Acceso a bases de datos con Python

Miguel Rodríguez Penabad

Laboratorio de Bases de Datos

Universidade da Coruña

5 de xaneiro de 2022

Índice

Introdución

Acceso a bases de datos desde Python

Existen varios métodos para escribir programas en Python que accedan a SXBD relacionais:

- **Python DB-API**: é o método máis manual e cercano á base de datos, e tamén o que ofrece un maior control.
  - Ofrece unha API (maiormente) unificada e independente do SXBD.
  - É imprescindible saber SQL para utilizala.
- **Integrado con ferramentas** ou librerías como `pandas`, especialmente usado en Ciencia de Datos.
- **Mapeabores Obxecto-Relacionais** como `SQLAlchemy`. Sería comparable a Hibernate en Java.

Neste documento describiremos o uso de Python DB-API.

Nota sobre versións de Python

Python 2.x (a última versión foi 2.7.19) está considerado obsoleto desde o 1 de xaneiro de 2020 (`https://www.python.org/doc/sunset-python-2/`).
Neste documento úsase a versión 3.7 de Python.
Nesta versión facilítase a impresión de valores con `print` usando os *f strings*:

```
>>> id=3
>>> valor='Lápiz'
>>> # Válido en python 2.x e 3.x
... print("Id: %d, Valor: %s" % (id, valor))
Id: 3, Valor: Lápiz

>>> # Válido en python 2.x e 3.x
... print("Id: {}, Valor: {}".format(id, valor))
Id: 3, Valor: Lápiz

>>> # Válido en python 3.6 ou superior: "f strings"
... print(f"Id: {id}, Valor: {valor}")
Id: 3, Valor: Lápiz
```

Introdución

Miniexemplo

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# 1. Importamos o adaptador específico para o SXBD (PostgreSQL)
import psycopg2

# 2. Obtemos unha conexión coa base de datos
con=psycopg2.connect(host='localhost',
                     user='testuser',
                     password='testpass',
                     dbname='testdb')

# 3. Creamos un cursor
cur = con.cursor()

# 4. Executamos consultas SQL utilizando o cursor
#    Por exemplo, un select, mostrando logo os datos
cur.execute('select nome, datanac from persoa where id=1')
row = cur.fetchone()
print(f"Nome persoa: {row[0]}, data de nacemento: {row[1]}")

# 5. Liberamos recursos
cur.close()
con.close()
```

Introdución

Estrutura básica para Python DB-API

Bloques fundamentais

En todo programa que utilice Python para acceder a unha base de datos veremos estes elementos

- Importación do adaptador da base de datos (a veces chamado driver):
  `psycopg2`, `pymysql`, `mariadb`, `cx_Oracle`, `sqlite3`, ...
- Obtención dunha conexión coa base de datos (obxecto de tipo `connection`).
- Obtención dun cursor a partir da conexión (obxecto de tipo `cursor`).
- Uso do cursor para lanzar consultas SQL contra a base de datos.
  As sentencias SQL poden ser DDL, DML ou DCL.
- Desconexión e liberación de recursos.

Carencias no miniexemplo

Para ter un código mínimo e funcional, no miniexemplo non se incluíron algúns aspectos que serán fundamentais.

- Control de erros (con `try .. except .. finally`).
- Control transaccional (`commit`/`rollback`).

Tampouco se inclúen aspectos máis avanzados, como

- Modos de aillamento das transaccións (`isolation levels`)

Índice

Conexión á base de datos

Adaptadores de bases de datos

Adaptadores

- Antes de conectarse a un SXBD hai que importar o adaptador de BD (a veces chamado *conector* ou *driver*) específico.
- Cada adaptador é específico para SXBD, pero logo o funcionamento é moi similar.
- Poden existir varias implementacións para un mesmo SXBD.
- Exemplos:
  - PostgreSQL: `psycopg2` (e `psycopg2.extras` para funcionalidade extendida) é o máis común (implementado en C), pero existen outros, como `pg8000` ou `py-postgresql` (Python puro).
  - MySQL/MariaDB: `mysql.connector`, `pymysql`, `mariadb`.
  - Oracle: `cx_Oracle`.
  - SQLite3: `sqlite3`.
- As implementacións en C son normalmente máis eficientes, mentras que as de Python puro son máis portables entre diferentes plataformas.

Instalación

- Unha instalación normal de Python non inclúe normalmente todos os adaptadores de bases de datos.
- A instalación do módulo de Python necesario pode facerse de varias formas, dependendo da plataforma.
- Exemplos:
  - Debian: `apt install python3-psycopg2`.
  - Windows: Descargar e executar `easy_install <driver.exe>`.
  - *The Python way*, co instalador de paquetes de Python(`pip3`):
    `pip3 install psycopg2` (`pip3 search` para buscar nos repositorios oficiais).
    Alternativa: descargar o arquivo Wheel (extensión `.whl`) e usar `pip3 install <driver.whl>`.

Conexión á base de datos

PosgreSQL: psycopg2

- Método `connect` especificando o DSN (Data Source Name), como como unha lista de argumentos (clave=valor) ou como un string.
  - `host`, predeterminado `localhost`.
  - `port`, predeterminado o 5432.
  - `user`, predeterminado o usuario do SO.
  - `password`
  - `dbname` (a opción 1, con parámetros separados, tamén admite `database`). Predeterminado: o nome de usuario (`user`).
- Opcionalmente poden especificarse outras características
- Desde Python podemos obter axuda: `import psycopg2; help(psycopg2.connect)`
- Se PostgreSQL está configurado con autenticación `peer` para o usuario, non necesita a password.

