



DESCRIÇÃO

Desenvolvimento de aplicações para armazenamento e recuperação de informação em banco de dados utilizando a linguagem de programação Python.

PROPÓSITO

Compreender os passos e ações necessárias para realizar a manipulação de registro em banco de dados, de forma a garantir o correto funcionamento de programas que tenham essas características.

PREPARAÇÃO

Antes de iniciar o conteúdo, sugerimos que você tenha o software Python 3.7, ou posterior, instalado em seu computador. Ele será utilizado nos módulos deste tema.

Você pode pesquisar e baixar o Python 3.7 no site oficial do Python.

Para programar os exemplos apresentados neste módulo, utilizamos a IDE PyCharm Community. Você pode baixar essa IDE gratuitamente no site oficial da JetBrains.

Para visualizar os dados e tabelas, vamos utilizar o DB Browser for SQLite, também disponível gratuitamente.

Para que você possa acompanhar melhor os exemplos apresentados clique aqui para realizar o download do arquivo “Banco de dados.zip” que contém os scripts utilizados neste conteúdo.

OBJETIVOS

MÓDULO 1

Reconhecer as funcionalidades de frameworks e bibliotecas para gerenciamento de banco de dados

MÓDULO 2

Empregar as funcionalidades para conexão, acesso e criação de bancos de dados e tabelas

MÓDULO 3

Aplicar as funcionalidades para inserção, remoção e atualização de registros em tabelas

MÓDULO 4

Empregar as funcionalidades para recuperação de registros em tabelas

INTRODUÇÃO

Neste tema, aprenderemos a utilizar o Python para criar aplicações que utilizam banco de dados.

Atualmente, muitas aplicações, como gerenciadores de conteúdo, jogos e sistemas de controle de vendas utilizam banco de dados. É muito importante que o desenvolvedor saiba manipular as informações de um banco de dados.

Nos próximos módulos, mostraremos como criar novos bancos de dados, tabelas e relacionamentos, inserir e remover registros, além de recuperar as informações presentes no banco por meio de consultas.

Para demonstrar a utilização desses comandos, usaremos o banco de dados SQLite, que é um banco de dados leve, baseado em arquivo, que não necessita de um servidor para ser utilizado.

O SQLite está disponível em diversas plataformas, inclusive em celulares com sistema operacional Android.

O SQLite também é muito utilizado durante o desenvolvimento das aplicações, para então ser substituído por um banco de dados mais robusto, como PostgreSQL ou MySQL, quando em produção.

MÓDULO 1

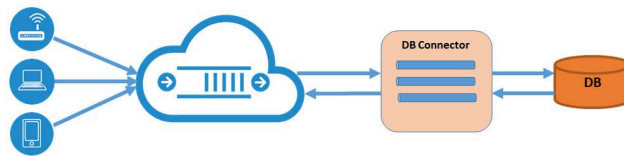
🕒 Reconhecer as funcionalidades de frameworks e bibliotecas para gerenciamento de banco de dados

CONCEITOS

CONECTORES PARA BANCO DE DADOS

Quando estamos trabalhando com banco de dados, precisamos pesquisar por bibliotecas que possibilitem a conexão da aplicação com o banco de dados que iremos utilizar.

Na área de banco de dados, essas bibliotecas se chamam conectores ou adaptadores.



📷 Figura: 1.

Como estamos trabalhando com Python, devemos procurar por conectores que atendam a PEP 249 (Python Database API Specification v2.0 - DB-API 2.0) .

Essa PEP (Python Enhancement Proposal) especifica um conjunto de padrões que os conectores devem seguir, a fim de garantir um melhor entendimento dos módulos e maior portabilidade entre bancos de dados.

Como exemplo de padrões definidos pela DB-API 2.0, podemos citar:

Nomes de métodos:

close, commit e cursor

Nomes de Classes:

Connection e Cursor

Tipos de exceção:

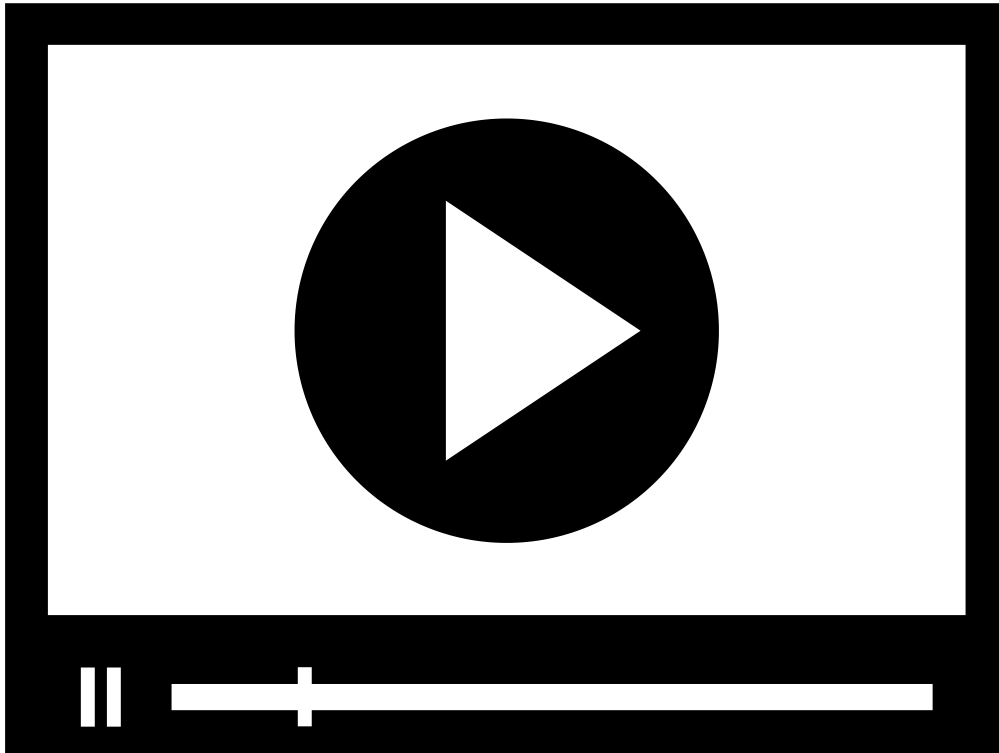
IntegrityError e InternalError

Como exemplo de conectores que implementam a DB-API 2.0, temos:

psycopg2: Conector mais utilizado para o banco de dados PostgreSQL.

mysqlclient, PyMySQL e mysql-connector-python: Conectores para o banco de dados MySQL.

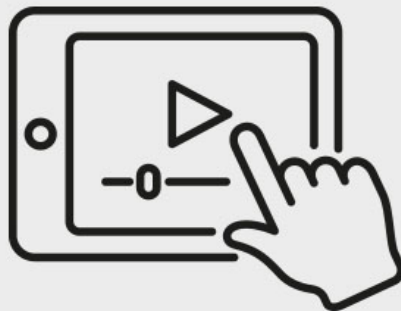
sqlite3: Adaptador padrão do Python para o banco de dados SQLite.



EXPLORANDO CONECTORES PARA BANCOS DE DADOS EM PYTHON

Neste vídeo, vamos falar dos conectores para bancos de dados em Python. Vamos abordar diferentes bibliotecas, como `psycopg2`, `mysqlclient`, `PyMySQL` e `mysql-connector-python`, além do `sqlite3`. Você aprenderá como se conectar, consultar e manipular dados em diversos tipos de bancos de dados usando Python. Prepare-se para ampliar suas habilidades em manipulação de dados e integração com bancos de dados em suas aplicações Python.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



COMENTÁRIO

Apenas o conector para SQLite está disponível nativamente no Python 3.7. Todos os demais precisam ser instalados.

Como dito na introdução, neste tema, utilizaremos o banco de dados SQLite para demonstrar o desenvolvimento de aplicações Python para banco de dados.

Apesar do nome Lite (leve), o SQLite é um banco de dados completo, que permite a criação de tabelas, relacionamentos, índices, gatilhos (trigger) e visões (views).

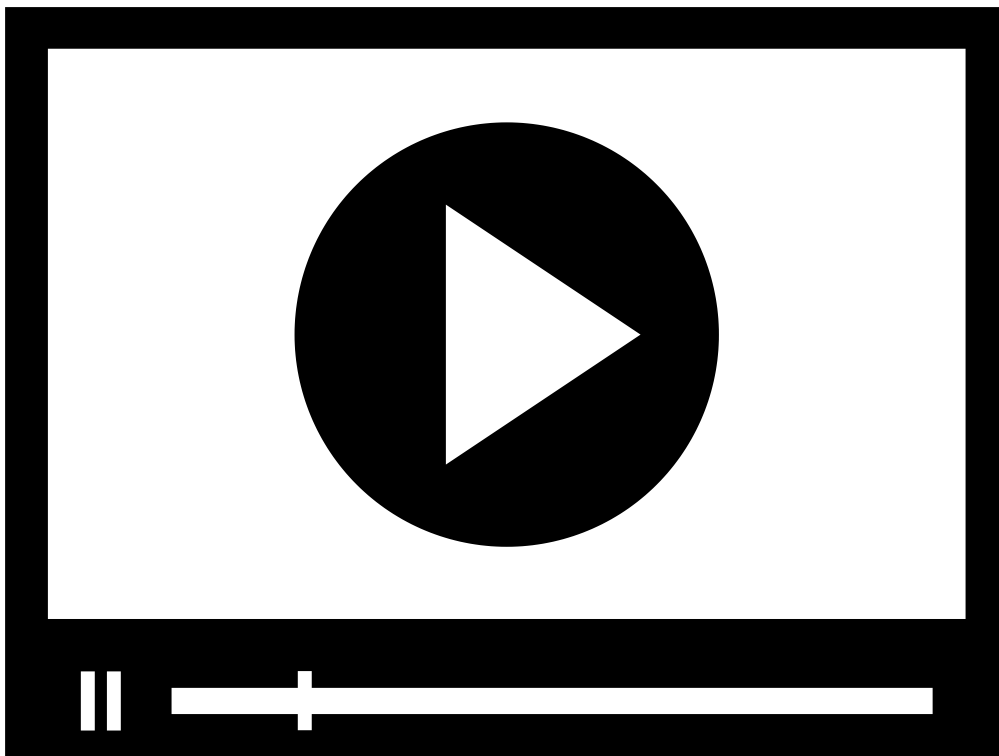
O SQLite também tem suporte a subconsultas, transações, junções, pesquisa em texto (full text search), restrições de chave estrangeira, entre outras funcionalidades.

Porém, por não ter um servidor para seu gerenciamento, o SQLite não provê um acesso direto pela rede. As soluções existentes para acesso remoto são “caras” e ineficientes.

Além disso, o SQLite não tem suporte à autenticação, com usuários e permissões definidas. Estamos falando aqui sobre usuários do banco de dados, que têm permissão para criar tabela, inserir registros etc.

Observe que, apesar do SQLite não ter suporte à autenticação, podemos e devemos implementar nosso próprio controle de acesso para nossa aplicação.

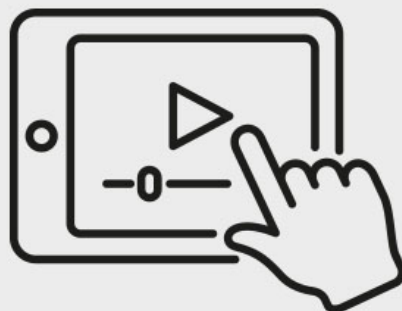
PRINCIPAIS MÉTODOS DOS CONECTORES EM PYTHON



EXPLORANDO OS MÉTODOS ESSENCIAIS DOS CONECTORES EM PYTHON

Neste vídeo, vamos aprofundar nosso conhecimento sobre os principais métodos dos conectores em Python, incluindo `Connect`, `Connection` e `Cursor`. Você aprenderá como usar essas funções fundamentais para estabelecer conexões com bancos de dados, gerenciar transações e executar consultas SQL de maneira eficaz. Prepare-se para dominar o essencial na interação com bancos de dados usando Python.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Antes de começarmos a trabalhar diretamente com banco de dados, precisamos conhecer algumas classes e métodos disponibilizados pelos conectores e previstos na PEP 249.

Esses métodos são a base para qualquer aplicação que necessite de acesso a banco de dados e estão descritos a seguir:

CONNECT

Função global do conector para criar uma conexão com o banco de dados.

Retorna um objeto do tipo Connection.

CONNECTION

Classe utilizada para gerenciar todas as operações no banco de dados.

Principais métodos:

commit: Confirma todas as operações pendentes;

rollback: Desfaz todas as operações pendentes;

cursor: Retorna um objeto do tipo Cursor; e

close: Encerra a conexão com o banco de dados.

CURSOR

Classe utilizada para enviar os comandos ao banco de dados.

Principais métodos:

execute: Prepara e executa a operação passada como parâmetro;

fetchone: Retorna a próxima linha encontrada por uma consulta; e

fetchall: Retorna todas as linhas encontradas por uma consulta.

A utilização desses métodos segue basicamente o mesmo fluxo de trabalho para todas as aplicações que utilizam banco de dados. A seguir, vamos descrever esse fluxo:

Criar uma conexão com o banco de dados utilizando a função connect.

Utilizar a conexão para criar um cursor, que será utilizado para enviar comandos ao banco de dados.

Utilizar o cursor para enviar comandos ao banco de dados, por exemplo:

Criar tabelas.

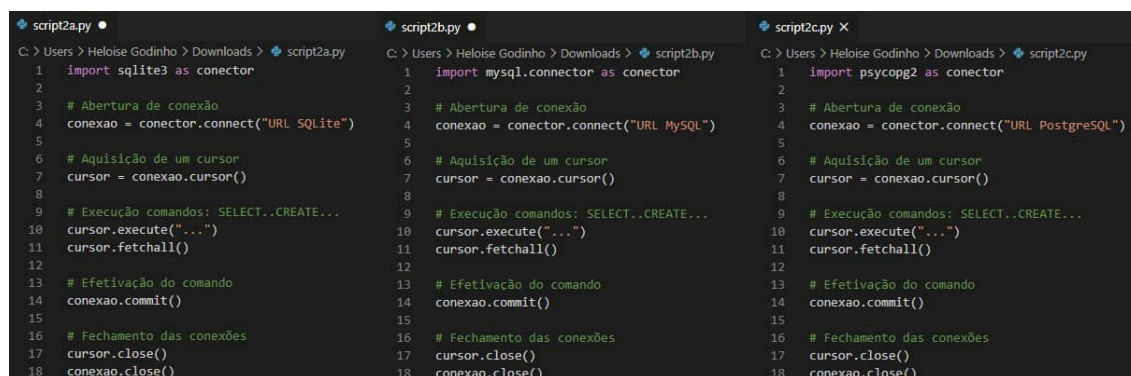
Inserir linhas.

Selecionar linhas.

Efetivar as mudanças no banco de dados utilizando o método commit da conexão.

Fechar o cursor e a conexão

Observe os scripts a seguir na Figura 2, na qual temos esse fluxo de trabalho descrito na forma de código Python.



```
script2a.py
C: > Users > Heloise Godinho > Downloads > script2a.py
1 import sqlite3 as conector
2
3 # Abertura de conexão
4 conexao = conector.connect("URL SQLite")
5
6 # Aquisição de um cursor
7 cursor = conexao.cursor()
8
9 # Execução comandos: SELECT..CREATE...
10 cursor.execute("...")
11 cursor.fetchall()
12
13 # Efetivação do comando
14 conexao.commit()
15
16 # Fechamento das conexões
17 cursor.close()
18 conexao.close()

script2b.py
C: > Users > Heloise Godinho > Downloads > script2b.py
1 import mysql.connector as conector
2
3 # Abertura de conexão
4 conexao = conector.connect("URL MySQL")
5
6 # Aquisição de um cursor
7 cursor = conexao.cursor()
8
9 # Execução comandos: SELECT..CREATE...
10 cursor.execute("...")
11 cursor.fetchall()
12
13 # Efetivação do comando
14 conexao.commit()
15
16 # Fechamento das conexões
17 cursor.close()
18 conexao.close()

script2c.py
C: > Users > Heloise Godinho > Downloads > script2c.py
1 import psycopg2 as conector
2
3 # Abertura de conexão
4 conexao = conector.connect("URL PostgreSQL")
5
6 # Aquisição de um cursor
7 cursor = conexao.cursor()
8
9 # Execução comandos: SELECT..CREATE...
10 cursor.execute("...")
11 cursor.fetchall()
12
13 # Efetivação do comando
14 conexao.commit()
15
16 # Fechamento das conexões
17 cursor.close()
18 conexao.close()
```

Fonte: O Autor

 Figura: 2

Neste momento, vamos focar apenas no script1, mais à esquerda da figura.

Na primeira linha, importamos o módulo sqlite3, conector para o banco de dados SQLite disponível nativamente no Python 3.7. Atribuímos um alias (apelido) a esse import chamado conector. Veremos ao final por que criamos esse alias.

Na linha 4, abrimos uma conexão com o banco de dados utilizando uma URL. O retorno dessa conexão é um objeto da classe Connection, que foi atribuído à variável conexao.

Na linha 7, utilizamos o método cursor da classe Connection para criar um objeto do tipo Cursor, que foi atribuído à variável cursor.

Na linha 10, utilizamos o método `execute`, da classe `Cursor`. Este método permite enviar comandos SQL para o banco de dados. Entre os comandos, podemos citar: `SELECT`, `INSERT` e `CREATE`.

Na linha 11, usamos o método `fetchall`, da classe `Cursor`, que retorna os resultados de uma consulta.

Na linha 14, utilizamos o método `commit`, da classe `Connection`, para efetivar todos os comandos enviados anteriormente. Se desejássemos desfazer os comandos, poderíamos utilizar o método `rollback`.

Nas linhas 17 e 18, fechamos o cursor e a conexão, respectivamente.

A estrutura apresentada irá se repetir ao longo deste tema, onde, basicamente, preencheremos o parâmetro do método `execute` com o comando SQL pertinente.

ATENÇÃO

Agora observe os scripts 2 e 3 da Figura 2. e veja que ambas as bibliotecas ***mysql.connector*** e ***psycopg2*** contêm os mesmos métodos e funções da biblioteca ***sqlite3***.

Como estamos utilizando o ***alias conector*** no `import`, para trocar de banco de dados, bastaria apenas alterar o `import` da primeira linha. Isso se deve ao fato de que todas elas seguem a especificação **DB-API 2.0**.

Cada biblioteca tem suas particularidades, que podem adereçar alguma funcionalidade específica do banco de dados referenciado, mas todas elas implementam, da mesma maneira, o básico para se trabalhar com banco de dados.

Isso é de grande utilidade, pois podemos alterar o banco de dados a qualquer momento, sem a necessidade de realizar muitas alterações no código-fonte.

Mais adiante, mostraremos algumas situações onde as implementações podem diferir entre os conectores.

PRINCIPAIS EXCEÇÕES DOS CONECTORES

EM PYTHON

Além dos métodos, a DB-API 2.0 prevê algumas exceções que podem ser lançadas pelos conectores.

Vamos listar e explicar algumas dessas exceções:

Error	IntegrityError	OperationalError
DatabaseError	ProgrammingError	NotSupportedError

- ☐ **Atenção!** Para visualizaçãocompleta da tabela utilize a rolagem horizontal

ERROR

Classe base para as exceções. É a mais abrangente.

DATABASEERROR

Exceção para tratar erros relacionados ao banco de dados. Também é uma exceção abrangente. É uma subclasse de Error.

INTEGRITYERROR

Exceção para tratar erros relacionados à integridade do banco de dados, como falha na checagem de chave estrangeira e falha na checagem de valores únicos. É uma subclasse de DatabaseError.

PROGRAMMINGERROR

Exceção para tratar erros relacionados à programação, como sintaxe incorreta do comando SQL, tabela não encontrada etc. É uma subclasse de DatabaseError.

OPERATIONALERROR

Exceção para tratar erros relacionados a operações no banco de dados, mas que não estão sob controle do programador, como desconexão inesperada. É uma subclasse de DatabaseError.

NOTSUPPORTEDERROR

Exceção para tratar erros relacionados a operações não suportadas pelo banco de dados, como chamar o método rollback em um banco de dados que não suporta transações. É uma subclasse de DatabaseError.

TIPOS DE DADOS

Cada dado armazenado em um banco de dados contém um tipo, como inteiro, texto, ponto flutuante, entre outros.

Os tipos suportados pelos bancos de dados não são padronizados e, por isso, é necessário verificar na sua documentação quais tipos são disponibilizados.

O SQLite trata os dados armazenados de um modo um pouco diferente de outros bancos, nos quais temos um número limitado de tipos.

No SQLite, cada valor armazenado no banco é de uma das classes a seguir:

NULL	Para valores nulos.
INTEGER	Para valores que são números inteiros, com sinal.
REAL	Para valores que são números de ponto flutuante.
TEXT	Para valores que são texto (string).
BLOB	Para armazenar valores exatamente como foram inseridos, ex. bytes.

☐ **Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

Apesar de parecer um número limitado de classes, quando comparado com outros bancos de dados, o SQLite suporta o conceito de afinidades de tipo para as colunas.

AFINIDADES DE TIPO

Afinidade é a classe preferida para se armazenar um dado em uma determinada coluna.

No SQLite, quando definimos uma coluna durante a criação de uma tabela, ao invés de especificar um tipo estático, dizemos qual a afinidade dela. Isso nos permite armazenar diferentes tipos de dados em uma mesma coluna. Observe as afinidades disponíveis a seguir:

TEXT	Coluna para armazenar dados das classes NULL, TEXT e BLOB.
NUMERIC	Coluna para armazenar dados de qualquer uma das cinco classes.
INTEGER	Similar ao NUMERIC, diferenciando apenas no processo de conversão de valores.
REAL	Similar ao NUMERIC, porém os valores inteiros são forçados a serem representados como ponto flutuante.
NONE	Coluna sem preferência de armazenamento, não é realizada nenhuma conversão de valores.

☐ **Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

A afinidade também permite ao motor do SQLite fazer um mapeamento entre tipos não suportados e tipos suportados. Por exemplo, o tipo *VARCHAR(n)*, disponível no *MySQL* e *PostgreSQL*, é convertido para *TEXT* no *SQLite*.

Esse mapeamento nos permite definir atributos no *CREATE TABLE* com outros tipos, não suportados pelo *SQLite*. Esses tipos são convertidos para tipos conhecidos utilizando afinidade.

