria(

    evento text,
    tabla name,
    usuario text,
    fecha timestamp
);
```

Una vez creada la tabla de auditoría, creamos **la función que ejecutarán los disparadores de auditoría de cada tabla**.

```
CREATE OR REPLACE FUNCTION peliculas.fn_auditoria() RETURNS TRIGGER AS $fn_auditoria$
BEGIN
    IF current_user = 'critico' THEN
        PERFORM peliculas.da_permiso_critico();
    END IF;
    IF TG_OP = 'INSERT' THEN
        INSERT INTO peliculas.auditoria(evento, tabla, usuario, fecha) VALUES ('INSERT',
TG_TABLE_NAME, current_user, current_timestamp);
    ELSIF TG_OP = 'UPDATE' THEN
        INSERT INTO peliculas.auditoria(evento, tabla, usuario, fecha) VALUES ('UPDATE',
TG_TABLE_NAME, current_user, current_timestamp);
    ELSIF TG_OP = 'DELETE' THEN
        INSERT INTO peliculas.auditoria(evento, tabla, usuario, fecha) VALUES ('DELETE',
TG_TABLE_NAME, current_user, current_timestamp);
    END IF;
    IF current_user = 'critico' THEN
        PERFORM peliculas.quita_permiso_critico();
    END IF;
    RETURN NEW;
END; $fn_auditoria$ LANGUAGE plpgsql;
```

Este trigger recoge el tipo de la consulta que hemos hecho, si es un **INSERT**, introduce en la tabla de auditoría, el tipo de operación que se ha hecho, el nombre de la tabla que hizo disparar al trigger (con **TG_TABLE_NAME**), el usuario que lo ejecuto (con **current_user**) y la hora en la cual tuvo lugar dicha operación (con **current_timestamp**). Con las demás operaciones, el trigger actúa de la misma forma, variando exclusivamente el tipo de operación que añade a la tabla (**INSERT**, **UPDATE** o **DELETE**)

Como podemos ver en la imagen anterior, al principio y al final hay dos **funciones auxiliares**, las cuales hemos utilizado para solucionar el problema de que cuando el crítico insertaba una crítica, a la hora de introducir el evento en la tabla de auditoría, aparecía el usuario *postgres* en lugar de crítico. Lo que hacen estas funciones auxiliares es dar el permiso al rol de critico de poder insertar en auditoría, y cuando acaba la operación, se le quita dicho permiso, el acceso a auditoría siempre estará controlado por el disparador esto solo se ejecutará cuando el usuario sea un crítico, otro usuario (que no es crítico) ejecuta la función, esta parte no se ejecutará. Estas funciones pueden dar y quitar permisos ya que tienen un ***SECURITY DEFINER***, que lo que hace es que las funciones las ejecuta el rol o usuario que creó esas funciones, que en este caso es el usuario ***postgres***, el cual tiene todos los permisos. En primer lugar, se adjunta el código de la función que da el permiso, y después la función que quita el permiso:

```
CREATE OR REPLACE FUNCTION peliculas.da_permiso_critico()
RETURNS void
SECURITY DEFINER
AS
$BODY$
BEGIN
GRANT INSERT ON peliculas.auditoria TO critico;
END;
$BODY$
LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION peliculas.quita_permiso_critico()
RETURNS void
SECURITY DEFINER
AS
$BODY$
BEGIN
REVOKE INSERT ON peliculas.auditoria FROM critico;
END;
$BODY$
LANGUAGE plpgsql;
```

También hacemos uso de las variables **current_timestamp** (que devuelve un timestamp con la fecha, día y hora actual) y del **current_user** (que devuelve el nombre del usuario que está activo en ese momento) además de las operaciones de los triggers o disparadores que encontramos en la **documentación oficial de PostgreSQL** como son el **TG_OP** (que devuelve el tipo de operación que ha disparado el trigger) y **TG_TABLE_NAME** (que devuelve el nombre de la tabla que invocó al trigger), cuyas documentaciones oficiales están aquí.

Una vez que hemos creado todas las funciones necesarias para que el disparador de auditoría funcione correctamente, **creamos los disparadores de auditorías de todas las tablas**:

Trigger de auditoria para la tabla *peliculas.crticas*:

```
CREATE TRIGGER tg_criticas_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.criticas
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.personal*:

```
CREATE TRIGGER tg_personal_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.personal
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.actor*:

```
CREATE TRIGGER tg_actor_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.actor
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.director*:

```
CREATE TRIGGER tg_director_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.director
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.actua*:

```
CREATE TRIGGER tg_actua_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.actua
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.peliculas*:

```
CREATE TRIGGER tg_peliculas_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.peliculas
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.generos*:

```
CREATE TRIGGER tg_generos_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.generos
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.caratulas*:

```
CREATE TRIGGER tg_caratulas_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.caratulas
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Trigger de auditoria para la tabla *peliculas.pag_web*:

```
CREATE TRIGGER tg_paginaweb_audit

    AFTER INSERT OR UPDATE OR DELETE
    ON peliculas.pag_web
    FOR EACH ROW
    EXECUTE PROCEDURE peliculas.fn_auditoria();
```

Este disparador se encargará de almacenar dentro de la tabla ***películas.auditoria*** los eventos que van teniendo lugar en las distintas tablas de la base de datos, los cuales tienen esta información / atributos:

- La tabla en la que ha ocurrido el evento.
- El tipo de evento u operación.
- El usuario que lo ha llevado a cabo.
- La fecha y la hora en la cual tuvo lugar la inserción.

Trigger de insertar críticas

Este trigger está relacionado con las críticas. Debe comprobar si, al insertar una crítica, la página web se encuentra o no en la tabla. Si ésta no se encuentra, se añadirá a la tabla.

En primer lugar, **creamos la función que ejecutará el trigger de insertar críticas**. Cuyo funcionamiento se explica a continuación

Primero, vamos a comprobar que el usuario ha introducido todos los atributos necesarios, es decir, los que en la tabla de ***peliculas.criticas*** son ***NOTNULL***.

```
CREATE OR REPLACE FUNCTION peliculas.fn_inserta_critica() RETURNS TRIGGER
SECURITY DEFINER AS $fn_inserta_critica$

BEGIN

    IF NEW.critico ISNULL THEN
        RAISE EXCEPTION 'El nombre del critico no puede ser nulo, debes
imprimir uno';
    END IF;

    IF NEW.anno_peliculas ISNULL THEN
        RAISE EXCEPTION 'El anno de la pelicula no puede ser nulo, debes
aportar uno';
    END IF;

    IF NEW.titulo_peliculas ISNULL THEN
        RAISE EXCEPTION 'El titulo de la pelicula no puede tener valor
nulo, debes aportar uno';
    END IF;

    IF NEW.nombre_pag_web ISNULL THEN
        RAISE EXCEPTION 'El nombre o url de la pagina web no puede ser
nulo, debes aportar uno';
    END IF;
```

Si el usuario no ha introducido todos los datos, el propio trigger producirá un error avisando al usuario de que falta ese atributo en concreto.

Si el usuario ha introducido todos los datos, se procederá a comprobar si la página web se encuentra o no en la tabla:

```
IF NEW.nombre_pag_web NOT IN (SELECT nombre FROM peliculas.pag_web) THEN
        INSERT INTO peliculas.pag_web(nombre) VALUES
(NEW.nombre_pag_web);

    END IF;

    RETURN NEW;

END;

$fn_inserta_critica$ LANGUAGE plpgsql;
```

Una vez creada la función a ejecutar, **creamos el trigger *tg_inserta_critica***, que hará uso de la función anteriormente descrita:

```
CREATE TRIGGER tg_inserta_critica

    BEFORE INSERT
    ON peliculas.criticas
    FOR EACH ROW
    EXECUTE FUNCTION peliculas.fn_inserta_critica();
```

Trigger de media

Para este tercer y último trigger se creará una nueva tabla para almacenar las medias de las películas. La tabla deberá de ser actualizada cada vez que se introduzca una nueva crítica.

La tabla donde se almacenarán las medias será ***películas.nota_media_peliculas***:

```
CREATE TABLE peliculas.nota_media_peliculas(

    titulo_peliculas text,
    anno_peliculas smallint,
    media integer
);
```

Para un correcto funcionamiento de este trigger, creamos una vista que contiene todas las medias de las películas actualizadas cada vez que la ejecutamos (este código es reutilizado de nuestra tercera parte de la práctica 2):

```
CREATE VIEW peliculas.media_peliculas as
(SELECT titulo_peliculas, anno_peliculas, avg(puntuacion) as
puntuacion_media
FROM
peliculas.criticas
GROUP BY
titulo_peliculas, anno_peliculas
ORDER BY
avg(puntuacion));
```

Este trigger se activará después de una inserción dentro de la tabla ***películas.criticas***, y actualizará la media de la película de la que se acaba de insertar la crítica, ya que actualizar las demás no tiene sentido, ya que seguirán teniendo la misma media.

```
CREATE OR REPLACE FUNCTION peliculas.fn_actualiza_media_peliculas()
RETURNS TRIGGER SECURITY DEFINER AS $fn_actualiza_media_peliculas$

BEGIN

    UPDATE peliculas.nota_media_peliculas SET (media) =
    (SELECT puntuacion_media
    FROM peliculas.media_peliculas
    WHERE (peliculas.media_peliculas.titulo_peliculas =
NEW.titulo_peliculas) and (peliculas.media_peliculas.anno_peliculas =
NEW.anno_peliculas));

    RETURN NEW;

END;

$fn_actualiza_media_peliculas$ LANGUAGE plpgsql;
```

Se crea el trigger ***tg_actualiza_medias_peliculas*** que se disparará después de una inserción dentro de la tabla ***películas.criticas*** para **actualizar la media de las películas**:

```
CREATE TRIGGER tg_actualiza_medias_peliculas
    AFTER INSERT
    ON peliculas.criticas
    FOR EACH ROW
    EXECUTE FUNCTION peliculas.fn_actualiza_media_peliculas();
```

Utilizamos esta consulta auxiliar para poblar la tabla de las medias basándonos en nuestra vista anteriormente creada:

```
INSERT INTO peliculas.nota_media_peliculas(titulo_peliculas,
anno_peliculas, media)
SELECT titulo_peliculas, anno_peliculas, puntuacion_media FROM
peliculas.media_peliculas;
```

Creación de usuarios

El programa constará de 4 tipos distintos de usuarios:

- **Administrador** → Todos los permisos.
- **Gestor** → Inserción, modificación, borrado y consulta.
- **Crítico** → Consulta de todas las tablas a excepción de la tabla "auditoria" e inserción en la tabla "críticas".
- **Cliente** → Consulta de todas las tablas a excepción de la tabla "auditoria".

Primero, crearemos los usuarios con sus respectivas contraseñas.

```
CREATE USER admin PASSWORD 'admin';
CREATE USER gestor PASSWORD 'gestor';
CREATE USER critico PASSWORD 'critico';
CREATE USER cliente PASSWORD 'cliente';
```

Una vez creados, les asignamos los permisos en función del rol.

```
REVOKE ALL PRIVILEGES ON ALL TABLES IN SCHEMA peliculas FROM gestor,
critico, cliente;
GRANT USAGE ON SCHEMA peliculas TO admin, gestor, critico, cliente;
GRANT SELECT ON ALL TABLES IN SCHEMA peliculas TO critico;
GRANT INSERT ON peliculas.criticas TO critico;
GRANT SELECT ON ALL TABLES IN SCHEMA peliculas TO cliente;

CREATE TABLE peliculas.auditoria(

    evento text,
    tabla name,
    usuario text,
    fecha timestamp
);

GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA peliculas TO admin WITH
GRANT OPTION;
GRANT INSERT, UPDATE, DELETE, SELECT ON ALL TABLES IN SCHEMA peliculas TO
gestor;
```

Como se muestra en la imagen, en primer lugar, hacemos uso de la función ***REVOKE*** para eliminar todos los permisos de todos los roles que puedan tener predeterminados. Acto seguido, damos un ***GRANT USAGE*** a todos los roles para que puedan tener acceso al esquema ***películas***. Y después damos permisos de ***SELECT*** e ***INSERT*** en todas las tablas del esquema ***películas*** al usuario *crítico* y el permiso ***SELECT*** en todas las tablas del esquema ***películas*** a ***cliente***. Nótese que hemos creado la **tabla de auditoría** justo después de dar estos permisos, porque hemos considerado que la información de auditoría solo la podrían ver los usuarios ***admin*** y ***gestor*** ya que es información que no le interesa saber a un ***cliente*** o a un ***crítico***, y así, dicha tabla es invisible tanto para ***crítico*** como para ***cliente***.

Después de crear la tabla de auditoría damos todos los permisos en el esquema ***películas*** a ***admin*** con la opción de dar permisos. Y, por último, damos los permisos de *INSERT*, ***UPDATE***, ***DELETE*** y ***SELECT*** en todas las tablas del esquema *películas* a ***gestor***.

Aquí adjuntamos un link para ver la documentación oficial de la sentencia ***GRANT*** y de los permisos que ofrece.

Conexión con programas externos y seguridad

Se ha creado un programa en **Python** para que cualquier usuario creado previamente pueda conectarse a la base de datos, cuyo funcionamiento será explicado a continuación:

En primer lugar, se llevará a cabo la elección del usuario, se tendrá que elegir uno de los 4 usuarios creados previamente, e introducir la contraseña asignada a cada uno. Si la contraseña es correcta, el programa mostrará el texto 'Contrasena correcta!'. Si ésta fuese incorrecta, no se podría asignar el usuario correspondiente. Si se introduce una opción no valida al elegir usuario, el programa seguirá pidiendo esa opción, hasta que la opción introducida sea adecuada. Si todo ha ido bien, devolverá la información del usuario elegido para que éste pueda conectarse a la base de datos.

```
def user_choice() -> str:

    global user
    correct_choice = False

    while (not correct_choice):
        try:
            print('\t\t------------------=[ELECCION DE USUARIO]=----------
-------\n\n\t1. Admin\n\t2. Gestor\n\t3. Critico\n\t4. Cliente\n\n\t\t---
--------------=[ELECCION DE USUARIO]=-----------------\n\n')
            choice = int(input('Elige el usuario con el que deseas
conectarte a la base de datos (numero): '))
            if (choice.__eq__(1)):
                password = str(input('Introduce la contrasena del usuario
admin: '))
                if (password.__eq__(os.getenv('admin_password'))):
                    user = 'admin'
                    correct_choice = True
                else:
                    print('Contrasena incorrecta para el usuario admin')
                    time.sleep(2)
            elif (choice.__eq__(2)):
                password = str(input('Introduce la contrasena del usuario
gestor: '))
                if (password.__eq__(os.getenv('gestor_password'))):
                    user = 'gestor'
                    correct_choice = True
                else:
                    print('Contrasena incorrecta para el usuario gestor')
                    time.sleep(2)
            elif (choice.__eq__(3)):
```

```
                password = str(input('Introduce la contrasena del usuario
critico: '))
                if (password.__eq__(os.getenv('critico_password'))):
                    user = 'critico'
                    correct_choice = True
                else:
                    print('Contrasena incorrecta para el usuario
critico')
                    time.sleep(2)
            elif (choice.__eq__(4)):
                password = str(input('Introduce la contrasena del usuario
cliente: '))
                if (password.__eq__(os.getenv('cliente_password'))):
                    user = 'cliente'
                    correct_choice = True
                else:
                    print('Contrasena incorrecta para el usuario
cliente')
                    time.sleep(2)
            else:
                print('Introduce una opcion correcta (numero del 1 al
4)')
                time.sleep(2)
        except ValueError:
            print('Introduce una opcion valida (numero del 1 al 4)')

        os.system('cls')

    print('Contrasena correcta!, estableciendo conexion...')
    time.sleep(2)
    user_info = os.getenv(user + '_user')

    return user_info
```

Se crearán dos funciones: una para **establecer conexión** con la base de datos y otra para **cerrar la conexión** con la misma, respectivamente. La información se recoge en el **archivo *.env*** por medio de la librería ***dotenv***.