```
import psycopg2
# Opción 1: Lista de argumentos clave=valor
con=psycopg2.connect(host='localhost',
                     user='testuser',
                     password='testpass',
                     dbname='testdb') #Tamén database='testdb'
# Opción 2: String
con=
 psycopg2.connect("host='localhost' user='testuser' password='testpass' dbname='testdb'")

# Opción 2b: String (con todo predeterminado)
#  Se o usuario do SO fose "usuario", con autenticación de PostgreSQL "peer",
#  sería equivalente a host='localhost', user='usuario', dbname='usuario'
con=psycopg2.connect("")
```

Conexión á base de datos

Outros xestores

MySQL/MariaDB

- Hai moitos adaptadores: `MySQLdb`, `pymysql`, `mysql.connector`, `mariadb`, ...
- Os parámetros de `connect` poden variar entre estes adaptadores.

```
import pymysql
con=pymysql.connect(host='localhost',
                    user='testuser',
                    password='testpass',
                    database='testdb')
```

SQLite3

- Non hai unha conexión a un servidor.
- Accede a un ficheiro coa base de datos.
- Se o ficheiro non existe, créase.

```
import sqlite3
con=sqlite3.connect('database_file.db')
```

Conexión á base de datos

Inciso: Xestión de excepcións en Python

- En Python existe unha xerarquía de clases excepcións.
- Aínda que a raíz desa xerarquía é `BaseException`, todas as que nos interesan derivan de `Exception`.
- A xestión faise mediante sentencias `try .. except`.
- Na parte `except` podemos poñer unha lista de excepcións (entre parénteses) ou unha soa.
- Con PostgreSQL normalmente usaremos `psycopg2.Error` ou `psycopg2.OperationalError`.
- Unha vez temos a conexión con PosgreSQL, poderemos acceder ós campos `pgcode` e `pgerror`. (Un erro de conexión non devolve ningún `pgcode` nin `pgerror`, ambos son `None`). En `https://www.psycopg.org/docs/errors.html` pode verse unha lista de erros (que corresponden co estándar SQLSTATE) exposta por `psycopg2`.

```
try:
    # Accións
except <ListaTiposExcepcion> [as <NomeExcepcion> ]:
    # Xestión de NomeExcepcion
[else:
    # Esta parte execútase se non pasa pola parte "except" ]
[finally:
    # Esta parte execútase sempre, ocurra ou non un erro ]
```

```
try:
    b=7/0
except Exception as e:
    print(e) # Imprime a mensaxe de erro
    print(type(e)) # Vemos o tipo de excepción xenerada
```

```
--
division by zero
<class 'ZeroDivisionError'>
```

Conexión á base de datos

Un exemplo máis completo

- Configuración externalizada (ficheiro `dbconfig.json`):
  `{ "host":"localhost", "user":"testuser", "password":"testpass", "dbname":"testdb" }`
- Parámetros adicionais: cursores accesibles como dicionarios (máis adiante).
- Control (moi básico) de excepcións.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
import psycopg2, psycopg2.extras, json, sys

def read_config(file):
    with open(file) as f:
        cfg=json.load(f)
    return cfg

def connect_db():
    try:
        config=read_config('dbconfig.json')
        # **config "desempaqueta" o diccionario que contén a configuración
        # para pasalo como argumentos clave=valor
        conn=psycopg2.connect(cursor_factory=psycopg2.extras.DictCursor, **config)
        return conn
    except psycopg2.OperationalError as e:
        print(f"Imposible conectar: {e}")
        sys.exit(-1)
```