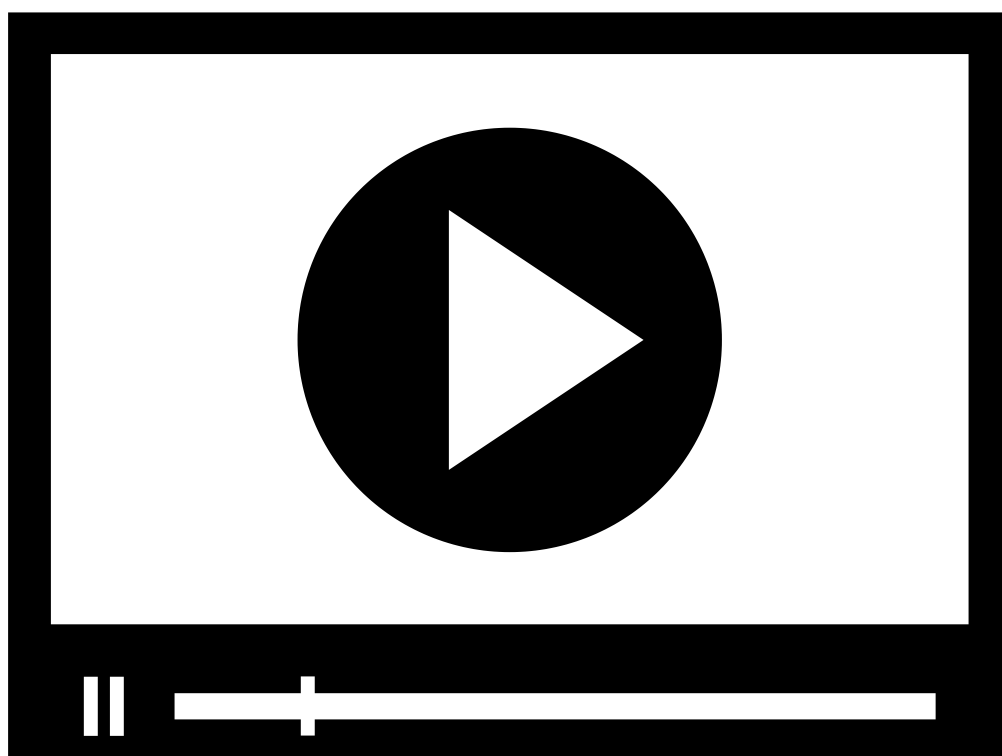
A tabela de afinidades do *SQLite* está descrita a seguir:

Tipo no CREATE TABLE	Afinidade
INT INTEGER TINYINT SMALLINT MEDIUMINT BIGINT UNSIGNED BIG INT INT2 INT8	INTEGER
CHARACTER(20) VARCHAR(255) VARYING CHARACTER(255) NCHAR(55) NATIVE CHARACTER(70) NVARCHAR(100) TEXT CLOB	TEXT
BLOB	BLOB
REAL DOUBLE	REAL

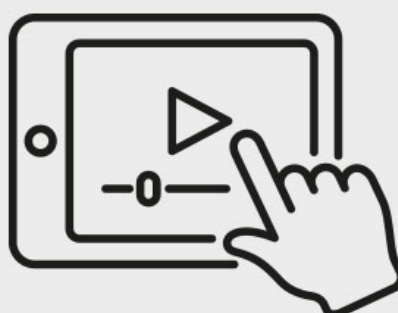
DOUBLE PRECISION FLOAT	
NUMERIC DECIMAL(10,5) BOOLEAN DATE DATETIME	NUMERIC

- ❑ **Atenção! Para visualizaçãocompleta da tabela utilize a rolagem horizontal**

A partir desta tabela, mesmo utilizando o SQLite para desenvolvimento, podemos definir os atributos de nossas tabelas com os tipos que utilizaremos em ambiente de produção.



Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Fontes das Informações:

Casos de dengues (datasus) e

Estimativas da População (IBGE)

VERIFICANDO O APRENDIZADO

1. NESTE MÓDULO, APRENDEMOS SOBRE AS BIBLIOTECAS PARA ACESSO A BANCO DE DADOS EM PYTHON. VIMOS QUE AS PRINCIPAIS BIBLIOTECAS IMPLEMENTAM A PEP 249, QUE PROPÕE PADRÕES QUE DEVEM SER SEGUIDOS POR ESSAS BIBLIOTECAS.

CONSIDERANDO AS CLASSES E MÉTODOS PROPOSTOS NA PEP 249, MARQUE A ALTERNATIVA CORRETA QUE COMPLETA A SEGUINTE AFIRMAÇÃO:

A FUNÇÃO GLOBAL _____ DEVE SER UTILIZADA PARA CRIAR UMA CONEXÃO COM O BANCO DE DADOS. ESSA FUNÇÃO RETORNA UM OBJETO DO TIPO _____, QUE É UTILIZADO PARA CRIAR UM OBJETO DO TIPO _____ UTILIZANDO O MÉTODO _____.

- A) connect – Connection – Cursor – cursor
- B) execute – Connection – cursor – Cursor
- C) Connection – connection – cursor – Cursor
- D) connect – Execute – Cursor – cursor

2. SOBRE O BANCO DE DADOS SQLITE E SEU CONECTOR SQLITE3 PARA PYTHON, ASSINALE A ALTERNATIVA CORRETA:

- A) Mesmo não tendo um servidor para seu gerenciamento, o SQLite provê acesso direto e otimizado pela rede.

B) O sqlite3 não permite criar colunas com tipos diferentes de TEXT, NUMERIC, INTEGER e REAL.

C) No SQLite, os dados de uma mesma coluna podem ser de tipos diferentes.

D) O sqlite3 não implementa a DB API 2.0.

GABARITO

1. Neste módulo, aprendemos sobre as bibliotecas para acesso a banco de dados em Python. Vimos que as principais bibliotecas implementam a PEP 249, que propõe padrões que devem ser seguidos por essas bibliotecas.

Considerando as classes e métodos propostos na PEP 249, marque a alternativa correta que completa a seguinte afirmação:

A função global _____ deve ser utilizada para criar uma conexão com o banco de dados. Essa função retorna um objeto do tipo _____, que é utilizado para criar um objeto do tipo _____ utilizando o método _____.

A alternativa "A " está correta.

Os programas que precisam se conectar a um banco de dados que utiliza bibliotecas que seguem a PEP, normalmente seguem o mesmo fluxo de desenvolvimento. A função global connect da biblioteca deve ser utilizada para criar um objeto do tipo Connect. O método cursor da classe Connect retorna um objeto do tipo Cursor, que é utilizado para enviar comandos ao banco de dados.

2. Sobre o banco de dados SQLite e seu conector sqlite3 para Python, assinale a alternativa correta:

A alternativa "C " está correta.

A afinidade definida no SQLite indica a classe preferida para o armazenamento dos dados de uma determinada coluna, porém, diferente de outros bancos de dados, podemos ter dados de classes diversas em uma mesma coluna.

MÓDULO 2

- ⦿ Empregar as funcionalidades para conexão, acesso e criação de bancos de dados e tabelas

CONCEITOS

CONECTANDO A UM BANCO DE DADOS

Neste módulo, vamos aprender a criar e a conectar-se a um banco de dados, criar e editar tabelas e seus relacionamentos.



Fonte: Shutterstock

Como o **SQLite** trabalha com arquivo e não tem suporte à autenticação, para se conectar a um banco de dados **SQLite**, basta chamar a função **connect** do módulo **sqlite3**, passando como argumento o caminho para o arquivo que contém o banco de dados.

Veja a sintaxe a seguir:

```
>>> import sqlite3
```

```
>>> conexao = sqlite3.connect('meu_banco.db')
```

Pronto! Isso é o suficiente para termos uma conexão com o banco de dados meu_banco.db e iniciar o envio de comandos SQL para criar tabelas e inserir registros.

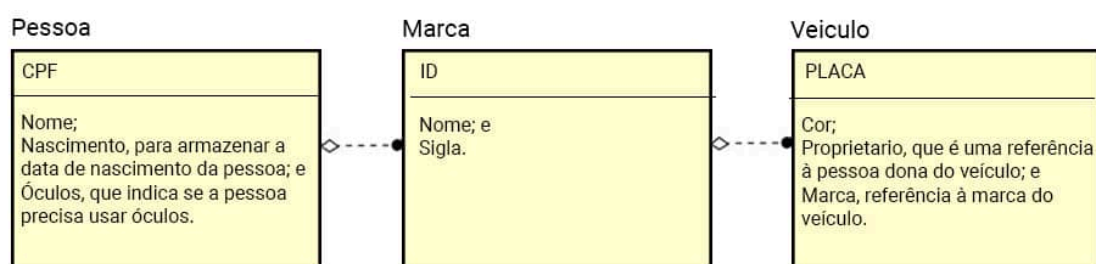
Caso o arquivo não exista, ele será criado automaticamente! O arquivo criado pode ser copiado e compartilhado.

Se quisermos criar um banco de dados em memória, que será criado para toda execução do programa, basta utilizar o comando `conexao = sqlite3.connect(':memory:')`.

CRIANDO TABELAS

Agora que já sabemos como criar e se conectar a um banco de dados SQLite, vamos começar a criar nossas tabelas.

Antes de colocarmos a mão na massa, vamos verificar o nosso modelo entidade relacionamento (ER) que utilizaremos para criar nossas tabelas neste primeiro momento. Veja o modelo na Figura 3.



Fonte: O Autor

📷 Figura: 3.

Nosso modelo é composto por três entidades: Pessoa, Veiculo e Marca.

Nossa primeira entidade se chama Pessoa e contém os atributos:

- cpf, como chave primária;
- nome;

- nascimento, para armazenar a data de nascimento da pessoa; e
- olhos, que indica se a pessoa precisa usar óculos.

A segunda entidade se chama Marca, que contém os atributos:

- id, como chave primária,
- nome; e
- sigla.

Nossa terceira e última entidade se chama Veículo, que contém os atributos:

- placa, como chave primária;
- cor;
- proprietário, que é uma referência à pessoa dona do veículo; e
- marca, referência à marca do veículo.

Para os relacionamentos do nosso modelo, uma pessoa pode ter zero, um ou mais veículos e um veículo só pode ter um proprietário. Uma marca pode estar em zero, um ou mais veículos e um veículo só pode ter uma marca.

Agora que já temos nosso modelo ER, precisamos definir os tipos de cada atributo das nossas entidades. Como estamos trabalhando com SQLite, precisamos ter em mente a tabela de afinidades disponibilizada no módulo anterior.

Vamos definir a nossa entidade Pessoa, de forma que os atributos tenham os seguintes tipos e restrições:

CPF	INTEGER (chave primária, não nulo)
NOME	TEXT (não nulo)
NASCIMENTO	DATE (não nulo)
OCULOS	BOOLEAN (não nulo)

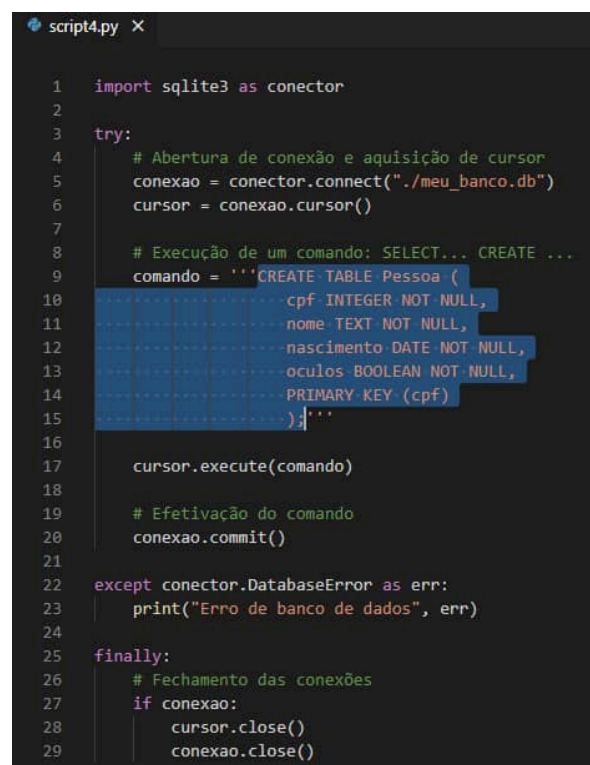
☐ **Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

Para criar uma tabela que represente essa entidade, vamos utilizar o comando SQL:

```
CREATE TABLE Pessoa (  
    cpf INTEGER NOT NULL,  
    nome TEXT NOT NULL,  
    nascimento DATE NOT NULL,  
    olhos BOOLEAN NOT NULL,  
    PRIMARY KEY (cpf)  
);
```

Observe pela nossa tabela de afinidades, que os tipos DATE e BOOLEAN serão convertidos por afinidade para NUMERIC. Na prática, os valores do tipo DATE serão da classe TEXT e os do tipo BOOLEAN da classe INTEGER, pois armazenaremos os valores True como 1 e False como 0.

Definido o comando SQL, vamos ver como criar essa tabela em Python no exemplo da Figura 4 a seguir.



```
script4.py X  
  
1  import sqlite3 as conector  
2  
3  try:  
4      # Abertura de conexão e aquisição de cursor  
5      conexao = conector.connect("./meu_banco.db")  
6      cursor = conexao.cursor()  
7  
8      # Execução de um comando: SELECT... CREATE ...  
9      comando = '''CREATE TABLE Pessoa (  
10         ..... cpf INTEGER NOT NULL,  
11         ..... nome TEXT NOT NULL,  
12         ..... nascimento DATE NOT NULL,  
13         ..... olhos BOOLEAN NOT NULL,  
14         ..... PRIMARY KEY (cpf)  
15         ..... );'''  
16  
17      cursor.execute(comando)  
18  
19      # Efetivação do comando  
20      conexao.commit()  
21  
22  except conector.DatabaseError as err:  
23      print("Erro de banco de dados", err)  
24  
25  finally:  
26      # Fechamento das conexões  
27      if conexao:  
28          cursor.close()  
29          conexao.close()
```

Fonte: O Autor

📷 Figura: 4.

Na linha 1, importamos o módulo sqlite3 e atribuímos o alias conector.

Observe que envolvemos todo o nosso código com a cláusula try/catch, capturando as exceções do tipo DatabaseError.

Na linha 5, criamos uma conexão com o banco de dados meu_banco.db. Caso esse arquivo não exista, será criado um arquivo no mesmo diretório do script sendo

executado.

Na linha 6, criamos um cursor para executar as operações no nosso banco.

Nas linhas 9 a 15, definimos a variável comando, que é uma string contendo o comando SQL para criação da nossa tabela Pessoa.

Na linha 17, utilizamos o método execute do cursor para executar o comando SQL passado como argumento.

Na linha 19, efetivamos a transação pendente utilizando o método commit da variável conexão. Nesse caso, a transação pendente é a criação da tabela.

Nas linhas 22 e 23, capturamos e tratamos a exceção DatabaseError.

Nas linhas 28 e 29 fechamos o cursor e a conexão na cláusula finally, para garantir que nenhuma conexão fique aberta em caso de erro.

Observe como ficou a árvore de diretórios à esquerda da figura após a execução do programa. Veja que agora temos o arquivo meu_banco.db.

ATENÇÃO

Nos exemplos ao longo deste tema, vamos utilizar a mesma estrutura de código do script anterior, porém, vamos omitir as cláusulas try/catch para fins de brevidade.

Agora vamos tratar de nossa próxima entidade, Marca. Vamos defini-la de forma que os atributos tenham os seguintes tipos e restrições:

id	INTEGER (chave primária, não nulo)
nome	TEXT (não nulo)
sigla	CHARACTER (não nulo, tamanho 2)

☐ **Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

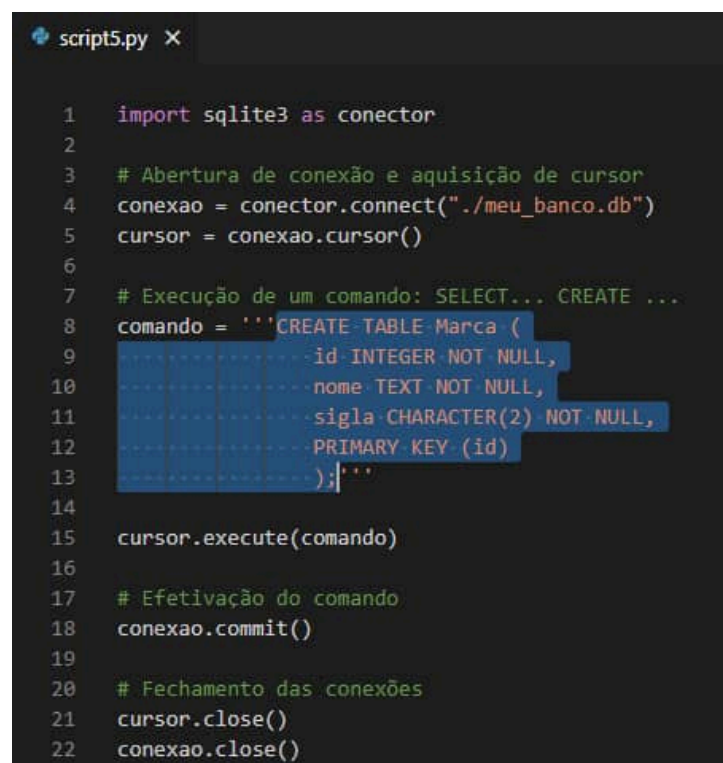
A codificação latin-1, muito utilizada no Brasil, utiliza um byte por caractere. Como a sigla da marca será composta por 2 caracteres, nosso atributo sigla terá tamanho 2 (CHAR(2) ou CHARACTER(2)).

Para criar uma tabela que represente essa entidade, vamos utilizar o comando SQL:

```
CREATE TABLE Marca (  
id INTEGER NOT NULL,  
nome TEXT NOT NULL,  
sigla CHARACTER(2) NOT NULL,  
PRIMARY KEY (id)  
);
```

Pela tabela de afinidades, o tipo CHARACTER(2) será convertido para TEXT.

Confira a imagem a seguir, Figura 5, para verificar como ficou o script de criação dessa tabela.



```
script5.py X  
  
1  import sqlite3 as conector  
2  
3  # Abertura de conexão e aquisição de cursor  
4  conexao = conector.connect("./meu_banco.db")  
5  cursor = conexao.cursor()  
6  
7  # Execução de um comando: SELECT... CREATE ...  
8  comando = '''CREATE TABLE Marca (  
9      .....id INTEGER NOT NULL,  
10     .....nome TEXT NOT NULL,  
11     .....sigla CHARACTER(2) NOT NULL,  
12     .....PRIMARY KEY (id)  
13     .....);'''  
14  
15  cursor.execute(comando)  
16  
17  # Efetivação do comando  
18  conexao.commit()  
19  
20  # Fechamento das conexões  
21  cursor.close()  
22  conexao.close()
```

Fonte: O Autor

📷 Figura: 5.

Após a criação da conexão e do cursor, linhas 4 e 5, definimos uma string com o comando SQL para criação da tabela Marca, linhas 8 a 13.

O comando foi executado na linha 15 e efetivado na linha 18.

Nas linhas 21 e 22, fechamos o cursor e a conexão.

A próxima entidade a ser criada será a entidade Veiculo. Os seus atributos terão os seguintes tipos e restrições.

PLACA	CHARACTER (chave primária, não nulo, tamanho 7)
ANO	INTEGER (não nulo)
COR	TEXT (não nulo)
PROPRIETÁRIO	INTEGER (chave estrangeira, não nulo)
MARCA	INTEGER (chave estrangeira, não nulo)

☐ **Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

Como nossas placas são compostas por 7 caracteres, nosso atributo placa terá tamanho 7 (CHAR(7) ou CHARACTER(7)). Por afinidade, ele será convertido para TEXT.

O atributo proprietario será utilizado para indicar um relacionamento da entidade Veiculo com a entidade Pessoa. O atributo da tabela Pessoa utilizado no relacionamento será o CPF. Como o CPF é um INTEGER, o atributo relacionado proprietario também precisa ser INTEGER.

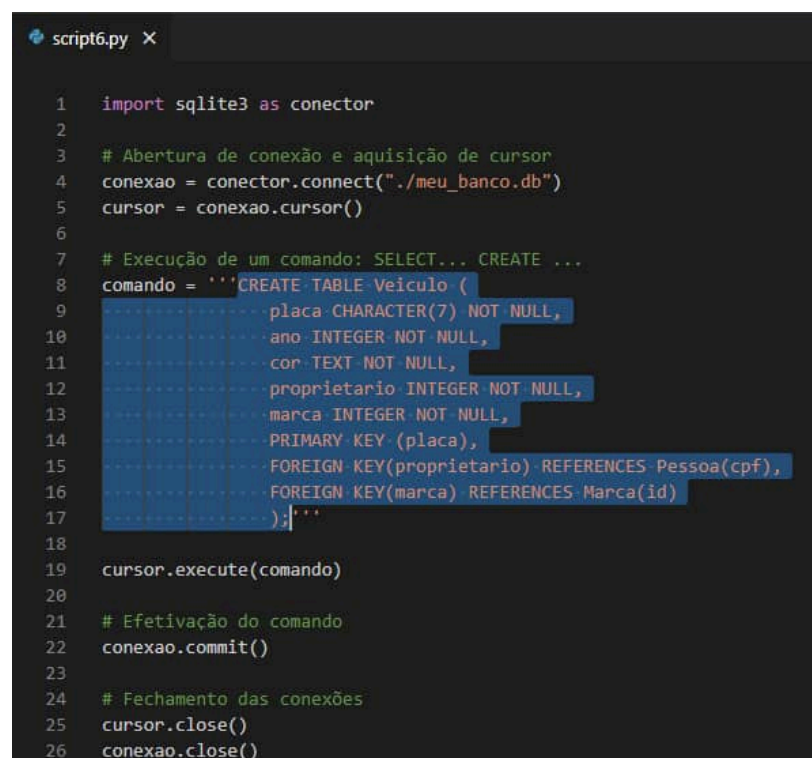
Analogamente, o atributo marca será utilizado para indicar um relacionamento da entidade Veiculo com a entidade Marca, por meio do atributo id da tabela Marca, que também é um INTEGER.

Veja o comando SQL a seguir para criar a tabela Veiculo e o relacionamento com as tabelas Pessoa e Marca.

```
CREATE TABLE Veiculo (  
placa CHARACTER(7) NOT NULL,  
ano INTEGER NOT NULL,  
cor TEXT NOT NULL,
```

```
proprietario INTEGER NOT NULL,  
marca INTEGER NOT NULL,  
PRIMARY KEY (placa),  
FOREIGN KEY(proprietario) REFERENCES Pessoa(cpf),  
FOREIGN KEY(marca) REFERENCES Marca(id)  
);
```

Definido o comando SQL, vamos ver como criar essa tabela em Python no exemplo da Figura 6 a seguir.



```
script6.py X  
  
1  import sqlite3 as conector  
2  
3  # Abertura de conexão e aquisição de cursor  
4  conexao = conector.connect("./meu_banco.db")  
5  cursor = conexao.cursor()  
6  
7  # Execução de um comando: SELECT... CREATE ...  
8  comando = '''CREATE TABLE Veiculo (  
9      ..... placa CHARACTER(7) NOT NULL,  
10     ..... ano INTEGER NOT NULL,  
11     ..... cor TEXT NOT NULL,  
12     ..... proprietario INTEGER NOT NULL,  
13     ..... marca INTEGER NOT NULL,  
14     ..... PRIMARY KEY (placa),  
15     ..... FOREIGN KEY(proprietario) REFERENCES Pessoa(cpf),  
16     ..... FOREIGN KEY(marca) REFERENCES Marca(id)  
17     ..... );'''  
18  
19  cursor.execute(comando)  
20  
21  # Efetivação do comando  
22  conexao.commit()  
23  
24  # Fechamento das conexões  
25  cursor.close()  
26  conexao.close()
```

Fonte: O Autor

📷 Figura: 6.

Este script também segue os mesmos passos do script anterior até a linha 8, onde definimos a string com o comando SQL para criação da tabela Veiculo.

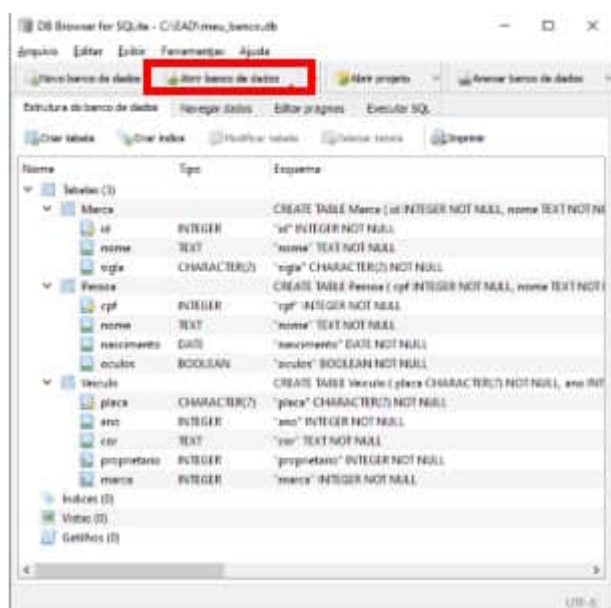
Esse comando foi executado na linha 19 e efetivado na linha 22.

