Python com banco de dados

Vamos estudar o desenvolvimento de aplicações para armazenamento e recuperação de informação em banco de dados utilizando a linguagem de programação Python, compreendendo os passos e as 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.

Prof. Frederico Tosta de Oliveira

Preparação

Para estudar este conteúdo, tenha a versão 3.7 do Python ou posterior instalada em seu computador. É possível pesquisar e baixar a versão mínima recomendada (3.7) no site oficial do Python. Utilizamos a IDE PyCharm Community para programar os exemplos aqui apresentados, e você pode baixá-la gratuitamente no site oficial da JetBrains. Para visualizar os dados e tabelas, utilizaremos o DB Browser for SQLite, também disponível gratuitamente. Para acompanhar melhor os exemplos, clique aqui para baixar o arquivo **Banco de dados.zip**, que contém os scripts utilizados neste material.

Objetivos

- Reconhecer as funcionalidades de frameworks e bibliotecas para gerenciamento de banco de dados.
- Empregar as funcionalidades para conexão, acesso e criação de bancos de dados e tabelas.
- Aplicar as funcionalidades para inserção, remoção e atualização de registros em tabelas.
- Empregar as funcionalidades para recuperação de registros em tabelas.

Introdução

Olá! Vamos abordar alguns dos principais tópicos relacionados à integração de programas escritos em Python e os SGBD mais adotados. Veremos como realizar as conexões, criar tabelas, relacionamentos e inserir novos registros no banco.

Além disto, aprenderemos a recuperar dados do banco utilizando consultas simples e até as mais complexas, implementando o comando JOIN para selecionar dados relacionados. Assista ao vídeo e confira!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Conectores para banco de dados

Vamos conhecer os principais conectores de bancos de dados em Python, essenciais para desenvolvedores que buscam integrar suas aplicações com diferentes sistemas de gerenciamento de bancos de dados (SGBDs).

> Conectores, como psycopg2 para PostgreSQL, MySQLclient, PyMySQL, MySQL Connector/Python para MySQL, e sqlite3 para SQLite, seguem a especificação Python Enhancement Proposals (PEP) 249, garantindo uma interface padronizada e consistente.

A escolha do conector adequado pode impactar a performance, a facilidade de instalação e a compatibilidade com recursos avançados do banco de dados, permitindo uma comunicação eficiente e segura entre o aplicativo e o SGBD.

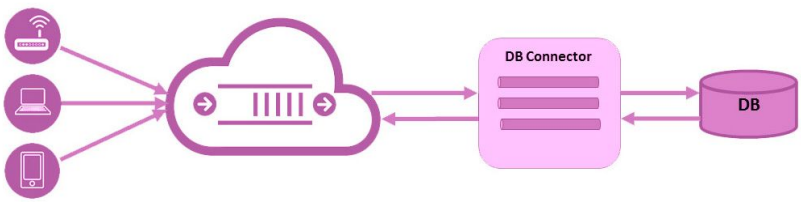
Confira neste vídeo os principais conectores de bancos de dados em Python e saiba como escolher o mais adequado para integrar suas aplicações com PostgreSQL, MySQL e SQLite.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

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.



Conectores.

Como estamos trabalhando com Python, devemos procurar por conectores que atendam a **PEP 249**.

PEP 249

Python Database API Specification v2.0 - DB-API 2.0.

Essa **PEP** 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.

PEP

Python Enhancement Proposal.

Veja a seguir exemplos de padrões definidos pela DB-API 2.0.

Nomes de métodos

Close, commit e cursor.

Nomes de classes

Connection e Cursor.

Tipos de exceção

IntegrityError e InternalError.

Acompanhe agora exemplos de conectores que implementam a DB-API 2.0.

1
psycopg2
Conector mais utilizado para o banco de dados PostgreSQL.

2
mysqlclient, PyMySQL e mysql-connector-python
Conectores para o banco de dados MySQL.

3
sqlite3
Adaptador padrão do Python para o banco de dados SQLite.

Explorando conectores para banco de dados em Python

Conectores para bancos de dados são bibliotecas escritas para facilitar as operações entre programa e aplicações de banco. Essas bibliotecas possuem interfaces mais enxutas e uma forma comum de criar conexões, executar operações CRUD e outras tarefas relacionadas à comunicação com bancos de dados.

