Capítulo
Gerenciando banco de dados SQLite3 com Python

Para entender o uso de classes e métodos leia o capítulo Introdução a Classes e Métodos em Python. Já os comandos SQL e a manipulação de registros no SQLite3 são apresentados no capítulo Guia rápido de comandos SQLite3.

Para os exemplos considere a tabela `clientes` e seus campos:

Campo	Tipo	Requerido
id	inteiro	sim
nome	texto	sim
idade	inteiro	não
cpf	texto (11)	sim
email	texto	sim
fone	texto	não
cidade	texto	não
uf	texto (2)	sim
criado_em	data	sim
bloqueado	booleano	não

Obs: O campo `bloqueado` nós vamos inseri-lo depois com o comando `ALTER TABLE`.

Conectando e desconectando do banco

Podemos criar o banco de dados de duas formas: na **memória RAM**

```
# conectando...
conn = sqlite3.connect(':memory:')
```

ou persistindo em um **banco de dados**. Vamos usar sempre esta opção!

```
# conectando...
conn = sqlite3.connect('clientes.db')
```

Uma sintaxe mínima para se conectar a um banco de dados é:

```
# connect_db.py
# 01_create_db.py
import sqlite3

conn = sqlite3.connect('clientes.db')
conn.close()
```

O último método (close( )) faz a desconexão do banco, fechando-o.

Criando um banco de dados

O código para criar um banco de dados é o mesmo mencionado anteriormente.

Para rodar este programa abra o **terminal** (cmd.exe) e digite:

```
> python3 01_create_db.py
> dir *.db
```

Digitando `dir` você verá que o banco foi criado.

Criando uma tabela

Para criar uma tabela no banco de dados usamos dois métodos fundamentais:

- **cursor**: é um interador que permite navegar e manipular os registros do bd.
- **execute**: lê e executa comandos SQL puro diretamente no bd.

```
# 02_create_schema.py
import sqlite3

# conectando...
conn = sqlite3.connect('clientes.db')
# definindo um cursor
cursor = conn.cursor()

# criando a tabela (schema)
cursor.execute("""
CREATE TABLE clientes (
        id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
        nome TEXT NOT NULL,
        idade INTEGER,
        cpf     VARCHAR(11) NOT NULL,
        email TEXT NOT NULL,
        fone TEXT,
        cidade TEXT,
        uf VARCHAR(2) NOT NULL,
        criado_em DATE NOT NULL
);
""")

print('Tabela criada com sucesso.')
# desconectando...
conn.close()
```

Para executar digite no **terminal (cmd.exe)**:

```
> python3 02_create_schema.py
> sqlite3 clientes.db '.tables'
> sqlite3 clientes.db 'PRAGMA table_info(clientes)'
```

Digitando `sqlite3 clientes.db '.tables'` você verá que a tabela foi criada.

E o comando `sqlite3 clientes.db 'PRAGMA table_info(clientes)'` retorna os campos da tabela.

Nota: Caso você use *Python 2, a* única alteração a ser feita é no comando print, onde você deve tirar os parênteses. E no início do arquivo é recomendável que se defina a codificação `utf-8`, que no caso do Python 3 já é padrão.

```
# 02_create_schema.py
# -*- coding: utf-8 -*-
```

```
# usando Python 2
import sqlite3
...
print 'Tabela criada com sucesso.'
# desconectando...
conn.close()
```

Agora vamos fazer o CRUD. Começando com a letra

Create - Inserindo um registro com comando SQL

A única novidade aqui é o método **commit()**. É ele que grava de fato as alterações na tabela. *Lembrando que uma tabela é alterada com as instruções SQL ``INSERT, UPDATE`` e ``DELETE``.*

```
# 03_create_data_sql.py
import sqlite3

conn = sqlite3.connect('clientes.db')
cursor = conn.cursor()

# inserindo dados na tabela
cursor.execute("""
INSERT INTO clientes (nome, idade, cpf, email, fone, cidade, uf, criado_em)
VALUES ('Regis', 35, '00000000000', 'regis@email.com', '11-98765-4321', 'Sao
Paulo', 'SP', '2014-06-08')
""")

cursor.execute("""
INSERT INTO clientes (nome, idade, cpf, email, fone, cidade, uf, criado_em)
VALUES ('Aloisio', 87, '11111111111', 'aloisio@email.com', '98765-4322',
'Porto Alegre', 'RS', '2014-06-09')
""")

cursor.execute("""
INSERT INTO clientes (nome, idade, cpf, email, fone, cidade, uf, criado_em)
VALUES ('Bruna', 21, '22222222222', 'bruna@email.com', '21-98765-4323', 'Rio
de Janeiro', 'RJ', '2014-06-09')
""")

cursor.execute("""
INSERT INTO clientes (nome, idade, cpf, email, fone, cidade, uf, criado_em)
VALUES ('Matheus', 19, '33333333333', 'matheus@email.com', '11-98765-4324',
'Campinas', 'SP', '2014-06-08')
""")

# gravando no bd
conn.commit()

print('Dados inseridos com sucesso.')

conn.close()
```

Para executar digite no **terminal (cmd.exe)**:

```
> python3 03_create_data_sql.py
```

Inserindo n registros com uma tupla de dados

Usando uma *lista* podemos inserir vários registros de uma vez, e o método `executemany` faz essa ação.

```
# 04_create_data_nrecords.py
import sqlite3

conn = sqlite3.connect('clientes.db')
cursor = conn.cursor()

# criando uma lista de dados
lista = [(
    'Fabio', 23, '44444444444', 'fabio@email.com', '1234-5678', 'Belo
Horizonte', 'MG', '2014-06-09'),
    ('Joao', 21, '55555555555', 'joao@email.com',
     '11-1234-5600', 'Sao Paulo', 'SP', '2014-06-09'),
    ('Xavier', 24, '66666666666', 'xavier@email.com', '12-1234-5601',
'Campinas', 'SP', '2014-06-10')]

# inserindo dados na tabela
cursor.executemany("""
INSERT INTO clientes (nome, idade, cpf, email, fone, cidade, uf, criado_em)
VALUES (?,?,?,?,?,?,?,?)
""", lista)

conn.commit()
print('Dados inseridos com sucesso.')
conn.close()
```

Observe o uso de "**?**": isso significa que no lugar de cada "**?**" entrarão os valores da lista na sua posição respectiva. É o que nós chamamos de *parâmetros de entrada*.