Índice

Execución de sentencias SQL

Introdución

Tipos de sentencias SQL

- **DDL** (Data Definition Language): `create table...` etc.
- **DCL** (Data Control Language): `grant`, `revoke`, xestión transaccional.
  NOTA: Para a xestión transaccional veremos tamén métodos do obxecto `connection` e xestión transaccional (`set transaction...`).
- **DML** (Data Manipulation Language): `insert, delete, update, select`, etc.
  A selección de datos (`select`) requerirá métodos para obter e recorrer as filas obtidas.

Mecanismos

- Para as sentencias DML e DDL usaremos cursores.
- Para as sentencias DCL podemos usar cursores (para `grant` e `revoke`, por exemplo) ou atributos ou métodos do obxecto `connection`.
- Atributos e métodos importantes da `connection`:
  - `cursor()` para crear un cursor e executar SQL (a continuación).
  - `commit()` para confirmar unha transacción.
  - `rollback()` para anular unha transacción, desfacendo os cambios.
  - `autocommit` para establecer o modo autocommit. Se é `True` cada execución do cursor remata cun `commit()` implícito, polo que non permitiría definir transaccións explícitas (de máis dunha sentencia SQL, por exemplo).
  - `isolation_level`, `set_isolation_level()` para especificar o nivel de aillamento da transacción actual.

Execución de sentencias SQL

Cursores

- Para executar sentencias SQL hai que crear un `cursor` a partir da conexión.
- O cursor ten un método `execute` para executar a sentencia.

```
conn = connect_db()
cur = conn.cursor([<parámetros opcionais>])
...
cur.execute('sentencia SQL' [, <valores> ])
...
cur.close()
conn.close()
```

- Outra opción, que cerra automaticamente o cursor:

```
conn = connect_db()
with conn.cursor([<parámetros opcionais>]) as cur:
    cur.execute('sentencia SQL' [, <valores> ])
    ...
# Non necesitamos cur.close()
...
conn.close()
```

Execución de sentencias SQL

Cursores: Sentencias DDL/DCL

Nota: Algúns SXBDs, como Oracle, fai un commit implícito antes e despois das sentencias DDL e DCL. PostgreSQL non o fai.

```
cur=conn.cursor()
sentencia_create="""create table persoa(
                    id serial primary key,
                    nome varchar(50) not null,
                    datanac date)"""
cur.execute(sentencia_create)
conn.commit()



cur.execute("grant select on persoa to usuario")
conn.commit()
```

Execución de sentencias SQL

Cursores: Sentencias DML

Podemos executar sentencias que inclúen directamente os datos, ou construir o string que representa a sentencia a partir de variables.
É unha mala idea (relacionado co problema de SQL injection).

```
sentencia="""insert into persoa(nome, datanac)
            values('Ada Lovelace', '12-10-1815')"""
cur.execute(sentencia)
conn.commit()

nome=input("Nome: ")
data_nac=input("Data nacemento: ")
sentencia2 = f"""insert into persoa(nome, datanac)
                values('{nome}', '{data_nac}')"""
cur.execute(sentencia2)
conn.commit()
```

Posibles problemas:

1. Caracteres especiais como o apóstrofe.
   Que pasa se o nome tecleado é `O'Connor`?
2. Tipos e formatos de datos, especialmente a data.
   A data introducida `12-10-1815` é o 12 de outubro (Europa) ou o 10 de decembro (EUA)?
3. SQL injection.

