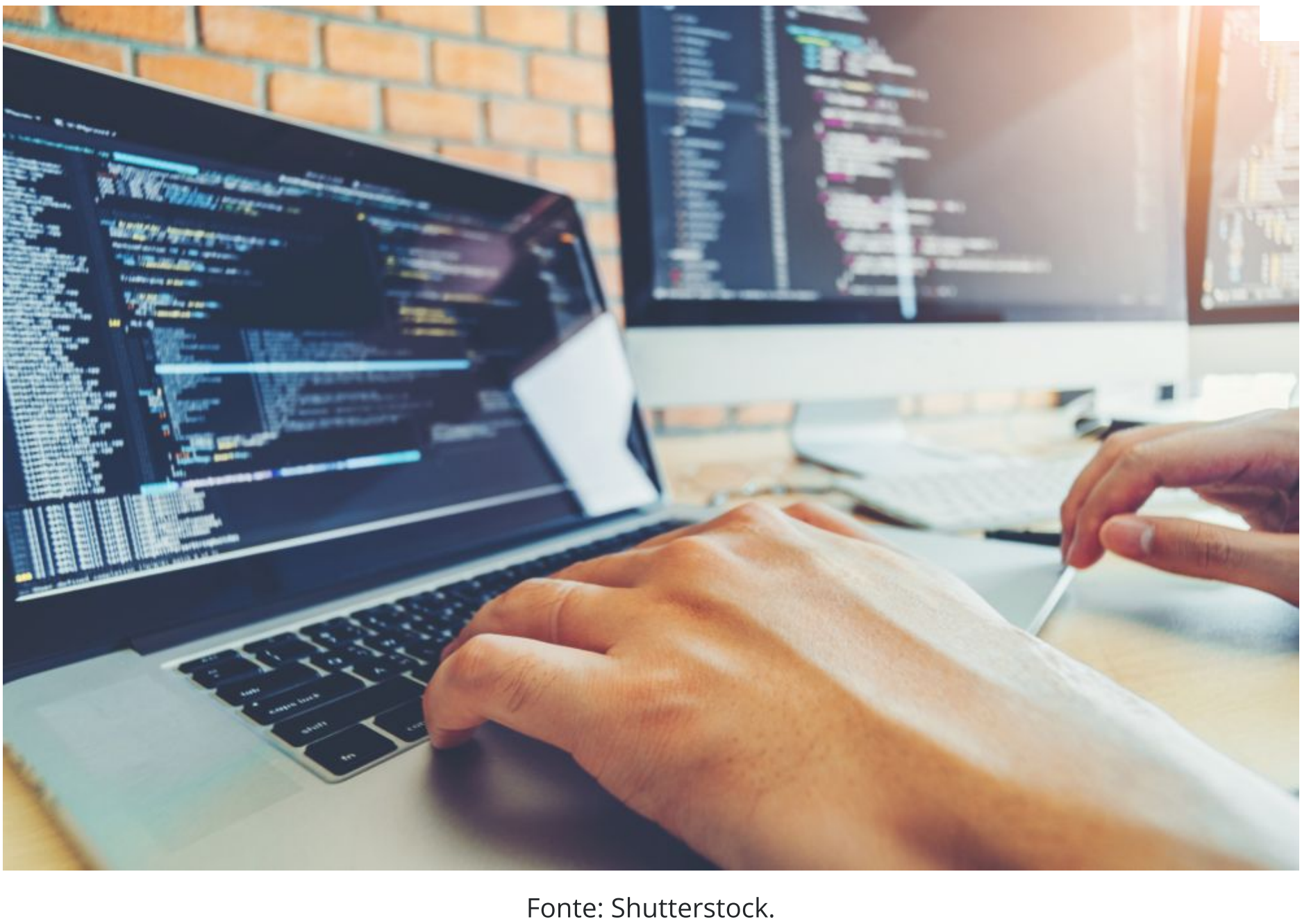
NÃO PODE FALTAR

APLICAÇÃO DE BANCO DE DADOS COM PYTHON

Vanessa Cadan Scheffer

LINGUAGEM DE CONSULTA ESTRUTURADA - SQL

O SQL é a linguagem que permite aos usuários se comunicarem com banco de dados relacionais.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

INTRODUÇÃO A BANCO DE DADOS

Grande parte dos softwares que são desenvolvidos (se não todos) acessa algum tipo de mecanismo para armazenar dados. Podem ser dados da aplicação, como cadastro de clientes, ou então dados sobre a execução da solução, os famosos

logs. Esses dados podem ser armazenados em arquivos, cenário no qual se destacam os arquivos delimitados, com extensão CSV (*comma separated values*), os arquivos JSON (*JavaScript Object Notation*) e os arquivos XML (*Extensible Markup Language*). Outra opção para persistir os dados é utilizar um sistema de banco de dados.