Devido à variedade de fornecedores de bancos de dados, existem também diversos conectores para o Python. Inclusive, em alguns casos, podemos escolher entre diferentes conectores para o mesmo tipo de banco, veja!

Psycopg2

É uma biblioteca de código aberto em Python que serve para bancos de dados PostgreSQL, sendo a escolha ideal para conectar programas Python a este tipo de banco. É implementado em C e Python, oferecendo uma combinação de desempenho e facilidade de uso. O psycopg2 ainda suporta muitas funcionalidades avançadas do PostgreSQL, como tipos de dados específicos, transações complexas e recursos de notificação. Oferece um bom desempenho em razão de sua implementação parcialmente em C.

MySQLclient

É uma biblioteca baseada em C, resultando em um desempenho mais rápido em comparação com bibliotecas puramente em Python. É um sucessor do MySQL-python e oferece compatibilidade com a API DB-API 2.0, padrão em bibliotecas de banco de dados Python. Sua instalação pode ser mais complexa, especialmente no Windows, pois requer a presença de um compilador C e bibliotecas de desenvolvimento do MySQL.

PyMySQL

É escrito inteiramente em Python, facilitando a instalação, pois não há necessidade de um compilador C ou bibliotecas de desenvolvimento adicionais. Pode ser instalado facilmente em qualquer ambiente que suporte Python, tornando-o uma boa escolha para desenvolvimento e ambientes onde a instalação de compiladores é difícil. Geralmente mais lento que outros conectores devido à ausência de código nativo em C.

MySQL Connector

É desenvolvido e mantido pela Oracle, a mesma empresa responsável pelo MySQL, o que garante maior compatibilidade e suporte. Assim como o PyMySQL, é escrito totalmente em Python, facilitando a instalação, pois não exige compiladores ou bibliotecas adicionais. Pode ser instalado via pip, sem pré-requisitos extras. Embora seja mais lento que o MySQLclient devido à ausência de código nativo em C, seu desempenho é geralmente comparável ao do PyMySQL. Além disso, este conector suporta recursos avançados do MySQL, como conectividade SSL/TLS, autenticação pluggable e suporte completo para Unicode.

SQLite

É um banco de dados embutido que não requer um servidor separado. Todo o banco de dados é armazenado em um único arquivo, sem necessidade de instalação ou configuração complexa, o que o torna ideal para aplicações pequenas e médias, desenvolvimento e testes. É muito rápido para ler e escrever em arquivos de tamanho pequeno a médio, mas pode não ser tão eficiente para grandes volumes de dados ou cargas de trabalho intensas. Por ser fácil de usar, é comum em aplicativos móveis, navegadores e outras aplicações que valorizam simplicidade e leveza.

Atividade 1

Questão

Para desenvolver aplicações Python que interajam com diferentes sistemas de gerenciamento de bancos de dados (SGBDs), precisamos selecionar o conector apropriado. Considere as seguintes opções de conectores para PostgreSQL e MySQL. Qual conector é mais adequado para se conectar a um banco de dados PostgreSQL em uma aplicação Python?

A

MySQLclient

B

PyMySQL

C

psycopg2

D

MySQL Connector/Python

E

sqlite3

A alternativa C está correta.

O psycopg2 é especificamente projetado para conectar aplicações Python a bancos de dados PostgreSQL, oferecendo suporte avançado e desempenho otimizado para esse SGBD. As outras alternativas são conectores para MySQL ou SQLite.

Principais métodos e exceções dos conectores em Python

Para o desenvolvimento eficiente de aplicações que interagem com bancos de dados, é necessário compreendermos os principais métodos e exceções dos conectores em Python.

A partir da sua utilização adequada, seus desenvolvedores podem gerenciar conexões e manipular dados de forma robusta, seguindo padrões da DB-API 2.0 e garantindo a integridade dos dados e a eficácia das operações em ambientes empresariais.

Neste vídeo, veremos como utilizar connect, commit e execute, bem como manejar exceções como IntegrityError e OperationalError. Assista!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Ao utilizarmos um conector de banco de dados no nosso programa, adicionamos diversas possibilidades e também desafios ao desenvolvimento. Enquanto podemos utilizar classes e métodos para executar tarefas e criar representações, temos que lidar com as possíveis exceções retornadas pelo conector.

