**Introdução da aula**



Qual é o foco da aula?

Nesta aula, vamos explorar como utilizar um banco de dados relacional em Python.

Objetivos gerais de aprendizagem

Ao longo desta aula, você irá:

* relatar a linguagem de consulta estruturada;
* descrever o banco de dados SQlite;
* identificar os acrônimos de operações.

**Situação-problema**

Utilizar bancos de dados para persisti-los faz parte da realidade de toda empresa que tenha algum sistema computadorizado, mesmo o destinado somente para cadastro de cliente. Na edição do dia 6 de março de 2017, a revista *[The Economist](https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data" \t "_blank)* publicou um artigo cujo título afirma "The world’s most valuable resource is no longer oil, but data" (O recurso mais valioso do mundo não é mais petróleo, mas dados). Esse texto trouxe os holofotes para o que muitas empresas já sabiam: uma vez que os dados são preciosos, precisamos armazená-los e tratá-los para extrair informações.

Hoje é comum encontrar nas empresas equipes totalmente focadas em soluções que envolvem dados. Parte do time é responsável por construir componentes (softwares ou microsserviços) voltados às necessidades de armazenamento e recuperação de dados. Esses componentes precisam ser parametrizáveis, o que significa que o componente deve ser capaz de fazer a mesma tarefa para diferentes tecnologias. Para os casos de recuperação de dados em banco de dados relacional, construir um componente parametrizável significa recuperar dados de diferentes RDBMS com o mesmo código mas com diferentes parâmetros.

Como desenvolvedor em uma empresa de consultoria de software, você foi alocado para construir uma solução parametrizável capaz de criar banco de dados na tecnologia SQLite, criar e apagar tabelas, o nome do banco, das tabelas e a DDL necessária para criar uma tabela (tudo deve ser parametrizável). Você também deve construir uma solução capaz de fazer um CRUD em um banco SQLite – veja que o componente deve funcionar para qualquer nome de banco e de tabela, ou seja, parâmetros. As regras que lhe foram passadas são as seguintes:

* Para criar uma nova base de dados, é necessário informar o nome.
* Para criar uma nova tabela, é necessário informar o nome do banco que receberá a tabela e a DDL de criação.
* Para excluir uma tabela, é necessário informar o nome do banco e da tabela.
* Para inserir um novo registro em uma tabela, é preciso informar: o nome do banco e da tabela e um dicionário contendo a chave e o valor a ser inserido. Por exemplo, {'nome': 'João', 'idade': 30}.
* Para recuperar, os dados é preciso informar o nome do banco e da tabela.
* Para atualizar um novo registro em uma tabela é preciso informar: o nome do banco e da tabela e dois dicionários, um contendo a chave e o valor a ser atualizado e outro contendo a chave e o valor da condição do registro que será atualizado.
* Para excluir um registro em uma tabela, é preciso informar: o nome do banco e da tabela e um dicionário contendo a chave e o valor da condição para localizar o registro a ser inserido. Por exemplo, {'id\_cliente': 10}.

Então, mãos à obra!

**Introdução à 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)**



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 de dados SQL 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 os 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.

