

**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.

**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.

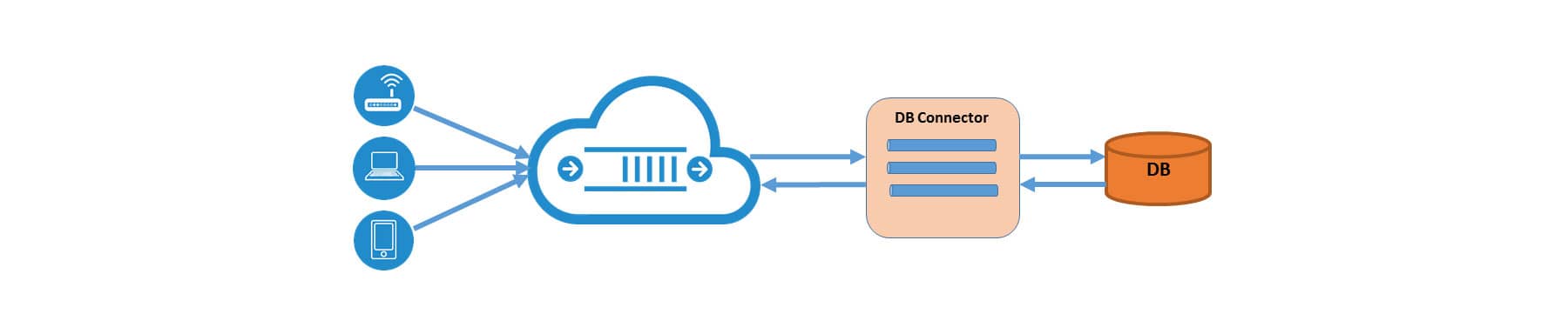
O SQLite também é muito utilizado durante o desenvolvimento das aplicações, para então ser substituído por um banco de dados mais robusto, como PostgreSQL ou MySQL, quando em produção.

**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.



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.

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**

Antes de começarmos a trabalhar diretamente com banco de dados, precisamos conhecer algumas classes e métodos disponibilizados pelos conetores 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:

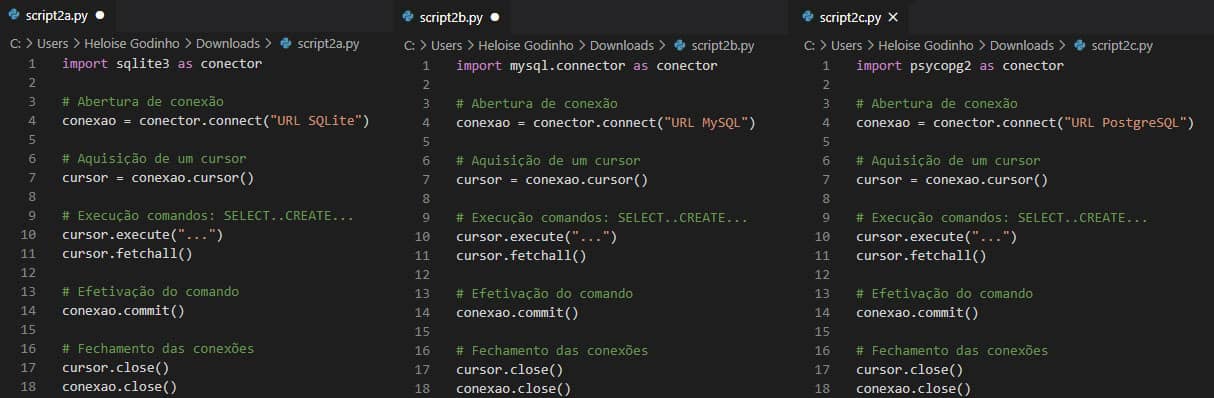
* Classe utilizada para gerenciar todas as operações no banco de dados.
* Principais métodos:
  + commit: Confirma todas as operações pendentes;
  + rollback: Desfaz todas as operações pendentes;
  + cursor: Retorna um objeto do tipo Cursor; e
  + close: Encerra a conexão com o banco de dados.

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.

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



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.

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.

**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

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. |

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

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.

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. |

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.

