

Soporte a la Gestión de Datos con Programación Visual

GESTION DE BASES DE DATOS

GENERALIDADES

En python, el acceso a bases de datos está estandarizado por la especificación Database API (DB-API), actualmente en la versión 2.0 ([PEP 249: Python Database API Specification v2.0](#))

Gracias a esto, se puede acceder a cualquier base de datos utilizando la misma interfaz (ya sea un motor remoto, local, ODBC, etc.).

El mismo código se podría llegar a usar para cualquier base de datos, tomando siempre los recaudos necesarios (lenguaje SQL estándar, estilo de parámetros soportado, etc.)

CONECTORES

Hay que descargar los conectores que se vayan a usar, compatibles con la versión del motor de base de datos y de Python a emplear.

Ej: para MySql

<https://dev.mysql.com/downloads/connector/python/>

(descargar para Python 3.4, 32-bit o 64-bit)

También:

pip install PyMySQL

Para Odbc:

<https://pypi.python.org/pypi/pypyodbc>

O también: pip install pypyodbc

ODBC

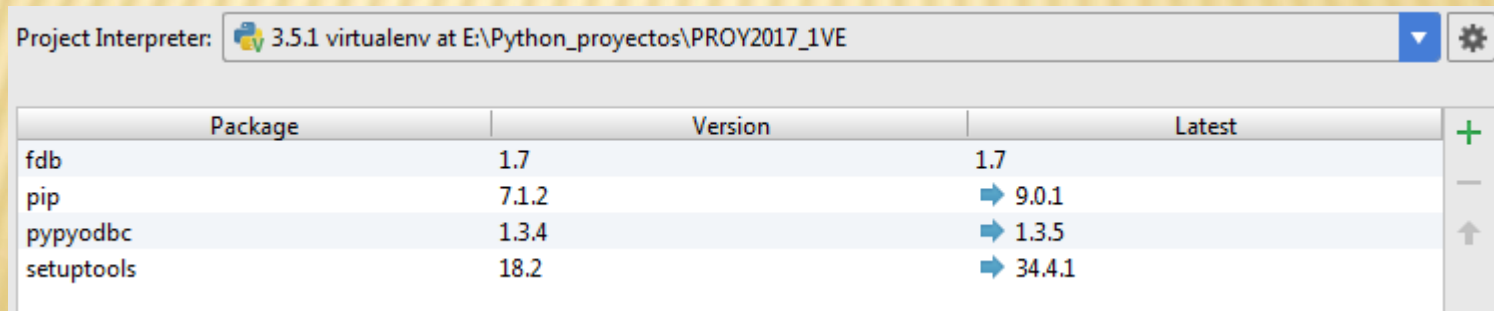
Open DataBase Connectivity (**ODBC**) es un estándar de acceso a las bases de datos desarrollado por SQL Access Group (SAG) en 1992.

El objetivo de **ODBC** es hacer posible el acceder a cualquier dato desde cualquier aplicación, sin importar qué sistema de gestión de bases de datos (DBMS) almacene los datos.

Casi todos los motores de B.Datos ofrecen un driver ODBC. Hay que instalar ese driver en cada computadora que vaya a acceder por esta vía a una base de datos.

INSTALACION DE CONECTORES

Para que los proyectos hagan uso de los conectores, hay que incorporarlos al entorno virtual en uso. Se puede emplear la herramienta PIP, que a su vez descarga el conector y lo copia en el entorno virtual. En Settings... del proyecto,



Project Interpreter: 3.5.1 virtualenv at E:\Python_proyectos\PROY2017_1VE				
Package	Version	Latest		
fdb	1.7	1.7		+
pip	7.1.2	➔ 9.0.1		-
pypyodbc	1.3.4	➔ 1.3.5		↑
setuptools	18.2	➔ 34.4.1		

INSTALACION DE CONECTORES - ALTERNATIVA

En caso de no poder usar la herramienta PIP, basta con copiar la carpeta que contiene el código del conector en:

...(entorno virtual)..\Lib\site-packages\

Por ej.: carpetas

\pypyodbc y

\pymysql,

que pueden descargarse del campus de la cátedra.

ESTRUCTURA BÁSICA DE LA GESTIÓN DE DATOS

- Importar el conector
- Conectarse a la base de datos (crear un objeto conexión con connect del módulo conector)
- Crear un objeto Cursor de la conexión
- Ejecutar una consulta (método execute del cursor)
- Obtener los datos (método fetch o iterar sobre el cursor)
- Si hubo modificaciones, usar commit() del cursor
- Cerrar el cursor (método close del cursor)

CONEXIÓN

`conn = conector.connect(.....)`

Los parámetros de la cadena de conexión dependen del conector:

Ej. MySql: (con pymysql)

“host='localhost', port=3306, user='root', passwd='1234', db='mydb'”

Ej Access: (con pypyodbc)

“Driver= {Microsoft Access Driver (*.mdb, *.accdb)};
DBQ=D:\database.mdb”