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

Utilizaremos o banco de dados SQLite para demonstrar o desenvolvimento de aplicações Python para banco de dados. Apesar do nome Lite, o SQLite é um banco de dados completo, que permite a criação de tabelas, relacionamentos, índices, gatilhos e visões.

Representação de banco de dados.

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.

Atenção

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. Veja 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.
- 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.
- 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. Acompanhe a descrição desse fluxo!

Conectar

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

Executar

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

Enviar comandos

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

- Criar tabelas.
- Inserir linhas.
- Selecionar linhas.

Confirmar

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

Fechar

Fechar o cursor e a conexão.

Observe o script a seguir, no qual temos esse fluxo de trabalho descrito na forma de código Python.

```
python
```

```python
import sqlite3 as conector

# Abertura de conexão
conexao = conector.connect("URL SQLite")

# Aquisição de um cursor
cursor = conexao.cursor()

# Execução comandos: SELECT..CREATE...
cursor.execute("...")
cursor.fetchall()

# Efetivação do comando
conexao.commit()

# Fechamento das conexões
cursor.close()
conexao.close()
```

Agora vamos revisar o código que criamos para interagir com o banco de dados SQLite.

Na linha 1

Importamos o módulo sqlite3, conector para o banco de dados SQLite disponível nativamente a partir do Python 3.7. Atribuímos um alias (apelido) a esse import chamado conector.

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 conexão.

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 conteúdo. Basicamente, preencheremos o parâmetro do método execute com o comando SQL pertinente.

A seguir, veja o mesmo código anterior adaptado para conexão com o **MySQL** e o **PostgreSQL**.

```python
import mysql.connector as conector

# Abertura de conexão
conexao = conector.connect("URL MySQL")

# Aquisição de um cursor
cursor = conexao.cursor()

# Execução comandos: SELECT..CREATE...
cursor.execute("...")
cursor.fetchall()

# Efetivação do comando
conexao.commit()

# Fechamento das conexões
cursor.close()
conexao.close()
```

```python
import psycopg2 as conector

# Abertura de conexão
conexao = conector.connect("URL PostgreSQL")

# Aquisição de um cursor
cursor = conexao.cursor()

# Execução comandos: SELECT..CREATE...
cursor.execute("...")
cursor.fetchall()

# Efetivação do comando
conexao.commit()

# Fechamento das conexões
cursor.close()
conexao.close()
```

Agora observe os scripts 2 e 3 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 porque todas elas seguem a especificação DB-API 2.0.

Comentário

Cada biblioteca tem suas particularidades, que podem atender a funcionalidades específicas 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 em que 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. Veja!

Error

Classe base para as exceções. É a mais abrangente.

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.

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.

DatabaseError

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

ProgrammingError

Exceção para tratar erros relacionados à programação, como sintaxe incorreta do comando SQL, tabela não encontrada etc. 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.

Atividade 2

Questão

Em um ambiente empresarial, garantir a integridade e a eficiência das operações de banco de dados é fundamental para o sucesso de uma aplicação. Portanto, após cada transação, é necessário executar um comando de confirmação no banco de dados. Qual método é utilizado para confirmar todas as operações pendentes em uma conexão de banco de dados em Python?

A

rollback

B

execute

C

fetchall

D

commit

E

close

A alternativa D está correta.

O método commit é utilizado para confirmação de todas as operações pendentes em uma conexão de banco de dados, garantindo que as alterações sejam salvas permanentemente.

Tipos de dados

Para garantir a eficiência e integridade das aplicações, é importante entender os tipos de dados usados em bancos de dados, como inteiros, texto e ponto flutuante. Cada tipo tem características que afetam como são armazenados e manipulados. Escolher os tipos certos melhora o desempenho, facilita a manutenção e assegura a compatibilidade entre diferentes SGBDs, sendo uma habilidade essencial no mercado.

Neste vídeo, você vai aprender sobre a importância dos tipos de dados em bancos de dados e como eles impactam o armazenamento e a eficiência das aplicações. Descubra os diferentes tipos, como INTEGER, TEXT, REAL e BLOB, e como escolher o correto para garantir a integridade e o desempenho dos seus sistemas.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

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 **afinidades de tipo** para as colunas.

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.

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.