Execución de sentencias SQL

Cursores: Sentencias DML

- Recoméndase o uso de **consultas con parámetros**.
  Un parámetro é un oco ou *placeholder* que tomará o valor na chamada ó método `execute`.
- Hai 2 variantes:
  - Parámetros sen nome:
    - Na sentencia represéntanse con `%s`.
    - Na chamada ó método `execute` utilízase unha tupla cos valores.
      NOTA: debe ser sempre unha tupla. Se necesitamos só un valor, usaremos a sintaxe `(valor,)`.
  - Parámetros con nome:
    - Na sentencia represéntanse con `%(nome)s`.
    - Na chamada ó método `execute` utilízase un dicionario `{'nome1':valor1, 'nome2': valor2...}`.
- Facilita unha mellor de xestión de tipos e formatos de datos, incluíndo nulos.
  - Podemos constuir datas co módulo `datetime`, que debemos importar.
  - Podemos usar a constante `None` para insertar un nulo.

```
import datetime
nome='Alan Turing'
data=datetime.datetime(1912,6,23)
cur.execute("insert into persoa(nome, datanac) values(%s, %s)", (nome, data))
conn.commit()

#Parámetros con nome
cur.execute("insert into persoa(nome, datanac) values(%(nome)s, %(data_nac)s)",
          {'nome': 'Matusalén', 'data_nac': None})
```

Execución de sentencias SQL

Consultas (`select`)

- Execútase a consulta co método `execute()` do cursor, e logo hai que recuperar e procesar as filas:
  - `fetchone()` devolve a seguinte fila, ou `None` se non existe.
  - `fetchall()` devolve unha lista con todas as filas restantes (inicialmente serán todas as filas).
- Cando a consulta devolve unha fila, usaremos `fetchone()`

```
cur.execute("select id, nome, datanac from persoa where id=%s", (1,))
row=cur.fetchone()
print(row)
(1, 'Ada Lovelace', datetime.date(1815, 10, 12))
```

- Cando a consulta devolve unha colección de filas, temos 2 alternativas:
  - Usando `cur.fetchall()` e un bucle `for` para recorrer a colección.
    - Ventaxa: é moi sinxelo.
    - Inconveniente: pode ter unha gran demanda de memoria, xa que recupera todas as filas.
  - Usando `cur.fetchone()` nun bucle.
    - Ventaxa: menor demanda de recursos.
    - Inconveniente: Código lixeiramente máis complexo?

```
cur.execute("""select id, nome, datanac
               from persoa""")
rows=cur.fetchall()
for row in rows:
    print(row)
```

```
cur.execute("""select id, nome, datanac
               from persoa""")
row=cur.fetchone()
while row:
    print(row)
    row=cur.fetchone()
```

Tipos de cursores

- O tipo de cursor predeterminado devolve tuplas, de xeito que podemos acceder ós datos polo seu índice (o índice empeza en 0).

```
# select id, nome, datanac from persoa
>>> print(row)
(1, 'Ada Lovelace', datetime.date(1815, 10, 12))

>>> print(row[1])
Ada Lovelace
```

- Podemos especificar un tipo especial de cursor, o **`DictCursor`**, que ademais permite acceder a fila como un dicionario, polo nome do campo, indicando `cursor_factory=psycopg2.extras.DictCursor`:
  - Na creación dun cursor específico: `conn.cursor(cursor_factory=psycopg2.extras.DictCursor)`.
  - Na creación da conexión, para todos os cursores que se creen: `conn=psycopg2.connect(<DSN>, cursor_factory=psycopg2.extras.DictCursor)`.

```
>>> cur=conn.cursor(cursor_factory=psycopg2.extras.DictCursor)
>>> cur.execute("select id, nome, datanac from persoa where id=%s", (1,))
>>> row=cur.fetchone()
>>> row
[1, 'Ada Lovelace', datetime.date(1815, 10, 12)]

>>> row[1]
'Ada Lovelace'

>>> row['nome']
'Ada Lovelace'
```

Execución de sentencias SQL

Propiedades dos cursores

Con `help(cur)` (sendo `cur` un cursor) podemos ver as propiedades e métodos dos que dispoñemos.
Son especialmente interesantes:

- `rowcount`: O número de filas afectadas pola sentencia SQL. Pode ser o número de filas que recupera un select, ou o número de filas que se ven afectadas por outras sentencias DML (por exemplo, número de filas borradas nun `delete`).
  Para as sentencias DDL (ex: create table), `rowcount` é -1.
- `rownumber`: O número de fila que se está procesando na actualidade (tras un `fetchone()`), por exemplo no bucle que recupera cada fila.
  Para as sentencias DDL e DML de modificación de datos `rownumber` é 0.

```
>>> cur.execute("select * from persoa")
>>> row = cur.fetchone()
>>> while row:
...     print(f"Fila número {cur.rownumber} de {cur.rowcount}: {row}")
...     row = cur.fetchone()
...
Fila número 1 de 2: (5, 'Ada Lovelace', datetime.date(1815, 10, 12))
Fila número 2 de 2: (6, 'Alan Turing', datetime.date(1912, 6, 23))
>>> cur.execute("delete from persoa")
>>> cur.rowcount
2
>>> cur.execute("delete from persoa where id=555")
>>> cur.rowcount
0
```