Para executar digite no **terminal (cmd.exe)**:

```
> python3 04_create_data_nrecords.py
```

Inserindo um registro com parâmetros de entrada definido pelo usuário

Neste exemplo usaremos parâmetros de entrada, que deverá ser digitado pelo usuário. Esta é a forma mais desejável de entrada de dados porque o usuário pode digitar os dados em tempo de execução.

```
# 05_create_data_param.py
import sqlite3

conn = sqlite3.connect('clientes.db')
cursor = conn.cursor()

# solicitando os dados ao usuário
p_nome = input('Nome: ')
p_idade = input('Idade: ')
p_cpf = input('CPF: ')
p_email = input('Email: ')
p_fone = input('Fone: ')
p_cidade = input('Cidade: ')
p_uf = input('UF: ')
p_criado_em = input('Criado em (yyyy-mm-dd): ')
```

```
# inserindo dados na tabela
cursor.execute("""
INSERT INTO clientes (nome, idade, cpf, email, fone, cidade, uf, criado_em)
VALUES (?,?,?,?,?,?,?,?)
""", (p_nome, p_idade, p_cpf, p_email, p_fone, p_cidade, p_uf, p_criado_em))

conn.commit()

print('Dados inseridos com sucesso.')

conn.close()
```

Nota: Caso use *Python 2* use o método `raw_input()` em

```
# python 2
p_nome = raw_input('Nome: ')
...
print 'Dados inseridos com sucesso.'
conn.close()
```

Para executar digite no **terminal (cmd.exe):**

```
> python3 05_create_data_param.py
```

Veja a interação do programa:

```
Nome: Regis
Idade: 35
CPF: 30020030011
Email: regis@email.com
Fone: 11 9537-0000
Cidade: Sao Paulo
UF: SP
Criado em (yyyy-mm-dd): 2014-06-15
Dados inseridos com sucesso.
```

Read - Lendo os dados

Aqui nós usamos o famoso `SELECT`. O método `fetchall()` retorna o resultado do `SELECT`.

```
# 06_read_data.py
import sqlite3

conn = sqlite3.connect('clientes.db')
cursor = conn.cursor()

# lendo os dados
cursor.execute("""
SELECT * FROM clientes;
""")

for linha in cursor.fetchall():
    print(linha)
conn.close()
```

Para executar digite no **terminal (cmd.exe)**:

```
> python3 06_read_data.py
```

Eis o resultado:

```
(1, 'Regis', 35, '00000000000', 'regis@email.com', '11-98765-4321', 'Sao
Paulo', 'SP', '2014-06-08')
(2, 'Aloisio', 87, '11111111111', 'aloisio@email.com', '98765-4322', 'Porto
Alegre', 'RS', '2014-06-09')
(3, 'Bruna', 21, '22222222222', 'bruna@email.com', '21-98765-4323', 'Rio de
Janeiro', 'RJ', '2014-06-09')
(4, 'Matheus', 19, '33333333333', 'matheus@email.com', '11-98765-4324',
'Campinas', 'SP', '2014-06-08')
(5, 'Fabio', 23, '44444444444', 'fabio@email.com', '1234-5678', 'Belo
Horizonte', 'MG', '2014-06-09')
(6, 'Joao', 21, '55555555555', 'joao@email.com', '11-1234-5600', 'Sao
Paulo', 'SP', '2014-06-09')
(7, 'Xavier', 24, '66666666666', 'xavier@email.com', '12-1234-5601',
'Campinas', 'SP', '2014-06-10')
(8, 'Regis', 35, '30020030011', 'regis@email.com', '11 9750-0000', 'Sao
Paulo', 'SP', '2014-06-15')
```

Update - Alterando os dados

Observe o uso das variáveis `id_cliente` onde definimos o `id` a ser alterado, `novo_fone` e `novo_criado_em` usados como parâmetro para alterar os dados. Neste caso, salvamos as alterações com o método `commit()`.

```
# 07_update_data.py
import sqlite3

conn = sqlite3.connect('clientes.db')
cursor = conn.cursor()

id_cliente = 1
novo_fone = '11-1000-2014'
novo_criado_em = '2014-06-11'

# alterando os dados da tabela
cursor.execute("""
UPDATE clientes
SET fone = ?, criado_em = ?
WHERE id = ?
""", (novo_fone, novo_criado_em, id_cliente))

conn.commit()

print('Dados atualizados com sucesso.')

conn.close()
```

Para executar digite no **terminal (cmd.exe)**:

```
> python3 07_update_data.py
```

Delete - Deletando (apagando) os dados

Vamos excluir um registro pelo seu `id`.

```
# 08_delete_data.py
import sqlite3

conn = sqlite3.connect('clientes.db')
cursor = conn.cursor()

id_cliente = 8

# excluindo um registro da tabela
cursor.execute("""
DELETE FROM clientes
WHERE id = ?
""", (id_cliente,))

conn.commit()

print('Registro excluido com sucesso.')

conn.close()
```

Para executar digite no terminal (cmd.exe):

```
> python3 08_delete_data.py
```

Adicionando uma nova coluna

Para inserir uma nova coluna na tabela usamos o comando SQL `ALTER TABLE`.

```
# 09_alter_table.py
import sqlite3

conn = sqlite3.connect('clientes.db')
cursor = conn.cursor()

# adicionando uma nova coluna na tabela clientes
cursor.execute("""
ALTER TABLE clientes
ADD COLUMN bloqueado BOOLEAN;
""")

conn.commit()

print('Novo campo adicionado com sucesso.')

conn.close()
```