💡 DICA

Caso a referência da chave estrangeira seja feita a um atributo inexistente, será lançado um erro de programação: `ProgrammingError`.

Para visualizar como ficaram nossas tabelas, vamos utilizar o programa DB Browser for SQLite.

Após abrir o programa DB Browser for SQLite, basta clicar em Abrir banco de dados e selecionar o arquivo meu_banco.db. Observe a Figura 7 a seguir.



Fonte: O Autor

📷 Figura: 7.

Observe que está tudo conforme o previsto, inclusive a ordem dos atributos obedece a sequência na qual foram criados.

ALTERAÇÃO E REMOÇÃO DE TABELA

Neste momento, temos o nosso banco com as três tabelas exibidas no modelo ER da Figura 3.

Durante o desenvolvimento, pode ser necessário realizar alterações no nosso modelo e, consequentemente, nas nossas tabelas. Nesta parte do módulo, vamos ver como podemos fazer para adicionar um novo atributo e como fazemos para remover uma tabela.

Para alterar uma tabela e adicionar um novo atributo, precisamos utilizar o comando **ALTER TABLE** do SQL.

Para ilustrar, vamos adicionar mais um atributo à entidade Veiculo.

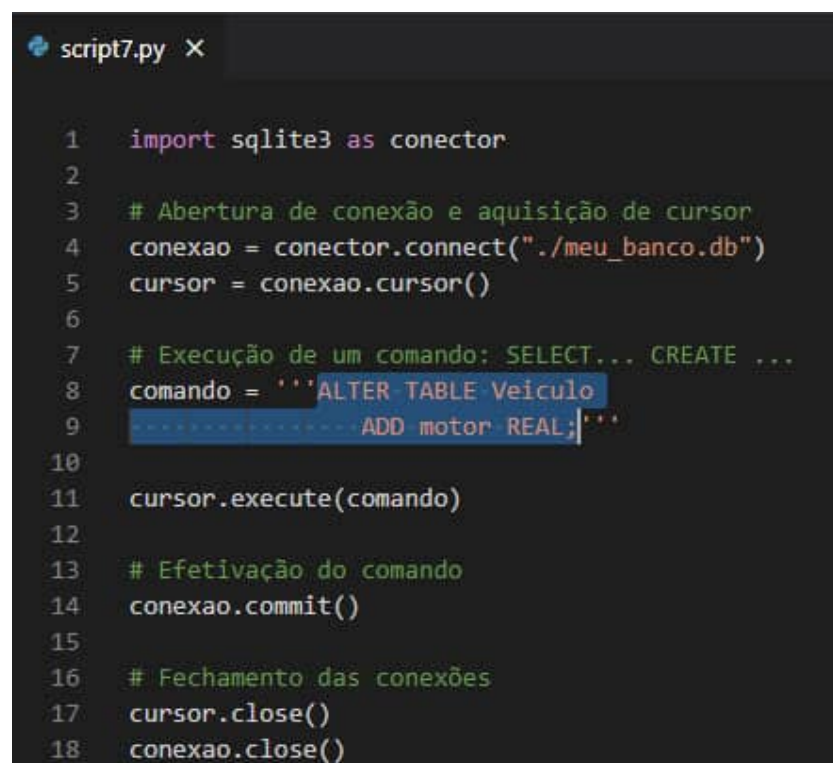
O atributo se chama motor e corresponde à motorização do carro: 1.0, 1.4, 2.0 etc. Esse atributo deverá conter pontos flutuantes e, por isso, vamos defini-lo como do tipo REAL.

Para alterar a tabela Veículo e adicionar a coluna motor, utilizamos o seguinte comando SQL.

ALTER TABLE Veiculo

ADD motor REAL;

Confira o script da Figura 8, onde realizamos a alteração da tabela Veiculo.



```
1  import sqlite3 as conector
2
3  # Abertura de conexão e aquisição de cursor
4  conexao = conector.connect("./meu_banco.db")
5  cursor = conexao.cursor()
6
7  # Execução de um comando: SELECT... CREATE ...
8  comando = '''ALTER TABLE Veiculo
9             ADD motor REAL;'''
10
11 cursor.execute(comando)
12
13 # Efetivação do comando
14 conexao.commit()
15
16 # Fechamento das conexões
17 cursor.close()
18 conexao.close()
```

Fonte: O Autor

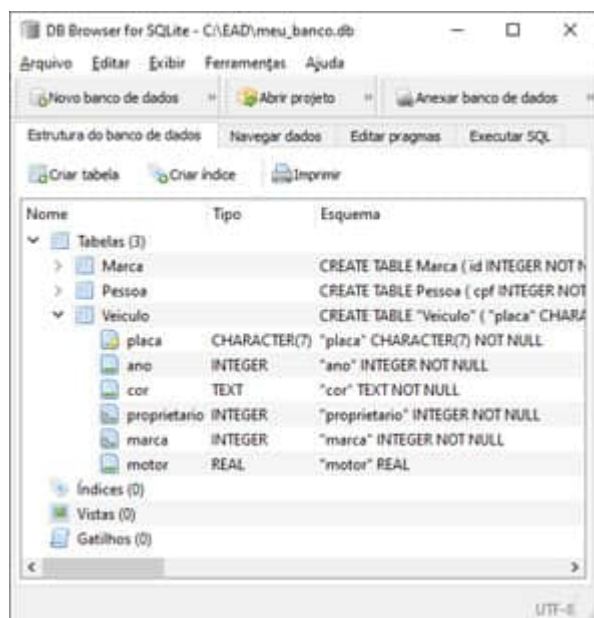
📷 Figura: 8.

Após se conectar ao banco e obter um cursor, linhas 4 e 5, construímos uma string com o comando ALTER TABLE nas linhas 8 e 9.

Na linha 11, executamos o comando e na linha 14 efetivamos a modificação.

Nas linhas 17 e 18, fechamos o cursor e a conexão.

Observe como ficou nossa tabela após a criação da nova coluna.



Fonte: O Autor

📷 Figura: 9.

Veja, pela Figura 9, que o atributo motor foi adicionado ao final da entidade, obedecendo a ordem de criação.

★ EXEMPLO

Em algumas situações, pode ser necessário que as colunas sigam uma ordem predeterminada. Um exemplo é quando precisamos carregar os dados de uma planilha diretamente para um banco de dados, para realizarmos o chamado bulk insert (inserção em massa). Nesses casos, as colunas da planilha precisam estar na mesma sequência das colunas do banco.

Como nem todos os bancos de dados, incluindo o SQLite, dão suporte à criação de colunas em posição determinada, vamos precisar remover nossa tabela para recriá-la com os atributos na posição desejada.

Para o nosso exemplo, desejamos a seguinte sequência: placa, ano, cor, motor, proprietario e marca.

No exemplo da Figura 10, vamos remover a tabela Veiculo, utilizando o comando **DROP TABLE** do SQL e, posteriormente, vamos criá-la novamente com a sequência desejada.

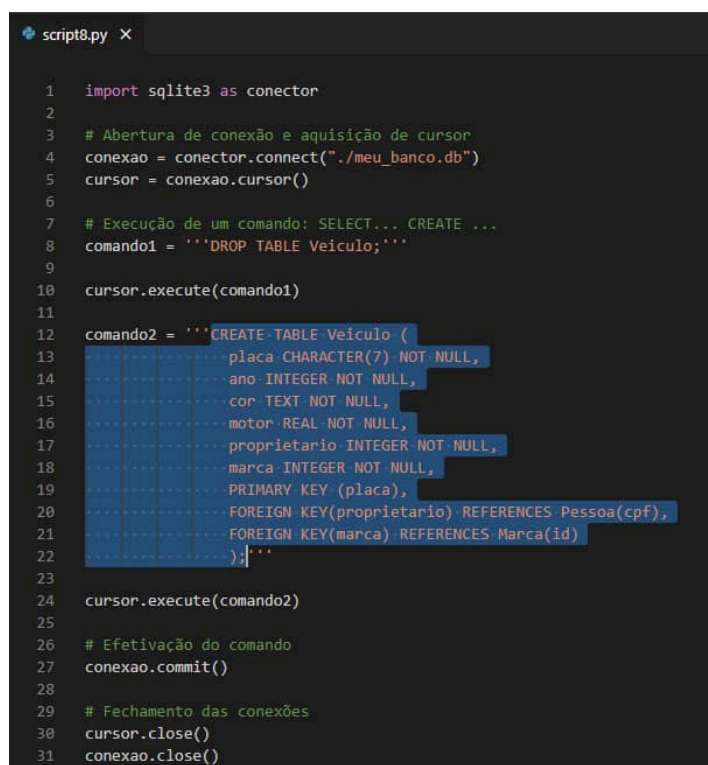
Para remover a tabela Veiculo, utilizamos o seguinte comando SQL.

DROP TABLE Veiculo;

Para recriar a tabela, utilizamos o seguinte comando SQL:

```
CREATE TABLE Veiculo (  
placa CHARACTER(7) NOT NULL,  
ano INTEGER NOT NULL,  
cor TEXT NOT NULL,  
motor REAL NOT NULL,  
proprietario INTEGER NOT NULL,  
marca INTEGER NOT NULL,  
PRIMARY KEY (placa),  
FOREIGN KEY(proprietario) REFERENCES Pessoa(cpf),  
FOREIGN KEY(marca) REFERENCES Marca(id)  
);
```

Confira como ficou nosso script na Figura 10 a seguir:



```
script8.py x  
1  import sqlite3 as conector  
2  
3  # Abertura de conexão e aquisição de cursor  
4  conexao = conector.connect("./meu_banco.db")  
5  cursor = conexao.cursor()  
6  
7  # Execução de um comando: SELECT... CREATE ...  
8  comando1 = '''DROP TABLE Veiculo;'''  
9  
10 cursor.execute(comando1)  
11  
12 comando2 = '''CREATE TABLE Veiculo (  
13 .....placa CHARACTER(7) NOT NULL,  
14 .....ano INTEGER NOT NULL,  
15 .....cor TEXT NOT NULL,  
16 .....motor REAL NOT NULL,  
17 .....proprietario INTEGER NOT NULL,  
18 .....marca INTEGER NOT NULL,  
19 .....PRIMARY KEY (placa),  
20 .....FOREIGN KEY(proprietario) REFERENCES Pessoa(cpf),  
21 .....FOREIGN KEY(marca) REFERENCES Marca(id)  
22 .....);'''  
23  
24 cursor.execute(comando2)  
25  
26 # Efetivação do comando  
27 conexao.commit()  
28  
29 # Fechamento das conexões  
30 cursor.close()  
31 conexao.close()
```

Fonte: O Autor

📷 Figura: 10.

Após se conectar ao banco e obter o cursor, criamos a string comando1 com o comando para remover a tabela Veiculo, na linha 8. Na linha 10, executamos esse comando.

Nas linhas 12 a 22, criamos o comando para criar novamente a tabela Veiculo com os atributos na ordem mostrada anteriormente e, na linha 24, executamos esse comando.

Na linha 27, efetivamos todas as modificações pendentes, tanto a remoção da tabela, quanto a criação. Observe que não é preciso efetuar um commit para cada comando!

Nas linhas 30 e 31, liberamos o cursor e fechamos a conexão.

Chegamos ao final do módulo 2 e agora nosso modelo ER está conforme a Figura 11 a seguir:



Fonte: O Autor

📷 Figura: 11.

⊕ SAIBA MAIS

Algumas bibliotecas de acesso a banco de dados oferecem uma funcionalidade chamada mapeamento objeto-relacional, do inglês object-relational mapping (ORM).

ORM

ORM (Object Relational Mapper) é uma mapeamento objeto-relacional, isto é uma técnica de mapeamento objeto relacional que permite fazer uma relação dos objetos com os dados que os mesmos representam.

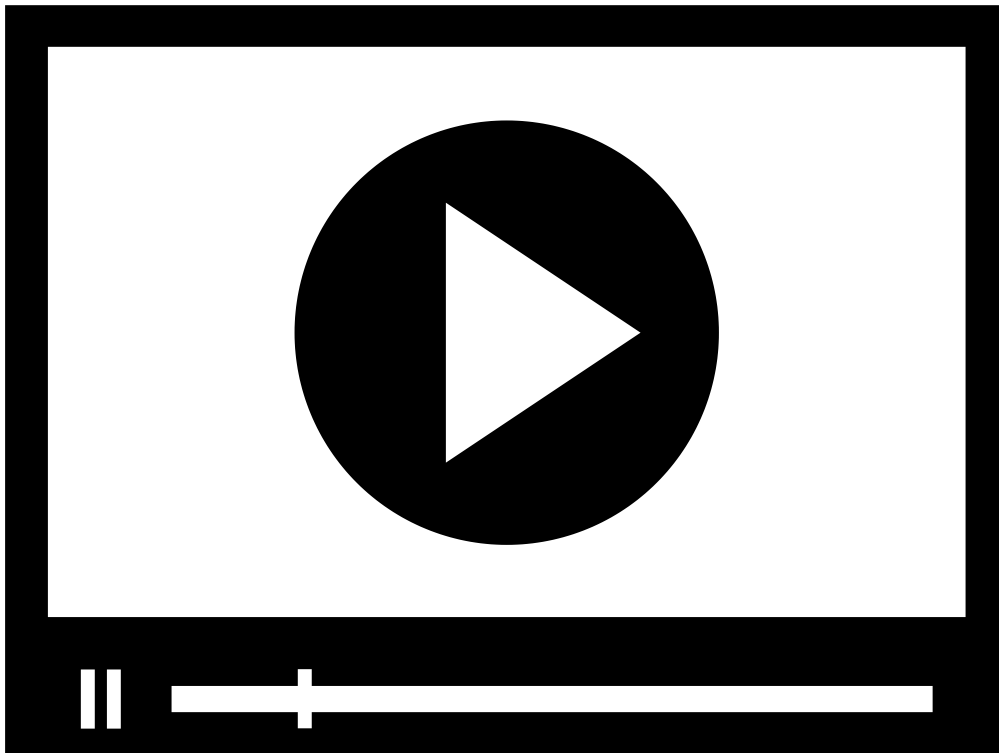
Esse mapeamento permite associar classes definidas em Python com tabelas em banco de dados, onde cada objeto dessas classes corresponde a um registro da tabela.

Os comandos SQL de inserções e consultas são todos realizados por meio de métodos, não sendo necessário escrever o comando SQL em si. Os ORM nos permitem trabalhar com um nível mais alto de abstração.

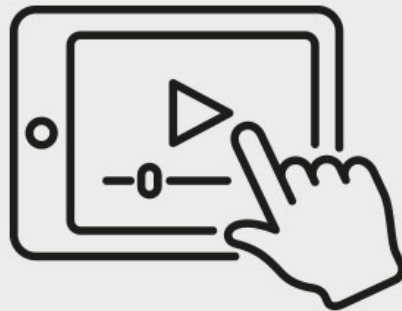
Como exemplo dessas bibliotecas, podemos citar SQLAlchemy e Peewee.

Visite as páginas dessas bibliotecas e veja como elas facilitam a manipulação de registros em banco de dados.

No próximo módulo, vamos mostrar como inserir e atualizar os dados em uma tabela.



Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. CONSIDERE QUE PRECISAMOS ESCREVER UM PROGRAMA PARA SE CONECTAR A UM BANCO DE DADOS RELACIONAL E CRIAR A ENTIDADE

CIDADE, COM OS ATRIBUTOS A SEGUIR:

ID: INTEIRO, NÃO NULO

NOME: TEXTO

ESTADO: TEXTO

DADO QUE UTILIZAMOS O CONECTOR SQLITE3, SELECIONE O SCRIPT QUE EXECUTA TODAS AS OPERAÇÕES NECESSÁRIAS CORRETAMENTE:

```
atividade2-a.py
1 import sqlite3 as conector
2
3 conexao = conector.connect("banco.db")
4 cursor = conexao.cursor()
5
6 comando = '''CREATE Cidade (
7     id INTEGER NOT NULL,
8     nome TEXT,
9     estado TEXT
10 );'''
11
12 cursor.execute(comando)
13 cursor.close()
14 conexao.close()
15

atividade2-b.py
1 import sqlite3 as conector
2
3 conexao = conector.connect("banco.db")
4 cursor = conexao.cursor()
5
6 comando = '''CREATE TABLE Cidade (
7     id INTEGER NOT NULL,
8     nome TEXT,
9     estado TEXT
10 );'''
11
12 conexao.execute(comando)
13 conexao.commit()
14 conexao.close()
15

atividade2-c.py
1 import sqlite3 as conector
2
3 conexao = conector.connect("banco.db")
4 cursor = conexao.cursor()
5
6 comando = '''CREATE TABLE Cidade (
7     id INTEGER NOT NULL,
8     nome TEXT,
9     estado TEXT
10 );'''
11
12 cursor.execute(comando)
13 conexao.commit()
14 cursor.close()
15 conexao.close()
16

atividade2-d.py
1 import sqlite3 as conector
2
3 conexao = conector.connect("banco.db")
4 cursor = conexao.cursor()
5
6 comando = '''CREATE TABLE Cidade (
7     id INTEGER NOT NULL,
8     nome TEXT,
9     estado TEXT
10 );'''
11
12 conexao.commit(comando)
13 cursor.close()
14 conexao.close()
```

FONTE: O AUTOR

A) script atividade2-a.py

B) script atividade2-b.py

C) script atividade2-c.py

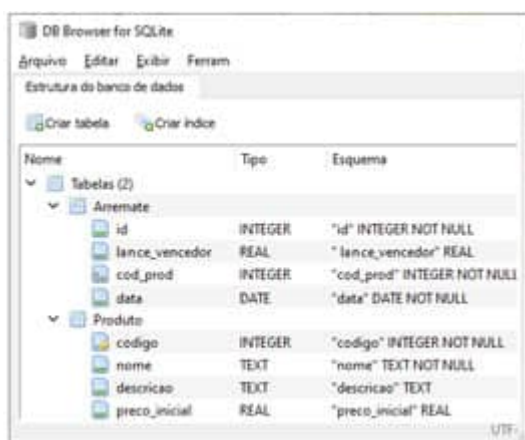
D) script atividade2-d.py

2. CONSIDERE O SCRIPT A SEGUIR, ONDE CRIAMOS AS TABELAS PRODUTO E ARREIMATE E O RELACIONAMENTO ENTRE ELAS.

```
atividade1.py
1 import sqlite3 as conector
2 conexao = conector.connect("banco.db")
3 cursor = conexao.cursor()
4
5 comando1 = '''CREATE TABLE Produto (
6     codigo INTEGER NOT NULL,
7     nome TEXT NOT NULL,
8     PRIMARY KEY (codigo));'''
9 cursor.execute(comando1)
10
11 comando2 = '''CREATE TABLE Arremate (
12     id INTEGER NOT NULL,
13     data DATE NOT NULL,
14     cod_prod INTEGER NOT NULL,
15     FOREIGN KEY(cod_prod) REFERENCES Produto(codigo));'''
16 cursor.execute(comando2)
17
18 cursor.execute('ALTER TABLE Produto ADD descricao TEXT;')
19 cursor.execute('ALTER TABLE Produto ADD preco_inicial REAL;')
20 cursor.execute('ALTER TABLE Arremate ADD lance_vencedor REAL;')
21
22 conexao.commit()
23 cursor.close()
24 conexao.close()
```

FONTE: O AUTOR

MARQUE A ALTERNATIVA QUE CORRESPONDE ÀS TABELAS CRIADAS PELO SCRIPT.

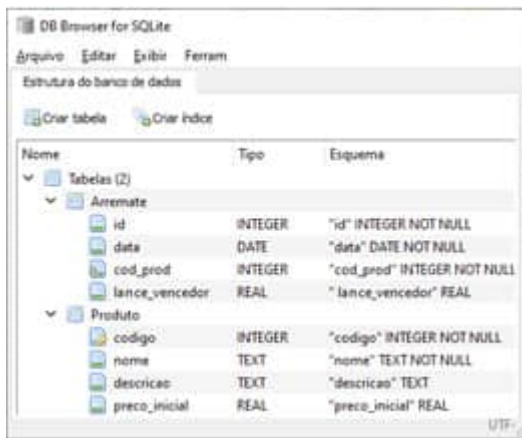


The screenshot shows the 'DB Browser for SQLite' application. The 'Estrutura do banco de dados' (Database Structure) tab is active, displaying a tree view of the database schema. Two tables are listed: 'Arremate' and 'Produto'. The 'Arremate' table has four columns: 'id' (INTEGER), 'lance_vencedor' (REAL), 'cod_prod' (INTEGER), and 'data' (DATE). The 'Produto' table has four columns: 'codigo' (INTEGER), 'nome' (TEXT), 'descricao' (TEXT), and 'preco_inicial' (REAL). The 'cod_prod' column in 'Arremate' is marked as a foreign key referencing the 'codigo' column in 'Produto'.