**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.



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.**

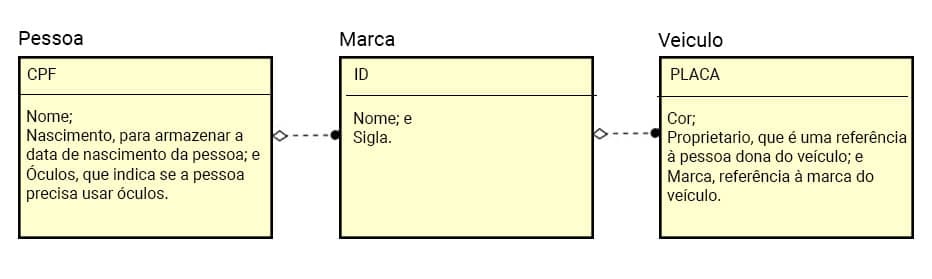
Caso o arquivo não exista, ele será criado automaticamente! O arquivo criado pode ser copiado e compartilhado.

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.

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  
- **oculos**, 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 Veiculo, que contém os atributos:  
- **placa**, como chave primária;  
- **cor**;  
- **proprietario**, 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:

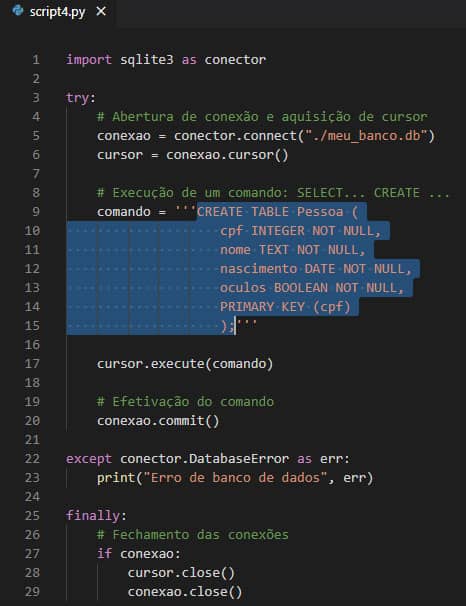
|  |  |
| --- | --- |
| CPF | INTEGER (chave primária, não nulo) |
| NOME | TEXT (não nulo) |
| NASCIMENTO | DATE (não nulo) |
| OCULOS | BOOLEAN (não nulo) |

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,  
oculos 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.



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.

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) |

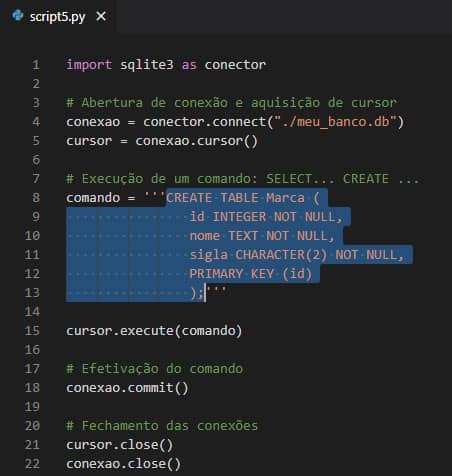
A codificação latin-1, muito utilizada no Brasil, utiliza um byte por caractere. Como a sigla da marca será composta por 2 caracteres, nosso atributo sigla terá tamanho 2 (CHAR(2) ou CHARACTER(2)).

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.



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) |

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.

**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.

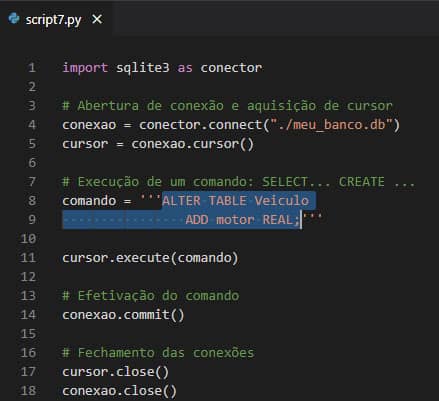
Para ilustrar, vamos adicionar mais um atributo à entidade Veiculo.

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.



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.

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.