**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.

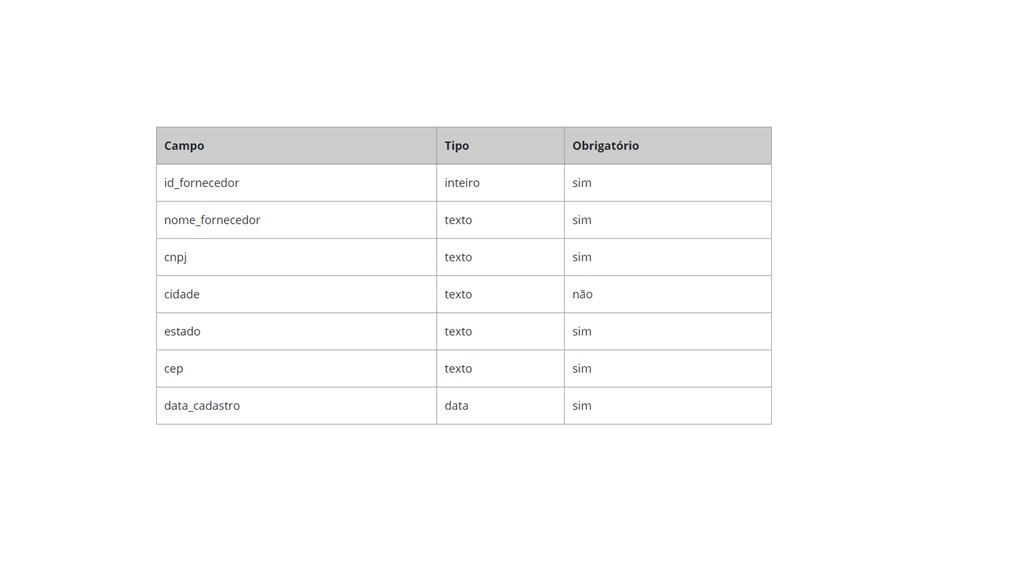
Essa tecnologia pode ser embutida em telefones celulares e computadores e vem incluída em inúmeros outros aplicativos que as pessoas usam todos os dias. Ao passo que a maioria dos bancos de dados SQL usa um servidor para rodar e gerenciar, o SQLite não possui um processo de servidor separado. O SQLite 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 abaixo:

Tabela fornecedor. Fonte: elaborado pelo autora.

**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'>

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.

\_\_\_\_\_\_\_

**➕ Pesquise 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](https://www.python.org/dev/peps/pep-0249/" \l "description" \t "_blank) (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.

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 e Read**



**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) fechar 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

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 exige 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

**UPdate e Delete**



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')

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()