Nome	Tipo	Esquema
Tabelas (2)		
Arremate		
id	INTEGER	"id" INTEGER NOT NULL
lance_vencedor	REAL	"lance_vencedor" REAL
cod_prod	INTEGER	"cod_prod" INTEGER NOT NULL
data	DATE	"data" DATE NOT NULL
Produto		
codigo	INTEGER	"codigo" INTEGER NOT NULL
nome	TEXT	"nome" TEXT NOT NULL
descricao	TEXT	"descricao" TEXT
preco_inicial	REAL	"preco_inicial" REAL

A)

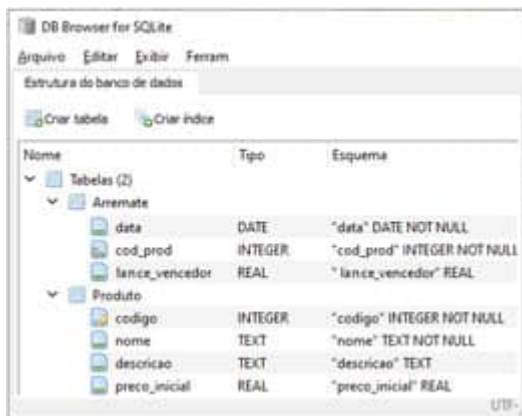
Fonte: O autor



Nome	Tipo	Esquema
Anemate		
id	INTEGER	"id" INTEGER NOT NULL
data	DATE	"data" DATE NOT NULL
cod_prod	INTEGER	"cod_prod" INTEGER NOT NULL
lance_vencedor	REAL	"lance_vencedor" REAL
Produto		
codigo	INTEGER	"codigo" INTEGER NOT NULL
nome	TEXT	"nome" TEXT NOT NULL
descricao	TEXT	"descricao" TEXT
preco_inicial	REAL	"preco_inicial" REAL

B)

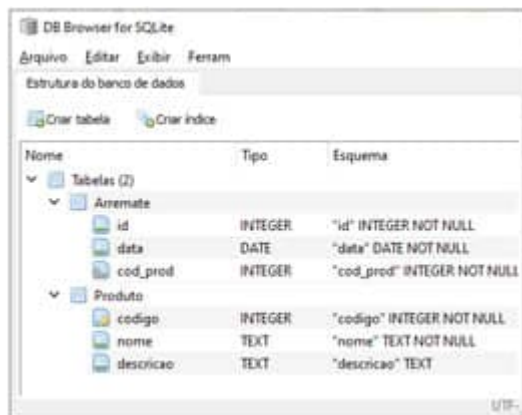
Fonte: O autor



Nome	Tipo	Esquema
Anemate		
data	DATE	"data" DATE NOT NULL
cod_prod	INTEGER	"cod_prod" INTEGER NOT NULL
lance_vencedor	REAL	"lance_vencedor" REAL
Produto		
codigo	INTEGER	"codigo" INTEGER NOT NULL
nome	TEXT	"nome" TEXT NOT NULL
descricao	TEXT	"descricao" TEXT
preco_inicial	REAL	"preco_inicial" REAL

C)

Fonte: O autor



Nome	Tipo	Esquema
Anemate		
id	INTEGER	"id" INTEGER NOT NULL
data	DATE	"data" DATE NOT NULL
cod_prod	INTEGER	"cod_prod" INTEGER NOT NULL
Produto		
codigo	INTEGER	"codigo" INTEGER NOT NULL
nome	TEXT	"nome" TEXT NOT NULL
descricao	TEXT	"descricao" TEXT

D)

Fonte: O autor

GABARITO

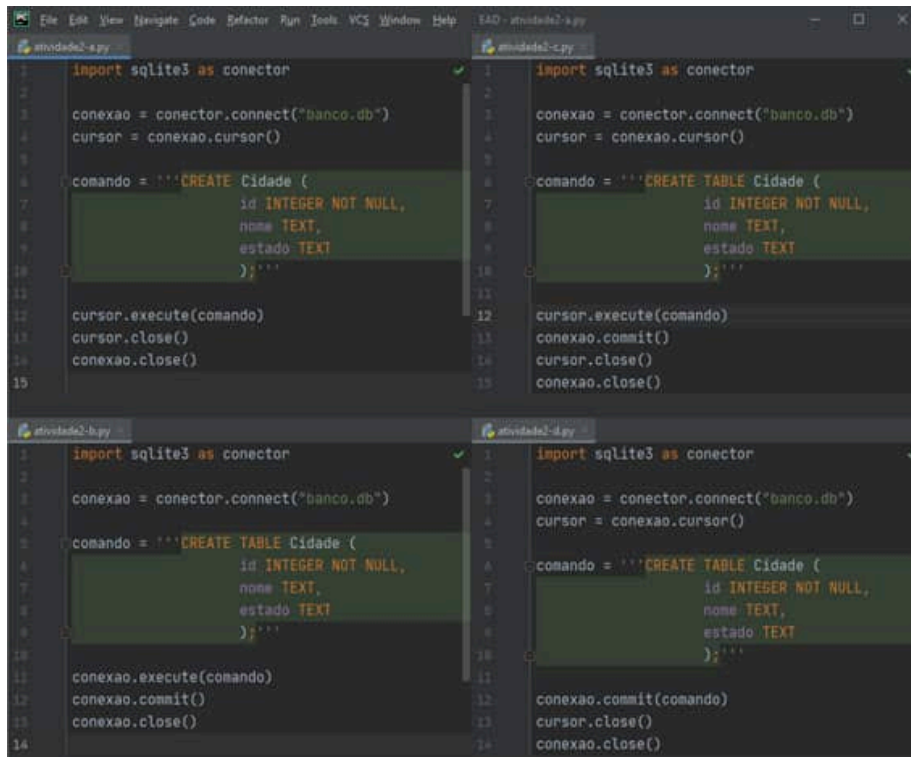
1. Considere que precisamos escrever um programa para se conectar a um banco de dados relacional e criar a entidade Cidade, com os atributos a seguir:

id: inteiro, não nulo

nome: texto

estado: texto

Dado que utilizamos o conector sqlite3, selecione o script que executa todas as operações necessárias corretamente:



Fonte: O Autor

A alternativa "C " está correta.

No script (c) temos todos os passos necessários para criar uma tabela. Primeiro, é necessário criar uma conexão (linha 3) e, em seguida, um cursor (linha 4). Na sequência, precisamos definir o comando SQL de criar tabela (linhas 6 a 10) e utilizar o cursor para executar esse comando (linha 12). Após a execução do comando, precisamos efetivar a transação utilizando o método commit da conexão (linha 13). Posteriormente, fechamos o cursor e então fechamos a conexão.

2. Considere o script a seguir, onde criamos as tabelas Produto e Arremate e o relacionamento entre elas.

```
1 import sqlite3 as conector
2 conexao = conector.connect("banco.db")
3 cursor = conexao.cursor()
4
5 comando1 = '''CREATE TABLE Produto (
6     codigo INTEGER NOT NULL,
7     nome TEXT NOT NULL,
8     PRIMARY KEY (codigo));'''
9 cursor.execute(comando1)
10
11 comando2 = '''CREATE TABLE Arremate (
12     id INTEGER NOT NULL,
13     data DATE NOT NULL,
14     cod_prod INTEGER NOT NULL,
15     FOREIGN KEY(cod_prod) REFERENCES Produto(codigo));'''
16 cursor.execute(comando2)
17
18 cursor.execute('ALTER TABLE Produto ADD descricao TEXT;')
19 cursor.execute('ALTER TABLE Produto ADD preco_inicial REAL;')
20 cursor.execute('ALTER TABLE Arremate ADD lance_vencedor REAL;')
21
22 conexao.commit()
23 cursor.close()
24 conexao.close()
```

Fonte: O Autor

Marque a alternativa que corresponde às tabelas criadas pelo script.

A alternativa "B " está correta.

O script cria, inicialmente, a tabela Produto, com os atributos: codigo e nome (linhas 5 a 8). Posteriormente, alteramos a tabela para criar mais dois atributos: descricao, preco_inicial (linhas 18 e 19). A tabela Arremate foi criada inicialmente com os atributos id, data e cod_prod (linhas 11 a 15) e posteriormente foi adicionado o atributo lance_vencedor (linha 20). Lembre-se de que o SQLite obedece à ordem de criação dos atributos. A única opção que contém todos os atributos ordenados é a opção b.

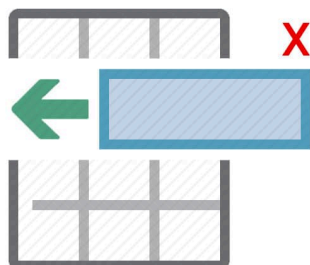
MÓDULO 3

⦿ Aplicar as funcionalidades para inserção, remoção e atualização de registros em tabelas

CONCEITOS

INSERÇÃO DE DADOS EM TABELA

Neste módulo, vamos aprender a inserir, alterar e remover registros de tabelas. Para inserir registros em uma tabela, utilizamos o comando **INSERT INTO**, do SQL.



Para ilustrar a utilização desse comando, vamos inserir o seguinte registro na tabela **Pessoa**.

CPF: 12345678900

Nome: João

Data de Nascimento: 31/01/2000

Usa óculos: Sim (True)

O Comando SQL para inserção desses dados é o seguinte:

```
INSERT INTO Pessoa (cpf, nome, nascimento, olhos)
```

```
VALUES (12345678900, 'João', '2000-01-31', 1);
```

Observe que alteramos a formatação da data para se adequar ao padrão de alguns bancos de dados, como MySQL e PostgreSQL. Para o SQLite será indiferente, pois o tipo **DATE** será convertido por afinidade para **NUMERIC**, que pode ser de qualquer classe. Na prática, será convertido para classe **TEXT**.

Além disso, utilizamos o valor “1” para representar o booleano **True**. Assim como o **DATE**, o **BOOLEAN** será convertido para **NUMERIC**, porém, na prática, será convertido para classe **INTEGER**.

Confira como ficou o script para inserção dos dados, Figura 12.

```

script9.py X

1  import sqlite3 as conector
2
3  # Abertura de conexão e aquisição de cursor
4  conexao = conector.connect("./meu_banco.db")
5  cursor = conexao.cursor()
6
7  # Execução de um comando: SELECT... CREATE ...
8  comando = '''INSERT INTO Pessoa (cpf, nome, nascimento, olhos)
9             VALUES (12345678900, 'João', '2000-01-31', 1);'''
10
11  cursor.execute(comando)
12
13  # Efetivação do comando
14  conexao.commit()
15
16  # Fechamento das conexões
17  cursor.close()
18  conexao.close()

```

Fonte: O Autor

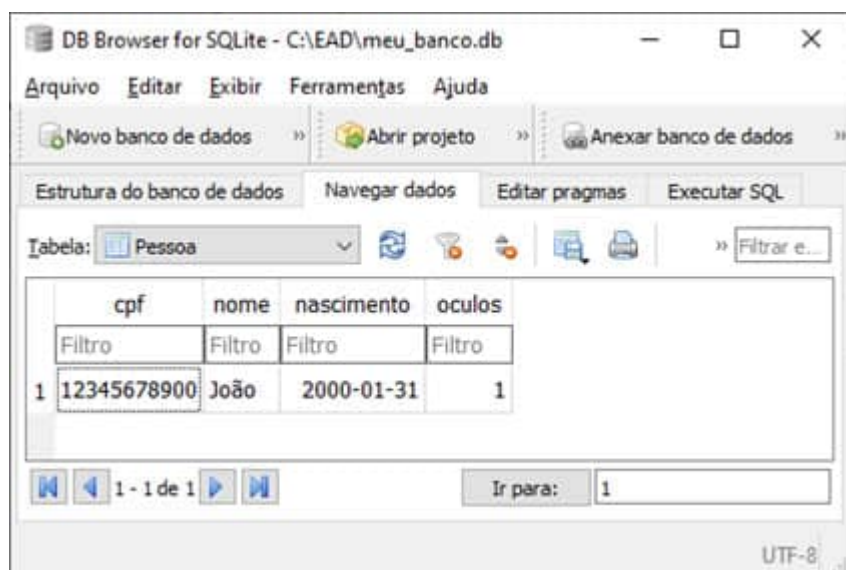
📷 Figura: 12.

Após se conectar ao banco e obter o cursor, criamos a string com o comando para inserir um registro na tabela Pessoa nas linhas 8 e 9.

Na linha 11, executamos esse comando com a execução do commit na linha 14.

Ao final do script, fechamos o cursor e a conexão.

Observe na figura 13 como ficou nossa tabela pelo DB Browser for SQLite.



The screenshot shows the DB Browser for SQLite interface. The title bar indicates the database file is 'C:\EAD\meu_banco.db'. The 'Executar SQL' tab is active. The table 'Pessoa' is selected, and its structure is displayed as follows:

cpf	nome	nascimento	olhos
12345678900	João	2000-01-31	1

At the bottom, the status bar shows '1 - 1 de 1' and 'Ir para: 1'.

Fonte: O Autor

📷 Figura: 13.

INSERÇÃO DE DADOS EM TABELA COM QUERIES DINÂMICAS

É muito comum reutilizarmos uma mesma string para inserir diversos registros em uma tabela, alterando apenas os dados a serem inseridos.

Para realizar esse tipo de operação, o método `execute`, da classe `Cursor`, prevê a utilização de parâmetros de consulta, que é uma forma de criar comandos SQL dinamicamente.

COMENTÁRIO

De uma forma geral, as APIs `sqlite3`, `psycopg2` e `PyMySQL` fazem a concatenação da string e dos parâmetros antes de enviar ao banco de dados.

Essa concatenação é realizada de forma correta, evitando brechas de segurança, como SQL Injection, e convertendo os dados para os tipos e formatos esperados pelo banco de dados.

Como resultado final, temos um comando pronto para ser enviado ao banco de dados.

Para indicar que a string de um comando contém parâmetros que precisam ser substituídos antes da sua execução, utilizamos delimitadores. Esses delimitadores também estão previstos na PEP 249 e podem ser: `"?"`, `"%s"`, entre outros.

Na biblioteca do SQLite, utilizamos o delimitador `"?"`.

Para ilustrar a utilização do delimitador `"?"` em SQLite, considere o comando a seguir:

```
>>> comando = "INSERT INTO tabela1 (atributo1, atributo2) VALUES (?, ?);"
```

Esse comando indica que, ao ser chamado pelo método `execute`, devemos passar dois parâmetros, um para cada interrogação. Esses parâmetros precisam estar em um iterável, como em uma tupla ou lista.

Veja a seguir como poderia ficar a chamada do método `execute` para esse comando:

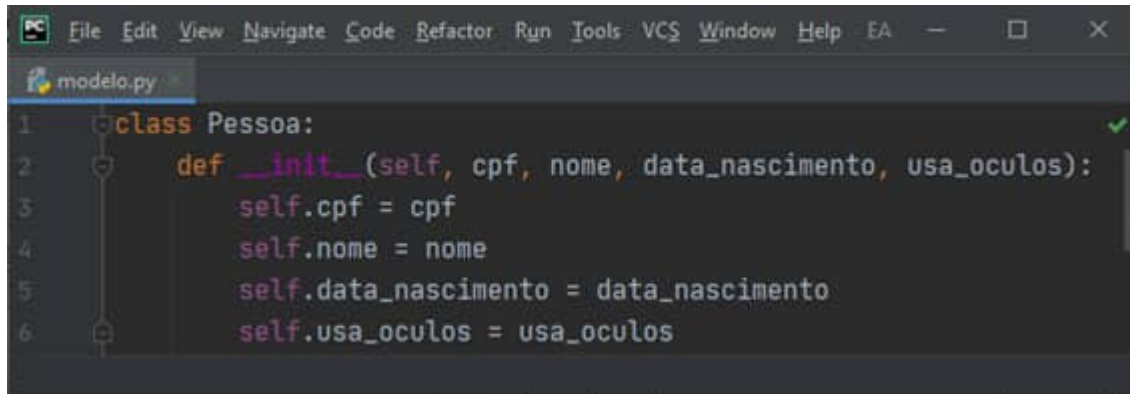
```
>>> cursor.execute(comando, ("Teste", 123))
```

A partir da string e da tupla, é montado o comando final, que é traduzido para:

“INSERT INTO tabela1 (atributo1, atributo2) VALUES ('Teste', 123);”

A concatenação é feita da forma correta para o banco de dados em questão, aspas simples para textos e números sem aspas.

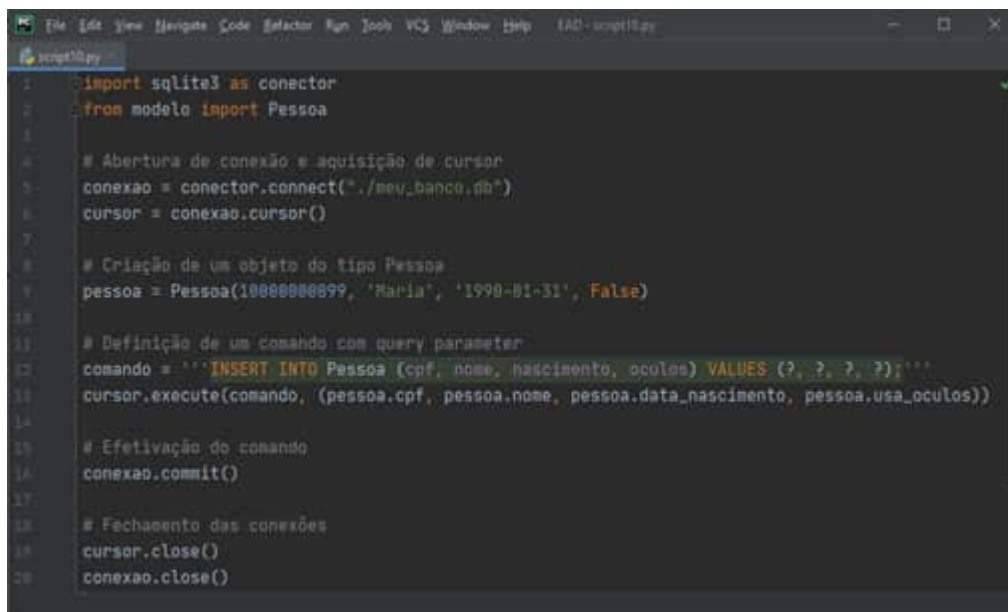
No exemplo a seguir, vamos detalhar a utilização de parâmetros dinâmicos, porém, antes, vamos definir uma classe chamada Pessoa, com os mesmos atributos da nossa entidade Pessoa.



```
1 class Pessoa:
2     def __init__(self, cpf, nome, data_nascimento, usa_olhos):
3         self.cpf = cpf
4         self.nome = nome
5         self.data_nascimento = data_nascimento
6         self.usa_olhos = usa_olhos
```

autor

A definição dessa classe pode ser vista no script modelo.py. Foi retirado a referência para a figura.



```
1 import sqlite3 as conector
2 from modelo import Pessoa
3
4 # Abertura de conexão e aquisição de cursor
5 conexao = conector.connect("./meu_banco.db")
6 cursor = conexao.cursor()
7
8 # Criação de um objeto do tipo Pessoa
9 pessoa = Pessoa(10000000000, 'Maria', '1990-01-31', False)
10
11 # Definição de um comando com query parameter
12 comando = 'INSERT INTO Pessoa (cpf, nome, nascimento, olhos) VALUES (?, ?, ?, ?);'
13 cursor.execute(comando, (pessoa.cpf, pessoa.nome, pessoa.data_nascimento, pessoa.usa_olhos))
14
15 # Efetivação do comando
16 conexao.commit()
17
18 # Fechamento das conexões
19 cursor.close()
20 conexao.close()
```

autor

Observe que temos os mesmos atributos que a entidade equivalente, mas com nomes diferentes. Fizemos isso para facilitar o entendimento dos exemplos deste módulo.

Confira agora o script10.

Primeiro, importamos o conector na linha 1 e, na linha 2, importamos a classe Pessoa. Ela será utilizada para criar um objeto do tipo Pessoa.

Nas linhas 5 e 6, conectamos ao banco de dados e criamos um cursor.

Na linha 9, utilizamos o construtor da classe Pessoa para criar um objeto com os seguintes atributos: CPF: 10000000099, Nome: Maria, Data de Nascimento: 31/01/1990 e Usa óculos: Não (False). O valor False será convertido para 0 durante a execução do método execute.

Na linha 12, definimos a string que conterá o comando para inserir um registro na tabela Pessoa. Observe como estão representados os valores dos atributos! Estão todos com o delimitador representado pelo caractere interrogação (!)?

Como dito anteriormente, isso serve para indicar ao método execute que alguns parâmetros serão fornecidos, a fim de substituir essas interrogações por valores.

Na linha 13, chamamos o método execute utilizando, como segundo argumento, uma tupla com os atributos do objeto pessoa. Cada elemento dessa tupla irá ocupar o lugar de uma interrogação, respeitando a ordem com que aparecem na tupla.

O comando final enviado ao banco de dados pelo comando execute foi:

```
INSERT INTO Pessoa (cpf, nome, nascimento, olhos)
VALUES (10000000099, 'Maria', '1990-01-31', 0);
```

Na linha 16, efetivamos o comando utilizando o método commit.

Nas linhas 18 e 19, fechamos o cursor e a conexão.

DICA

Nem todos os conectores utilizam o mesmo caractere como delimitador. Os conectores psycopg2, do PostgreSQL, e PyMySQL, do MySQL, utilizam o “%s”. É necessário ver a documentação de cada conector para verificar o delimitador correto!

INSERÇÃO DE DADOS EM TABELA COM

QUERIES DINÂMICAS E NOMES

Além da utilização do caractere “?” como delimitador de parâmetros, o `sqlite3` também possibilita a utilização de argumentos nomeados.

A utilização de argumentos nomeados funciona de forma similar à chamada de funções utilizando os nomes dos parâmetros.

Nessa forma de construção de queries dinâmicas, ao invés de passar uma tupla, devemos passar um dicionário para o método `execute`. Ele será utilizado para preencher corretamente os valores dos atributos.

A utilização de argumentos nomeados nos permite utilizar argumentos sem a necessidade de manter a ordem.

Para ilustrar a utilização dos argumentos nomeados em *SQLite*, considere o comando a seguir:

```
>>> comando = INSERT INTO tabela1 (atributo1, atributo2) VALUES (:atrib1, :atrib2);
```

Esse comando indica que ao ser chamado pelo método `execute`, devemos passar um dicionário com duas chaves, sendo uma “atrib1” e outra “atrib2”. Observe que há dois pontos (“:”) antes do argumento nomeado!

Veja a seguir como poderia ficar a chamada do método `execute` para esse comando:

```
>>> cursor.execute(comando, {"atrib1": "Teste", "atrib2": 123})
```

A partir da string e do dicionário é montado o comando final, que é traduzido para:

```
INSERT INTO tabela1 (atributo1, atributo2) VALUES ('Teste', 123);
```

Observe o exemplo da Figura 14, onde vamos criar um script similar ao anterior, no qual vamos utilizar novamente a classe `Pessoa`, porém o comando para inserir um registro no banco de dados utiliza os argumentos nomeados.

```

script11.py X
1  import sqlite3 as conector
2  from modelo import Pessoa
3
4  # Abertura de conexão e aquisição de cursor
5  conexao = conector.connect("./meu_banco.db")
6  cursor = conexao.cursor()
7
8  # Criação de um objeto do tipo Pessoa
9  pessoa = Pessoa(20000000099, 'José', '1990-02-28', False)
10
11 # Definição de um comando com query parameter
12 comando = '''INSERT INTO Pessoa (cpf, nome, nascimento, olhos)
13 .....VALUES (:cpf,:nome,:data_nascimento,:usa_olhos);'''
14 cursor.execute(comando, {"cpf": pessoa.cpf,
15                           "nome": pessoa.nome,
16                           "data_nascimento": pessoa.data_nascimento,
17                           "usa_olhos": pessoa.usa_olhos})
18
19 # Efetivação do comando
20 conexao.commit()
21
22 # Fechamento das conexões
23 cursor.close()
24 conexao.close()

```

Fonte: O Autor

📷 Figura: 14.

Nas linhas 5 e 6, conectamos ao banco de dados e criamos um cursor.

Na linha 9, utilizamos o construtor da classe Pessoa para criar um objeto com os seguintes atributos: CPF: 20000000099, Nome: José, Data de Nascimento: 28/02/1990 e Usa óculos: Não (False).

Nas linhas 12 e 13, definimos a string que conterá o comando para inserir um registro na tabela Pessoa. Observe os nomes dos argumentos nomeados: cpf, nome, data_nascimento e usa_olhos. Cada um desses nomes deve estar presente no dicionário passado para o método execute!

Na linha 14, chamamos o método execute utilizando, como segundo argumento, um dicionário com os atributos do objeto pessoa, definido na linha 10. Cada argumento nomeado do comando será substituído pelo valor da chave correspondente do dicionário.

O comando final enviado ao banco de dados pelo comando execute foi:

```

INSERT INTO Pessoa (cpf, nome, nascimento, olhos)
VALUES (20000000099,'José','1990-02-28',0);

```

Na linha 20, efetivamos o comando utilizando o método commit.

Observe que nosso código está crescendo. Imagine se tivéssemos uma entidade com dezenas de atributos? A chamada do método execute da linha 14 cresceria proporcionalmente, comprometendo muito a leitura do nosso código.