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

Tipo no CREATE TABLE	Afinidade
REAL DOUBLE DOUBLE PRECISION FLOAT	REAL
NUMERIC DECIMAL(10,5) BOOLEAN DATE DATETIME	NUMERIC

Tabela: Afinidades do SQLite.
Frederico Tosta de Oliveira.

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

Atividade 3

Questão

A escolha adequada dos tipos de dados é fundamental para o desempenho e a integridade dos bancos de dados em aplicações reais. Qual tipo de dado é usado para armazenar valores de texto em um banco de dados SQLite?

A

INTEGER

B

TEXT

C

BLOB

D

REAL

E

NUMERIC

A alternativa B está correta.

No SQLite, o tipo de dado TEXT é utilizado para armazenar valores de texto (strings). Isso permite o armazenamento de dados textuais de forma eficiente e adequada, conforme suas características.

Conexão a banco de dados e criação de tabelas

Para um desenvolvedor, é fundamental saber conectar um script Python a um banco de dados e criar tabelas para organizar os dados corretamente. Isso permite definir tipos de dados, chaves primárias e estrangeiras, garantindo a integridade dos dados e possibilitando o desenvolvimento de sistemas robustos e escaláveis.

Neste vídeo, você aprenderá a conectar scripts Python a bancos de dados como SQLite, PostgreSQL e MySQL, e a criar tabelas usando SQL. Veja como implementar o comando CREATE TABLE em Python para criar tabelas e definir chaves, garantindo a integridade dos dados. Confira!

> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

Conectando a um banco de dados

Agora 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.

```
python

>>> 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.

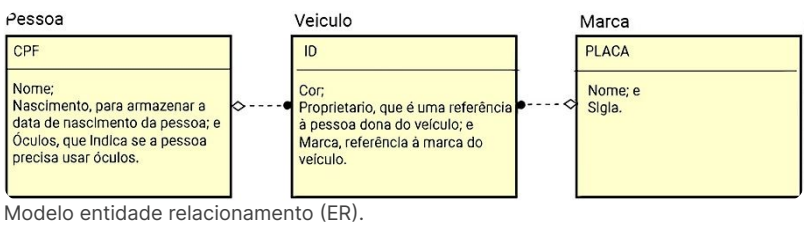
Caso o arquivo não exista, ele será criado automaticamente! O arquivo criado pode ser copiado e compartilhado.

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 tabelas.

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



Modelo entidade relacionamento (ER).

Nosso modelo é composto por três entidades: Pessoa, Veículo e Marca. Conheça mais sobre elas!

Pessoa

Contém estes atributos:

- CPF, como chave primária.
- Nome.
- Nascimento, para armazenar a data de nascimento da pessoa.
- Óculos, que indica se a pessoa precisa usar óculos.

Marca

Contém estes atributos:

- Id, como chave primária.
- Nome.
- Sigla.

Veículo

Contém estes atributos:

- Placa, como chave primária.
- Cor.
- Proprietário, que é uma referência à pessoa dona do veículo.
- 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 do SQLite**.

Vamos definir a entidade Pessoa, de forma que os atributos tenham os seguintes tipos e restrições. Observe!

CPF

INTEGER (chave primária, não nulo).

Nome

TEXT (não nulo).

Nascimento

DATE (não nulo).

Óculos

BOOLEAN (não nulo).

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

```plain-text
CREATE TABLE Pessoa (
  cpf INTEGER NOT NULL,
  nome TEXT NOT NULL,
  nascimento DATE NOT NULL,
  oculos BOOLEAN NOT NULL,
  PRIMARY KEY (cpf)
);
```

Observe pela 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 próximo exemplo.

```
python
 import sqlite3 as conector

 try:
    # Abertura de conexão e aquisição de cursor
    conexao = conector.connect("./meu_banco.db")
    cursor = conexao.cursor()

   # Execução de um comando: SELECT... CREATE ...
    comando = '''CREATE TABLE Pessoa (
                    cpf INTEGER NOT NULL,
                    nome TEXT NOT NULL,
                    nascimento DATE NOT NULL,
                    oculos BOOLEAN NOT NULL,
                    PRIMARY KEY (cpf)
                    );'''

     cursor.execute(comando)

     # Efetivação do comando
     conexao.commit()

  except conector.DatabaseError as err:
     print("Erro de banco de dados", err)

  finally:
     # Fechamento das conexões
     if conexao:
         cursor.close()
         conexao.close()
```

Vamos agora analisar o passo a passo do código apresentado.

Na linha 1

Importamos o módulo sqlite3 e atribuímos o alias conector.

Na linha 5

Conectamos ao banco de dados meu_banco.db; se o arquivo não existir, ele será criado no diretório atual do script.

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

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

Atenção

Nos exemplos ao longo deste conteúdo, 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).

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.