```
def connection_establishment(user_info: str):

    global connection
    global cursor
    connection = psycopg2.connect(user_info)
    cursor = connection.cursor()
    print('\nConexion establecida correctamente!\n')
    time.sleep(1)
```

```
def connection_termination():

    global connection
    global cursor
    cursor.close()
    connection.close()
    print('Conexion cerrada.')
```

Una vez conectados a la base de datos, tendremos que elegir el tipo de consulta entre select e *insert*, *delete* o *update*. Se desplegará un "menú" por pantalla ofreciendo la elección de los dos tipos de consultas mencionados anteriormente, introduciendo su numero correspondiente (1 o 2). Si se introduce un número incorrecto se le avisa al usuario, y si se introduce un string directamente no funcionará y no se podrá avanzar hasta introducir una opción correcta controlado por el booleano ***correct_result***, que se pondrá a True cuando se haya elegido una opción válida. La función retornara 1 o 2, según la opción elegida.

```
def query_choice() -> int:

    possible_results = [1, 2]
    print('\t\t-----------------=[ELECCION DE TIPO DE CONSULTA]=---------
--------\n\n\t1. Consulta de tipo select\n\t2. Consulta de tipo insert,
delete o update\n\n')
    correct_result = False
    while (not correct_result):
        try:
            result = int(input('Elige la opción deseada (introduciendo el
numero de la opcion): '))
            if (result in possible_results):
                correct_result = True
        except ValueError:
            print(f'Introduce un valor valido ({possible_results})')

    return result
```

En el caso de que en la anterior función hubiésemos elegido la **opción 1**, nos iríamos a hacer una consulta de tipo select (o lo que es lo mismo, ejecutaremos la función ***select_query()***), que ejecutará el cursor con ***cursor.execute()*** la consulta que recibe la función como parámetro, y en el caso de que haya un error se avisará cuál ha sido el error y se proporciona información adicional sobre él, imprimiendo el propio error en el ***try – except***, que se encarga de controlar todos los errores posibles.

En el caso de que la consulta esté bien formulada, haremos un ***fetchall()*** dentro de la variable ***rows*** que almacena todas las filas de dicha consulta, y para imprimirlas por pantalla haremos uso de la sentencia ***for***.

```
def select_query(sql_command: str):

    global connection
    global cursor

    try:
        cursor.execute(sql_command)
        rows = cursor.fetchall()
        print('\n\nConsulta realizada correctamente, mostrando
resultados...\n\n')
        time.sleep(1)
        for row in rows:
            print(row)

    except (errors.UndefinedTable) as undefined_table:
        print(f'\n\nLa tabla que has introducido no existe ->
{undefined_table}')
    except (errors.UndefinedColumn) as undefined_column:
        print(f'\n\nLa columna introducida no existe ->
{undefined_column}')
    except (errors.InsufficientPrivilege) as permission_error:
        print(f'\n\nEl usuario elegido ({user}) no tiene permisos para
realizar esta accion -> {permission_error}')
    except (errors.SyntaxError) as syntax_error:
        print(f'\n\nError en la sintaxis de la consulta SQL ->
{syntax_error}')
    except (errors.ProgrammingError) as programming_error:
        print(f'\n\nError de programacion, ¿has hecho un insert en un
query select? -> {programming_error}')
```

En el caso de elegir la **opción 2**, ejecutaremos la en la función ***query_choice()*** para realizar una consulta de tipo ***insert*** seremos dirigidos a esta función, la cuál se encarga de ejecutar la consulta dada por el usuario. En el caso de que la consulta tenga algún error se avisará al usuario del error y se le dará información adicional sobre él. Todos los errores están controlados por el ***try – except***. En caso de que no haya ningún error, se ejecutará la consulta e inmediatamente despues haremos un ***cursor.commit()*** para que los cambios realizados se vean reflejados en la base de datos.