Segundo Date (2003), um sistema de banco de dados é basicamente apenas um sistema computadorizado de persistência de registros. O próprio banco de dados pode ser considerado como um repositório para uma coleção de dados computadorizados. Os sistema de banco de dados podem ser divididos em duas categorias: banco de dados relacional e banco de dados NoSQL.

A teoria base dos bancos de dados relacional existe desde a década de 1970 (MACHADO, 2014). Nessa abordagem, os dados são persistidos em uma estrutura bidimensional, chamada de relação (que é uma tabela), que está baseada na teoria dos conjuntos pertencentes à matemática. Cada unidade de dados é conhecida como coluna, ao passo que cada unidade do grupo é conhecida como linha, tupla ou registro.

Com o surgimento de grandes aplicações web e de outras tecnologias como o IoT, o volume de dados que trafegam na rede e são processados aumentou consideravelmente, o que pode ser um desafio em termos de performance para os bancos relacionais. Além do volume, o formato dos dados também é desafiador para a persistência nessa tecnologia, uma vez que eles são persistidos em linhas e colunas.

Como armazenar fotos, vídeos, e outros formatos? Para suprir essa nova demanda, pesquisadores se dedicaram a buscar soluções para o manuseio de dados em grande escala e em outros formatos não estruturados, obtendo, como um dos resultados, o banco de dados não relacional, que geralmente é referenciado como NoSQL (VAISH, 2013). Existe uma discussão sobre o significado de NoSQL: em algumas literaturas, ele é tratado como *not only SQL* (não somente SQL), o que remete à possibilidade de existir também o SQL nessa tecnologia. Mas, segundo

Vaish (2013), originalmente, NoSQL era a combinação das palavras *No* e *SQL*, que literalmente dizia respeito a não usar a linguagem SQL (*structured query language* - linguagem de consulta estruturada) para acessar e manipular dados em sistemas gerenciadores de banco de dados. Qualquer que seja a origem do termo, hoje o NoSQL é usado para abordar a classe de bancos de dados que não seguem os princípios do sistema de gerenciamento de banco de dados relacional (RDBMS) e são projetados especificamente para lidar com a velocidade e a escala de aplicações como Google, Facebook, Yahoo, Twitter, dentre outros.

LINGUAGEM DE CONSULTA ESTRUTURADA - SQL

Para se comunicar com um banco de dados relacional, existe uma linguagem específica conhecida como SQL, que, tal como dito, significa *structured query language* ou, traduzindo, *linguagem de consulta estruturada*. Em outras palavras, SQL é a linguagem que permite aos usuários se comunicarem com banco de dados relacionais (ROCKOFF, 2016). Em 1986, o American National Standards Institute (ANSI) publicou seu primeiro conjunto de padrões em relação à linguagem SQL e, desde então, passou por várias revisões. Algumas empresas de software para banco de dados, como Oracle e Microsoft, adaptaram a linguagem SQL adicionando inúmeras extensões e modificações à linguagem padrão. Mas, embora cada fornecedor tenha sua própria interpretação exclusiva do SQL, ainda existe uma linguagem base que é padrão a todos fornecedores.

As instruções da linguagem SQL são divididas em três grupos: DDL, DML, DCL (ROCKOFF, 2016), descritos a seguir.

- **DDL** é um acrônimo para *Data Definition Language* (linguagem de definição de dados). Fazem parte deste grupo as instruções destinadas a criar, deletar e modificar banco de dados e tabelas. Neste módulo vão aparecer comandos como CREATE, o ALTER e o DROP.

- **DML** é um acrônimo para *Data Manipulation Language* (linguagem de manipulação de dados). Fazem parte deste grupo as instruções destinadas a recuperar, atualizar, adicionar ou excluir dados em um banco de dados. Neste módulo vão aparecer comandos como INSERT, UPDATE e DELETE.
- **DCL** é um acrônimo para *Data Control Language* (linguagem de controle de dados). Fazem parte deste grupo as instruções destinadas a manter a segurança adequada para o banco de dados. Neste módulo vão aparecer comandos como GRANT e REVOKE.

DICA

Como sugestão de literatura para noções de SQL, uma opção é o *Guia mangá de bancos de dados* (TAKAHASHI; AZUMA, 2009).

BANCO DE DADOS RELACIONAL

CONEXÃO COM BANCO DE DADOS RELACIONAL

Ao criar uma aplicação em uma linguagem de programação que precisa acessar um sistema gerenciador de banco de dados relacional (RDBMS), uma vez que são processos distintos, é preciso criar uma conexão entre eles. Após estabelecida a conexão, é possível (de alguma forma) enviar comandos SQL para efetuar as ações no banco (RAMAKRISHNAN; GEHRKE, 2003). Para fazer a conexão e permitir que uma linguagem de programação se comunique com um banco de dados com a utilização da linguagem SQL, podemos usar as tecnologias ODBC (Open Database Connectivity) e JDBC (Java Database Connectivity).