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,

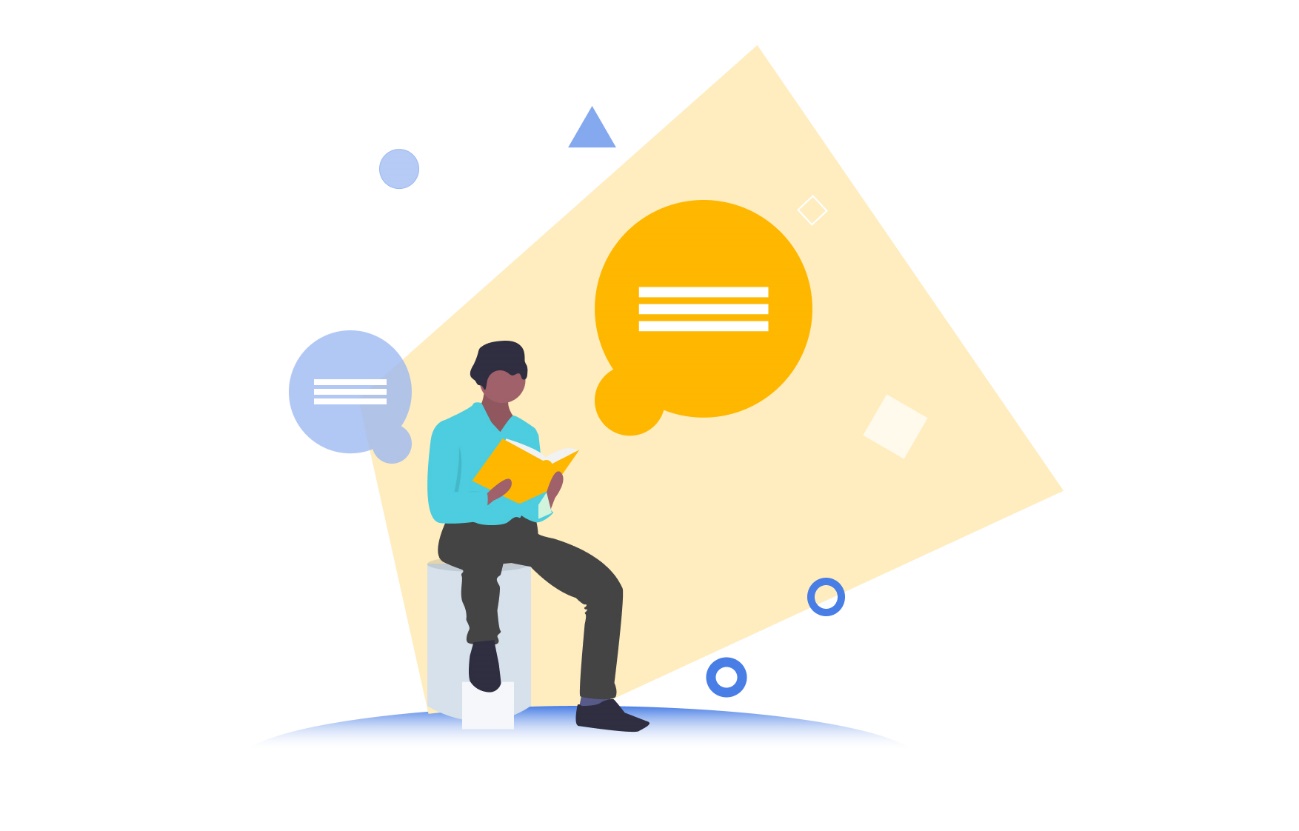
    cep VARCHAR(9) NOT NULL,

    data\_cadastro DATE NOT NULL

)

Que tal usar o [ambiente](https://trinket.io/python3/d88de3f6ed" \t "_blank)para testar a conexão e as operações com o banco SQLite?! Execute o código, altere e faça novos testes!

**Conclusão**



Chegou o momento de implementar a versão inicial da solução parametrizável que realiza  funções tanto de DDL quanto de DML em um banco de dados SQLite. Certamente, existem muitas possibilidades de implementação. Vejamos uma proposta inicial de solução.

Uma possibilidade é construir duas classes, uma com os métodos para DDL e outra para o CRUD (DML). Vamos, então, começar pela classe capaz de criar um banco, criar e apagar uma tabela. Veja a seguir a classe *DDLSQLite*. Criamos um método privado (lembre-se de que, em Python, trata-se somente uma convenção de nomenclatura) que retorna uma instância de conexão com um banco parametrizável.

Também criamos três método públicos, um para criar um banco de dados, outro para criar uma tabela e o último para apagar uma tabela. Todos os métodos são parametrizáveis, conforme foi solicitado. Chamo a atenção para o método *criar\_banco\_de\_dados()*, que cria o banco fazendo uma conexão e já fechando-a (lembre-se de que existem diferentes formas de fazer uma determinada implementação).

In [12]: import sqlite3

class DDLSQLite:

    def \_conectar(self, nome\_banco):

        nome\_banco **+=** '.db'

        conn **=** sqlite3**.**connect(nome\_banco)

        return conn

    def criar\_banco\_de\_dados(self, nome\_banco):

        nome\_banco **+=** '.db'

        sqlite3**.**connect(nome\_banco)**.**close()

        print(f"O banco de dados {nome\_banco} foi criado com sucesso!")

        return None

    def criar\_tabela(self, nome\_banco, ddl\_create):

        conn **=** self**.**\_conectar(nome\_banco)

        cursor **=** conn**.**cursor()

        cursor**.**execute(ddl\_create)

        cursor**.**close()

        conn**.**close()

        print(f"Tabela criada com sucesso!")

        return None

    def apagar\_tabela(self, nome\_banco, tabela):

        conn **=** self**.**\_conectar(nome\_banco)

        cursor **=** conn**.**cursor()

        cursor**.**execute(f"DROP TABLE {tabela}")

        cursor**.**close()

        conn**.**close()

        print(f"A tabela {tabela} foi excluída com sucesso!")

        return None

O próximo passo é construir a classe que faz o CRUD. Observe a proposta da classe *CrudSQLite*. Como todas as operações precisam receber como parâmetro o nome do banco de dados, colocamos o parâmetro no método construtor, ou seja, para instanciar essa classe, deve ser informado o nome da banco e, a partir do objeto, basta chamar os métodos e passar os demais parâmetros. O nome do banco será usado pelo método privado *\_conectar(),* que retorna uma instância da conexão – esse método será chamado internamente pelos demais métodos. O método inserir\_registro(), extrai como uma tupla as colunas (linha 13) e os valores (linha 14) a serem usados para construir o comando de inserção. O método ler\_registros() seleciona todos os dados de uma tabela e os retorna, lembrando que o retorno será uma lista de tuplas. O método atualizar\_registro(), precisa tando dos dados a serem atualizados (dicionário) quanto dos dados que serão usados para construir a condição. **Nessa primeira versão do componente, a atualização só pode ser feita desde que o valor da condição seja inteiro**. O método*apagar\_registro()* também só pode ser executado desde que o valor da condição seja inteiro.