Para executar digite no terminal (cmd.exe):

```
> python3 09_alter_table.py
```

Lendo as informações do banco de dados

Para ler as informações da tabela usamos o comando
PRAGMA.
Para listar as tabelas do banco usamos o comando
SELECT name FROM sqlite_master ...
Para ler o schema da tabela usamos o comando
SELECT sql FROM sqlite_master ...

```
# 10_view_table_info.py
import sqlite3

conn = sqlite3.connect('clientes.db')
cursor = conn.cursor()
nome_tabela = 'clientes'

# obtendo informações da tabela
cursor.execute('PRAGMA table_info({})'.format(nome_tabela))

colunas = [tupla[1] for tupla in cursor.fetchall()]
print('Colunas:', colunas)
# ou
# for coluna in colunas:
#    print(coluna)

# listando as tabelas do bd
cursor.execute("""
SELECT name FROM sqlite_master WHERE type='table' ORDER BY name
""")

print('Tabelas:')
for tabela in cursor.fetchall():
    print("%s" % (tabela))

# obtendo o schema da tabela
cursor.execute("""
SELECT sql FROM sqlite_master WHERE type='table' AND name=?
""", (nome_tabela,))

print('Schema:')
for schema in cursor.fetchall():
    print("%s" % (schema))

conn.close()
```

Para executar digite no terminal (cmd.exe):

```
> python3 10_view_table_info.py
```

Eis o resultado:

```
Colunas: ['id', 'nome', 'idade', 'cpf', 'email', 'fone', 'cidade', 'uf',
'criado_em', 'bloqueado']
```

```
Tabelas:
clientes
sqlite_sequence
Schema:
CREATE TABLE clientes (
        id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
        nome TEXT NOT NULL,
        idade INTEGER,
        cpf     VARCHAR(11) NOT NULL,
        email TEXT NOT NULL,
        fone TEXT,
        cidade TEXT,
        uf VARCHAR(2) NOT NULL,
        criado_em DATE NOT NULL
, bloqueado BOOLEAN)
```

Fazendo backup do banco de dados (exportando dados)

Talvez seja este o item mais importante: **backup**. Observe o uso da biblioteca **io** que salva os dados num arquivo externo através do método `write`, e o método `iterdump()` que exporta a estrutura e dados da tabela para o arquivo externo.

```
# 11_backup.py
import sqlite3
import io

conn = sqlite3.connect('clientes.db')

with io.open('clientes_dump.sql', 'w') as f:
    for linha in conn.iterdump():
        f.write('%s\n' % linha)

print('Backup realizado com sucesso.')
print('Salvo como clientes_dump.sql')

conn.close()
```

Para executar digite no terminal (cmd.exe):

```
> python3 11_backup.py
> type clientes_dump.sql  (linux: cat)
```

Com o comando type (ou `cat`) você poderá ler a estrutura da tabela salva.

Recuperando o banco de dados (importando dados)

Criaremos um novo banco de dados e iremos reconstruir a tabela e os dados com o arquivo *clientes_dump.sql*. O método `read()` lê o conteúdo do arquivo *clientes_dump.sql* e o método `executescript()` executa as instruções SQL escritas neste arquivo.

```
# 12_read_sql.py
import sqlite3
import io

conn = sqlite3.connect('clientes_recuperado.db')
```

```
cursor = conn.cursor()

f = io.open('clientes_dump.sql', 'r')
sql = f.read()
cursor.executescript(sql)

print('Banco de dados recuperado com sucesso.')
print('Salvo como clientes_recuperado.db')

conn.close()
```

Para executar digite no terminal (cmd.exe):

```
> python3 12_read_sql.py
Banco de dados recuperado com sucesso.
Salvo como clientes_recuperado.db
> sqlite3 clientes_recuperado.db 'SELECT * FROM clientes;'
```

Com o último comando você verá que os dados estão lá. São e salvo!!!

Próximo passo

Usaremos os pacotes names e rstr , o primeiro gera nomes randômicos e o segundo gera string e números randômicos. Você deve instalar os pacotes usando o PIP.

Criando valores randômicos

Antes de mexer no banco de fato vamos criar uns valores randômicos para popular o banco futuramente.

O arquivo gen_random_values.py gera idade, cpf, telefone, data e cidade aleatoriamente. Para isso vamos importar algumas bibliotecas.

```
# gen_random_values.py
import random
import rstr
import datetime
```

Vamos criar uma função `gen_age()` para gerar um número inteiro entre 15 e 99 usando o comando random.randint(a,b) .

```
def gen_age():
    return random.randint(15, 99)
```

A função `gen_cpf()` gera uma string com 11 caracteres numéricos. No caso, o primeiro parâmetro são os caracteres que serão sorteados e o segundo é o tamanho da string.

```
def gen_cpf():
    return rstr.rstr('1234567890', 11)
```

Agora vamos gerar um telefone com a função `gen_phone()` no formato (xx) xxxx-xxxx

```
def gen_phone():
    return '({0}) {1}-{2}'.format(
        rstr.rstr('1234567890', 2),
        rstr.rstr('1234567890', 4),
        rstr.rstr('1234567890', 4))
```

A função `gen_timestamp()` gera um *datetime* no formato `yyyy-mm-dd hh:mm:ss.000000`. Repare no uso do `random.randint(a,b)` com um intervalo definido para cada parâmetro.