> Ambos, ODBC e JDBC, expõem os recursos de banco de dados de uma forma padronizada ao programador de aplicativo através de uma interface de programação de aplicativo (API — application programming interface)
> — RAMAKRISHNAN; GEHRKE, 2003, p. 162.

A grande vantagem de utilizar as tecnologias ODBC ou JDBC está no fato de que uma aplicação pode acessar diferentes RDBMS sem precisar recompilar o código. Essa transparência é possível porque a comunicação direta com o RDBMS é feita por um driver. Um driver é um software específico responsável por traduzir as chamadas ODBC e JDBC para a linguagem do RDBMS.

O JDBC é uma API padrão em Java, inicialmente desenvolvida pela Sun Microsystems (MENON, 2005). Em outras palavras, JDBC é um conjunto de classes desenvolvidas em Java que abstraem a conexão com um RDBMS. Cada fornecedor de RDBMS, como Oracle e Microsoft, constrói e distribui, gratuitamente, um driver JDBC. Diferentes RDBMS necessitam de diferentes drivers para comunicação; por exemplo, em uma aplicação que se conecta a três RDBMS distintos, serão necessários três drivers distintos. ODBC também é uma API padronizada para conexão com os diversos RDBMS (MICROSOFT, 2017). As funções na API ODBC são implementadas por desenvolvedores de drivers específicos do RDBMS e, para utilizá-las, você deve configurar uma entrada nas propriedades do sistema.

CONEXÃO DE BANCO DADOS SQL EM PYTHON

Nesta aula, vamos explorar como utilizar um banco de dados relacional em Python. Agora que já sabemos que para acessar esse tipo de tecnologia precisamos de um mecanismo de conexão (ODBC ou JDBC) e uma linguagem para nos comunicarmos com ele (SQL), vamos ver como atuar em Python.

Para se comunicar com um RDBMS em Python, podemos utilizar bibliotecas já disponíveis, com uso das quais, por meio do driver de um determinado fornecedor, será possível fazer a conexão e a execução de comandos SQL no banco. Por exemplo, para se conectar com um banco de dados Oracle, podemos usar a biblioteca cx-Oracle, ou, para se conectar a um PostgreSQL, temos como opção o psycopg2. Visando à padronização entre todos os módulos de conexão com um RDBMS e o envio de comandos, o **PEP 249 (Python Database API Specification**

v2.0) elenca um conjunto de regras que os fornecedores devem seguir na construção de módulos para acesso a banco de dados. Por exemplo, a documentação diz que todos módulos devem implementar o método `connect(parameters...)` para se conectar a um banco. Veja que, dessa forma, caso seja necessário alterar o banco de dados, somente os parâmetros mudam, não o código.

Agora que já temos essa visão geral, vamos explorar a implementação em Python usando um mecanismo de banco de dados SQL chamado SQLite.

Ver anotações

BANCO DE DADOS SQLITE

> "O SQLite é uma biblioteca em linguagem C, que implementa um mecanismo de banco de dados SQL pequeno, rápido, independente, de alta confiabilidade e completo"
>
> — SQLITE, 2020, [s.p.], tradução nossa.

Essa tecnologia pode ser embutida em telefones celulares e computadores e vem incluída em inúmeros outros aplicativos que as pessoas usam todos os dias. Ao passo que a maioria dos bancos de dados SQL usa um servidor para rodar e gerenciar, o SQLite não possui um processo de servidor separado. O SQLite lê e grava diretamente em arquivos de disco, ou seja, um banco de dados SQL completo com várias tabelas, índices, triggers e visualizações está contido em um único arquivo de disco.

O interpretador Python possui o módulo *built-in sqlite3*, que permite utilizar o mecanismo de banco de dados SQLite.

> O módulo sqlite3 foi escrito por Gerhard Häring e fornece uma interface SQL compatível com a especificação DB-API 2.0 descrita pelo PEP 249.
>
> — PSF, 2020d, [s.p.], tradução nossa.

Para permear nosso estudo, vamos criar um banco de dados chamado *aulaDB,* no qual vamos criar a tabela fornecedor, conforme Quadro 3.1:

Quadro 3.1 | Tabela fornecedor

Campo	Tipo	Obrigatório
id_fornecedor	inteiro	sim
nome_fornecedor	texto	sim
cnpj	texto	sim
cidade	texto	não
estado	texto	sim
cep	texto	sim
data_cadastro	data	sim

Fonte: elaborado pelo autora.