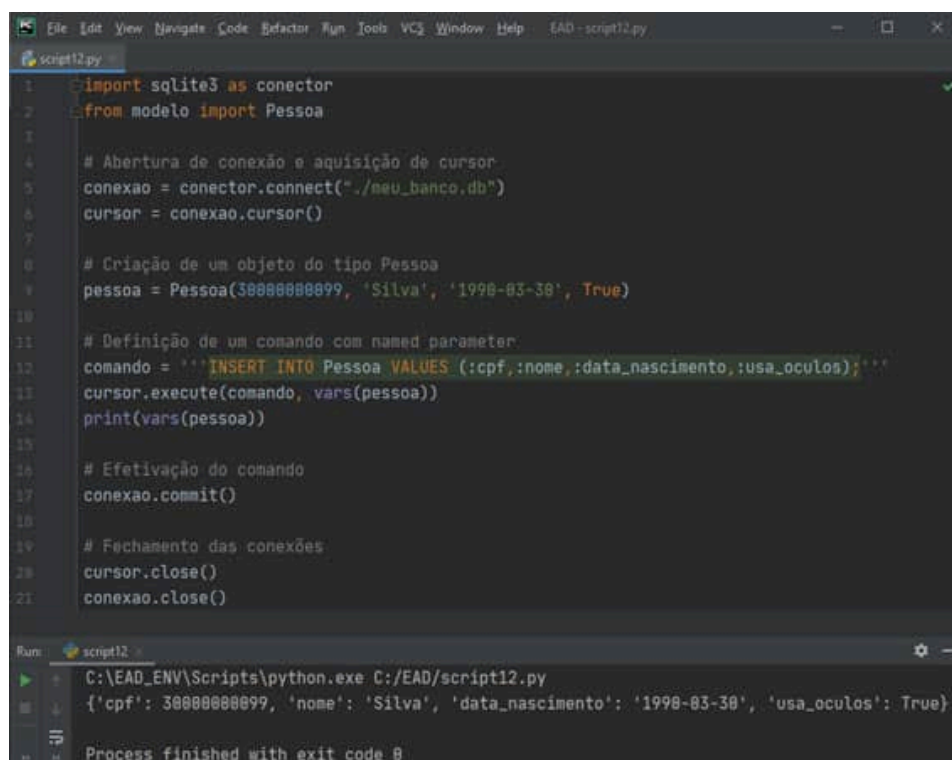
A seguir, vamos simplificar nosso código, de forma que ele permaneça legível, independentemente do número de atributos de uma entidade.

💡 DICA

Quando utilizamos o comando *INSERT INTO do SQL* para inserir um registro onde todos os atributos estão preenchidos, podemos suprimir o nome das colunas no comando.

Como vamos inserir uma pessoa com todos os atributos, vamos manter apenas os argumentos nomeados no comando SQL.

No próximo exemplo, vamos simplificar mais um pouco nosso código, removendo o nome das colunas no comando SQL e utilizando a função interna vars do Python que converte objetos em dicionários. Observe a Figura 15 a seguir.



```
1 import sqlite3 as conector
2 from modelo import Pessoa
3
4 # Abertura de conexão e aquisição de cursor
5 conexao = conector.connect("./meu_banco.db")
6 cursor = conexao.cursor()
7
8 # Criação de um objeto do tipo Pessoa
9 pessoa = Pessoa(30000000000, 'Silva', '1998-03-30', True)
10
11 # Definição de um comando com named parameter
12 comando = 'INSERT INTO Pessoa VALUES (:cpf,:nome,:data_nascimento,:usa_olhos);'
13 cursor.execute(comando, vars(pessoa))
14 print(vars(pessoa))
15
16 # Efetivação do comando
17 conexao.commit()
18
19 # Fechamento das conexões
20 cursor.close()
21 conexao.close()
```

Run: script12.py

C:\EAD_ENV\Scripts\python.exe C:/EAD/script12.py

{'cpf': 30000000000, 'nome': 'Silva', 'data_nascimento': '1998-03-30', 'usa_olhos': True}

Process finished with exit code 0

Fonte: O Autor

📷 Figura: 15.

Após a criação da conexão e do cursor, criamos um objeto do tipo Pessoa com todos os atributos preenchidos na linha 9. Observe que o valor True do atributo usa_olhos será

convertido para 1 durante a execução do método `execute`.

Na linha 12, criamos o comando SQL ***“INSERT INTO”***, onde suprimimos o nome das colunas após o nome da tabela **Pessoa**.

Na linha 13, utilizamos o método `execute` passando como segundo argumento `vars(pessoa)`.

A função interna `vars` retorna todos os atributos de um objeto na forma de dicionário, no qual cada chave é o nome de um atributo.

Observe na saída do console onde imprimimos o resultado de `vars(pessoa)`, linha 14, e destacado a seguir:

```
{'cpf': 30000000099, 'nome': 'Silva', 'data_nascimento': '1990-03-30', 'usa_olhos': True}
```

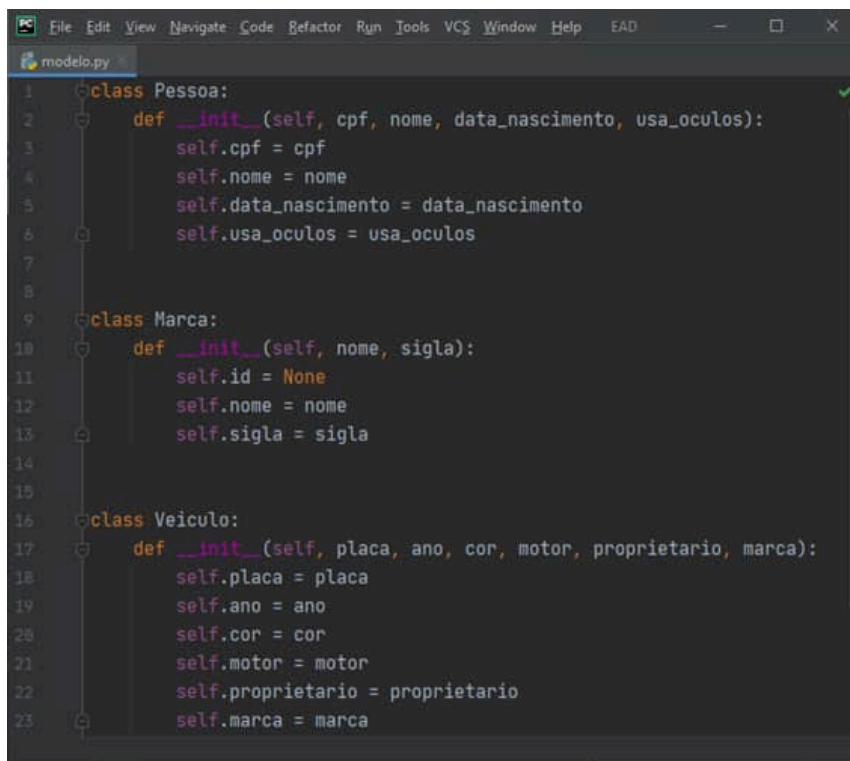
O comando final enviado ao banco de dados pelo comando `execute` foi:

```
INSERT INTO Pessoa VALUES (30000000099,'Silva','1990-03-30',1);
```

Na linha 17, efetivamos a transação e ao final do script fechamos o cursor e a conexão.

No exemplo a seguir, vamos inserir alguns registros nas outras tabelas, de forma a povoar nosso banco de dados. Vamos utilizar a mesma lógica do exemplo anterior, no qual utilizamos a função `vars()` e argumentos nomeados.

Primeiro, vamos criar mais duas classes no nosso módulo `modelo.py` para representar as entidades **Marca** e **Veiculo**. Confira essas classes no script da Figura 16.

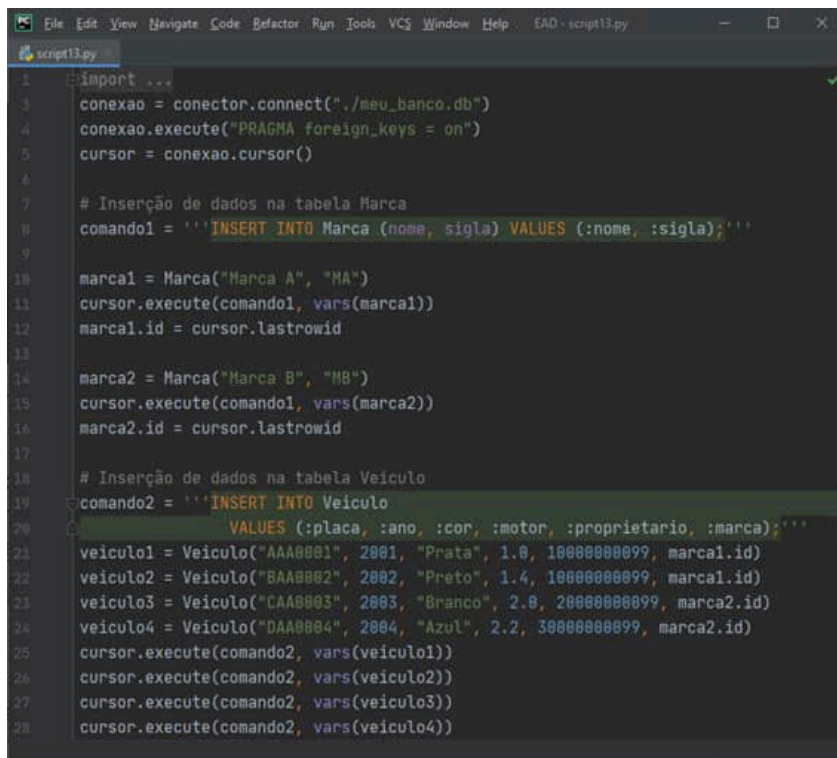


```
1 class Pessoa:
2     def __init__(self, cpf, nome, data_nascimento, usa_olhos):
3         self.cpf = cpf
4         self.nome = nome
5         self.data_nascimento = data_nascimento
6         self.usa_olhos = usa_olhos
7
8
9 class Marca:
10     def __init__(self, nome, sigla):
11         self.id = None
12         self.nome = nome
13         self.sigla = sigla
14
15
16 class Veiculo:
17     def __init__(self, placa, ano, cor, motor, proprietario, marca):
18         self.placa = placa
19         self.ano = ano
20         self.cor = cor
21         self.motor = motor
22         self.proprietario = proprietario
23         self.marca = marca
```

Fonte: O Autor

 Figura: 16.

Para inserir os registros, vamos criar o script definido na Figura 17 a seguir.



```
1 import ...
2
3 conexao = conector.connect("./meu_banco.db")
4 conexao.execute("PRAGMA foreign_keys = on")
5 cursor = conexao.cursor()
6
7 # Inserção de dados na tabela Marca
8 comando1 = '''INSERT INTO Marca (nome, sigla) VALUES (:nome, :sigla);'''
9
10 marca1 = Marca("Marca A", "MA")
11 cursor.execute(comando1, vars(marca1))
12 marca1.id = cursor.lastrowid
13
14 marca2 = Marca("Marca B", "MB")
15 cursor.execute(comando1, vars(marca2))
16 marca2.id = cursor.lastrowid
17
18 # Inserção de dados na tabela Veiculo
19 comando2 = '''INSERT INTO Veiculo
20     VALUES (:placa, :ano, :cor, :motor, :proprietario, :marca);'''
21 veiculo1 = Veiculo("AAA0001", 2001, "Prata", 1.0, 1000000000, marca1.id)
22 veiculo2 = Veiculo("BAA0002", 2002, "Preto", 1.4, 1000000000, marca1.id)
23 veiculo3 = Veiculo("CAA0003", 2003, "Branco", 2.0, 2000000000, marca2.id)
24 veiculo4 = Veiculo("DAA0004", 2004, "Azul", 2.2, 3000000000, marca2.id)
25 cursor.execute(comando2, vars(veiculo1))
26 cursor.execute(comando2, vars(veiculo2))
27 cursor.execute(comando2, vars(veiculo3))
28 cursor.execute(comando2, vars(veiculo4))
```

Fonte: O Autor

 Figura: 17.

No script13, após abrir conexão, vamos utilizar um comando especial do SQLite na linha 4. O comando PRAGMA.

ATENÇÃO

O comando PRAGMA é uma extensão do SQL específica para o SQLite. Ela é utilizada para modificar alguns comportamentos internos do banco de dados.

O SQLite, por padrão, não força a checagem das restrições de chave estrangeira. Isso ocorre por motivos históricos, visto que versões mais antigas do SQLite não tinham suporte à chave estrangeira.

No comando da linha 4, habilitamos a flag `foreign_keys`, de forma a garantir que as restrições de chave estrangeiras sejam checadas antes de cada operação.

Após a utilização do comando PRAGMA e da criação de um cursor, vamos inserir registros relacionados à entidade Marca e Veiculo no banco.

Vamos começar pela entidade Marca, pois ela é um requisito para criação de registros na tabela Veiculo, visto que a tabela Veiculo contém uma chave estrangeira para a tabela Marca, por meio do atributo `Veiculo.marca`, que referencia `Marca.id`.

Na linha 8, escrevemos o comando de inserção de registro na tabela Marca utilizando argumentos nomeados.

Como não iremos passar um valor para o id da marca, que é autoincrementado, foi necessário explicitar o nome das colunas no comando `INSERT INTO`. Caso omitíssemos o nome das colunas, seria gerada uma exceção do tipo `OperationalError`, com a descrição indicando que a tabela tem 3 colunas, mas apenas dois valores foram fornecidos.

Na linha 10, criamos um objeto do tipo `Marca`, que foi inserido no banco pelo comando execute da linha 11.

Para criar uma referência da marca que acabamos de inserir em um veículo, precisamos do id autoincrementado gerado pelo banco no momento da inserção.

Para isso, vamos utilizar o atributo `lastrowid` do `Cursor`. Esse atributo armazena o id da linha do último registro inserido no banco e está disponível assim que chamamos o método `execute` do cursor. O id da linha é o mesmo utilizado para o campo autoincrementado.

Na linha 12, atribuímos o valor do `lastrowid` ao atributo `id` do objeto `marca1`, recém-criado.

Nas linhas 14 a 16, criamos mais um objeto do tipo Marca, inserimos no banco de dados e recuperamos seu novo id.

Nas linhas 19 e 20, temos o comando para inserir registros na tabela Veiculo, também utilizando argumentos nomeados. Como vamos inserir um veículo com todos os atributos, omitimos os nomes das colunas.

Nas linhas 21 a 24, criamos quatro objetos do tipo Veiculo. Observe que utilizamos o atributo id dos objetos marca1 e marca2 para fazer referência à marca do veículo. Os CPF dos proprietários foram escolhidos aleatoriamente, baseando-se nos cadastros anteriores.

Lembre-se de que como os atributos proprietario e marca são referências a chaves de outras tabelas, eles precisam existir no banco! Caso contrário, será lançada uma exceção de erro de integridade (IntegrityError) com a mensagem Falha na Restrição de Chave Estrangeira (FOREIGN KEY constraint failed).

Na sequência, os veículos foram inseridos no banco pelos comandos execute das linhas 25 a 28.

Os comandos SQL gerados por esse script foram os seguintes:

```
INSERT INTO Marca (nome, sigla) VALUES ('Marca A', 'MA')
```

```
INSERT INTO Marca (nome, sigla) VALUES ('Marca B', 'MB')
```

```
INSERT INTO Veiculo VALUES ('AAA0001', 2001, 'Prata', 1.0, 10000000099, 1)
```

```
INSERT INTO Veiculo VALUES ('BAA0002', 2002, 'Preto', 1.4, 10000000099, 1)
```

```
INSERT INTO Veiculo VALUES ('CAA0003', 2003, 'Branco', 2.0, 20000000099, 2)
```

```
INSERT INTO Veiculo VALUES ('DAA0004', 2004, 'Azul', 2.2, 30000000099, 2)
```

Ao final do script, efetivamos as transações, fechamos a conexão e o cursor.

Neste momento, temos os seguintes dados nas nossas tabelas:

Tabela:

	id	nome	sigla
Filtro	Filtro	Filtro	Filtro
1	1	Marca A	MA
2	2	Marca B	MB

Tabela:

	cpf	nome	nascimento	olhos
Filtro	Filtro	Filtro	Filtro	Filtro
1	10000000099	Maria	1990-01-31	0
2	12345678900	João	2000-01-31	1
3	20000000099	José	1990-02-28	0
4	30000000099	Silva	1990-03-30	1

Tabela:

	placa	ano	cor	motor	proprietario	marca
Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro
1	AAA0001	2001	Prata	1.0	10000000099	1
2	BAA0002	2002	Preto	1.4	10000000099	1
3	CAA0003	2003	Branco	2.0	20000000099	2
4	DAA0004	2004	Azul	2.2	30000000099	2

Fonte: O Autor

ATUALIZAÇÃO DE DADOS EM UMA TABELA

Agora que já sabemos como inserir um registro em uma tabela, vamos aprender a atualizar os dados de um registro.

Para atualizar um registro, utilizamos o comando SQL UPDATE. Sua sintaxe é a seguinte:

UPDATE tabela1

SET coluna1 = valor1, coluna2 = valor2...

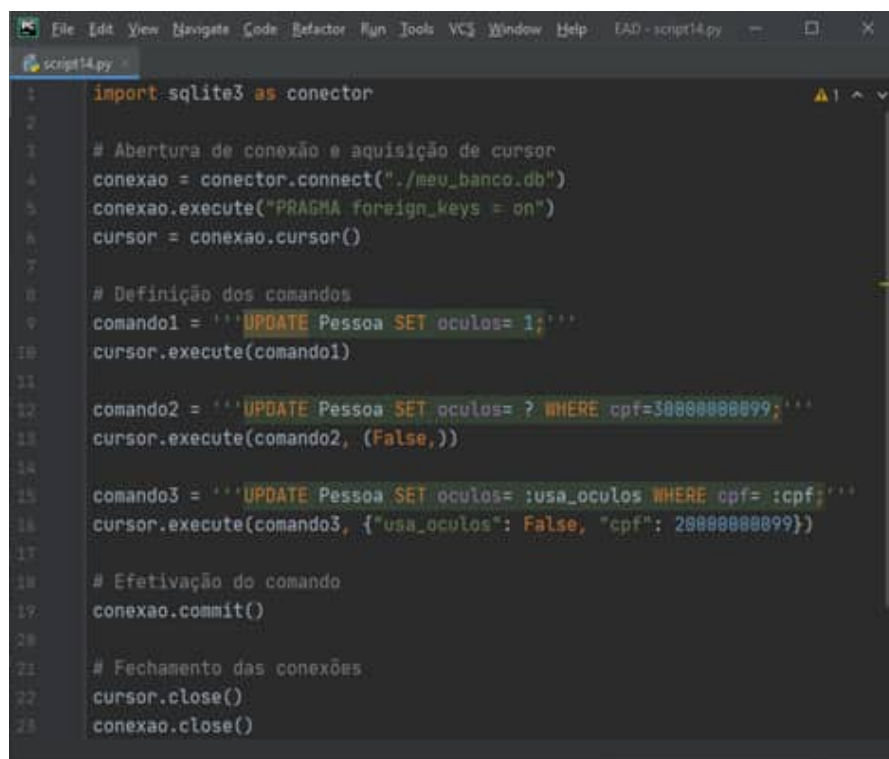
WHERE [condição];

Assim como no comando INSERT, podemos montar o comando UPDATE de três formas.

Uma string sem delimitadores, uma string com o delimitador “?” ou uma string com argumentos nomeados.

Também podemos utilizar os delimitadores na condição da cláusula WHERE.

No exemplo a seguir, Figura 18, vamos mostrar como atualizar registros de uma tabela utilizando os três métodos.



```
1 import sqlite3 as conector
2
3 # Abertura de conexão e aquisição de cursor
4 conexao = conector.connect("./meu_banco.db")
5 conexao.execute("PRAGMA foreign_keys = on")
6 cursor = conexao.cursor()
7
8 # Definição dos comandos
9 comando1 = '''UPDATE Pessoa SET olhos= 1;'''
10 cursor.execute(comando1)
11
12 comando2 = '''UPDATE Pessoa SET olhos= ? WHERE cpf=30000000099;'''
13 cursor.execute(comando2, (False,))
14
15 comando3 = '''UPDATE Pessoa SET olhos= :usa_olhos WHERE cpf= :cpf;'''
16 cursor.execute(comando3, {"usa_olhos": False, "cpf": 20000000099})
17
18 # Efetivação do comando
19 conexao.commit()
20
21 # Fechamento das conexões
22 cursor.close()
23 conexao.close()
```

Fonte: O Autor

📷 Figura: 18.

Após a abertura da conexão, habilitamos novamente a checagem de chave estrangeira, por meio da flag `foreign_keys`, ativada pelo comando PRAGMA.

Na sequência, criamos o cursor e definimos a string do nosso primeiro comando de atualização, onde passamos os valores de forma explícita, sem utilização de delimitadores. O string foi atribuído à variável comando1, linha 9.

No comando1, estamos atualizando o valor do atributo olhos para 1 (verdadeiro) para TODOS os registros da tabela. Isso ocorreu, pois a cláusula WHERE foi omitida.

Na linha 10, executamos o comando1.

Na linha 12, criamos um comando de UPDATE utilizando o delimitador “?” para o valor do atributo olhos e explicitamos o valor do cpf na cláusula WHERE. O comando executado pela linha 13 é:

```
UPDATE Pessoa SET olhos= 0 WHERE cpf=30000000099;
```

Ou seja, vamos alterar o valor do atributo olhos para zero (falso) apenas para quem tem cpf igual a 30000000099.

Na linha 15, criamos mais um comando de UPDATE, desta vez utilizando o argumento nomeado tanto para o atributo olhos quanto para o cpf da cláusula WHERE.

O comando final enviado pela linha 16 ao banco de dados é:

```
UPDATE Pessoa SET olhos= 0 WHERE cpf= 20000000099;
```

Onde vamos alterar o valor do atributo olhos para zero (falso) para quem tem cpf igual a 20000000099.

Na linha 19, efetivamos as transações e posteriormente fechamos o cursor e a conexão.

Se tentássemos alterar o CPF de uma pessoa referenciada pela tabela Veiculo, seria lançada uma exceção do tipo IntegrityError, devido à restrição da chave estrangeira.

DICA

Cuidado ao executar um comando UPDATE sem a cláusula WHERE, pois, dependendo do tamanho do banco de dados, essa operação pode ser muito custosa.

REMOÇÃO DE DADOS DE UMA TABELA

Nesta parte deste módulo, iremos aprender a remover registros de uma tabela.