Quando usamos o comando datetime.datetime.now().isoformat() ele retorna a data e hora atual no formato `yyyy-mm-ddThh:mm:ss.000000`. Para suprimir a letra T usamos o comando `.isoformat(" ")` que insere um espaço no lugar da letra T.

```
def gen_timestamp():
    year = random.randint(1980, 2015)
    month = random.randint(1, 12)
    day = random.randint(1, 28)
    hour = random.randint(1, 23)
    minute = random.randint(1, 59)
    second = random.randint(1, 59)
    microsecond = random.randint(1, 999999)
    date = datetime.datetime(year, month, day, hour,minute,
second,microsecond).isoformat(" ")
    return date
```

A função `gen_city()` escolhe uma cidade numa lista com o comando random.choice(seq) (suprimi alguns valores).

```
def gen_city():
    list_city = [
        [u'São Paulo', 'SP'],
        [u'Rio de Janeiro', 'RJ'],
        [u'Porto Alegre', 'RS'],
        [u'Campo Grande', 'MS']]
    return random.choice(list_city)
```

Conectando e desconectando do banco

Como mencionado antes, a intenção é criar um único arquivo. Mas, inicialmente, vamos usar um arquivo exclusivo para conexão o qual chamaremos de connect_db.py , assim teremos um arquivo que pode ser usado para vários testes de conexão com o banco de dados.

```
# connect_db.py
import sqlite3

class Connect(object):

    def __init__(self, db_name):
        try:
            # conectando...
            self.conn = sqlite3.connect(db_name)
            self.cursor = self.conn.cursor()
            # imprimindo nome do banco
            print("Banco:", db_name)
```

```
            # lendo a versão do SQLite
            self.cursor.execute('SELECT SQLITE_VERSION()')
            self.data = self.cursor.fetchone()
            # imprimindo a versão do SQLite
            print("SQLite version: %s" % self.data)
        except sqlite3.Error:
            print("Erro ao abrir banco.")
            return False
```

Aqui usamos o básico já visto na parte 1 que são os comandos `sqlite3.connect()` e `cursor()`. Criamos uma classe "genérica" chamada `Connect()` que representa o banco de dados. E no inicializador da classe `__init__` fazemos a conexão com o banco e imprimimos a versão do SQLite, definido em `self.cursor.execute('SELECT SQLITE_VERSION()')`.

O próximo passo é fechar a conexão com o banco:

```
def close_db(self):
    if self.conn:
        self.conn.close()
        print("Conexão fechada.")
```

Este método está dentro da classe `Connect()`, portanto atente-se a **endentação**.

Agora, criamos uma instância da classe acima e chamamos de `ClientesDb()`, representando um banco chamado *clientes.db*.

```
class ClientesDb(object):

    def __init__(self):
        self.db = Connect('clientes.db')

    def close_connection(self):
        self.db.close_db()
```

Fazendo desta forma é possível instanciar outras classes, uma para cada banco, como `PessoasDb()` que veremos mais pra frente.

Finalmente, para rodar o programa podemos escrever o código abaixo...

```
if __name__ == '__main__':
    cliente = ClientesDb()
    cliente.close_connection()
```

salvar... e no terminal digitar:

```
> python3 connect_db.py
> dir *.db
```

Pronto, o banco *clientes.db* está criado.

Modo interativo

Legal mesmo é quando usamos o modo interativo para rodar os comandos do python, para isso podemos usar o python3 ou ipython3. No terminal basta digitar python3 `ENTER` que vai aparecer o prompt abaixo (*na mesma pasta do projeto, tá?*)

```
> python3
Python 3.4.0 (default, Apr 11 2014, 13:05:18)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Agora vamos digitar os seguintes comandos. As explicações serão posteriormente apresentadas.

```
>>> from connect_db import Connect
>>> dir(Connect)
>>> db = Connect('clientes.db')
>>> dir(db)
>>> db.close_db()
>>> exit()
```

A primeira linha importa a classe `Connect` do arquivo *connect_db.py*.

O comando `dir(Connect)` lista todos os métodos da classe `Connect()`, inclusive `__init__` e `close_db()`.

`db = Connect('clientes.db')` cria uma instância da classe `Connect()` e usa o argumento `'clientes.db'` para criar o banco com o nome especificado.

o comando `dir(db)` lista os métodos da instância.

E `db.close_db()` fecha a conexão com o banco.

Criando um banco de dados

Nosso arquivo principal se chamará manager_db.py e iremos incrementá-lo aos poucos. Na verdade quando usamos o comando `c = ClientesDb()` já criamos o banco de dados com o nome especificado, e instanciamos uma classe chamada `ClientesDb`. Portanto esta fase já está concluida.

Repetindo o código inicial para criar e conectar o banco de dados:

```
# manager_db.py
import os
import sqlite3
import io
import datetime
import names
import csv
from gen_random_values import *


class Connect(object):

    def __init__(self, db_name):
        try:
            # conectando...
            self.conn = sqlite3.connect(db_name)
```

```
            self.cursor = self.conn.cursor()
            print("Banco:", db_name)
            self.cursor.execute('SELECT SQLITE_VERSION()')
            self.data = self.cursor.fetchone()
            print("SQLite version: %s" % self.data)
        except sqlite3.Error:
            print("Erro ao abrir banco.")
            return False

    def commit_db(self):
        if self.conn:
            self.conn.commit()

    def close_db(self):
        if self.conn:
            self.conn.close()
            print("Conexão fechada.")


class ClientesDb(object):

    tb_name = 'clientes'

    def __init__(self):
        self.db = Connect('clientes.db')
        self.tb_name

    def fechar_conexao(self):
        self.db.close_db()

if __name__ == '__main__':
    c = ClientesDb()
```

Rodando no **terminal (cmd.exe)**...

```
> python3 manager_db.py
> dir *.db
```

O banco `clientes.db` está criado.

Ou no **modo interativo**...

```
> python3
>>> from manager_db import *
>>> c = ClientesDb()
Banco: clientes.db
SQLite version: 3.8.2
>>> exit()
```

Criando uma tabela

Agora é tudo continuação do arquivo manager_db.py ...