```plain-text
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 o código a seguir, para verificar como ficou o script de criação dessa tabela.

```
python

  import sqlite3 as conector

  # Abertura de conexão e aquisição de cursor
  conexao = conector.connect("./meu_banco.db")
  cursor = conexao.cursor()

  # Execução de um comando: SELECT... CREATE ...
  comando = '''CREATE TABLE Marca (
                 id INTEGER NOT NULL,
                 nome TEXT NOT NULL,
                 sigla CHARACTER(2) NOT NULL,
                 PRIMARY KEY (id)
                 );'''

   cursor.execute(comando)

   # Efetivação do comando
   conexao.commit()

   # Fechamento das conexões
   cursor.close()
   conexao.close()
```

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 Veículo. 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).

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 proprietário será utilizado para indicar um relacionamento da entidade Veículo 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.

O atributo marca será utilizado para indicar um relacionamento da entidade Veículo 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 Veículo e o relacionamento com as tabelas Pessoa e Marca.

```plain-text
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 a seguir.

```python
import sqlite3 as conector

# Abertura de conexão e aquisição de cursor
conexao = conector.connect("./meu_banco.db")
cursor = conexao.cursor()

# Execução de um comando: SELECT... CREATE ...
comando = '''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)
                );'''

 cursor.execute(comando)