Ver anotações 0

CRIANDO UM BANCO DE DADOS

O primeiro passo é importar o módulo *sqlite3*. Como o módulo está baseado na especificação DB-API 2.0 descrita pelo PEP 249, ele utiliza o método *connnect()* para se conectar a um banco de dados. Em razão da natureza do SQLite (ser um arquivo no disco rígido), ao nos conectarmos a um banco, o arquivo é imediatamente criado na pasta do projeto (se estiver usando o projeto Anaconda, o arquivo é criado na mesma pasta em que está o Jupyter Notebook). Se desejar criar o arquivo em outra pasta, basta especificar o caminho juntamente com o nome, por exemplo: C:/Users/Documents/meu_projeto/meus_bancos/bancoDB.db. Observe do código a seguir.

In [1]:

```
import sqlite3


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

print(type(conn))
```

```
<class 'sqlite3.Connection'>
```

Ver anotações 0

Ao executar o código da entrada 1, o arquivo é criado, e a variável "conn" agora é um objeto da classe *Connection* pertencente ao módulo *sqlite3*.

CRIANDO UMA TABELA

Agora que temos uma conexão com um banco de dados, vamos utilizar uma instrução DDL da linguagem SQL para criar a tabela fornecedor. O comando SQL que cria a tabela fornecedor está no código a seguir e foi guardado em uma variável chamada *ddl_create*.

Observação: se tentar criar uma tabela que já existe, um erro é retornado. Caso execute todas as células novamente, certifique-se de apagar a tabela no banco, para evitar o erro.

In [2]:

```
ddl_create = """
CREATE TABLE fornecedor (
    id_fornecedor INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    nome_fornecedor TEXT NOT NULL,
    cnpj VARCHAR(18) NOT NULL,
    cidade TEXT,
    estado VARCHAR(2) NOT NULL,
    cep VARCHAR(9) NOT NULL,
    data_cadastro DATE NOT NULL
);
"""
```

O comando DDL implementado na entrada 2 faz parte do conjunto de instruções SQL, razão pela qual deve seguir a sintaxe que essa linguagem determina.

> **SAIBA MAIS**
>
> Caso queira saber mais sobre o comando CREATE, recomendamos a leitura das páginas 49 a 52 da seguinte obra, disponível na biblioteca virtual. RAMAKRISHNAN, R.; GEHRKE, J. **Sistemas de gerenciamento de banco de dados**. 3. ed. Porto Alegre : AMGH, 2003.

Podemos notar o padrão na instrução, que começa com o comando CREATE TABLE, seguido do nome da tabela a ser criada e, entre parênteses, o nome do campo, o tipo e a especificação de quando não são aceitos valores nulos. O primeiro campo possui uma instrução adicional, que é o autoincremento, ou seja, para cada novo registro inserido, o valor desse campo aumentará um.

Já temos a conexão e a DDL. Agora basta utilizar um mecanismo para que esse comando seja executado no banco. Esse mecanismo, segundo o PEP 249, deve estar implementado em um método chamado `execute()` de um objeto **cursor**. Os cursores desempenham o papel de pontes entre os conjuntos fornecidos como respostas das consultas e as linguagens de programação que não suportam conjuntos (RAMAKRISHNAN; GEHRKE, 2003). Portanto, sempre que precisarmos executar um comando SQL no banco usando a linguagem Python, usaremos um cursor para construir essa ponte. Observe o código a seguir.

In [3]:

```
cursor = conn.cursor()
cursor.execute(ddl_create)
print(type(cursor))


print("Tabela criada!")
print("Descrição do cursor: ", cursor.description)
print("Linhas afetadas: ", cursor.rowcount)
cursor.close()
conn.close()
```

```
<class 'sqlite3.Cursor'>
Tabela criada!
Descrição do cursor:  None
Linhas afetadas:  -1
```

Na linha 1 da entrada 3, a partir da conexão, criamos um objeto cursor. Na linha 2, invocamos o método *execute()* desse objeto para, enfim, criar a tabela pelo comando armazenado na variável *ddl_create*. Como o cursor é uma classe, ele possui métodos e atributos. Nas linhas 6 e 7 estamos acessando os atributos *description* e *rowcount*. O primeiro diz respeito a informações sobre a execução; e o segundo a quantas linhas foram afetadas. No módulo *sqlite3*, o atributo *description* fornece os nomes das colunas da última consulta. Como se trata de uma instrução DDL, a description retornou None e a quantidade de linhas afetadas foi -1. Todo cursor e toda conexão, após executarem suas tarefas, devem ser fechados pelo método *close()*.