```
def criar_schema(self, schema_name='sql/clientes_schema.sql'):
    print("Criando tabela %s ..." % self.tb_name)
```

```
    try:
        with open(schema_name, 'rt') as f:
            schema = f.read()
            self.db.cursor.executescript(schema)
    except sqlite3.Error:
        print("Aviso: A tabela %s já existe." % self.tb_name)
        return False

    print("Tabela %s criada com sucesso." % self.tb_name)

    ...

if __name__ == '__main__':
    c = ClientesDb()
    c.criar_schema()
```

Aqui nós criamos a função `criar_schema(self, schema_name)` dentro da classe `ClientesDb()`.

Com `with open(name)` abrimos o arquivo clientes_schema.sql .

Com `f.read()` lemos as linhas do arquivo.

E com cursor.executescript() executamos a instrução sql que está dentro do arquivo.

Modo interativo...

```
> python3
>>> from manager_db import *
>>> c = ClientesDb()
>>> c.criar_schema()
Criando tabela clientes ...
Tabela clientes criada com sucesso.
```

Se você digitar no terminal...

```
> sqlite3 clientes.db .tables
```

Você verá que a tabela foi criada com sucesso.

Create - Inserindo um registro com comando SQL

A função a seguir insere um registro na tabela. Repare no uso do comando `self.db.commit_db()` que grava de fato os dados.

```
def inserir_um_registro(self):
    try:
        self.db.cursor.execute("""
        INSERT INTO clientes (nome, idade, cpf, email,
        VALUES ('Regis da Silva', 35, '12345678901',
'regis@email.com', '(11) 9876-5342',
        'São Paulo', 'SP', '2014-07-30 11:23:00.199000')
        """)
        # gravando no bd
```

```
        self.db.commit_db()
        print("Um registro inserido com sucesso.")
    except sqlite3.IntegrityError:
        print("Aviso: O email deve ser único.")
        return False

    ...

if __name__ == '__main__':
    c = ClientesDb()
    c.criar_schema()
    c.inserir_um_registro()
```

Inserindo n registros com uma lista de dados

A função a seguir insere vários registros a partir de uma lista. Repare no uso do comando executemany(sql, [parâmetros])

```
self.db.cursor.executemany("""INSERT INTO tabela (campos) VALUES (?)""",
lista)
```

que executa a instrução sql várias vezes. Note também, pela sintaxe, que a quantidade de ? deve ser igual a quantidade de campos, e o parâmetro, no caso está sendo a lista criada.

```
def inserir_com_lista(self):
    # criando uma lista de dados
    lista = [('Agenor de Sousa', 23, '12345678901', 'agenor@email.com',
              '(10) 8300-0000', 'Salvador', 'BA', '2014-07-29
11:23:01.199001'),
             ('Bianca Antunes', 21, '12345678902', 'bianca@email.com',
              '(10) 8350-0001', 'Fortaleza', 'CE', '2014-07-28
11:23:02.199002'),
             ('Carla Ribeiro', 30, '12345678903', 'carla@email.com',
              '(10) 8377-0002', 'Campinas', 'SP', '2014-07-28
11:23:03.199003'),
             ('Fabiana de Almeida', 25, '12345678904', 'fabiana@email.com',
              '(10) 8388-0003', 'São Paulo', 'SP', '2014-07-29
11:23:04.199004'),
             ]
    try:
        self.db.cursor.executemany("""
        INSERT INTO clientes (nome, idade, cpf, email, fone, cidade, uf,
criado_em)
        VALUES (?,?,?,?,?,?,?,?)
        """, lista)
        # gravando no bd
        self.db.commit_db()
        print("Dados inseridos da lista com sucesso: %s registros." %
              len(lista))
    except sqlite3.IntegrityError:
        print("Aviso: O email deve ser único.")
        return False
```

Inserindo registros de um arquivo externo

Também podemos escrever as instruções sql num arquivo externo (clientes_dados.sql) e executá-lo com o comando `executescript(sql_script)`. Note que as instruções a seguir já foram vistas anteriormente.

```
def inserir_de_arquivo(self):
    try:
        with open('sql/clientes_dados.sql', 'rt') as f:
            dados = f.read()
            self.db.cursor.executescript(dados)
            # gravando no bd
            self.db.commit_db()
            print("Dados inseridos do arquivo com sucesso.")
    except sqlite3.IntegrityError:
        print("Aviso: O email deve ser único.")
        return False
```

Importando dados de um arquivo csv

Agora vamos importar os dados de clientes.csv . A única novidade é o comando csv.reader() .

```
import csv
...

def inserir_de_csv(self, file_name='csv/clientes.csv'):
    try:
        reader = csv.reader(
            open(file_name, 'rt'), delimiter=',')
        linha = (reader,)
        for linha in reader:
            self.db.cursor.execute("""
            INSERT INTO clientes (nome, idade, cpf, email, fone, cidade, uf,
criado_em)
            VALUES (?,?,?,?,?,?,?,?)
            """, linha)
        # gravando no bd
        self.db.commit_db()
        print("Dados importados do csv com sucesso.")
    except sqlite3.IntegrityError:
        print("Aviso: O email deve ser único.")
        return False
```

Obs: Veja em gen_csv.py como podemos gerar dados randômicos para criar um novo clientes.csv.

Inserindo um registro com parâmetros de entrada definido pelo usuário

Agora está começando a ficar mais interessante. Quando falamos *parâmetros de entrada* significa interação direta do usuário na aplicação. Ou seja, vamos inserir os dados diretamente pelo terminal em tempo de execução. Para isso nós usamos o comando `input()` para Python 3 ou `raw_input()` para Python 2.