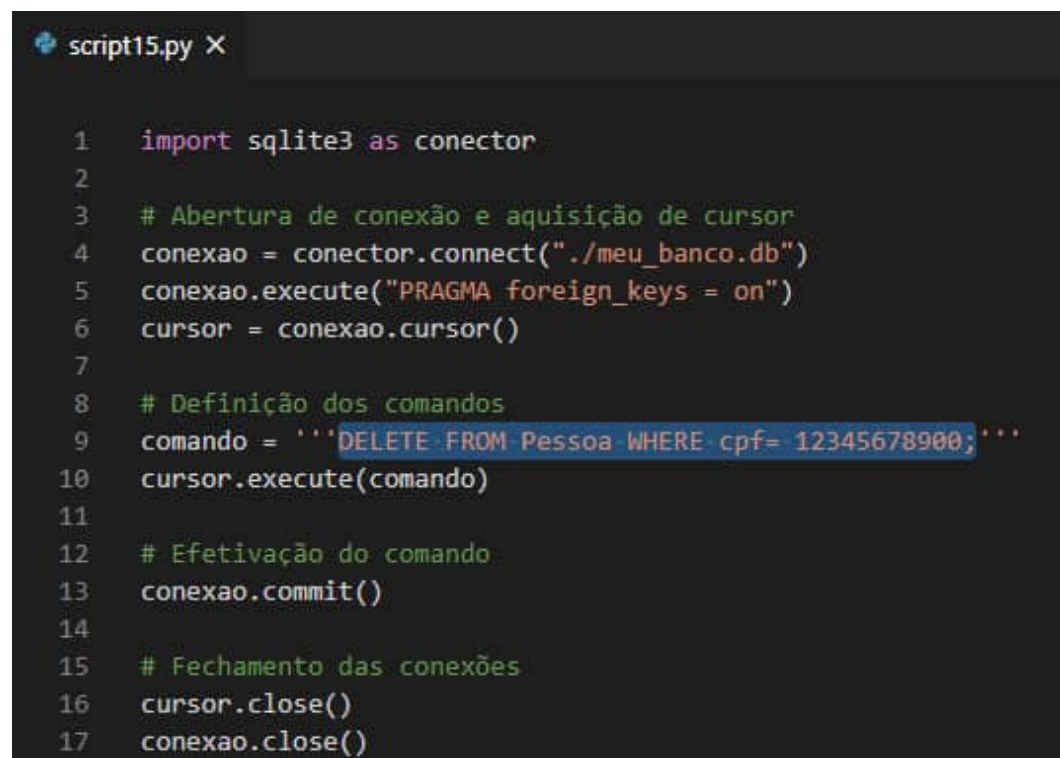
Para remover um registro, utilizamos o comando SQL DELETE. Sua sintaxe é a seguinte:

DELETE FROM tabela1

WHERE [condição];

Assim como nos outros comandos, podemos montar o comando DELETE de três formas. Uma string sem delimitadores, uma string com o delimitador “?” ou uma string com argumentos nomeados. Todos para a condição da cláusula WHERE.

Veja o exemplo da utilização do comando DELETE a seguir, na Figura 19.



```
script15.py X

1  import sqlite3 as conector
2
3  # Abertura de conexão e aquisição de cursor
4  conexao = conector.connect("./meu_banco.db")
5  conexao.execute("PRAGMA foreign_keys = on")
6  cursor = conexao.cursor()
7
8  # Definição dos comandos
9  comando = '''DELETE FROM Pessoa WHERE cpf= 12345678900;'''
10 cursor.execute(comando)
11
12 # Efetivação do comando
13 conexao.commit()
14
15 # Fechamento das conexões
16 cursor.close()
17 conexao.close()
```

Fonte: O Autor

📷 Figura: 19.

Após a criação da conexão, habilitação da checagem de chave estrangeira e aquisição do cursor, criamos o comando SQL “DELETE” na linha 9, onde explicitamos o CPF da pessoa que desejamos remover do banco.

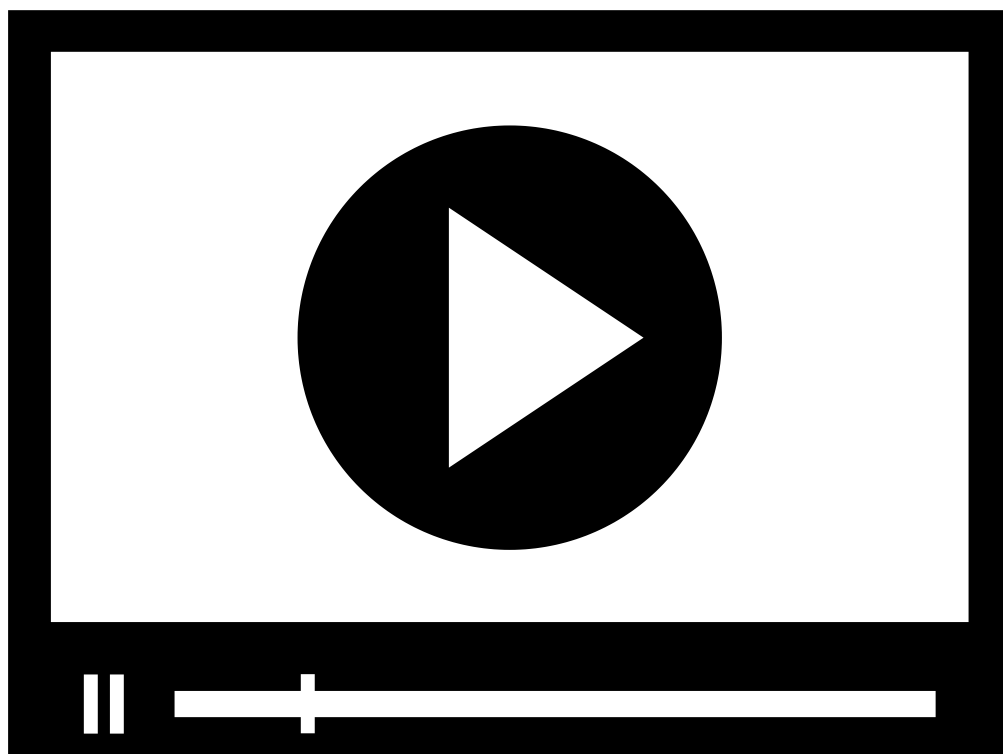
Na linha 10, utilizamos o método execute com o comando criado anteriormente.

O comando final enviado ao banco de dados pelo comando execute foi:

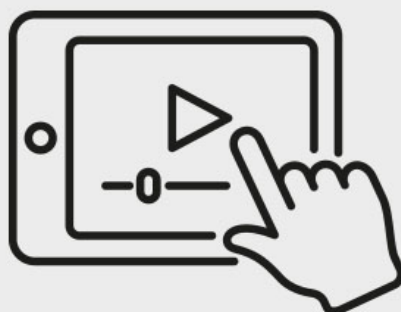
DELETE FROM Pessoa WHERE cpf=12345678900;

Na linha 13, efetivamos a transação e ao final do script fechamos o cursor e a conexão.

Observe que, como as outras pessoas cadastradas são referenciadas pela tabela Veiculo por meio de seu CPF, seria gerada uma exceção do tipo `IntegrityError` caso tentássemos removê-las.



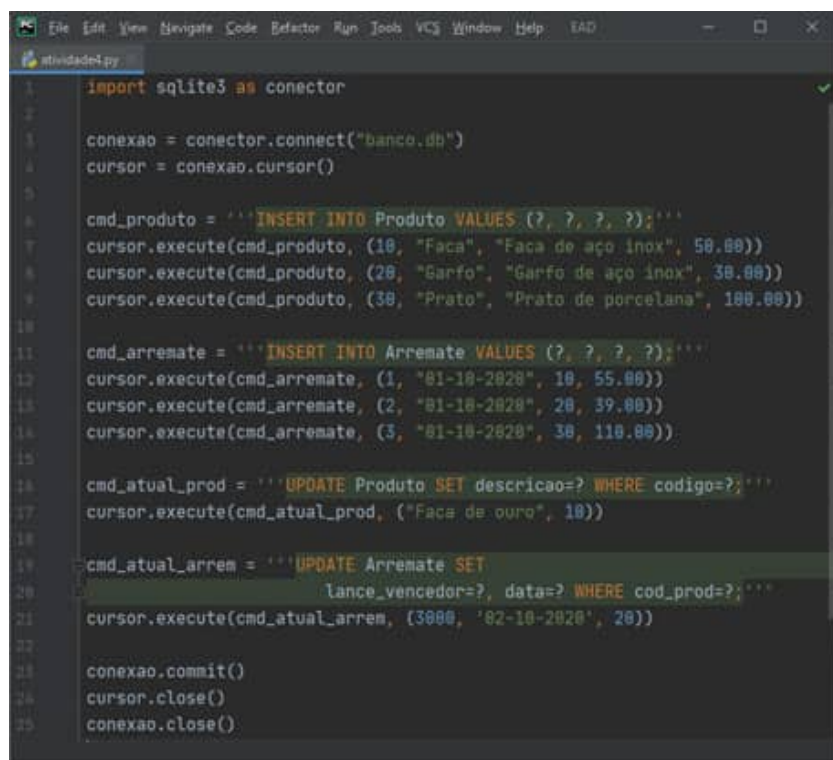
Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. CONSIDERE QUE TEMOS AS TABELAS ARREIMATE E PRODUTO, DEFINIDAS COM OS COMANDOS SQL A SEGUIR:

CREATE TABLE PRODUTO (
CODIGO INTEGER NOT NULL,
NOME TEXT NOT NULL,
DESCRICAO TEXT,
PRECO_INICIAL REAL,
PRIMARY KEY (CODIGO))
CREATE TABLE ARREMATE (
ID INTEGER NOT NULL,
DATA DATE NOT NULL,
COD_PROD INTEGER NOT NULL,
LANCE_VENCEDOR REAL,
FOREIGN KEY(COD_PROD) REFERENCES PRODUTO(CODIGO))
QUAL SERÁ O CONTEÚDO DA TABELA ARREMATE APÓS A EXECUÇÃO
DO SCRIPT A SEGUIR?



```
1 import sqlite3 as conector
2
3 conexao = conector.connect("banco.db")
4 cursor = conexao.cursor()
5
6 cmd_produto = '''INSERT INTO Produto VALUES (?, ?, ?, ?);'''
7 cursor.execute(cmd_produto, (10, "Faca", "Faca de aço inox", 50.00))
8 cursor.execute(cmd_produto, (20, "Garfo", "Garfo de aço inox", 30.00))
9 cursor.execute(cmd_produto, (30, "Prato", "Prato de porcelana", 100.00))
10
11 cmd_arremate = '''INSERT INTO Arremate VALUES (?, ?, ?, ?);'''
12 cursor.execute(cmd_arremate, (1, "01-10-2020", 10, 55.00))
13 cursor.execute(cmd_arremate, (2, "01-10-2020", 20, 39.00))
14 cursor.execute(cmd_arremate, (3, "01-10-2020", 30, 110.00))
15
16 cmd_atual_prod = '''UPDATE Produto SET descricao=? WHERE codigo=?;'''
17 cursor.execute(cmd_atual_prod, ("Faca de ouro", 10))
18
19 cmd_atual_arrem = '''UPDATE Arremate SET
20     lance_vencedor=?, data=? WHERE cod_prod=?;'''
21 cursor.execute(cmd_atual_arrem, (3000, "02-10-2020", 20))
22
23 conexao.commit()
24 cursor.close()
25 conexao.close()
```

FONTE: O AUTOR

A)

A) id	A) Data	A) cod_prod	A) lance_vencedor
A) 1	A) 01-10-2020	A) 10	A) 3000.0
A) 2	A) 01-10-2020	A) 20	A) 39.0
A) 3	A) 01-10-2020	A) 30	A) 110.0

☐ **Atenção! Para visualizaçãocompleta da tabela utilize a rolagem horizontal**

A)

B)

B) id	B) Data	B) cod_prod	B) lance_vencedor
B) 1	B) 01-10-2020	B) 10	B) 55.0
B) 2	B) 02-10-2020	B) 20	B) 3000.0
B) 3	B) 01-10-2020	B) 30	B) 110.0

☐ **Atenção! Para visualizaçãocompleta da tabela utilize a rolagem horizontal**

B)

C)

C) id	C) Data	C) cod_prod	C) lance_vencedor
-------	---------	-------------	-------------------

C) 1	C) 01-10-2020	C) 20	C) 55.0
C) 2	C) 02-10-2020	C) 20	C) 3000.0
C) 3	C) 01-10-2020	C) 30	C) 110.0

☐ **Atenção! Para visualizaçãocompleta da tabela utilize a rolagem horizontal**

C)

D)

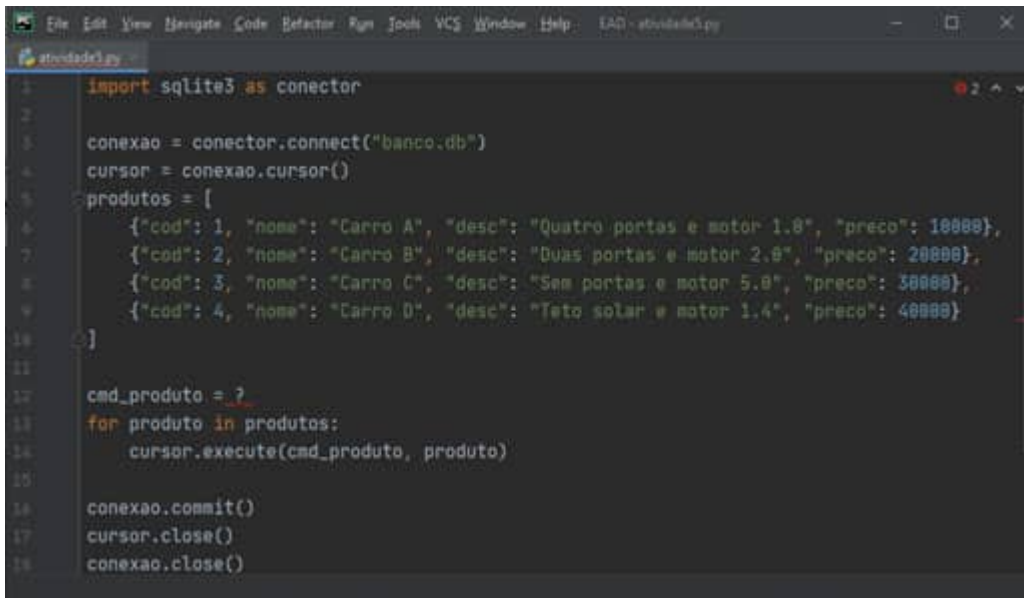
D) id	D) Data	D) cod_prod	D) lance_vencedor
D) 1	D) 01-10-2020	D) 10	D) 55.0
D) 2	D) 01-10-2020	D) 20	D) 39.0
D) 3	D) 01-10-2020	D) 30	D) 110.0

☐ **Atenção! Para visualizaçãocompleta da tabela utilize a rolagem horizontal**

D)

2. CONSIDERE NOVAMENTE AS TABELAS DO EXEMPLO ANTERIOR, PORÉM SEM REGISTROS CADASTRADOS. OBSERVE O SCRIPT A

SEGUIR E RESPONDA:



```
1 import sqlite3 as conector
2
3 conexao = conector.connect("banco.db")
4 cursor = conexao.cursor()
5 produtos = [
6     {"cod": 1, "nome": "Carro A", "desc": "Quatro portas e motor 1.8", "preco": 10000},
7     {"cod": 2, "nome": "Carro B", "desc": "Duas portas e motor 2.0", "preco": 20000},
8     {"cod": 3, "nome": "Carro C", "desc": "Sem portas e motor 5.0", "preco": 30000},
9     {"cod": 4, "nome": "Carro D", "desc": "Teto solar e motor 1.4", "preco": 40000}
10 ]
11
12 cmd_produto = _?
13 for produto in produtos:
14     cursor.execute(cmd_produto, produto)
15
16 conexao.commit()
17 cursor.close()
18 conexao.close()
```

FONTE: O AUTOR

QUAL STRING DEVE SER INCLUÍDA NA LINHA 12, SUBSTITUINDO A INTERROGAÇÃO, PARA QUE O PROGRAMA INSIRA CORRETAMENTE OS ELEMENTOS DA LISTA PRODUTOS NA TABELA PRODUTO?

- A) `""INSERT INTO Produto VALUES (?, ?, ?, ?);""`
- B) `""INSERT INTO Produto VALUES (:codigo, :nome, :descricao, :preco_inicial);""`
- C) `""INSERT INTO Produto VALUES (:cod, :nome, :desc, :preco);""`
- D) `""INSERT INTO Produto VALUES ([cod, nome, desc, preco]);""`

GABARITO

1. Considere que temos as tabelas Arremate e Produto, definidas com os comandos SQL a seguir:

```
CREATE TABLE Produto (
    codigo INTEGER NOT NULL,
    nome TEXT NOT NULL,
    descricao TEXT,
    preco_inicial REAL,
    PRIMARY KEY (codigo))
CREATE TABLE Arremate (
```

id INTEGER NOT NULL,
data DATE NOT NULL,
cod_prod INTEGER NOT NULL,
lance_vencedor REAL,
FOREIGN KEY(cod_prod) REFERENCES Produto(codigo))

Qual será o conteúdo da tabela Arremate após a execução do script a seguir?

```
atividades.py
1 import sqlite3 as conector
2
3 conexao = conector.connect("banco.db")
4 cursor = conexao.cursor()
5
6 cmd_produto = '''INSERT INTO Produto VALUES (?, ?, ?, ?);'''
7 cursor.execute(cmd_produto, (10, "Faca", "Faca de aço inox", 50.00))
8 cursor.execute(cmd_produto, (20, "Garfo", "Garfo de aço inox", 30.00))
9 cursor.execute(cmd_produto, (30, "Prato", "Prato de porcelana", 100.00))
10
11 cmd_arremate = '''INSERT INTO Arremate VALUES (?, ?, ?, ?);'''
12 cursor.execute(cmd_arremate, (1, "01-10-2020", 10, 55.00))
13 cursor.execute(cmd_arremate, (2, "01-10-2020", 20, 39.00))
14 cursor.execute(cmd_arremate, (3, "01-10-2020", 30, 110.00))
15
16 cmd_atual_prod = '''UPDATE Produto SET descricao=? WHERE codigo=?;'''
17 cursor.execute(cmd_atual_prod, ("Faca de ouro", 10))
18
19 cmd_atual_arrem = '''UPDATE Arremate SET
20 lance_vencedor=?, data=? WHERE cod_prod=?;'''
21 cursor.execute(cmd_atual_arrem, (3000, "02-10-2020", 20))
22
23 conexao.commit()
24 cursor.close()
25 conexao.close()
```

Fonte: O Autor

A alternativa "B " está correta.

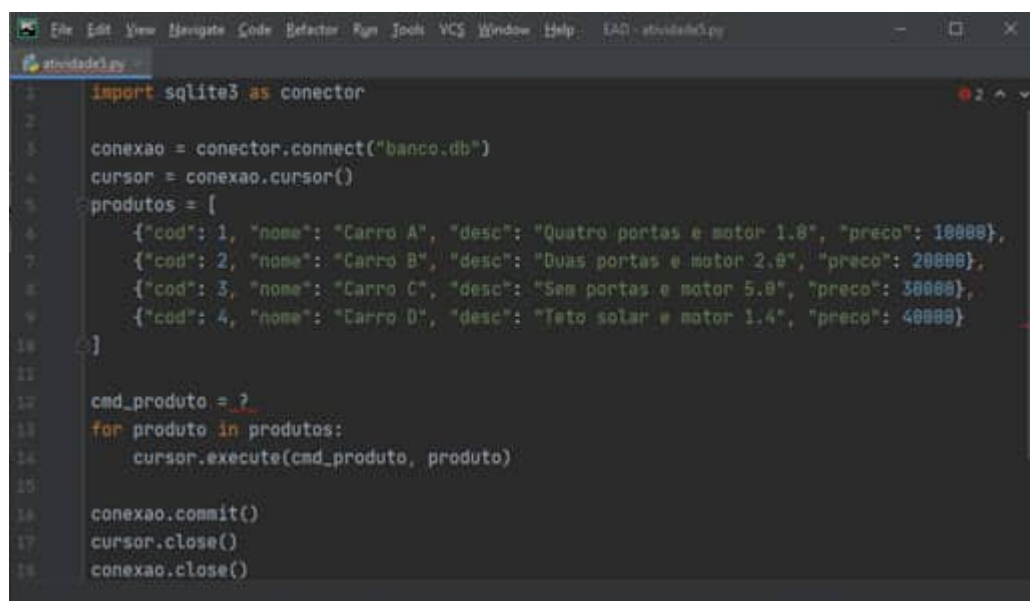
O script insere os dados iniciais na tabela Arremate nas linhas 11 a 14, deixando a tabela Arremate com os registros:

id	data	cod_prod	lance_vencedor
1	01-10-2020	10	55.0

2	01-10-2020	20	39.0
3	01-10-2020	30	110.0

Na linha 19, atualizamos a data e lance_vencedor do registro cujo cod_prod é 20. Com isso, obtivemos o resultado exibido na letra b.

2. Considere novamente as tabelas do exemplo anterior, porém sem registros cadastrados. Observe o script a seguir e responda:



```

1 import sqlite3 as conector
2
3 conexao = conector.connect("banco.db")
4 cursor = conexao.cursor()
5 produtos = [
6     {"cod": 1, "nome": "Carro A", "desc": "Quatro portas e motor 1.8", "preco": 10000},
7     {"cod": 2, "nome": "Carro B", "desc": "Duas portas e motor 2.0", "preco": 20000},
8     {"cod": 3, "nome": "Carro C", "desc": "Sem portas e motor 5.0", "preco": 30000},
9     {"cod": 4, "nome": "Carro D", "desc": "Teto solar e motor 1.4", "preco": 40000}
10 ]
11
12 cmd_produto = _?
13 for produto in produtos:
14     cursor.execute(cmd_produto, produto)
15
16 conexao.commit()
17 cursor.close()
18 conexao.close()

```

Fonte: O Autor

Qual string deve ser incluída na linha 12, substituindo a interrogação, para que o programa insira corretamente os elementos da lista produtos na tabela Produto?

A alternativa "C " está correta.

Para inserir dados de um dicionário utilizando queries dinâmicas, precisamos definir uma string que utilize parâmetros nomeados. Os nomes dos parâmetros precisam ser os nomes das chaves do dicionário, não o nome dos atributos. Lembrando que os nomes das chaves precisam ser precedidos por dois pontos (:).

MÓDULO 4

- ⦿ Empregar as funcionalidades para recuperação de registros em tabelas

SELEÇÃO DE REGISTROS DE UMA TABELA

Neste módulo, aprenderemos a recuperar os registros presentes no banco de dados.

Partiremos de consultas mais simples, utilizando apenas uma tabela, até consultas mais sofisticadas, envolvendo os relacionamentos entre tabelas.



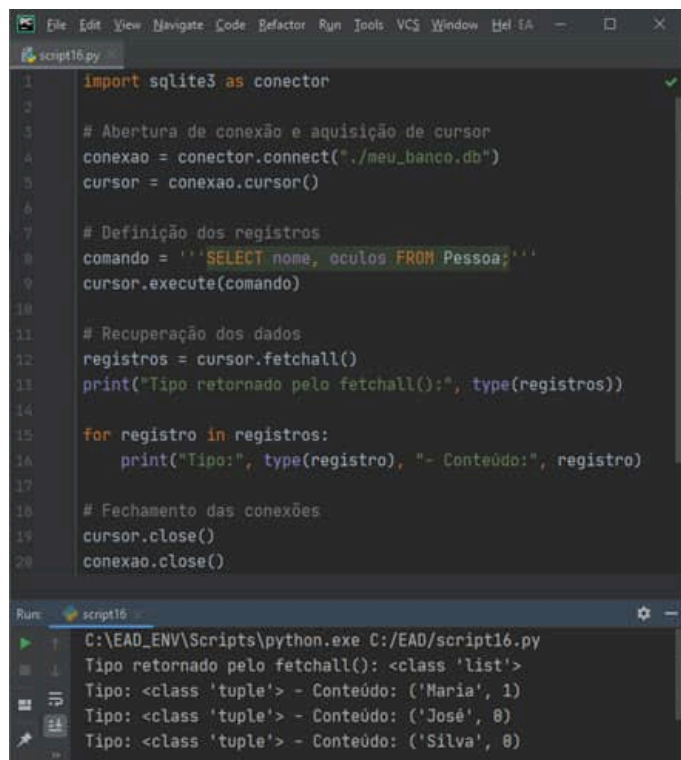
Para selecionar e recuperar registros de um banco de dados, utilizamos o comando SQL **SELECT**. Sua sintaxe é a seguinte:

```
SELECT coluna1, coluna2, ... FROM tabela1
```

```
WHERE [condição];
```

Assim como nos outros comandos, podemos utilizar uma string sem delimitadores, uma string com o delimitador “?” ou uma string com argumentos nomeados para a condição da cláusula **WHERE**.