Segundo o PEP 249 (2020), todos os módulos devem implementar 7 campos para o resultado do atributo description: name, type_code, display_size, internal_size, precision, scale e null_ok (https://www.python.org/dev/peps/pep-0249/#description).

Além de criar uma tabela, também podemos excluí-la. A sintaxe para apagar uma tabela (e todos seus dados) é "DROP TABLE table_name".

CRUD - CREATE, READ, UPDATE, DELETE

CRUD é um acrônimo para as quatro operações de DML que podemos fazer em uma tabela no banco de dados. Podemos inserir informações (create), ler (read), atualizar (update) e apagar (delete). Os passos necessários para efetuar uma das operações do CRUD são sempre os mesmos: (i) estabelecer a conexão com um banco; (ii) criar um cursor e executar o comando; (iii) gravar a operação; (iv) fecha o cursor e a conexão.

CREATE

Vamos começar inserindo registros na tabela fornecedor. Observe o código a seguir.

In [4]:

```
# Só é preciso importar a biblioteca uma vez. Importamos novamente para manter
todo o código em uma única célula
import sqlite3

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

cursor.execute("""
INSERT INTO fornecedor (nome_fornecedor, cnpj, cidade, estado, cep,
data_cadastro)
VALUES ('Empresa A', '11.111.111/1111-11', 'São Paulo', 'SP', '11111-111',
'2020-01-01')
""")

cursor.execute("""
INSERT INTO fornecedor (nome_fornecedor, cnpj, cidade, estado, cep,
data_cadastro)
VALUES ('Empresa B', '22.222.222/2222-22', 'Rio de Janeiro', 'RJ', '22222-222',
'2020-01-01')
""")

cursor.execute("""
INSERT INTO fornecedor (nome_fornecedor, cnpj, cidade, estado, cep,
data_cadastro)
VALUES ('Empresa C', '33.333.333/3333-33', 'Curitiba', 'PR', '33333-333', '2020-
01-01')
""")

conn.commit()

print("Dados inseridos!")
print("Descrição do cursor: ", cursor.description)
print("Linhas afetadas: ", cursor.rowcount)
```

```
cursor.close()
conn.close()
```

```
Dados inseridos!
Descrição do cursor:  None
Linhas afetadas:  1
```

Ver anotações 0

Na entrada 4, fizemos a conexão e criamos um cursor (linhas 4 e 5). Através do cursor, inserimos 3 registros na tabela fornecedor. A sintaxe para a inserção exig que se passe os campos a serem inseridos e os valores. Veja que não passamos o campo *id_fornecedor*, pois este foi criado como autoincremento. Após a execução das três inserções, na linha 22, usamos o método `commit()` para gravar as alterações na tabela. Veja que a quantidade de linhas afetadas foi 1, pois mostra o resultado da última execução do cursor, que foi a inserção de 1 registro.

Uma maneira mais prática de inserir vários registros é passar uma lista de tuplas, na qual cada uma destas contém os dados a serem inseridos em uma linha. Nesse caso, teremos que usar o método `executemany()` do cursor. Observe o código a seguir.

In [5]:

```
# Só é preciso importar a biblioteca uma vez. Importamos novamente para manter
todo o código em uma única célula
import sqlite3

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

dados = [
    ('Empresa D', '44.444.444/4444-44', 'São Paulo', 'SP', '44444-444', '2020-
01-01'),
    ('Empresa E', '55.555.555/5555-55', 'São Paulo', 'SP', '55555-555', '2020-
01-01'),
    ('Empresa F', '66.666.666/6666-66', 'São Paulo', 'SP', '66666-666', '2020-
01-01')
]

cursor.executemany("""
INSERT INTO fornecedor (nome_fornecedor, cnpj, cidade, estado, cep,
data_cadastro)
VALUES (?, ?, ?, ?, ?, ?)""", dados)

conn.commit()

print("Dados inseridos!")
print("Descrição do cursor: ", cursor.description)
print("Linhas afetadas: ", cursor.rowcount)
cursor.close()
conn.close()
```

```
Dados inseridos!
Descrição do cursor:  None
Linhas afetadas:  3
```

Ver anotações 0

Na entrada 5, criamos uma lista de tuplas chamada *dados*. Na linha 13 invocamos o método *executemany()* para inserir a lista. Veja que agora os valores foram substituídos por interrogações, e, além da instrução SQL, o método exige a passagem dos dados. Veja que agora foram afetadas 3 linhas no banco, pois esse foi o resultado do método do cursor.

READ

Agora que temos dados na tabela fornecedor, podemos avançar para a segunda operação, que é recuperar os dados. Também precisamos estabelecer uma conexão e criar um objeto cursor para executar a instrução de seleção. Ao executar a seleção, podemos usar o método *fetchall()* para capturar todas as linhas, através de uma lista de tuplas. Observe o código a seguir.