```
def inserir_com_parametros(self):
    # solicitando os dados ao usuário
    self.nome = input('Nome: ')
    self.idade = input('Idade: ')
    self.cpf = input('CPF: ')
    self.email = input('Email: ')
```

```
    self.fone = input('Fone: ')
    self.cidade = input('Cidade: ')
    self.uf = input('UF: ') or 'SP'
    date = datetime.datetime.now().isoformat(" ")
    self.criado_em = input('Criado em (%s): ' % date) or date

    try:
        self.db.cursor.execute("""
        INSERT INTO clientes (nome, idade, cpf, email, fone, cidade, uf,
criado_em)
        VALUES (?,?,?,?,?,?,?,?)
        """, (self.nome, self.idade, self.cpf, self.email, self.fone,
              self.cidade, self.uf, self.criado_em))
        # gravando no bd
        self.db.commit_db()
        print("Dados inseridos com sucesso.")
    except sqlite3.IntegrityError:
        print("Aviso: O email deve ser único.")
        return False
```

Note que, em `criado_em` se você não informar uma data ele insere a data atual. E os parâmetros informados são passados no final do comando `execute()`.

Veja a interação:

```
> python3
>>> from manager_db import *
>>> c = ClientesDb()
>>> c.criar_schema()
>>> c.inserir_com_parametros()
Nome: Regis
Idade: 35
CPF: 11100011100
Email: regis@email.com
Fone: (11) 1111-1111
Cidade: São Paulo
UF: SP
Criado em (2014-10-07 01:40:48.836683):
Dados inseridos com sucesso.
```

Inserindo valores randômicos

Se lembra de gen_random_values.py? Agora vamos usar ele.

Para preencher *criado_em* usamos a data atual `.now()`.

Para gerar o *nome* usamos a função `names.get_first_name()` e `names.get_last_name()`.

Para o *email* pegamos a primeira letra do nome e o sobrenome + `@email.com`, ou seja, o formato r.silva@email.com, por exemplo.

Para a *cidade* e *uf* usamos a função `gen_city()` retornando os dois elementos de `list_city`.

O `repeat` é 10 por padrão, mas você pode mudar, exemplo `inserir_randomico(15)` na chamada da função.

```
def inserir_randomico(self, repeat=10):
    ''' Inserir registros com valores randomicos names '''
    lista = []
    for _ in range(repeat):
        date = datetime.datetime.now().isoformat(" ")
        fname = names.get_first_name()
        lname = names.get_last_name()
        name = fname + ' ' + lname
        email = fname[0].lower() + '.' + lname.lower() + '@email.com'
        c = gen_city()
        city = c[0]
        uf = c[1]
        lista.append((name, gen_age(), gen_cpf(),
                      email, gen_phone(),
                      city, uf, date))
    try:
        self.db.cursor.executemany("""
        INSERT INTO clientes (nome, idade, cpf, email, fone, cidade, uf,
criado_em)
        VALUES (?,?,?,?,?,?,?,?)
        """, lista)
        self.db.commit_db()
        print("Inserindo %s registros na tabela..." % repeat)
        print("Registros criados com sucesso.")
    except sqlite3.IntegrityError:
        print("Aviso: O email deve ser único.")
        return False
```

Read - Lendo os dados

Eu preferi fazer duas funções `ler_todos_clientes()` e `imprimir_todos_clientes()`. A primeira apenas retorna os valores com o comando `fetchall()`, pois eu irei usá-lo mais vezes. E a segunda imprime os valores na tela. No caso, eu usei uma tabulação mais bonitinha...

```
def ler_todos_clientes(self):
    sql = 'SELECT * FROM clientes ORDER BY nome'
    r = self.db.cursor.execute(sql)
    return r.fetchall()

def imprimir_todos_clientes(self):
    lista = self.ler_todos_clientes()
    print('{:>3s} {:20s} {:<5s} {:15s} {:21s} {:14s} {:15s} {:s}
{:s}'.format(
        'id', 'nome', 'idade', 'cpf', 'email', 'fone', 'cidade', 'uf',
'criado_em'))
    for c in lista:
        print('{:3d} {:23s} {:2d} {:s} {:>25s} {:s} {:15s} {:s}
{:s}'.format(
            c[0], c[1], c[2],
            c[3], c[4], c[5],
            c[6], c[7], c[8]))
```

mas se quiser você pode usar simplesmente

```
def imprimir_todos_clientes(self):
```

```
    lista = self.ler_todos_clientes()
    for c in lista:
        print(c)
```

Mais SELECT

Exemplo: Vamos explorar um pouco mais o `SELECT`. Veja a seguir como localizar um cliente pelo `id`. Uma *sutileza* é a vírgula logo depois do `id`, isto é necessário porque quando usamos a `?` é esperado que os parâmetros sejam uma tupla.

```
def localizar_cliente(self, id):
    r = self.db.cursor.execute(
        'SELECT * FROM clientes WHERE id = ?', (id,))
    return r.fetchone()

def imprimir_cliente(self, id):
    if self.localizar_cliente(id) == None:
        print('Não existe cliente com o id informado.')
    else:
        print(self.localizar_cliente(id))
```

O `fetchone()` retorna apenas uma linha de registro.

Exemplo: Veja um exemplo de como contar os registros.

```
def contar_cliente(self):
    r = self.db.cursor.execute(
        'SELECT COUNT(*) FROM clientes')
    print("Total de clientes:", r.fetchone()[0])
```

Exemplo: Contar os clientes maiores que 50 anos de idade. Veja novamente a necessidade da vírgula em `(t,)`.

```
def contar_cliente_por_idade(self, t=50):
    r = self.db.cursor.execute(
        'SELECT COUNT(*) FROM clientes WHERE idade > ?', (t,))
    print("Clientes maiores que", t, "anos:", r.fetchone()[0])
```

Caso queira outra idade mude o valor ao chamar a função:

```
c.contar_cliente_por_idade(18)
```

Exemplo: Localizar clientes por idade.

```
def localizar_cliente_por_idade(self, t=50):
    resultado = self.db.cursor.execute(
        'SELECT * FROM clientes WHERE idade > ?', (t,))
    print("Clientes maiores que", t, "anos:")
    for cliente in resultado.fetchall():
        print(cliente)
```

Exemplo: Localizar clientes por uf.

```
def localizar_cliente_por_uf(self, t='SP'):
    resultado = self.db.cursor.execute(
```

```
        'SELECT * FROM clientes WHERE uf = ?', (t,))
    print("Clientes do estado de", t, ":")
    for cliente in resultado.fetchall():
        print(cliente)
```

SELECT personalizado

Exemplo: Vejamos agora como fazer nosso próprio `SELECT`.