In [13]: import sqlite3

class CrudSQLite:

    def \_\_init\_\_(self, nome\_banco):

        self**.**nome\_banco **=** nome\_banco **+** '.db'

    def \_conectar(self):

        conn **=** sqlite3**.**connect(self**.**nome\_banco)

        return conn

    def inserir\_registro(self, tabela, registro):

        colunas **=** tuple(registro**.**keys())

        valores **=** tuple(registro**.**values())

        conn **=** self**.**\_conectar()

        cursor **=** conn**.**cursor()

        query **=** f"""INSERT INTO {tabela} {colunas} VALUES {valores}"""

        cursor**.**execute(query)

        conn**.**commit()

        cursor**.**close()

        conn**.**close()

        print("Dados inseridos com sucesso!")

        return None

    def ler\_registros(self, tabela):

        conn **=** self**.**\_conectar()

        cursor **=** conn**.**cursor()

        query **=** f"""SELECT \* FROM {tabela}"""

        cursor**.**execute(query)

        resultado **=** cursor**.**fetchall()

        cursor**.**close()

        conn**.**close()

        return resultado

    def atualizar\_registro(self, tabela, dado, condicao):

        campo\_alterar **=** list(dado**.**keys())[0]

        valor\_alterar **=** dado**.**get(campo\_alterar)

        campo\_condicao **=** list(condicao**.**keys())[0]

        valor\_condicao **=** condicao**.**get(campo\_condicao)

        conn **=** self**.**\_conectar()

        cursor **=** conn**.**cursor()

        query **=** f"""UPDATE {tabela} SET {campo\_alterar} = '{valor\_alterar}' WHERE {campo\_condicao} = {valor\_condicao}"""

        cursor**.**execute(query)

        conn**.**commit()

        cursor**.**close()

        conn**.**close()

        print("Dado atualizado com sucesso!")

        return None

    def apagar\_registro(self, tabela, condicao):

        campo\_condicao **=** list(condicao**.**keys())[0]

        valor\_condicao **=** condicao**.**get(campo\_condicao)

        conn **=** self**.**\_conectar()

        cursor **=** conn**.**cursor()

        query **=** f"""DELETE FROM {tabela} WHERE {campo\_condicao} = {valor\_condicao}"""

        cursor**.**execute(query)

        conn**.**commit()

        cursor**.**close()

        conn**.**close()

        print("Dado excluído com sucesso!")

        return None

Antes de entregar uma solução, é preciso testar muitas vezes. Portanto, vamos aos testes. Primeiro vamos testar a classe DDLSQLite. Como podemos observar no resultado, tudo ocorreu como esperado: tanto o método para criar o banco quanto a tabela funcionaram corretamente.

In [14]: # instancia um objeto

objeto\_ddl **=** DDLSQLite()

# Cria um banco de dados

objeto\_ddl**.**criar\_banco\_de\_dados('desafio')

# Cria uma tabela chamada cliente

ddl\_create **=** """

CREATE TABLE cliente (

    id\_cliente INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,

    nome\_cliente TEXT NOT NULL,

    cpf VARCHAR(14) NOT NULL,

    email TEXT NOT NULL,

    telefone VARCHAR(15),

    cidade TEXT,

    estado VARCHAR(2) NOT NULL,

    data\_cadastro DATE NOT NULL

);

"""

objeto\_ddl**.**criar\_tabela(nome\_banco**=**'desafio', ddl\_create**=**ddl\_create)

# Caso precise excluir a tabela, o comando a seguir deverá ser usado

# objeto\_ddl.apagar\_tabela(nome\_banco='desafio', tabela='cliente')

O banco de dados desafio.db foi criado com sucesso!

Tabela criada com sucesso!