In [6]:

```
# Só é preciso importar a biblioteca uma vez. Importamos novamente para manter
todo o código em uma única célula
import sqlite3


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


cursor.execute("SELECT * FROM fornecedor")
resultado = cursor.fetchall()


print("Descrição do cursor: ", cursor.description)
print("Linhas afetadas: ", cursor.rowcount)


resultado[:2]
```

```
Descrição do cursor:  (('id_fornecedor', None, None, None, None, None, None),
('nome_fornecedor', None, None, None, None, None, None), ('cnpj', None, None,
None, None, None, None), ('cidade', None, None, None, None, None, None),
('estado', None, None, None, None, None, None), ('cep', None, None, None, None
None, None), ('data_cadastro', None, None, None, None, None, None))
Linhas afetadas:  -1
```

Out[6]:

```
[(1,
  'Empresa A',
  '11.111.111/1111-11',
  'São Paulo',
  'SP',
  '11111-111',
  '2020-01-01'),
 (2,
  'Empresa B',
  '22.222.222/2222-22',
  'Rio de Janeiro',
  'RJ',
  '22222-222',
  '2020-01-01')]
```

In [7]:

```
for linha in resultado:
    print(linha)
```

Ver anotações

```
(1, 'Empresa A', '11.111.111/1111-11', 'São Paulo', 'SP', '11111-111', '2020-01-
01')
(2, 'Empresa B', '22.222.222/2222-22', 'Rio de Janeiro', 'RJ', '22222-222',
'2020-01-01')
(3, 'Empresa C', '33.333.333/3333-33', 'Curitiba', 'PR', '33333-333', '2020-01-
01')
(4, 'Empresa D', '44.444.444/4444-44', 'São Paulo', 'SP', '44444-444', '2020-01-
01')
(5, 'Empresa E', '55.555.555/5555-55', 'São Paulo', 'SP', '55555-555', '2020-01-
01')
(6, 'Empresa F', '66.666.666/6666-66', 'São Paulo', 'SP', '66666-666', '2020-01-
01')
```


Na entrada 6, usamos a instrução SQL "select * from fornecedor" para selecionar todos (*) os dados da tabela fornecedor. O comando é executado pelo cursor e, através do método *fetchall(),* guardamos o resultado na variável "resultado". O resultado do método é uma lista de tuplas. Para ficar claro, na linha 13 imprimimos uma fatia da lista. Outro detalhe interessante é o resultado do atributo *description*, que retornou tuplas, informando o nome da coluna afetada. Os outros 6 campos da tupla retornaram *None* graças à implementação do módulo *sqlite3* (PSF, 2020d).

Na entrada 7, usamos uma estrutura de decisão para iterar na lista e imprimir cada valor. Veja que cada linha é uma tupla com as informações que inserimos.

A linguagem SQL é muito poderosa para manipulação de dados. Podemos selecionar somente os registros que satisfaçam uma determinada condição usando a cláusula "where", que funciona como uma estrutura condicional. Observe o código a seguir, no qual selecionamos somente o registro cujo id_fornecedor é igual a 5.

In [8]:

```
cursor.execute("SELECT * FROM fornecedor WHERE id_fornecedor = 5")
resultado = cursor.fetchall()
print(resultado)


cursor.close()
conn.close()
```

```
[(5, 'Empresa E', '55.555.555/5555-55', 'São Paulo', 'SP', '55555-555', '2020-
01-01')]
```


UPDATE

Ao inserir um registro no banco, pode ser necessário alterar o valor de uma coluna, o que pode ser feito por meio da instrução SQL UPDATE. Observe o código a seguir.

In [9]:

```
# Só é preciso importar a biblioteca uma vez. Importamos novamente para manter
todo o código em uma única célula
import sqlite3


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


cursor.execute("UPDATE fornecedor SET cidade = 'Campinas' WHERE id_fornecedor =
5")
conn.commit()


cursor.execute("SELECT * FROM fornecedor")
for linha in cursor.fetchall():
    print(linha)


cursor.close()
conn.close()
```

```
(1, 'Empresa A', '11.111.111/1111-11', 'São Paulo', 'SP', '11111-111', '2020-01-
01')
(2, 'Empresa B', '22.222.222/2222-22', 'Rio de Janeiro', 'RJ', '22222-222',
'2020-01-01')
(3, 'Empresa C', '33.333.333/3333-33', 'Curitiba', 'PR', '33333-333', '2020-01-
01')
(4, 'Empresa D', '44.444.444/4444-44', 'São Paulo', 'SP', '44444-444', '2020-01-
01')
(5, 'Empresa E', '55.555.555/5555-55', 'Campinas', 'SP', '55555-555', '2020-01-
01')
(6, 'Empresa F', '66.666.666/6666-66', 'São Paulo', 'SP', '66666-666', '2020-01-
01')
```

0

Ver anotações

Na entrada 9, alteramos o campo *cidade* do registro com id_fornecedor 5. No comando *update* é necessário usar a cláusula *where* para identificar o registro a ser alterado, caso não use, todos são alterados. Como estamos fazendo uma alteração no banco, precisamos gravar, razão pela qual usamos o *commit()* na linha 8. Para checar a atualização fizemos uma leitura mostrando todos os registros.