Execución de sentencias SQL

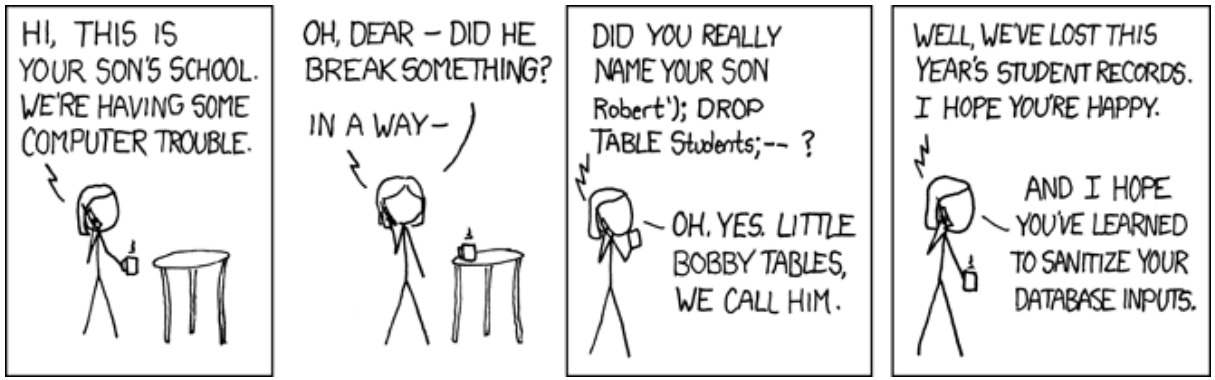
DML con cláusula `returning`

PostgreSQL (e outros xestores) admiten unha cláusula `returning` para devolver un ou varios valores nas sentencias DML `insert`, `delete` ou `update`.
Aquí veremos un exemplo de especial utilidade para `insert`: cando queremos insertar unha fila que ten unha clave primaria autoxenerada (`serial`) e queremos usar noutra sentencia o valor desa clave primaria.

```
# empresa(idemp serial (PK), nomeemp text)
# seccion(idsec serial (PK), nomesec text, idemp int (FK a empresa) )
...
cur.execute("""insert into empresa(nomeemp) values(%(nome)s) returning idemp""",
           {'nome': 'Empresa S.A.'})

id_empresa = cur.fetchone()[0]
print(f"Id insertado: {id_empresa}")

# Agora reusamos o id xenerado para a clave foránea en sección.
# Vemos tamén un exemplo de cláusula returning con varios campos.

cur.execute("""insert into seccion(nomesec, idemp) values(%(nome)s, %(id)s)
                 returning idsec,nomesec,idemp""",
           {'nome': 'Ventas', 'id': id_empresa})
row=cur.fetchone()
print(f"Sección insertada: {row}")
...
# Exemplo de saída:
# Id insertado: 7
# Sección insertada: (23, 'Ventas', 7)
```

Execución de sentencias SQL

Unha nota sobre SQL injection



(Imaxe: `https://xkcd.com/327/`

- Debemos recordar a necesidade de evitar problemas de seguridade como a inxección SQL. Por iso debemos usar sempre parámetros nas consultas.
- Código **inseguro**: Construir a consulta completa como un string.
  Que pasa se executamos a consulta deste exemplo co texto modificado para incluir o nome da persoa?

```
query = "select datanac from persoa where nome = '{}'"
nompers='Ada Lovelace'
# Parece correcto:
# query.format(nompers) # => "select datanac from persoa where nome = 'Ada Lovelace'"
cursor.execute(query.format(nompers)) # => Ok
# Pero, OLLO!
nompers="'; DROP TABLE persoa; -- "
query.format(nompers) # => "select datanac from persoa where nome = ''; DROP TABLE persoa; -- '"

cursor.execute(query.format(nompers)) # => Elimina a táboa persoa
```