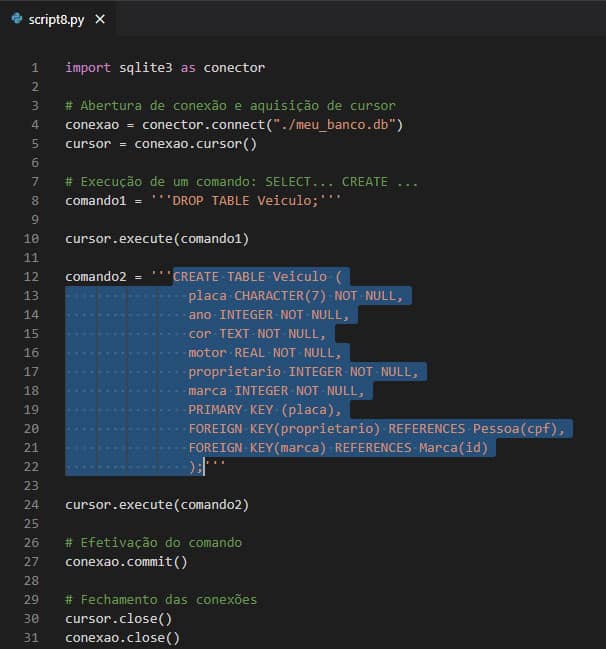
Para remover a tabela Veiculo, utilizamos o seguinte comando SQL.

***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:



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:



Algumas bibliotecas de acesso a banco de dados oferecem uma funcionalidade chamada mapeamento objeto-relacional, do inglês object-relational mapping [**(ORM)**](javascript:void(0)).

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 ***S*QLAlchemy** e **Peewee*.***

**MÓDULO 3**

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

**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)*

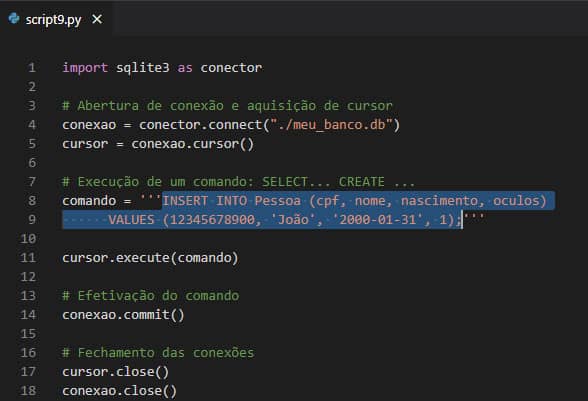
O Comando SQL para inserção desses dados é o seguinte:

***INSERT INTO Pessoa (cpf, nome, nascimento, oculos)*  
*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.



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.

**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))***

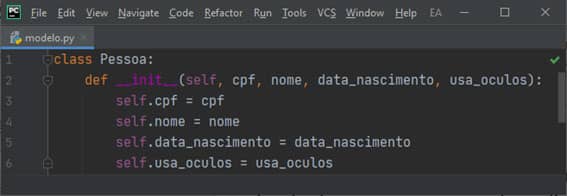
A partir da string e da tupla, é montado o comando final, que é traduzido para:

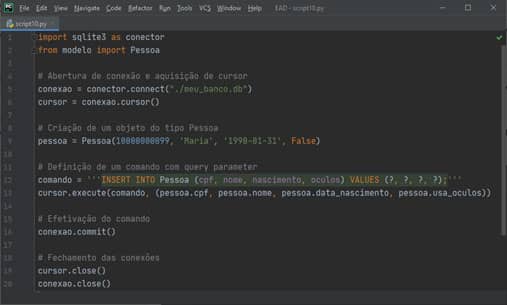
**“INSERT INTO tabela1 (atributo1, atributo2) VALUES (‘Teste’, 123);”**

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.

**Clique nas figuras abaixo.**

**[](javascript:void(0))**

**[](javascript:void(0))**

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, oculos)  
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.