DELETE

Ao inserir um registro no banco, pode ser necessário removê-lo no futuro, o que pode ser feito por meio da instrução SQL DELETE. Observe o código a seguir.

In [10]:

```
# Só é preciso importar a biblioteca uma vez. Importamos novamente para manter
todo o código em uma única célula
import sqlite3

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

cursor.execute("DELETE FROM fornecedor WHERE id_fornecedor = 2")
conn.commit()

cursor.execute("SELECT * FROM fornecedor")
for linha in cursor.fetchall():
    print(linha)

cursor.close()
conn.close()
```

```
(1, 'Empresa A', '11.111.111/1111-11', 'São Paulo', 'SP', '11111-111', '2020-01-
01')
(3, 'Empresa C', '33.333.333/3333-33', 'Curitiba', 'PR', '33333-333', '2020-01-
01')
(4, 'Empresa D', '44.444.444/4444-44', 'São Paulo', 'SP', '44444-444', '2020-01-
01')
(5, 'Empresa E', '55.555.555/5555-55', 'Campinas', 'SP', '55555-555', '2020-01-
01')
(6, 'Empresa F', '66.666.666/6666-66', 'São Paulo', 'SP', '66666-666', '2020-01-
01')
```

Na entrada 10, apagamos o registro com id_fornecedor 2. No comando *delete,* é necessário usar a cláusula *where* para identificar o registro apagado. Como estamos fazendo uma alteração no banco, precisamos gravar, razão pela qual usamos o *commit()* na linha 8. Para checar a atualização, fizemos uma leitura que mostra todos os registros.

Com a operação *delete*, concluímos nosso CRUD em um banco de dados SQLite usando a linguagem Python. O mais interessante e importante é que todas as etapas e todos os comandos que usamos podem ser aplicados em qualquer banco de dados relacional, uma vez que os módulos devem seguir as mesmas regras.

INFORMAÇÕES DO BANCO DE DADOS E DAS TABELAS

Além das operações de CRUD, é importante sabermos extrair informações estruturais do banco de dados e das tabelas. Por exemplo, considerado um banco de dados, quais tabelas existem ali? Quais são os campos de uma tabela? Qual é a estrutura da tabela, ou seja, qual DDL foi usada para gerá-la? Os comandos necessários para extrair essas informações podem mudar entre os bancos, mas vamos ver como extraí-las do SQLite. No código a seguir (entrada 11), temos uma instrução SQL capaz de retornar as tabelas no banco SQLite (linha 8) e outra capaz de extrair as DDLs usadas para gerar as tabelas (linha 15).

In [11]:

```
# Só é preciso importar a biblioteca uma vez. Importamos novamente para manter
todo o código em uma única célula
import sqlite3

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

# Lista as tabelas do banco de dados
cursor.execute("""SELECT name FROM sqlite_master WHERE type='table' ORDER BY
name""")
print('Tabelas:')
for tabela in cursor.fetchall():
    print(tabela)

# Captura a DDL usada para criar a tabela
tabela = 'fornecedor'
cursor.execute(f"""SELECT sql FROM sqlite_master WHERE type='table' AND
name='{tabela}'""")
print(f'\nDDL da tabela {tabela}:')
for schema in cursor.fetchall():
    print("%s" % (schema))

cursor.close()
conn.close()
```