No exemplo a seguir, Figura 20, vamos mostrar como recuperar todos os registros da tabela Pessoa.

The image shows a screenshot of a code editor window titled 'script16.py' and a console window below it. The code in the editor is a Python script that connects to a SQLite database, executes a SQL query, and prints the results. The console shows the output of the script, including the type of the data returned and the specific records.

```
1 import sqlite3 as conector
2
3 # Abertura de conexão e aquisição de cursor
4 conexao = conector.connect("./meu_banco.db")
5 cursor = conexao.cursor()
6
7 # Definição dos registros
8 comando = 'SELECT nome, olhos FROM Pessoa;'
9 cursor.execute(comando)
10
11 # Recuperação dos dados
12 registros = cursor.fetchall()
13 print("Tipo retornado pelo fetchall():", type(registros))
14
15 for registro in registros:
16     print("Tipo:", type(registro), "- Conteúdo:", registro)
17
18 # Fechamento das conexões
19 cursor.close()
20 conexao.close()
```

Run: script16

C:\EAD_ENV\Scripts\python.exe C:/EAD/script16.py
Tipo retornado pelo fetchall(): <class 'list'>
Tipo: <class 'tuple'> - Conteúdo: ('Maria', 1)
Tipo: <class 'tuple'> - Conteúdo: ('José', 8)
Tipo: <class 'tuple'> - Conteúdo: ('Silva', 8)

Fonte: O Autor

 Figura: 20.

Após criar uma conexão e obter um cursor, criamos o comando SQL “SELECT” na linha 8 e destacado a seguir:

SELECT nome, olhos FROM Pessoa;

Observe que estamos selecionando todas as pessoas da tabela, visto que não há clausulas WHERE. Porém, estamos recuperando apenas os dados das colunas nome e olhos.

Executamos o comando na linha 9 e utilizamos o método fetchall do cursor para recuperar os registros selecionados.

Atribuímos o retorno do método à variável registros.

O objeto retornado pelo método fetchall é do tipo lista, impresso pela linha 13 e que pode ser observado pelo console.

Na linha 15, iteramos sobre os elementos retornados e, na linha 16, imprimimos o tipo e o conteúdo dos registros.

Observe que cada registro é uma tupla, composta pelos atributos nome e olhos da entidade Pessoa.

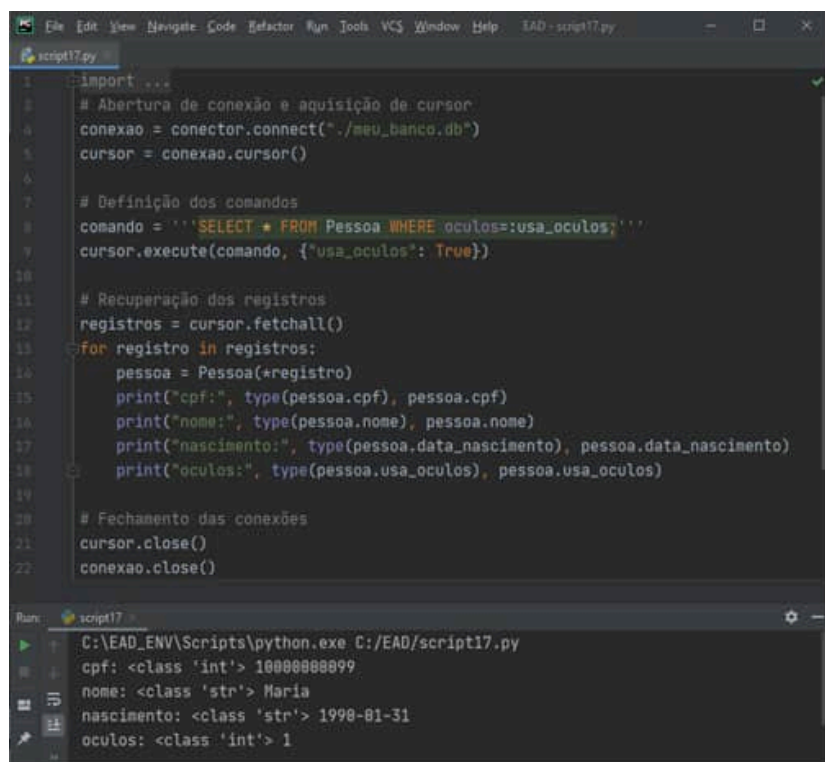
Os registros são sempre retornados em forma de tupla, mesmo que contenham apenas um atributo!

Ao final, fechamos o cursor e a conexão.

📢 ATENÇÃO

Como o SQLite não cria uma transação para o comando SELECT, não é necessário executar o commit.

No exemplo a seguir, Figura 21, vamos criar uma consulta para retornar as pessoas que usam óculos. Observe como ficou o exemplo.



```
1 import ...
2 # Abertura de conexão e aquisição de cursor
3 conexao = conector.connect("./meu_banco.db")
4 cursor = conexao.cursor()
5
6 # Definição dos comandos
7 comando = '''SELECT * FROM Pessoa WHERE olhos=:usa_olhos;'''
8 cursor.execute(comando, {"usa_olhos": True})
9
10 # Recuperação dos registros
11 registros = cursor.fetchall()
12 for registro in registros:
13     pessoa = Pessoa(*registro)
14     print("cpf:", type(pessoa.cpf), pessoa.cpf)
15     print("nome:", type(pessoa.nome), pessoa.nome)
16     print("nascimento:", type(pessoa.data_nascimento), pessoa.data_nascimento)
17     print("olhos:", type(pessoa.usa_olhos), pessoa.usa_olhos)
18
19 # Fechamento das conexões
20 cursor.close()
21 conexao.close()
```

Run: script17

```
C:\EAD_ENV\Scripts\python.exe C:/EAD/script17.py
cpf: <class 'int'> 100000000000
nome: <class 'str'> Maria
nascimento: <class 'str'> 1990-01-31
olhos: <class 'int'> 1
```

Fonte: O Autor

📷 Figura: 21.

Após abrir a conexão e criar um cursor, definimos o comando para selecionar apenas as pessoas que usam óculos.

Para isso, definimos o comando SQL da linha 8, que, após executado, fica da seguinte forma:

SELECT * FROM Pessoa WHERE olhos=1;

Observe que utilizamos o asterisco (*) para representar quais dados desejamos receber. No SQL, o asterisco representa todas as colunas.

Na linha 12, recuperamos todos os dados selecionados utilizando o fetchall, que foram iterados na linha 13.

Para cada registro retornado, que é uma tupla com os atributos cpf, nome, nascimento e oculos, nessa ordem, criamos um objeto do tipo Pessoa, na linha 14.

Observe que utilizamos o operador * do Python. Esse operador “desempacota” um iterável, passando cada elemento como um argumento para uma função ou construtor.

Como sabemos que a ordem das colunas é a mesma ordem dos parâmetros do construtor da classe Pessoa, garantimos que vai funcionar corretamente.

Das linhas 15 a 18, imprimimos cada atributo do registro e seu respectivo tipo.

Ao final do script fechamos a conexão e o cursor.

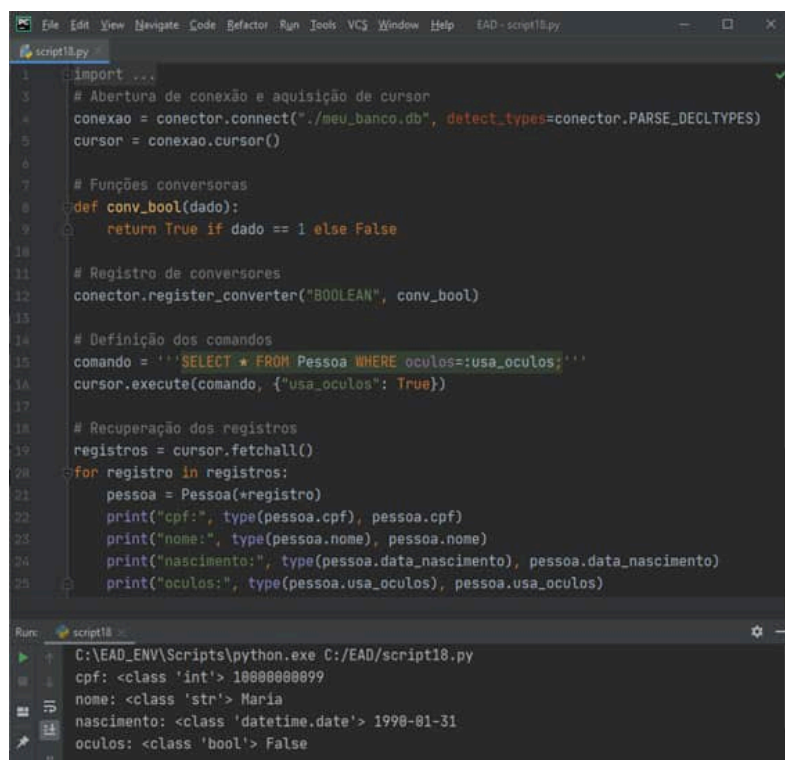
Verifique que os dados dos atributos nascimento e oculos estão exatamente como no banco de dados, o nascimento é uma string e o oculos é do tipo inteiro.

COMENTÁRIO

Se estivéssemos utilizando o banco de dados *PostgreSQL* com o conector *psycopg2*, como os tipos *DATE* e *BOOLEAN* são suportados, esses valores seriam convertidos para o tipo correto.

O conector *sqlite3* também nos permite fazer essa conversão automaticamente, mas precisamos fazer algumas configurações a mais.

No exemplo a seguir, Figura 22, vamos mostrar como fazer a conversão de datas e booleanos.



```
1 import ...
2
3 # Abertura de conexão e aquisição de cursor
4 conexao = conector.connect("./meu_banco.db", detect_types=conector.PARSE_DECLTYPES)
5 cursor = conexao.cursor()
6
7 # Funções conversoras
8 def conv_bool(dado):
9     return True if dado == 1 else False
10
11 # Registro de conversores
12 conector.register_converter("BOOLEAN", conv_bool)
13
14 # Definição dos comandos
15 comando = 'SELECT * FROM Pessoa WHERE olhos=:usa_olhos;'
16 cursor.execute(comando, {"usa_olhos": True})
17
18 # Recuperação dos registros
19 registros = cursor.fetchall()
20 for registro in registros:
21     pessoa = Pessoa(*registro)
22     print("cpf:", type(pessoa.cpf), pessoa.cpf)
23     print("nome:", type(pessoa.nome), pessoa.nome)
24     print("nascimento:", type(pessoa.data_nascimento), pessoa.data_nascimento)
25     print("olhos:", type(pessoa.usa_olhos), pessoa.usa_olhos)
```

Run: script18

C:\EAD_ENV\Scripts\python.exe C:/EAD/script18.py

cpf: <class 'int'> 10000000000

nome: <class 'str'> Maria

nascimento: <class 'datetime.date'> 1990-01-31

olhos: <class 'bool'> False

Fonte: O Autor

📷 Figura: 22.

A primeira modificação ocorre nos parâmetros da criação da conexão. Precisamos passar o argumento *PARSE_DECLTYPES* para o parâmetro *detect_types* da função *connect*. Observe como ficou a linha 4.

Isso indica que o conector deve tentar fazer uma conversão dos dados, tomando como base o tipo da coluna declarada no *CREATE TABLE*.

💬 COMENTÁRIO

Os tipos *DATE* e *TIMESTAMP* já possuem conversores embutidos no *sqlite3*, porém, o tipo *BOOLEAN* não.

Para informar ao conector como fazer a conversão do tipo *BOOLEAN*, precisamos definir e registrar a função conversora utilizando a função interna *register_converter* do *sqlite3*.

A função *register_converter* espera, como primeiro parâmetro, uma string com o tipo da coluna a ser convertido e, como segundo parâmetro, uma função que recebe o dado e retorna esse dado convertido.

Na linha 12, chamamos a função *register_converter*, passando a string “*BOOLEAN*” e a função *conv_bool* (converter booleano) como argumentos.

A função `conv_bool`, definida na linha 9, retorna `True` para o caso do dado ser 1, ou `False`, caso contrário. Com isso, convertemos os inteiros 0 e 1 para os booleanos `True` e `False`.

O restante do script é igual ao anterior, porém, os dados estão convertidos para o tipo correto.

Verifique a saída do console e observe os tipos dos atributos `nascimento` e `oculos`.

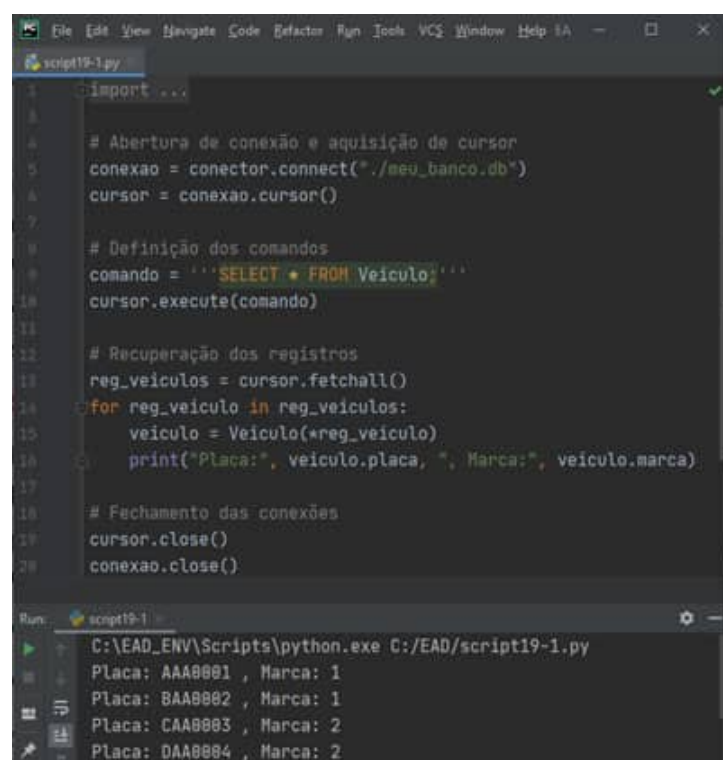
Agora são das classes `date` e `bool`!

SELEÇÃO DE REGISTROS UTILIZANDO JUNÇÃO

Na seção anterior, aprendemos a selecionar registros de apenas uma tabela, mas como podemos buscar registros de tabelas relacionadas?

No exemplo a seguir, Figura 23, vamos buscar os veículos e suas respectivas marcas.

Vamos fazer isso por meio da junção de tabelas.



```
1 import ...
2
3 # Abertura de conexão e aquisição de cursor
4 conexao = conector.connect("./meu_banco.db")
5 cursor = conexao.cursor()
6
7 # Definição dos comandos
8 comando = 'SELECT * FROM Veiculo;'
9 cursor.execute(comando)
10
11 # Recuperação dos registros
12 reg_veiculos = cursor.fetchall()
13 for reg_veiculo in reg_veiculos:
14     veiculo = Veiculo(*reg_veiculo)
15     print("Placa:", veiculo.placa, ", Marca:", veiculo.marca)
16
17 # Fechamento das conexões
18 cursor.close()
19 conexao.close()
```

Run: script19-1

C:\EAD_ENV\Scripts\python.exe C:/EAD/script19-1.py

Placa: AAA0001 , Marca: 1

Placa: BAA0002 , Marca: 1

Placa: CAA0003 , Marca: 2

Placa: DAA0004 , Marca: 2

Fonte: O Autor

📷 Figura: 23.

Após abrir a conexão e obter um cursor, selecionamos todos os registros da tabela Veiculo, utilizando o comando da linha 9, executado na linha 10.

Na linha 13, recuperamos todos os registros de veículos utilizando o método fetchall, que foram iterados na linha 14.

Na linha 15, criamos um objeto do tipo Veiculo utilizando o operador * e imprimimos os atributos placa e marca na linha 16.

Verifique no console que os valores do atributo marca são seus respectivos ids, 1 e 2. Isso ocorre, pois no banco de dados, armazenamos apenas uma referência à chave primária da entidade Marca.

E se quisermos substituir o id das marcas pelos seus respectivos nomes?

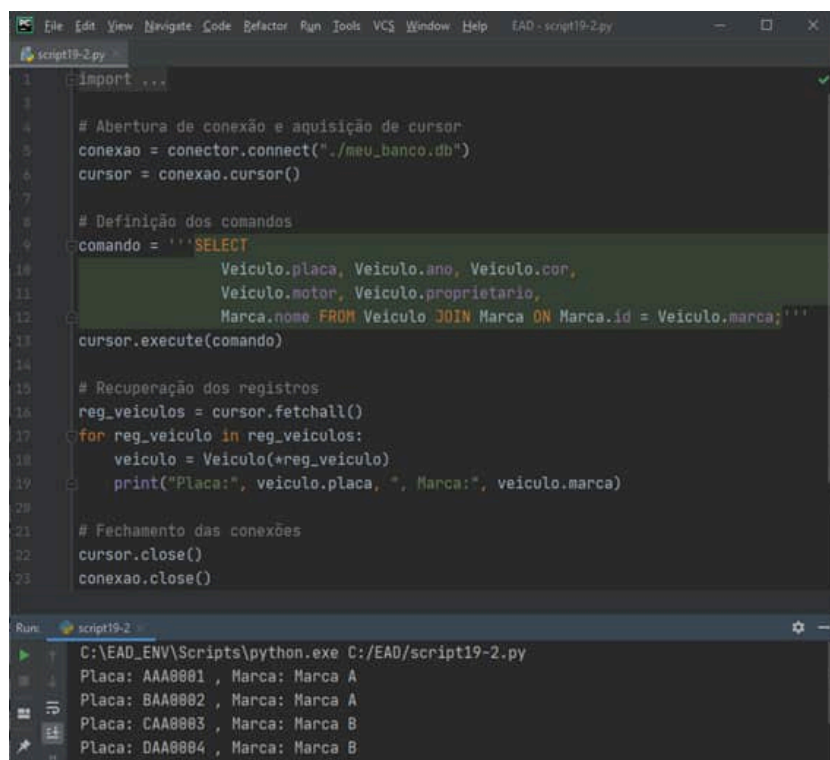
Para isso, precisamos realizar uma junção das tabelas Veiculo e Marca no comando SELECT.

O comando SELECT para junção de duas tabelas tem a seguinte sintaxe:

```
SELECT tab1.col1, tab1.col2, tab2.col1... FROM tab1 JOIN tab2 ON tab1.colN = tab2.colM;
```

Primeiro, definimos quais colunas serão retornadas utilizando a sintaxe nome_tabela.nome_coluna, depois indicamos as tabelas que desejamos juntar e, por último, indicamos como alinhar os registros de cada tabela, ou seja, quais são os atributos que devem ser iguais (colN e colM).

No exemplo a seguir, Figura 24, vamos criar um script de forma que o Veiculo tenha acesso ao nome da Marca, não ao id.



```
1 import ...
2
3 # Abertura de conexão e aquisição de cursor
4 conexao = conector.connect("./meu_banco.db")
5 cursor = conexao.cursor()
6
7 # Definição dos comandos
8 comando = '''SELECT
9     Veiculo.placa, Veiculo.ano, Veiculo.cor,
10     Veiculo.motor, Veiculo.proprietario,
11     Marca.nome FROM Veiculo JOIN Marca ON Marca.id = Veiculo.marca;'''
12 cursor.execute(comando)
13
14 # Recuperação dos registros
15 reg_veiculos = cursor.fetchall()
16 for reg_veiculo in reg_veiculos:
17     veiculo = Veiculo(*reg_veiculo)
18     print("Placa:", veiculo.placa, ", Marca:", veiculo.marca)
19
20 # Fechamento das conexões
21 cursor.close()
22 conexao.close()
```

Run: script19-2

C:\EAD_ENV\Scripts\python.exe C:/EAD/script19-2.py

Placa: AAA0001 , Marca: Marca A

Placa: BAA0002 , Marca: Marca A

Placa: CAA0003 , Marca: Marca B

Placa: DAA0004 , Marca: Marca B

Fonte: O Autor

📷 Figura: 24.

Após criar uma conexão e obter um cursor, definimos o comando SQL para recuperar os dados das tabelas Veiculo e Marca de forma conjunta.

Observe o comando SQL nas linhas 9 a 12 e destacado a seguir:

SELECT Veiculo.placa, Veiculo.ano, Veiculo.cor, Veiculo.motor, Veiculo.proprietario, Marca.nome FROM Veiculo JOIN Marca ON Marca.id = Veiculo.marca;

Vamos selecionar os atributos placa, ano, cor, motor e proprietário do Veiculo e juntar com o atributo nome da tabela Marca. Observe que não vamos utilizar o atributo id da Marca.

As tuplas retornadas serão similares à seguinte:

('AAA0001', 2001, 'Prata', 1.0, 10000000099, 'Marca A')

Na linha 16, recuperamos todos os registros de veículos, que foram iterados na linha 17.

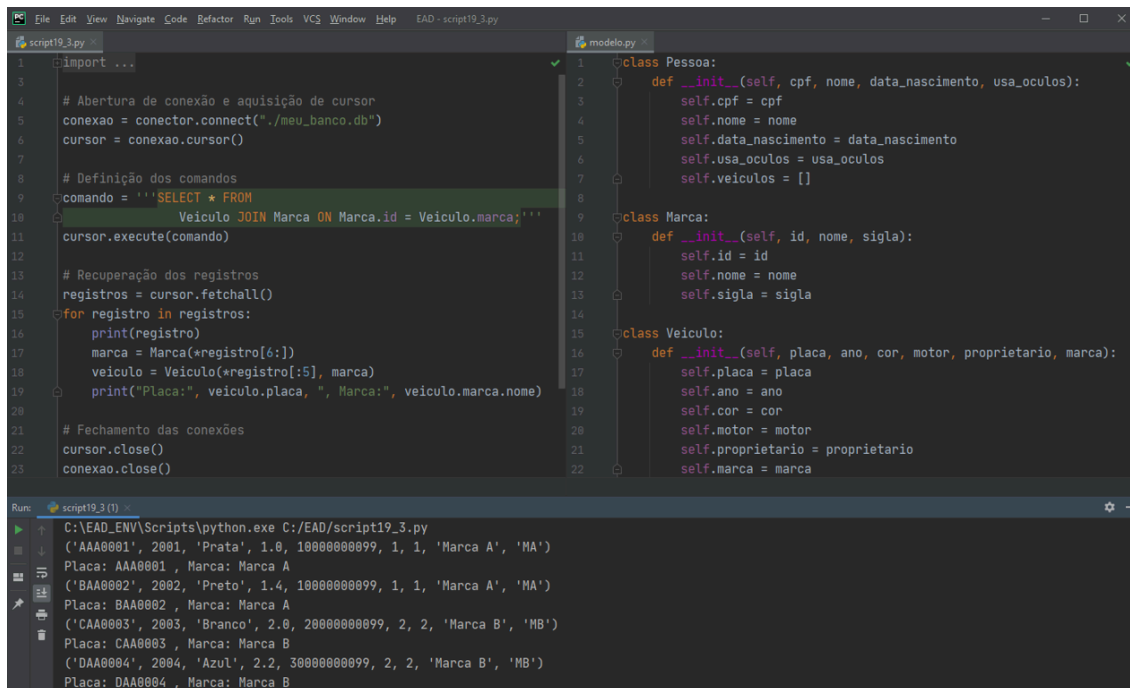
Na linha 18, criamos um objeto do tipo Veiculo utilizando o operador * e imprimimos os atributos placa e marca na linha 19.