```
def insert_query(sql_command: str):

    global connection
    global cursor

    try:
        cursor.execute(sql_command)
        connection.commit()
        print('\nConsulta realizada correctamente!')
    except (errors.UndefinedTable) as undefined_table:
        print(f'\n\nLa tabla que has introducido no existe ->
{undefined_table}')
    except (errors.UndefinedColumn) as undefined_column:
        print(f'\n\nLa columna introducida no existe ->
{undefined_column}')
    except (errors.InsufficientPrivilege) as permission_error:
        print(f'\n\nEl usuario elegido ({user}) no tiene permisos para
realizar esta accion -> {permission_error}')
    except (errors.UniqueViolation) as unique_violation:
        print(f'\n\nLa consulta viola una restriccion de unicidad (llave
ya existente) -> {unique_violation}')
    except (errors.ForeignKeyViolation) as foreign_key_violation:
        print(f'\n\nViolacion de clave foranea (no esta presente en la
tabla) -> {foreign_key_violation}')
    except (errors.SyntaxError) as syntax_error:
        print(f'\n\nError en la sintaxis de la consulta SQL ->
{syntax_error}')
```

Una vez que hacemos una consulta, preguntaremos si se quieren hacer más consultas:

- En el caso de que introduzca si, volverá al menú de elegir usuario para volver a hacer todo el proceso anteriormente mencionado
- Por el contrario, de introducir no, el programa acabará ejecución e imprimir por pantalla que el programa ha finalizado
- En el caso de que se introduzca distinta de si o no, no surtirá efecto hasta que se introduzca si o no, ya que el error está controlado mediante un ***try – except***.

```
def more_querys() -> bool:
    result: bool
    choice_succeded = False
    choice = ''
    while (not choice_succeded):
        try:
            choice = str(input('\n\n¿Desea hacer mas consultas? (si / 
no): '))
            if (choice.__eq__('si')):
                result = True
                choice_succeded = True
            elif (choice.__eq__('no')):
                result = False
                choice_succeded = True
        except ValueError:
            print('\n\nIntroduce una opcion valida (si / no)')


    return result
```

Por último, una función main que llama al resto de funciones creadas. Actúa como el hilo principal del programa, es el encargado de realizar todas las operaciones descritas con anterioridad sin errores inesperados. A destacar, es que en el caso de que se interrumpa el programa mediante teclado (usando CTRL + C) aparecerá esta alerta en pantalla. Al igual que si falla la conexión a la base de datos.

```
def main():

    try:

        os.system('cls')
        running = True

        while (running):
            # Primero se deberá elegir el usuario, y acto seguido abrir
la conexion con la base de datos
            try:
                connection_establishment(user_choice())
            except (psycopg2.OperationalError):
                print('Error en la conexion a la base de datos')
                running = False
                print('\n\n\t\t-----------------=[PROGRAMA FINALIZADO POR
FALLO DE CONEXION]=-----------------\n\n')
                break
            # Una vez metidos en la base de datos, elegimos que consulta
queremos hacer
            query_type = query_choice()
            if (query_type.__eq__(1)):
                sql_command = str(input('Introduce la consulta a realizar
(tipo select): '))
                select_query(sql_command)
            elif (query_type.__eq__(2)):
                sql_command = str(input('Introduce la consulta a realizar
(tipo insert): '))
                insert_query(sql_command)
            # Preguntaremos al usuario si quiere hacer mas consultas
            if more_querys().__eq__(False):
                print('Cerrando...')
                connection_termination()
                time.sleep(0.5)
                print('\n\t\t-----------------=[PROGRAMA FINALIZADO]=----
-------------\n\n')
                running = False
    except KeyboardInterrupt:
        print('\n\n\t\t-----------------=[PROGRAMA FINALIZADO POR
TECLADO]=-----------------\n\n')
```

Por último, se llama a la función main para que el programa funcione.

```
main()
```