 # Efetivação do comando
 conexao.commit()

 # Fechamento das conexões
 cursor.close()
 conexao.close()
```

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

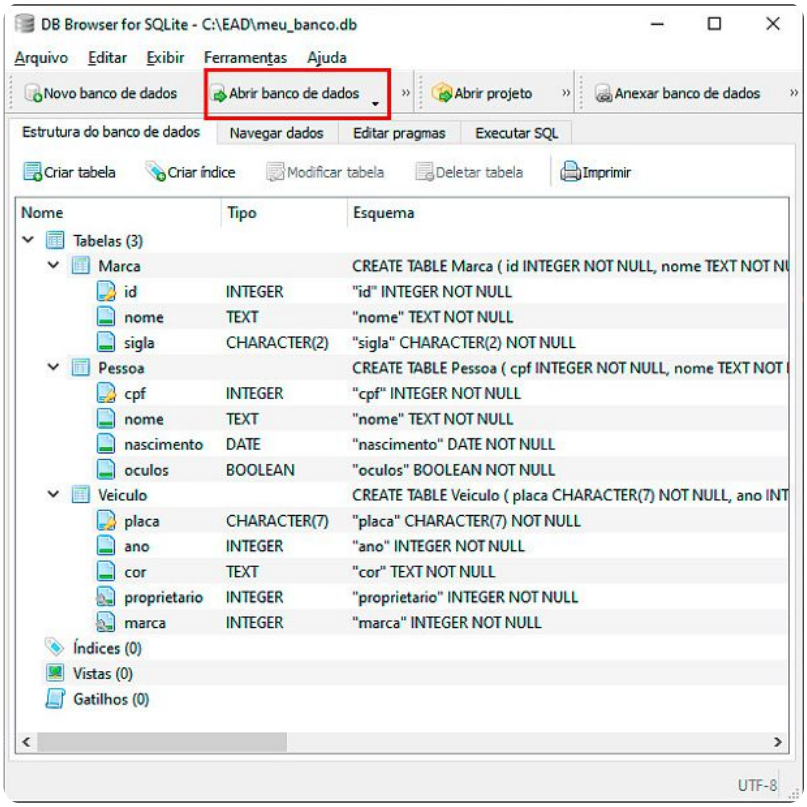
Esse comando foi executado na linha 19 e efetivado na linha 22.

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.



DB Browser for SQLite.

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

Atividade 1

Questão

A criação de tabelas em um banco de dados permite organizar e estruturar dados de forma eficiente. Qual comando SQL é utilizado para criar uma tabela no banco de dados?

A

INSERT TABLE

B

DELETE TABLE

C

UPDATE TABLE

D

CREATE TABLE

E

ALTER TABLE

A alternativa D está correta.

O comando SQL CREATE TABLE é utilizado para criar uma nova tabela no banco de dados, especificando os nomes das colunas e os tipos de dados para cada coluna.

Alteração e remoção de tabela

Os processos de alteração e remoção de tabelas em bancos de dados são essenciais para manter a flexibilidade e adaptabilidade dos sistemas. Durante o desenvolvimento, pode ser necessário ajustar tabelas para adicionar ou remover colunas. Dominar comandos como ALTER TABLE e DROP TABLE permite modificar o banco de dados rapidamente à medida que o projeto evolui, mantendo o sistema eficiente e funcional.

Neste vídeo, veremos como alterar e remover tabelas em bancos de dados usando SQL. Adicionaremos novos atributos com ALTER TABLE e removeremos tabelas com DROP TABLE, garantindo a flexibilidade das estruturas de dados nas suas aplicações. Assista!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Neste momento, temos o nosso banco com as três tabelas exibidas no modelo ER. Durante o desenvolvimento, pode ser necessário realizar alterações no nosso modelo e, consequentemente, nas nossas tabelas. Vamos ver agora como podemos fazer para adicionar um novo atributo e como 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 Veículo. 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 definilo como do tipo REAL.

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

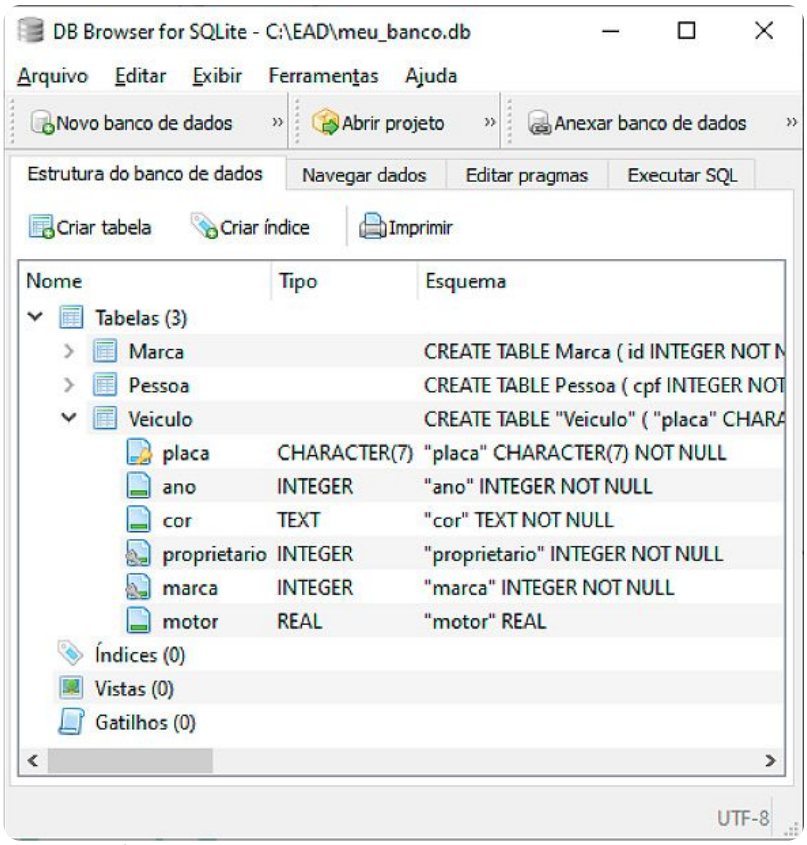
```plain-text
ALTER TABLE Veiculo
ADD motor REAL;
```

Confira o script a seguir, no qual realizamos a alteração da tabela Veículo.