Execución de sentencias SQL

Unha nota sobre SQL injection

- Código máis **seguro** usando parámetros.
- O texto da sentencia SQL nunca se ve modificado.

```
query = "select datanac from persoa where nome = %(nompers)s"
nompers='Ada Lovelace'

# Non usamos query.format(nompers) como no caso anterior.
cur.execute(query, {'nompers' : nompers})
cur.fetchall()
[(datetime.date(1815, 10, 12))]

# Non permite a inxección SQL.
# Simplemente toma o valor de nompers para comparar co campo nome,
# e non atopa ningunha fila.
nompers="'; DROP TABLE persoa; -- "
cur.execute(query, {'nompers' : nompers})
cur.fetchall()
[]
```

Índice

Control de erros

Introdución

- Debemos asegurar un bo control de erros e das transaccións.
- Podemos controlar as excepcións xenéricas de Python (`Exception`) ou directamente `psycopg2.Error` e as súas subclases.
  - `pgcode` é o código de erro (ver, por exemplo, `https://www.psycopg.org/docs/errors.html`).
  - `pgerror` é a mensaxe de PostgreSQL.
  - Con `import psycopg2.errorcodes` temos constantes que representan múltiples códigos de erro. Xenera un código máis lexible. Ex: `psycopg2.errorcodes.UNIQUE_VIOLATION` en vez de `"23505"`.
  - Como alternativa, podemos usar subclases específicas. Ex: `psycopg2.errors.UniqueViolation`.

```
try:
    cur.execute("update persoa set id=%(novoid)s where id=%(velloid)s",
                {'novoid': 2, 'velloid': 1})
    conn.commit()
except psycopg2.Error as e:
    print("Erro actualizando.")
    print(f"Tipo de excepción: {type(e)}")
    print(f"Código: {e.pgcode}")
    print(f"Mensaxe: {e.pgerror}")
    conn.rollback()
```

Resultado da execución:

```
Erro actualizando.
Tipo de excepción: <class 'psycopg2.errors.UniqueViolation'>
Código: 23505
Mensaxe: ERROR:  duplicate key value violates unique constraint "persoa_pkey"
DETAIL:  Key (id)=(2) already exists.
```

Control de erros

Visualización de erros en PostgreSQL

Vendo os erros con psql

Usamos`\set VERBOSITY verbose`. A liña ERROR inclúe o equivalente a pgcode e pgerror.
Exemplo:

```
create table ficheiro(
    id int constraint pk_ficheiro primary key,
    nome text constraint nn_ficheiro_nome not null,
    tamano int constraint nn_ficheiro_nome not null);
insert into ficheiro values(1,'F.txt', 324);

penabad=> \set VERBOSITY verbose
penabad=> insert into ficheiro values(1,'F.txt', 324);
ERROR:  23505: duplicate key value violates unique constraint "pk_ficheiro"
DETAIL:  Key (id)=(1) already exists.
SCHEMA NAME:  public
TABLE NAME:  ficheiro
CONSTRAINT NAME:  pk_ficheiro
LOCATION:  _bt_check_unique, nbtinsert.c:534

penabad=> insert into ficheiro values(1,'F.txt', NULL);
ERROR:  23502: null value in column "tamano" violates not-null constraint
DETAIL:  Failing row contains (1, F.txt, null).
SCHEMA NAME:  public
TABLE NAME:  ficheiro
COLUMN NAME:  tamano
LOCATION:  ExecConstraints, execMain.c:2032
```

Control de erros

Exemplo usando psycopg2.Error e constantes

- Debemos usar `import psycopg2.errorcodes`.
- Usando como excepción `Error` ou `OperationalError`, comprobamos `pgcode`.
- Opcionalmente podemos buscar na mensaxe (en `pgerror`) nomes de campo ou restricións.

```
import psycopg2
import psycopg2.errorcodes
try:
    conn = psycopg2.connect("")
    cur=conn.cursor()
    cur.execute("insert into ficheiro values(%s,%s,%s)", (1,'file.txt',324))
    conn.commit()
except psycopg2.Error as e:
    if e.pgcode == psycopg2.errorcodes.UNIQUE_VIOLATION:
        print("Xa existe un ficheiro con ese código")
    elif e.pgcode == psycopg2.errorcodes.NOT_NULL_VIOLATION:
        if '"id"' in e.pgerror:
             print("É obrigatorio especificar o id")
        elif '"nome"' in e.pgerror:
            print("É obrigatorio especificar o nome")
        elif '"tamano"' in e.pgerror:
            print("É obrigatorio especificar o tamano")
    else:
         print(f"Erro de postgres: {e.pgcode} - {e.pgerror}")
    conn.rollback()
```