Observe pelo console, que agora o atributo marca do nosso objeto do tipo Veiculo contém o nome da Marca.

No próximo exemplo, vamos dar um passo além. Vamos atribuir ao atributo marca, do Veiculo, um objeto do tipo Marca. Desta forma, vamos ter acesso a todos os atributos da

marca de um veículo.

Para isso, vamos fazer alguns ajustes no nosso modelo. Observe a Figura 25 a seguir.



```
1 import ...
2
3 # Abertura de conexão e aquisição de cursor
4 conexao = conector.connect("./meu_banco.db")
5 cursor = conexao.cursor()
6
7 # Definição dos comandos
8 comando = '''SELECT * FROM
9         Veiculo JOIN Marca ON Marca.id = Veiculo.marca;'''
10 cursor.execute(comando)
11
12 # Recuperação dos registros
13 registros = cursor.fetchall()
14 for registro in registros:
15     print(registro)
16     marca = Marca(*registro[6:])
17     veiculo = Veiculo(*registro[:5], marca)
18     print("Placa:", veiculo.placa, ", Marca:", veiculo.marca.nome)
19
20 # Fechamento das conexões
21 cursor.close()
22 conexao.close()
```

```
1 class Pessoa:
2     def __init__(self, cpf, nome, data_nascimento, usa_olhos):
3         self.cpf = cpf
4         self.nome = nome
5         self.data_nascimento = data_nascimento
6         self.usa_olhos = usa_olhos
7         self.veiculos = []
8
9 class Marca:
10     def __init__(self, id, nome, sigla):
11         self.id = id
12         self.nome = nome
13         self.sigla = sigla
14
15 class Veiculo:
16     def __init__(self, placa, ano, cor, motor, proprietario, marca):
17         self.placa = placa
18         self.ano = ano
19         self.cor = cor
20         self.motor = motor
21         self.proprietario = proprietario
22         self.marca = marca
```

Run: script19_3 (1)

```
C:\EAD_ENV\Scripts\python.exe C:/EAD/script19_3.py
('AAA0001', 2001, 'Prata', 1.0, 10000000099, 1, 1, 'Marca A', 'MA')
Placa: AAA0001, Marca: Marca A
('BAA0002', 2002, 'Preto', 1.4, 10000000099, 1, 1, 'Marca A', 'MA')
Placa: BAA0002, Marca: Marca A
('CAA0003', 2003, 'Branco', 2.0, 20000000099, 2, 2, 'Marca B', 'MB')
Placa: CAA0003, Marca: Marca B
('DAA0004', 2004, 'Azul', 2.2, 30000000099, 2, 2, 'Marca B', 'MB')
Placa: DAA0004, Marca: Marca B
```

Fonte: O Autor

 Figura: 25.

Vamos começar pelo nosso modelo. Adicionamos o id ao construtor da classe Marca.

Essa alteração foi feita para facilitar a criação do objeto do tipo Marca a partir da consulta no banco. Veremos o motivo mais à frente.

No script principal à esquerda da Figura 27, iniciamos o script com a abertura da conexão e criação do cursor.

Na sequência, na linha 9, criamos o comando *SQL SELECT* para retornar todas as colunas da tabela Veiculo e Marca, utilizando junção.

Na linha 11, executamos esse comando e os resultados da consulta foram recuperados na linha 14.

Na linha 15, iteramos sobre os registros recuperados.

Na linha 16, imprimimos cada tupla do registro da forma como foram retornados pelo conector. Vamos destacar um exemplo a seguir, que também pode ser observado no console.

('AAA0001', 2001, 'Prata', 1.0, 10000000099, 1, 1, 'Marca A', 'MA')

Destacado em vermelho, temos os atributos relacionados à entidade Veiculo e, em azul, temos os atributos relacionados à entidade Marca.

Na linha 17, utilizamos array slice para selecionar apenas os atributos da Marca. O resultado do slice para o exemplo anterior é a tupla (1, 'Marca A', 'MA'), onde temos os atributos id, nome e sigla. Essa tupla é utilizada em conjunto com o operador * para criarmos um objeto do tipo Marca. Como agora temos acesso ao id da Marca, foi necessário adicionar o id ao construtor da classe Marca.

Para ilustrar, após o slice e desempacotamento, a linha 17 pode ser traduzida para o seguinte código:

```
marca = Marca(1, 'Marca A', 'MA')
```

Na linha 18, utilizamos array slice para selecionar apenas os atributos do Veiculo, que retornou a tupla ('AAA0001', 2001, 'Prata', 1.0, 10000000099), onde temos os atributos placa, ano, cor, motor e proprietário. Observe que removemos o id da Marca no slice. Fizemos isso, pois ele será substituído pelo objeto marca criado na linha 17.

Após o slice e desempacotamento, a linha 18 pode ser traduzida para o seguinte código:

```
veiculo = Veiculo('AAA0001', 2001, 'Prata', 1.0, 10000000099, marca)
```

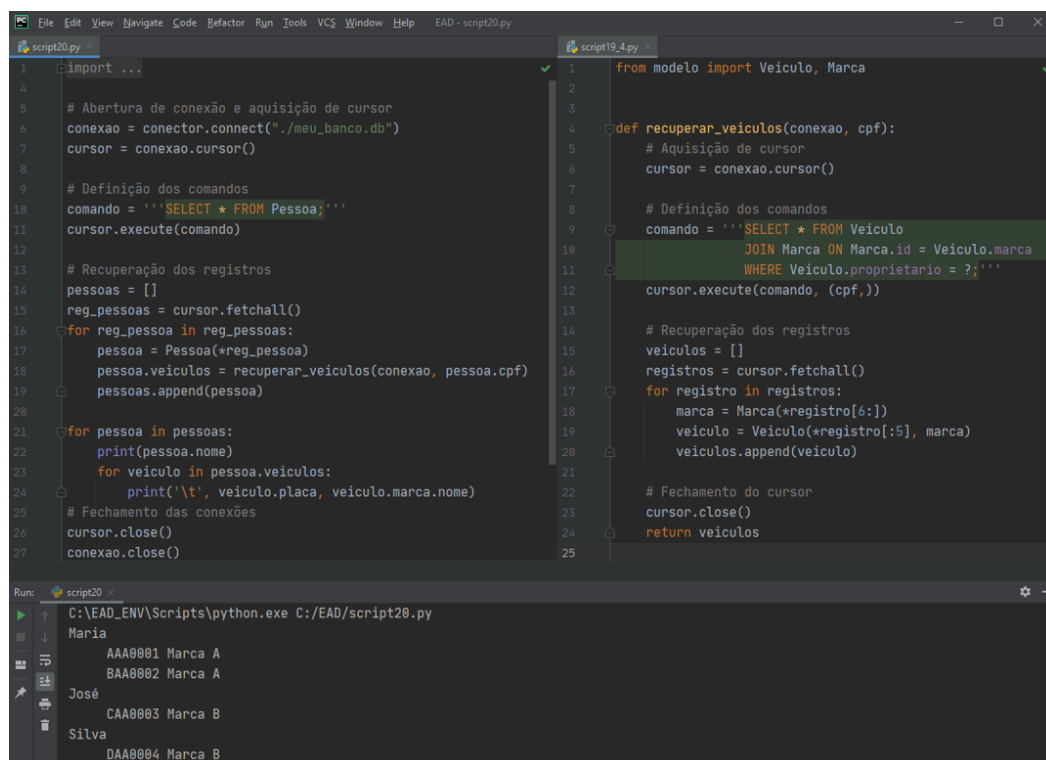
Na linha 19, imprimimos a placa do veículo e o nome da marca. Observe que o atributo marca, do Veiculo, faz uma referência ao objeto marca, por isso fomos capazes de acessar veiculo.marca.nome.

No final do script, fechamos a conexão e o cursor.

SELEÇÃO DE REGISTROS RELACIONADOS

Para finalizar, vamos recuperar todas as pessoas, com seus respectivos veículos e marcas.

Para isso, vamos transformar o script anterior em uma função, de forma que possamos utilizá-la no nosso script final. Observe a Figura 26 a seguir.



```
script20.py
1 import ...
2
3 # Abertura de conexão e aquisição de cursor
4 conexao = conector.connect("./meu_banco.db")
5 cursor = conexao.cursor()
6
7 # Definição dos comandos
8 comando = '''SELECT * FROM Pessoa;'''
9 cursor.execute(comando)
10
11 # Recuperação dos registros
12 pessoas = []
13 reg_pessoas = cursor.fetchall()
14
15 for reg_pessoa in reg_pessoas:
16     pessoa = Pessoa(*reg_pessoa)
17     pessoa.veiculos = recuperar_veiculos(conexao, pessoa.cpf)
18     pessoas.append(pessoa)
19
20 for pessoa in pessoas:
21     print(pessoa.nome)
22     for veiculo in pessoa.veiculos:
23         print('\t', veiculo.placa, veiculo.marca.nome)
24
25 # Fechamento das conexões
26 cursor.close()
27 conexao.close()

script19_4.py
1 from modelo import Veiculo, Marca
2
3 def recuperar_veiculos(conexao, cpf):
4     # Aquisição de cursor
5     cursor = conexao.cursor()
6
7     # Definição dos comandos
8     comando = '''SELECT * FROM Veiculo
9         JOIN Marca ON Marca.id = Veiculo.marca
10        WHERE Veiculo.proprietario = ?;'''
11     cursor.execute(comando, (cpf,))
12
13     # Recuperação dos registros
14     veiculos = []
15     registros = cursor.fetchall()
16
17     for registro in registros:
18         marca = Marca(*registro[6:])
19         veiculo = Veiculo(*registro[:5], marca)
20         veiculos.append(veiculo)
21
22     # Fechamento do cursor
23     cursor.close()
24     return veiculos

Run:
C:\EAD_ENV\Scripts\python.exe C:/EAD/script20.py
Maria
AAA0001 Marca A
BAA0002 Marca A
José
CAA0003 Marca B
Silva
DAA0004 Marca B
```

Fonte: O Autor

 Figura: 26.

À direita da imagem, temos o script 19_4. Utilizamos como base o script do exemplo anterior, script19_3, para criar uma função que retorne uma lista dos veículos de uma determinada pessoa.

Essa função tem como parâmetros uma conexão e o cpf de uma pessoa. Esse cpf será utilizado para filtrar os veículos que ela possui. Para isso, utilizamos o delimitador “?” na linha 11 e passamos o cpf da pessoa como argumento para o comando execute da linha 14.

Na linha 15, criamos uma lista vazia, chamada veiculos. Essa lista será povoada com os veículos recuperados pela consulta ao longo do laço for das linhas 17 a 20.

Ao final, fechamos o cursor e retornamos a lista veiculos.

À esquerda da figura, temos nosso script final, script20, cujo objetivo é ter uma lista com as pessoas cadastradas no banco de dados, incluindo seus veículos e marcas.

Após criar a conexão e o cursor, criamos o comando SQL para recuperar as pessoas na linha 10, que foi executado na linha 11.

Na linha 14, criamos a variável do tipo lista pessoas e, na linha 15, recuperamos todos os registros das pessoas utilizando a função fetchall do cursor.

Na linha 16, iteramos pelas pessoas recuperadas e para cada uma, criamos um objeto do tipo Pessoa (linha 17) e recuperamos seus veículos (linha 18).

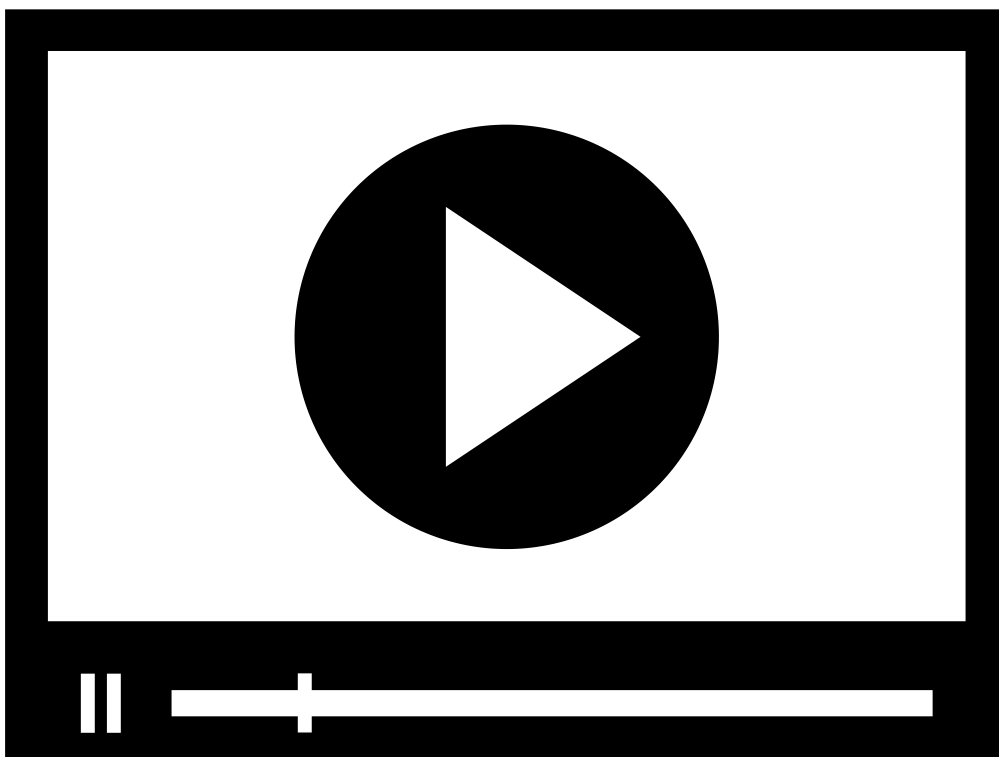
Para recuperar os veículos, utilizamos a função `recuperar_veiculos` do script 19_4, passando como argumento a conexão criada na linha 6 e o cpf da pessoa.

Na linha 19, adicionamos cada pessoa à lista `personas` criada anteriormente.

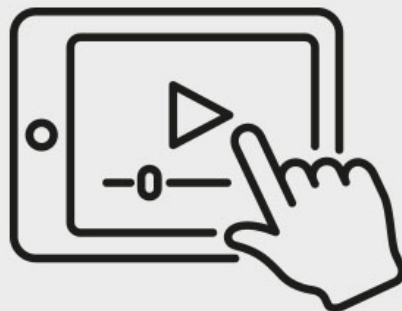
Na linha 21, iteramos sobre a lista `personas` e imprimimos seu nome e a lista de veículos que possui.

Ao final, fechamos a conexão e o cursor.

Verifique a saída do console abaixo da figura.



Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



VERIFICANDO O APRENDIZADO

1. PARA CONECTORES QUE IMPLEMENTEM A DB API 2.0, QUAL A CLASSE RETORNADA PELO MÉTODO FETCHALL DO TIPO CURSOR APÓS EXECUTAR UM COMANDO SQL SELECT QUE NÃO RETORNOU REGISTROS?

- A) list
- B) tuple
- C) dict
- D) None

2. CONSIDERE QUE TEMOS AS SEGUINTEs TABELAS E REGISTROS NO NOSSO BANCO DE DADOS:

ARREIMATE

ID	DATA	COD_PROD	LANCE_VENCEDOR
1	01-10-2020	10	55.0
2	01-10-2020	20	39.0
3	01-10-2020	30	110.0

❑ ATENÇÃO! PARA VISUALIZAÇÃO COMPLETA DA TABELA UTILIZE A ROLAGEM HORIZONTAL

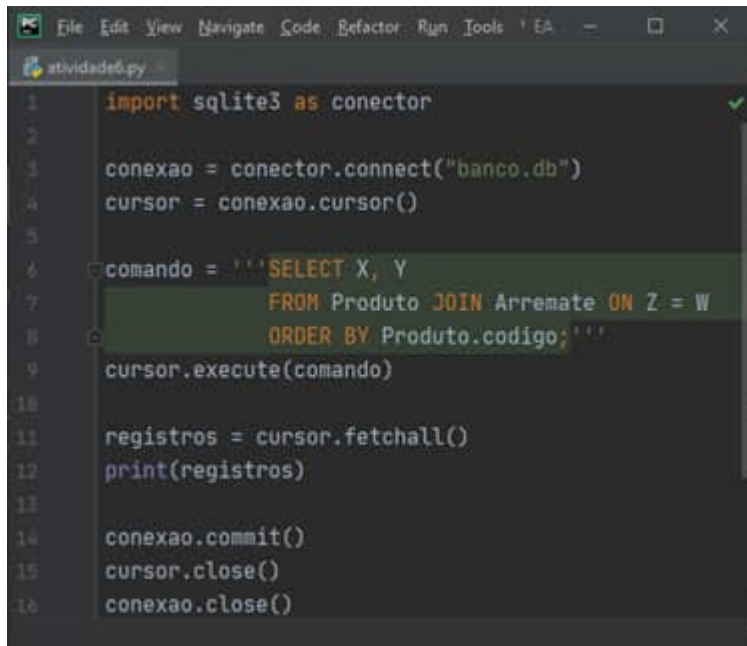
PRODUTO

CODIGO	NOME	DESCRIÇÃO	PRECO_INICIAL
10	FACA	FACA DE PORCELANA	50.0
20	GARFO	GARFO DE AÇO INOX	30.0
30	PRATO	PRATO DE PORCELANA	100.0

❑ ATENÇÃO! PARA VISUALIZAÇÃO COMPLETA DA TABELA UTILIZE A ROLAGEM HORIZONTAL

QUAIS OS VALORES DE X, Y, Z E W DO SCRIPT A SEGUIR PARA QUE A LINHA 12 PROGRAMA IMPRIMA OS VALORES:

[('FACA', 55.0), ('GARFO', 39.0), ('PRATO', 110.0)]



```
1 import sqlite3 as conector
2
3 conexao = conector.connect("banco.db")
4 cursor = conexao.cursor()
5
6 comando = '''SELECT X, Y
7             FROM Produto JOIN Arremate ON Z = W
8             ORDER BY Produto.codigo;'''
9 cursor.execute(comando)
10
11 registros = cursor.fetchall()
12 print(registros)
13
14 conexao.commit()
15 cursor.close()
16 conexao.close()
```

FONTE: O AUTOR

- A) Produto.codigo; Arremate.lance_vencedor; Produto.nome; e Arremate.cod_prod
- B) Arremate.cod_vencedor; Produto.codigo; Produto.nome; e Arremate.lance_vencedor
- C) Produto.nome; Arremate.preco_inicial; Arremate.cod_prod e Produto.codigo;
- D) Produto.nome; Arremate.lance_vencedor; Produto.codigo; e Arremate.cod_prod

GABARITO

1. Para conectores que implementem a DB API 2.0, qual a classe retornada pelo método fetchall do tipo Cursor após executar um comando SQL SELECT que não retornou registros?

A alternativa "A " está correta.

O método fetchall sempre retorna uma lista, mesmo que seja vazia!

2. Considere que temos as seguintes tabelas e registros no nosso banco de dados:

Arremate

ID	Data	cod_prod	lance_vencedor
----	------	----------	----------------

1	01-10-2020	10	55.0
2	01-10-2020	20	39.0
3	01-10-2020	30	110.0

☐ **Atenção! Para visualizaçãocompleta da tabela utilize a rolagem horizontal**

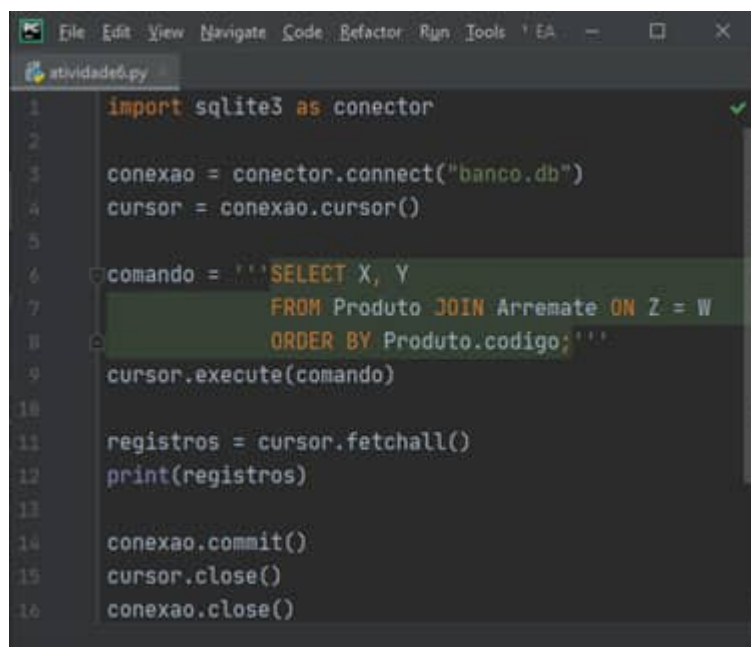
Produto

Codigo	Nome	descrição	preco_inicial
10	Faca	Faca de porcelana	50.0
20	Garfo	Garfo de aço inox	30.0
30	Prato	Prato de porcelana	100.0

☐ **Atenção! Para visualizaçãocompleta da tabela utilize a rolagem horizontal**

Quais os valores de X, Y, Z e W do script a seguir para que a linha 12 programa imprima os valores:

[('Faca', 55.0), ('Garfo', 39.0), ('Prato', 110.0)]



```
1 import sqlite3 as conector
2
3 conexao = conector.connect("banco.db")
4 cursor = conexao.cursor()
5
6 comando = '''SELECT X, Y
7             FROM Produto JOIN Arremate ON Z = W
8             ORDER BY Produto.codigo;'''
9 cursor.execute(comando)
10
11 registros = cursor.fetchall()
12 print(registros)
13
14 conexao.commit()
15 cursor.close()
16 conexao.close()
```

Fonte: O Autor

A alternativa "D " está correta.

Pela saída impressa pela linha 12, os dados dos registros retornados são Produto.nome e Arremate.lance_vencedor, que serão o X e Y, respectivamente. A junção precisa ser feita por colunas que possam ser “alinhadas”, ou seja, que possuam o mesmo tipo e valores em comum. Essas colunas são Produto.codigo; e Arremate.cod_prod, Z e W, respectivamente.

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Neste tema, visitamos as principais ações que podemos realizar na manipulação de registros em banco de dados.

Conhecemos as principais funcionalidades disponibilizadas pelas bibliotecas de gerenciamento de banco de dados que implementam a DB API 2.0, introduzida pela PEP 249.

Vimos como criar um banco de dados e iniciar uma conexão.

Além disso, aprendemos a aplicar as funcionalidades de inserção, remoção e atualização de registros.

Finalmente, empregamos as funcionalidades de recuperação de registros para selecionar e visualizar o conteúdo do banco de dados.

Para ouvir um *podcast* sobre o assunto, acesse a versão online deste conteúdo.



REFERÊNCIAS

HIPP, R. D. SQLite. *In*: SQLite. Consultado em meio eletrônico em: 08 out. 2020.

MYSQL. MySQL Connector. *In*: MySQL. Consultado em meio eletrônico em: 08 out. 2020.

PSYCOPG. Psycopg2. *In*: Psycopg. Consultado em meio eletrônico em: 08 out. 2020.

PYTHON. Python Software Foundation. *In*: Python. Consultado em meio eletrônico em: 08 out. 2020.

SQLITEBROWSER. DB Browser for SQLite. *In*: SQLitebrowser. Consultado em meio eletrônico em: 08 out. 2020.

EXPLORE+

Para saber mais sobre os assuntos tratados neste tema, pesquise na internet:

As páginas das bibliotecas SQLAlchemy e Peewee.

CONTEUDISTA

Frederico Tosta de Oliveira

 **CURRÍCULO LATTES**