**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!**

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, oculos)  
VALUES (12345678910,'Alice','20/11/1988',True);

**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!

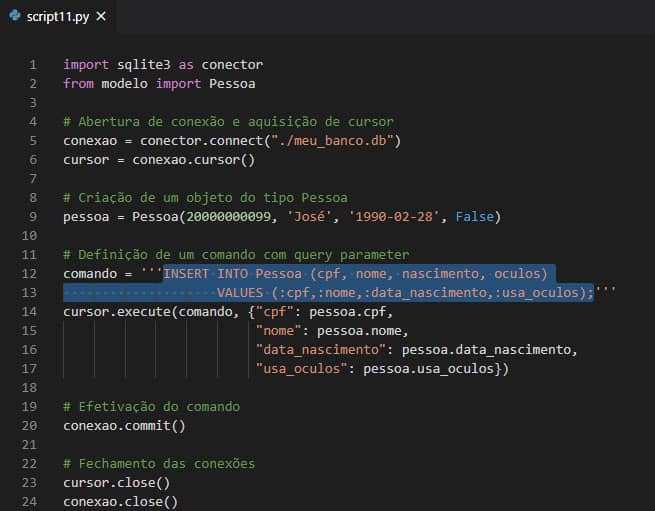
Veja a seguir como poderia ficar a chamada do método execute para esse comando:

**>>> 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.

****

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\_oculos. 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, oculos)  
VALUES (20000000099,'José','1990-02-28',0);**

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

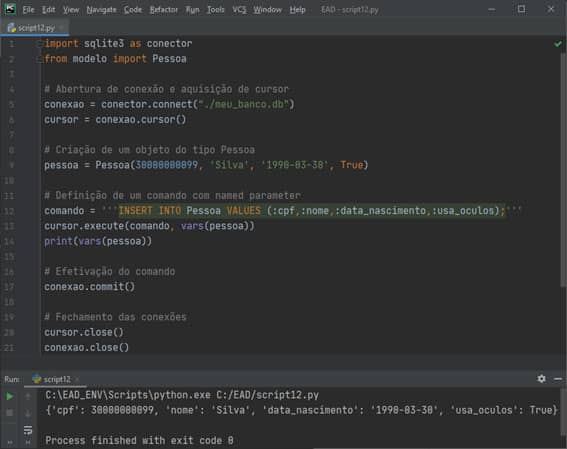
Observe que nosso código está crescendo. Imagine se tivéssemos uma entidade com dezenas de atributos? A chamada do método execute da linha 14 cresceria proporcionalmente, comprometendo muito a leitura do nosso código.

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

**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.



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\_oculos 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\_oculos': True}***

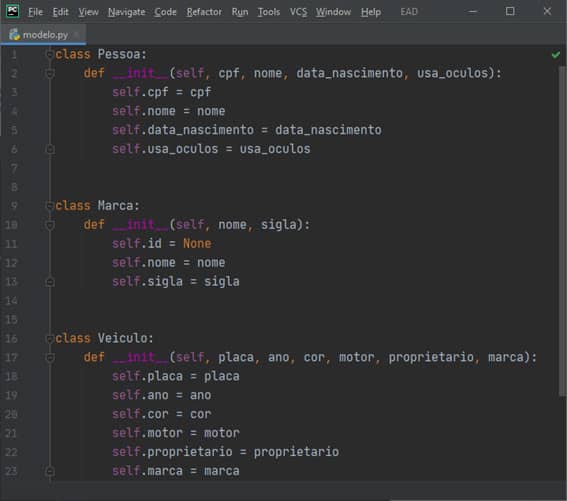
O comando final enviado ao banco de dados pelo comando execute foi:

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

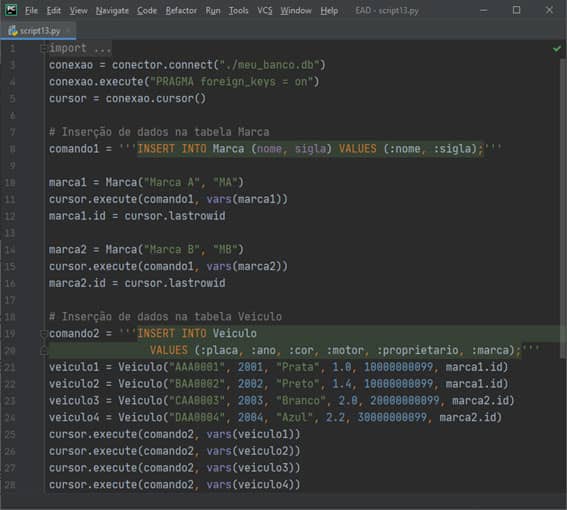
Na linha 17, efetivamos a transação e ao final do script fechamos o cursor e a conexão.

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.



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



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.