Control de erros

Exemplo usando psycopg2.Error e as súas subclases

- Teremos varias cláusulas except psycopg2.errors.<excepción>, de máis específica a menos. A última pode ser Error ou OperationalError.
- Os nomes son similares ás constantes, pero usando *CamelCase* en lugar de *SNAKE_CASE*.
- En lugar de nome de excepción pode usarse psycopg2.errors.lookup("<SQLSTATE>"), onde o SQLSTATE é pgcode.
- De novo, opcionalmente podemos buscar na mensaxe (en pgerror) nomes de campo ou restricións.

```
import psycopg2
try:
    conn = psycopg2.connect("")
    cur=conn.cursor()
    cur.execute("insert into ficheiro values(%s,%s,%s)", (1,'file.txt',3e24))
    conn.commit()
except psycopg2.errors.UniqueViolation as e:
    print("Xa existe un ficheiro con ese código")
    conn.rollback()
except psycopg2.errors.NotNullViolation as e:
    if '"id"' in e.pgerror:
        print("É obrigatorio especificar o id")
    elif '"nome"' in e.pgerror:
        print("É obrigatorio especificar o nome")
    elif '"tamano"' in e.pgerror:
        print("É obrigatorio especificar o tamano")
    conn.rollback()
except psycopg2.Error as e:
    print(f"Erro xeral de postgres: {e.pgcode} - {e.pgerror}")
    conn.rollback()
```

Control de erros

Uso da cláusula `with`

- A cláusula `with` de Python non é específica para control de erros, pero pode axudar.
- Se hai que facer algunha "limpeza" pode usarse a parte `finally:` de Python.
  Por exemplo, unha función pode obter un cursor, e executar unha sentencia:
  - Se todo funciona correctamente, faise commit.
  - Se algo vai mal, xestiónase a excepción e faise rollback.
  - En ambos casos hai que liberar o cursor e a conexión.
- Se só utilizamos `finally` para cerrar o cursor, podemos optar pola alternativa de usar `with`, que xa o cerra por nós.

```
try:
    cur=conn.cursor()

    cur.execute(...)
    conn.commit()
except ...
    ... xestión das excepcións
    conn.rollback()
finally:
    if (conn):
        cur.close()
```

```
with conn.cursor() as cur:
   try:
       cur.execute(...)
       conn.commit()
   except ...
       ... xestión das excepcións
       conn.rollback()
```

Índice

Control transaccional

Introdución

- Para xestionar as transaccións de forma explícita debemos observar o valor de `conn.autocommit` e desactivalo en caso de que estea. Temos 2 opcións:
  `conn.autocommit = False`
  `conn.set_session(autocommit = False)`
- Se saímos do programa ou pechamos a conexión mediante `conn.close()` faise un ROLLBACK implícito.
- PostgreSQL considera as sentencias DDL transaccionais (non como Oracle, por exemplo), polo que necesitan un COMMIT explícito.
- Se unha sentencia SQL en PostgreSQL falla nunha transacción, non permite a execución de máis sentencias SQL, e a transacción acabará en ROLLBACK (aínda que solicitemos un COMMIT).

```
penabad=> begin work;
BEGIN
penabad=*> create table t(id int primary key);
CREATE TABLE
penabad=*> insert into t values(1);
INSERT 0 1
penabad=*> insert into t values(1);
ERROR:  duplicate key value violates unique constraint "t_pkey"
DETAIL:  Key (id)=(1) already exists.
penabad=!> insert into t values(2);
ERROR:  current transaction is aborted, commands ignored until end of transaction block
penabad=!> commit;
ROLLBACK
penabad=> select * from t;
ERROR:  relation "t" does not exist
```

Control transaccional

Mecanismos de control

A clase `psycopg2.extensions.connection` que devolve a chamada a `conn=psycopg2.connect(...)` ten unha serie de atributos e métodos relacionados co control transaccional.
`help(psycopg2.extensions.connection)` da a lista completa.