```
def meu_select(self, sql="SELECT * FROM clientes WHERE uf='RJ';"):
    r = self.db.cursor.execute(sql)
    # gravando no bd
    self.db.commit_db()
    for cliente in r.fetchall():
        print(cliente)
```

Assim, podemos escrever qualquer `SELECT` direto na chamada da função:

```
c.meu_select("SELECT * FROM clientes WHERE uf='MG' ORDER BY nome;")
```

Acabamos de mudar a função original. Eu coloquei o `commit_db()` porque se quiser você pode escrever uma instrução SQL com `INSERT` ou `UPDATE`, por exemplo.

Exemplo: Lendo instruções de arquivos externos

No arquivo clientes_sp.sql eu escrevi várias instruções SQL.

```
SELECT * FROM clientes WHERE uf='SP';
SELECT COUNT(*) FROM clientes WHERE uf='SP';
SELECT * FROM clientes WHERE uf='RJ';
SELECT COUNT(*) FROM clientes WHERE uf='RJ';
```

Para que todas as instruções sejam lidas e retorne valores é necessário que usemos os comandos `split(';')` para informar ao interpretador qual é o final de cada linha. E o comando `execute()` dentro de um `for`, assim ele lê e executa todas as instruções SQL do arquivo.

```
def ler_arquivo(self, file_name='sql/clientes_sp.sql'):
    with open(file_name, 'rt') as f:
        dados = f.read()
        sqlcomandos = dados.split(';')
        print("Consulta feita a partir de arquivo externo.")
        for comando in sqlcomandos:
            r = self.db.cursor.execute(comando)
            for c in r.fetchall():
                print(c)
    # gravando no bd
    self.db.commit_db()
```

Novamente você pode usar qualquer instrução SQL porque o `commit_db()` já está ai.

```
c.ler_arquivo('sql/clientes_maior60.sql')
```

Update - Alterando os dados

Nenhuma novidade, todos os comandos já foram vistos antes. No caso, informamos o `id` do cliente. Veja que aqui usamos novamente a função `localizar_cliente(id)` para localizar o cliente.

```
def atualizar(self, id):
    try:
        c = self.localizar_cliente(id)
        if c:
            # solicitando os dados ao usuário
            # se for no python2.x digite entre aspas simples
            self.novo_fone = input('Fone: ')
            self.db.cursor.execute("""
            UPDATE clientes
            SET fone = ?
            WHERE id = ?
            """, (self.novo_fone, id,))
            # gravando no bd
            self.db.commit_db()
            print("Dados atualizados com sucesso.")
        else:
            print('Não existe cliente com o id informado.')
    except e:
        raise e
```

Chamando a função:

```
c.atualizar(10)
```

Delete - Deletando os dados

Novamente vamos localizar o cliente para depois deletá-lo.

```
def deletar(self, id):
    try:
        c = self.localizar_cliente(id)
        # verificando se existe cliente com o ID passado, caso exista
        if c:
            self.db.cursor.execute("""
            DELETE FROM clientes WHERE id = ?
            """, (id,))
            # gravando no bd
            self.db.commit_db()
            print("Registro %d excluído com sucesso." % id)
        else:
            print('Não existe cliente com o código informado.')
    except e:
        raise e
```

Chamando a função:

```
c.deletar(10)
```

Adicionando uma nova coluna

Para adicionar uma nova coluna é bem simples.

```
def alterar_tabela(self):
    try:
        self.db.cursor.execute("""
        ALTER TABLE clientes
        ADD COLUMN bloqueado BOOLEAN;
        """)
        # gravando no bd
        self.db.commit_db()
        print("Novo campo adicionado com sucesso.")
    except sqlite3.OperationalError:
        print("Aviso: O campo 'bloqueado' já existe.")
        return False
```

Lendo as informações do banco de dados

Obtendo informações da tabela

```
def table_info(self):
    t = self.db.cursor.execute(
        'PRAGMA table_info({})'.format(self.tb_name))
    colunas = [tupla[1] for tupla in t.fetchall()]
    print('Colunas:', colunas)
```

Chamando e vendo o resultado:

```
>>> c.table_info()
Colunas: ['id', 'nome', 'idade', 'cpf', 'email', 'fone', 'cidade', 'uf',
'criado_em']
```

Listando as tabelas do bd

```
def table_list(self):
    l = self.db.cursor.execute("""
    SELECT name FROM sqlite_master WHERE type='table' ORDER BY name
    """)
    print('Tabelas:')
    for tabela in l.fetchall():
        print("%s" % (tabela))
```

Chamando e vendo o resultado:

```
>>> c.table_list()
Tabelas:
clientes
sqlite_sequence
```

Obtendo o schema da tabela

```
def table_schema(self):
    s = self.db.cursor.execute("""
    SELECT sql FROM sqlite_master WHERE type='table' AND name=?
    """, (self.tb_name,))

    print('Schema:')
```

```
    for schema in s.fetchall():
        print("%s" % (schema))
```

Chamando e vendo o resultado:

```
>>> c.table_schema()
Schema:
CREATE TABLE clientes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    nome TEXT NOT NULL,
    idade INTEGER,
    cpf VARCHAR(11) NOT NULL,
    email TEXT NOT NULL UNIQUE,
    fone TEXT,
    cidade TEXT,
    uf VARCHAR(2) NOT NULL,
    criado_em DATETIME NOT NULL
)
```

Fazendo backup do banco de dados (exportando dados)

```
import io
...
def backup(self, file_name='sql/clientes_bkp.sql'):
    with io.open(file_name, 'w') as f:
        for linha in self.db.conn.iterdump():
            f.write('%s\n' % linha)

    print('Backup realizado com sucesso.')
    print('Salvo como %s' % file_name)
```

Se quiser pode salvar com outro nome.