EJEMPLO BÁSICO

```
import pymysql                # --- importar el conector correspondiente

#--- crear un objeto conexión con una base de datos:
conn = pymysql.connect( host='localhost', port=3306, user='root', passwd='1234', db='mydb' )

#--- crear un cursor:
cur = conn.cursor()

#--- usar el cursor creado para ejecutar distintos comandos SQL:
caux = "SELECT * FROM bancos"
cur.execute( caux )

cur.fetchone()               #--- recuperar la información obtenida con una Select.  1 registro
print(cur.descripcion)       #--- campos de la tabla BANCOS:  descripcion,
print(cur.rowcount)

cur.fetchall()               #--- recuperar la información obtenida con una Select.  Lista con todos los registros.
for row in cur:
    print(row)

#--- cerrar todo ...
cur.commit()
cur.close()
conn.close()
```

CONEXIÓN - ODBC

Ejemplo para ver los drivers ODBC instalados.

```
import pypyodbc
```

```
def lista_drivers():  
    ld = pypyodbc.drivers()  
    ld.sort()  
    for d in ld:  
        print(d)
```

```
lista_drivers()
```

CURSOR - CONSULTAS

```
cursor.execute("SELECT * FROM bancos" )
```

Para ver el resultado de la SELECT:

```
cursor.fetchall()                #--- obtenemos lista con todos los registros
for row in cursor:                #--- recorremos la lista:
    print( 'Id: {0}  Nombre: {1}'.format (row[0], row[1]))
```

cursor.description ***devuelve información de los campos.***

Da una lista con una tupla por cada campo. El primer elemento de la tupla es el nombre del campo.

Para obtener los nombres de las columnas:

```
columns = [column[0] for column in cursor.description]
```

Otra forma de obtener los registros:

```
for row in cursor:
    print(dict(zip(columns, row)))
```

Otra. Lista de diccionarios – Toda la consulta - en un objeto

```
query_results = [dict(zip(columns, row)) for row in cursor]
print(query_results)
```


CONSULTAS – CONTINUACION

#obtenemos lista con todos los registros:

cursor.**fetchall()**

de a un registro

cursor.**fetchone()**

varios

cursor.**fetchmany()**

PASO DE PARÁMETROS A SQL

Ej. pasaje de datos de un campo en Insert.

Ésta es la forma más segura - datos como tupla como 2do parámetro del execute:

```
caux = 'INSERT INTO persona (nombre, fechanacimiento, dni, altura) VALUES  
(%s,%s,%s,%s)'  
tdatos = ( cnom, dfecha_nac, ndni, naltura )  
cur.execute(caux, tdatos)
```

#--- otra forma - poner comillas donde hay que reemplazar textos o fechas (ver la diferencia entre '{0}' y {3} datos texto y números):

```
caux = "INSERT INTO persona (nombre, fechanacimiento, dni, altura) VALUES  
( '{0}', '{1}', {2}, {3} )"
```

```
cins = caux.format( cnom, dfecha_nac, ndni, naltura )  
cur.execute(cins)
```

#--- Esta forma es más insegura. Expone a SQL injection

OTRO MOTOR: SQLITE

Es un motor fácil de usar, no requiere programa corriendo, ni configuración. Muy útil para hacer pruebas.

```
import sqlite3  #--- cumple con DB-API 2.0
```

```
#--- se puede crear en memoria:
```

```
db=sqlite3.connect(':memory:')
```

```
#--- o bien en el disco:
```

```
db = sqlite3.connect('mibase')
```

```
cursor = db.cursor()
```

```
cSQL = 'CREATE TABLE IF NOT EXISTS ventas(id INTEGER PRIMARY KEY ASC,  
descrip TEXT(25), cant REAL(15,2))'
```

```
cursor.execute(cSQL)
```

```
db.commit()
```

```
db.close()
```


INSERTAR EN SQLITE

```
cur = db.cursor()
```

```
cSQL = 'CREATE TABLE IF NOT EXISTS ventas(id INTEGER PRIMARY KEY ASC,  
descrip TEXT(25), cant REAL(15,2))'
```

```
cur.execute(cSQL)
```

#--- si no se asigna id, el motor usa auto increment

```
cSQL = "INSERT into ventas2 (descrip,cant) VALUES (?,?)"
```

```
tdatos = ('queso', 12.5)      #--- tupla
```

```
cur.execute(cSQL, tdatos)    #--- 2 parámetros
```

```
id = cur.lastrowid           #--- devuelve el id del último ingresado
```

```
db.dbcommit()
```

```
db.close()
```

SELECT

Se puede hacer como en los ejemplos anteriores, con fetchall(), por ej, o bien:

```
cursor.execute("SELECT name, email, phone FROM users")
```

#--- el mismo objeto cursor es iterable. No necesita fetchall():

```
for fila in cursor:
```

fila[0] devuelve (name), el primer campo de la select

```
print('{0} : {1}, {2}'.format(fila[0], fila[1], fila[2]))
```

Para condición con parámetros, se usa ? también.

```
usu_id = 3
```

```
cursor.execute("SELECT name, email, phone FROM users WHERE  
id=?", (usu_id,))
```

```
usuario = cursor.fetchone()
```