```
python

  import sqlite3 as conector

  # Abertura de conexão e aquisição de cursor
  conexao = conector.connect("./meu_banco.db")
  cursor = conexao.cursor()

  # Execução de um comando: SELECT... CREATE ...
  comando = '''ALTER TABLE Veiculo
                ADD motor REAL;'''

   cursor.execute(comando)

   # Efetivação do comando
   conexao.commit()

   # Fechamento das conexões
   cursor.close()
   conexao.close()
```

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 a tabela após a criação da nova coluna.



DB Browser for SQLite.

Na imagem, o atributo motor foi adicionado ao final da entidade, obedecendo a ordem de criação.

Exemplo

Em algumas situações, as colunas precisam seguir uma ordem específica. Um cenário comum é quando carregamos dados de uma planilha diretamente para um banco de dados, para fazer uma inserção em massa (bulk insert). Nesses casos, as colunas da planilha devem estar na mesma ordem 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 esta sequência:

- Placa
- Ano
- Cor
- Motor
- Proprietário
- Marca

No exemplo a seguir, vamos remover a tabela Veículo, utilizando o comando DROP TABLE do SQL e, posteriormente, vamos criá-la novamente com a sequência desejada.

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

```plain-text
DROP TABLE Veiculo;
```

Para recriar a tabela, utilizamos este comando SQL:

```plain-text
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)
);
```

Veja como ficou nosso script!

```python
import sqlite3 as conector

# Abertura de conexão e aquisição de cursor
conexao = conector.connect("./meu_banco.db")
cursor = conexao.cursor()

# Execução de um comando: SELECT... CREATE ...
comando1 = '''DROP TABLE Veiculo;'''

cursor.execute(comando1)

comando2 = '''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)
               );'''

cursor.execute(comando2)

# Efetivação do comando
conexao.commit()

# Fechamento das conexões
cursor.close()
conexao.close()
```

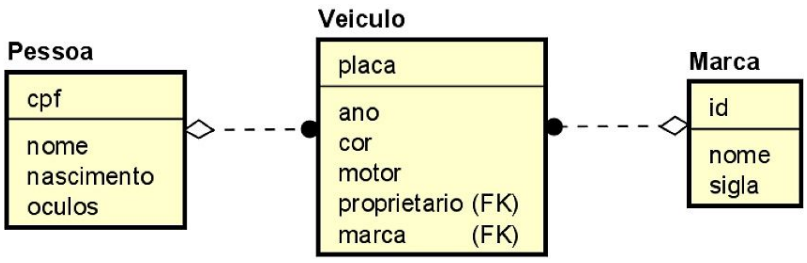
Após se conectar ao banco e obter o cursor, criamos a string comando1 com o comando para remover a tabela Veículo, na linha 8. Na linha 10, executamos esse comando.

Nas linhas 12 a 22, criamos o comando para criar novamente a tabela Veículo 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.

Veja o resultado do nosso modelo ER!



Modelo ER.

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 eles representam.

Esse mapeamento permite associar classes definidas em Python com tabelas em banco de dados, em que 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.

Exemplo

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

Atividade 2

Questão

Alterar e remover tabelas em um banco de dados ajuda a manter a flexibilidade e a adaptabilidade das aplicações. Qual comando SQL é utilizado para adicionar uma nova coluna a uma tabela existente?

A

CREATE COLUMN

B

ALTER TABLE

C

ADD COLUMN

D

MODIFY TABLE

E

UPDATE TABLE

A alternativa B está correta.

O comando SQL ALTER TABLE é utilizado para modificar a estrutura de uma tabela existente, incluindo a adição de novas colunas. Isso permite a atualização e a expansão das tabelas conforme necessário.

Desenvolvendo um script Python com conexão a banco de dados

No desenvolvimento de software moderno, é preciso criar programas com conexão a bancos de dados, especialmente quando utilizamos Python. Assim, as aplicações podem armazenar, recuperar e manipular dados de maneira segura, atendendo às diversas demandas empresariais e tecnológicas.

> Bancos de dados são fundamentais para a gestão de informações em sistemas de gerenciamento de conteúdo, e-commerce, aplicativos financeiros, jogos e muito mais.

Conectar uma aplicação Python a um banco de dados possibilita a implementação de funcionalidades dinâmicas, como registro de usuários, processamento de transações, geração de relatórios e análise de dados.

Assista ao vídeo e aprenda a criar um script em Python para gerenciar eventos, usando um banco de dados SQLite. Você verá como criar tabelas, inserir dados e consultar registros.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Você foi contratado como desenvolvedor de uma startup de tecnologia, a qual está desenvolvendo um sistema de gerenciamento de eventos. A aplicação precisa armazenar informações sobre eventos, participantes e locais. Para isso, você deve criar um script em Python que se conecte a um banco de dados SQLite e crie as tabelas necessárias.

Objetivo

Desenvolver um script em Python que:

1. Conecte-se a um banco de dados SQLite.
2. Crie três tabelas: Eventos, Participantes e Locais.
3. Selecione e exiba os dados armazenados.

Agora, siga o passo a passo a seguir.

Passo 1: Conectar-se ao banco de dados

- No arquivo gerenciamento_eventos.py, importe a biblioteca sqlite3.
- Crie uma função para conectar-se ao banco de dados.