- `commit()` e `rollback()` para confirmar ou anular a transacción actual.
- `set_session()` permite modificar varios aspectos (son atributos de `connection` que se poden consultar e/ou modificar directamente):
  - `autocommit = True / False` para o autocommit, xa descrito.
  - `deferrable = True / False`: pon todas as restricións aplazables en modo aplazado (`True`) ou inmediato(`False`).
  - `readonly = True / False` pon a sesión en modo de só lectura.
  - `isolation_level`: pon a sesión no nivel de aillamento especificado
- `set_isolation_level()`: Alternativa para especificar o nivel de aillamento.

Non hai métodos para crear `SAVEPOINT`s ou volver a eles, pero pode utilizarse directamente SQL nun cursor. Poden ser moi útiles para tratar o problema mostrado no exemplo da transparencia anterior.

```
cur.execute("SAVEPOIINT SP1")
...
cur.execute("ROLLBACK TO SP1")
```

En realidade, pode usarse SQL para outras funcionalidades. Por exemplo, as seguintes sentencias serían equivalentes:

```
conn.commit()
cur.execute("COMMIT")
```

Control transaccional

Modos de aillamento

Podemos consultar ou especificar o modo de aillamento da sesión, o que inclúe a transacción actual e as seguintes, con

```
conn.isolation_level #Consulta ou modificación
conn.set_isolation_level(<nivel>)
conn.set_session(isolation_level = <nivel>)
```

Os posibles valores, definidos en `psycopg2.extensions`, son

- `ISOLATION_LEVEL_AUTOCOMMIT = 0`
- `ISOLATION_LEVEL_READ_UNCOMMITTED = 4`
- `ISOLATION_LEVEL_READ_COMMITTED = 1`
- `ISOLATION_LEVEL_REPEATABLE_READ = 2`
- `ISOLATION_LEVEL_SERIALIZABLE = 3`
- `ISOLATION_LEVEL_DEFAULT = None`

Recordemos que o estándar SQL:

- Inclúe `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ` e `SERIALIZABLE`.
- Se a un xestor se lle solicita un nivel que non manexa, debe ofrecer outro superior.

Deste xeito, PostgreSQL con psycopg2:

- O modo read uncommitted trátase como read committed.
- O modo repeatable read trátase como serializable.
- Establecendo o nivel `ISOLATION_LEVEL_AUTOCOMMIT` activa o modo `autocommit`.
- Establecendo o nivel `ISOLATION_LEVEL_DEFAULT` utiliza o modo predeterminado configurado na instalación de PostgreSQL (habitualmente `read committed`).

Control transaccional

Modos de aillamento – Exemplo

Exemplo: Empezamos cunha táboa `test(id int)` sen filas.

Comprobamos o que verá o script de Python cos distintos niveis de aillamento.
As inserciónss realizaranse en `psql` en modo autocommit.

```
import psycopg2

def print_test():
    cur.execute("Select count(*) from test")
    print(f"Hai {cur.fetchone()[0]} filas.")

conn=psycopg2.connect("")
cur=conn.cursor()
# Modo predeterminado = READ COMMITTED
print("Modo READ COMMITTED")
print_test()   # "Hai 0 filas"
input("Pulsa ENTER despois de insertar 1 fila e confirmar")
                                                    # En psql: insert into test values(1)
print_test()   # "Hai 1 filas" xa que en psql se confirmou (autocommit)
conn.commit()
print("Nova transacción (sigue en modo READ COMMITTED)")
print_test()   # "Hai 1 filas" na nova transacción
conn.commit()

# Cambiamos a modo SERIALIZABLE
print("Modo SERIALIZABLE")
conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_SERIALIZABLE)
print_test()  # "Hai 1 filas"

input("Pulsa ENTER despois de insertar 1 fila e confirmar")
                                                    # En psql: insert into test values(2)
print_test()  # "Hai 1 filas", xa que en modo serializable
              # non se ven os cambios das outras transaccións
conn.commit()
print("Nova transacción (sigue en modo SERIALIZABLE)")
print_test()  # "Hai 2 filas" na nova transacción.
```

Índice

Bibliografía

- PEP 249 – Python Database API Specification v2.0.
  `https://www.python.org/dev/peps/pep-0249/`.
- Psycopg – PostgreSQL database adapter for Python.
  `https://www.psycopg.org/docs/`.
- PostgreSQL 13 documentation.
  `https://www.postgresql.org/docs/13/`.