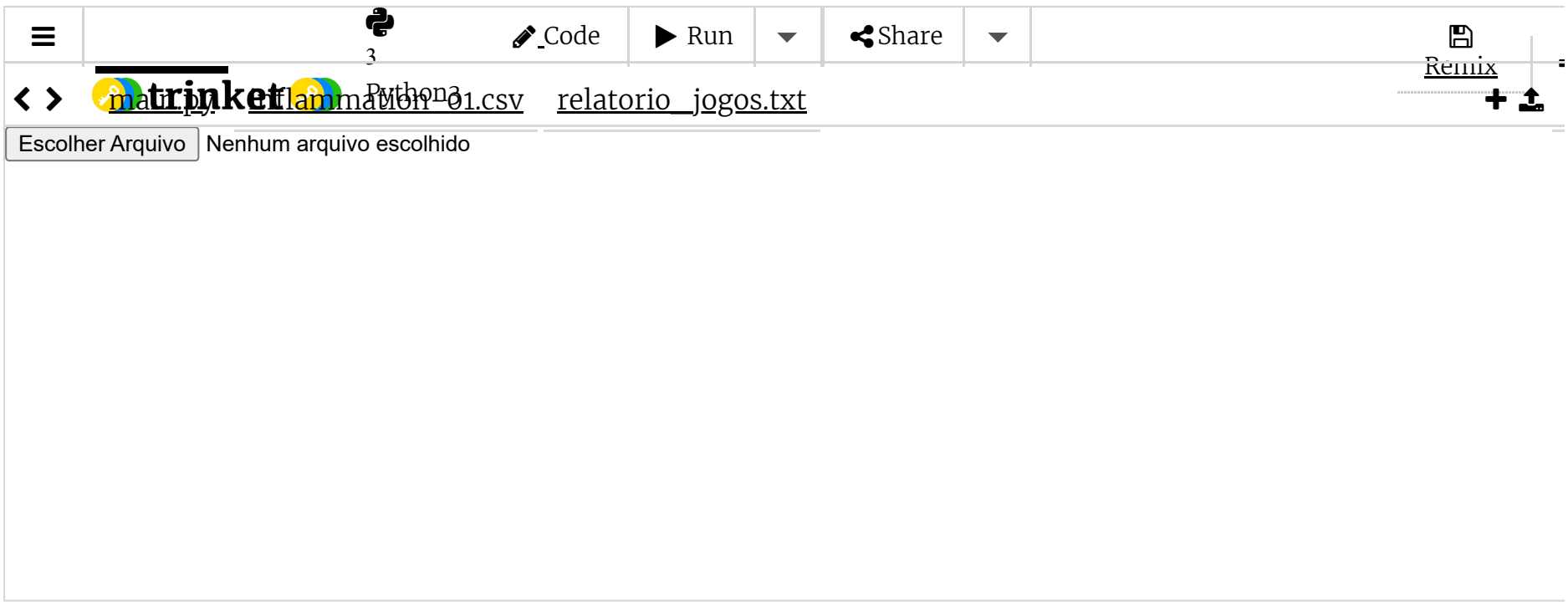
0

Ver anotações

```
Tabelas:
('fornecedor',)
('sqlite_sequence',)

DDL da tabela fornecedor:
CREATE TABLE fornecedor (
    id_fornecedor INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    nome_fornecedor TEXT NOT NULL,
    cnpj VARCHAR(18) NOT NULL,
    cidade TEXT,
    estado VARCHAR(2) NOT NULL,
    cep VARCHAR(9) NOT NULL,
    data_cadastro DATE NOT NULL
)
```

Que tal usar o ambiente a seguir para testar a conexão e as operações com o banco SQLite?! Execute o código, altere e faça novos testes!



REFERÊNCIAS E LINKS ÚTEIS

BANIN, S. L. **Python 3 - conceitos e aplicações**: uma abordagem didática. São Paulo: Érica, 2018.

MACHADO, F. N. R. **Projeto e implementação de banco de dados**. 3. ed. São Paulo: Érica, 2014.

MENON, R. M. Introduction to JDBC. In: MENON, R. M. **Expert Oracle JDBC Programming**. New York: Apress, 2005. p. 79-113.

MICROSOFT. **O que é o ODBC?** 2017. Disponível em: https://bit.ly/2XRHtuw. Acesso em: 31 jul. 2020.

PEP 249 - Python database API specification v.2.0. **Python**, 2020. Disponível em: https://bit.ly/3izcprq. Acesso em: 31 jul. 2020.

PSF - Python Software Fundation. **Python Module Index**. 2020c. Disponível em: https://bit.ly/3fTMkkY. Acesso em: 04 jun. 2020

PSF - Python Software Fundation. **sqlite3**. 2020d. Disponível em: https://bit.ly/3iE8ARY. Acesso em: 4 jun. 2020.

RAMAKRISHNAN, R.; GEHRKE, J. **Sistemas de gerenciamento de banco de dados**. 3. ed. Porto Alegre: AMGH, 2003.

ROCKOFF, L. **The Language of SQL**. 2. ed. [*S.l.*]: Pearson Education, 2016.

SQLITE. **What Is SQLite?** 2020. Disponível em: https://www.sqlite.org/index.html. Acesso em: 12 jun. 2020.

TAKAHASHI, M.; AZUMA, S. **Guia mangá de bancos de dados**. São Paulo: Novatec, 2009.

VAISH, G. **Getting Started with NoSQL**. Birmingham: Packt Publishing, 2013.