Vamos fazer os testes do CRUD em duas etapas. Na primeira, vamos inserir registros e consultarmos para checar se a inserção deu certo.

In [15]: objeto\_dml **=** CrudSQLite(nome\_banco**=**'desafio')

# Inserir registros

dados **=** [

    {

        'nome\_cliente': 'João',

        'cpf': '111.111.111-11',

        'email': 'joao@servidor',

        'cidade': 'São Paulo',

        'estado': 'SP',

        'data\_cadastro': '2020-01-01'

    },

    {

        'nome\_cliente': 'Maria',

        'cpf': '222.222.222-22',

        'email': 'maria@servidor',

        'cidade': 'São Paulo',

        'estado': 'SP',

        'data\_cadastro': '2020-01-01'

    },

]

# Para cada dicionário na lista de dados, invoca o método de inserção

for valor in dados:

    objeto\_dml**.**inserir\_registro(tabela**=**'cliente', registro**=**valor)

# Carrega dados salvos

dados\_carregados **=** objeto\_dml**.**ler\_registros(tabela**=**'cliente')

for dado in dados\_carregados:

    print(dado)

Dados inseridos com sucesso!

Dados inseridos com sucesso!

(1, 'João', '111.111.111-11', 'joao@servidor', None, 'São Paulo', 'SP', '2020-01-01')

(2, 'Maria', '222.222.222-22', 'maria@servidor', None, 'São Paulo', 'SP', '2020-01-01')

Agora vamos atualizar um registro e excluir outro.

In [16]: # Atualiza registro

dado\_atualizar **=** {'telefone': '(11)1.1111-1111'}

condicao **=** {'id\_cliente': 1}

objeto\_dml**.**atualizar\_registro(tabela**=**'cliente', dado**=**dado\_atualizar, condicao**=**condicao)

dados\_carregados **=** objeto\_dml**.**ler\_registros(tabela**=**'cliente')

for dado in dados\_carregados:

    print(dado)

# Apaga registro

condicao **=** {'id\_cliente': 1}

objeto\_dml**.**apagar\_registro(tabela**=**'cliente', condicao**=**condicao)

Dado atualizado com sucesso!

(1, 'João', '111.111.111-11', 'joao@servidor', '(11)1.1111-1111', 'São Paulo', 'SP', '2020-01-01')

(2, 'Maria', '222.222.222-22', 'maria@servidor', None, 'São Paulo', 'SP', '2020-01-01')

Dado excluído com sucesso!

Felizmente nosso componente funcionou conforme esperado. Gostaríamos de encerrar o desafio lembrando que são muitas as formas de implementação. Tratando-se de classes que se comunicam com banco de dados, o padrão de projeto singleton é o mais adequado para determinados casos, uma vez que permite instanciar somente uma conexão com o banco.

**Referências**



LJUBOMIR, P. **Introdução à computação usando Python**: um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016.

PSF. Python Software Fundation. **Classes**. 2020a. Disponível em: https://bit.ly/3fSURVx. Acesso em: 10 maio 2020.

PYTHON COURSE. Properties vs. Getters and Setters. Python 3 Tutorial. 2020. Disponível em: https://bit.ly/2XWY9Rn. Acesso em: 08 jun. 2021

RAMALHO, L. **Fluent Python**. Gravenstein: O’Reilly Media, 2014.

WEISFELD, M. A. **The Object-Oriented**: Thought Process. 4. ed. [S.l.]: Addison Wesley Professional, 2013.

API CNAE - Cadastro Nacional de Atividades Econômicas. **API e documentação**. Versão: 2.0.0. CNAE, 2017. Disponível em: https://bit.ly/3amaPGE. Acesso em: 30 jul. 2020.

LJUBOMIR, P. **Introdução à computação usando Python**: um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016.

PSF - Python Software Fundation. **Modules**. 2020b. Disponível em: https://bit.ly/30SmzNC. Acesso em: 04 jun. 2020.

PyPI. Python Package Index. Página inicial. 2020. Disponível em:  https://pypi.org/. Acesso em: 04 jun. 2020.

RAMALHO, L. **Fluent Python**. Gravenstein: O’Reilly Media, 2014.

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.