```
c.backup('sql/clientes_backup.sql')
```

Recuperando o banco de dados (importando dados)

Aqui nós usamos dois parâmetros: `db_name` para o banco de dados recuperado (no caso, um banco novo) e `file_name` para o nome do arquivo de backup com as instruções SQL salvas.

```
def importar_dados(self, db_name='clientes_recovery.db',
file_name='sql/clientes_bkp.sql'):
    try:
        self.db = Connect(db_name)
        f = io.open(file_name, 'r')
        sql = f.read()
        self.db.cursor.executescript(sql)
        print('Banco de dados recuperado com sucesso.')
        print('Salvo como %s' % db_name)
    except sqlite3.OperationalError:
        print(
```

```
            "Aviso: O banco de dados %s já existe. Exclua-o e faça
novamente." %
            db_name)
        return False
```

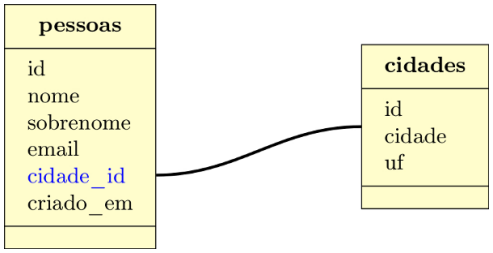
Fechando conexão:

```
def fechar_conexao(self):
    self.db.close_db()
```

Relacionando tabelas

Agora, no mesmo arquivo manager_db.py vamos criar uma outra instância chamada `PessoasDb()`. Neste exemplo vamos relacionar duas tabelas: `pessoas` e `cidades`.

Veja na figura a seguir como as tabelas se relacionam.



Agora os códigos:

```
class PessoasDb(object):

    tb_name = 'pessoas'

    def __init__(self):
        self.db = Connect('pessoas.db')
        self.tb_name
```

Criando o *schema* a partir de pessoas_schema.sql.

```
def criar_schema(self, schema_name='sql/pessoas_schema.sql'):
    print("Criando tabela %s ..." % self.tb_name)

    try:
        with open(schema_name, 'rt') as f:
            schema = f.read()
            self.db.cursor.executescript(schema)
    except sqlite3.Error:
        print("Aviso: A tabela %s já existe." % self.tb_name)
        return False

    print("Tabela %s criada com sucesso." % self.tb_name)
```

Inserindo as cidades a partir de cidades.csv.

```
def inserir_de_csv(self, file_name='csv/cidades.csv'):
    try:
```

```
        c = csv.reader(
            open(file_name, 'rt'), delimiter=',')
        t = (c,)
        for t in c:
            self.db.cursor.execute("""
            INSERT INTO cidades (cidade, uf)
            VALUES (?,?)
            """, t)
        # gravando no bd
        self.db.commit_db()
        print("Dados importados do csv com sucesso.")
    except sqlite3.IntegrityError:
        print("Aviso: A cidade deve ser única.")
        return False
```

Agora vamos contar quantas cidades temos na tabela...

```
def gen_cidade(self):
    ''' conta quantas cidades estão cadastradas e escolhe uma delas pelo id.
'''
    sql = 'SELECT COUNT(*) FROM cidades'
    q = self.db.cursor.execute(sql)
    return q.fetchone()[0]
```

para a partir dai gerar valores randômicos apenas com as cidades existentes.

```
def inserir_randomico(self, repeat=10):
    lista = []
    for _ in range(repeat):
        fname = names.get_first_name()
        lname = names.get_last_name()
        email = fname[0].lower() + '.' + lname.lower() + '@email.com'
        cidade_id = random.randint(1, self.gen_cidade())
        lista.append((fname, lname, email, cidade_id))
    try:
        self.db.cursor.executemany("""
        INSERT INTO pessoas (nome, sobrenome, email, cidade_id)
        VALUES (?,?,?,?)
        """, lista)
        self.db.commit_db()
        print("Inserindo %s registros na tabela..." % repeat)
        print("Registros criados com sucesso.")
    except sqlite3.IntegrityError:
        print("Aviso: O email deve ser único.")
        return False
```

Agora é só alegria!

```
def ler_todas_pessoas(self):
    sql = 'SELECT * FROM pessoas INNER JOIN cidades ON pessoas.cidade_id =
cidades.id'
    r = self.db.cursor.execute(sql)
    return r.fetchall()

def imprimir_todas_pessoas(self):
    lista = self.ler_todas_pessoas()
    for c in lista:
        print(c)
```

```
# myselect, imprime todos os nomes que começam com R
def meu_select(self, sql="SELECT * FROM pessoas WHERE nome LIKE 'R%' ORDER
BY nome;"):
    r = self.db.cursor.execute(sql)
    self.db.commit_db()
    print('Nomes que começam com R:')
    for c in r.fetchall():
        print(c)

def table_list(self):
    # listando as tabelas do bd
    l = self.db.cursor.execute("""
    SELECT name FROM sqlite_master WHERE type='table' ORDER BY name
    """)
    print('Tabelas:')
    for tabela in l.fetchall():
        print("%s" % (tabela))

def fechar_conexao(self):
    self.db.close_db()
```

Chamando tudo no **modo interativo**

```
>>> from manager_db import *
>>> p = PessoasDb()
>>> p.criar_schema()
>>> p.inserir_de_csv()
>>> p.gen_cidade()
>>> p.inserir_randomico(100)
>>> p.imprimir_todas_pessoas()
>>> p.meu_select()
>>> p.table_list()
>>> p.fechar_conexao()
```

Referências

1. sqlite3 — DB-API 2.0 interface for SQLite databases sqlite3 Embedded Relational Database Lets Talk to a SQLite Database with Python Advanced SQLite Usage in Python Python A Simple Step by Step SQLite Tutorial
2. sqlite3 Embedded Relational Database
3. Lets Talk to a SQLite Database with Python
4. Advanced SQLite Usage in Python
5. Python A Simple Step by Step SQLite Tutorial
6. Python docs, SQLite, Connection Objects