```
python

import sqlite3

def conectar_banco(nome_banco):
    conexao = sqlite3.connect(nome_banco)
    return conexao
```

Passo 2: Criar tabelas

Defina uma função para criar as tabelas Eventos, Participantes e Locais.

```
python

def criar_tabelas(conexao):
    cursor = conexao.cursor()

    cursor.execute('''CREATE TABLE IF NOT EXISTS Locais (
                      id INTEGER PRIMARY KEY AUTOINCREMENT,
                      nome TEXT NOT NULL,
                      endereco TEXT NOT NULL)''')

    cursor.execute('''CREATE TABLE IF NOT EXISTS Eventos (
                      id INTEGER PRIMARY KEY AUTOINCREMENT,
                      nome TEXT NOT NULL,
                      data TEXT NOT NULL,
                      local_id INTEGER NOT NULL,
                      FOREIGN KEY(local_id) REFERENCES Locais(id))''')

    cursor.execute('''CREATE TABLE IF NOT EXISTS Participantes (
                      id INTEGER PRIMARY KEY AUTOINCREMENT,
                      nome TEXT NOT NULL,
                      email TEXT NOT NULL,
                      evento_id INTEGER NOT NULL,
                      FOREIGN KEY(evento_id) REFERENCES Eventos(id))''')

    conexao.commit()
```

Passo 3: Montar o script principal

No final do arquivo gerenciamento_eventos.py, adicione o código principal para executar as funções definidas.

```
python

if __name__ == '__main__':
    conexao = conectar_banco('eventos.db')
    criar_tabelas(conexao)
    conexao.close()
```

Passo 4: Executar o script

1. Abra um terminal ou prompt de comando.
2. Navegue até o diretório onde o arquivo gerenciamento_eventos.py está salvo.
3. Execute o script com o comando. Veja!

```
bash

python gerenciamento_eventos.py
```

Resultado esperado

O script deve conectar-se ao banco de dados eventos.db, criar as tabelas Locais, Eventos e Participantes, inserir os dados fornecidos e exibir os dados das tabelas no console.

Atividade 3

Questão

Na criação de um sistema de gerenciamento de eventos, a ordem de criação das tabelas no banco de dados é um fator importante. O que aconteceria se o desenvolvedor criasse a tabela de eventos antes da tabela de locais?

A

A tabela de eventos seria criada corretamente.

B

Nada, a ordem das tabelas não importa.

C

A tabela de eventos criaria uma referência nula à tabela de locais.

D

A tabela de eventos seria criada sem chaves estrangeiras.

E

A tabela de eventos não seria criada devido à ausência da tabela de locais.

A alternativa E está correta.

Para criar uma tabela com uma chave estrangeira, a tabela referenciada deve existir. Portanto, tentar criar a tabela de eventos antes da tabela de locais resultaria em um erro, pois a referência à chave estrangeira não seria encontrada.

Inserção de dados em tabela

A partir de comandos como INSERT INTO, desenvolvedores podem adicionar novos registros, atualizando e expandindo continuamente o banco de dados com informações relevantes. Assim, as aplicações podem manipular dados de maneira efetiva, suportando operações de entrada de dados de usuários, integração de sistemas e manutenção de registros atualizados.

Neste vídeo, você aprenderá a usar o comando SQL INSERT INTO para inserir dados em tabelas de bancos de dados. Vamos abordar a sintaxe correta e exemplos práticos para que suas aplicações possam adicionar novos registros, mantendo seu banco de dados atualizado e funcional. Confira!

> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

Para ilustrar a utilização do comando INSERT INTO, do SQL, 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:

```plain-text
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.

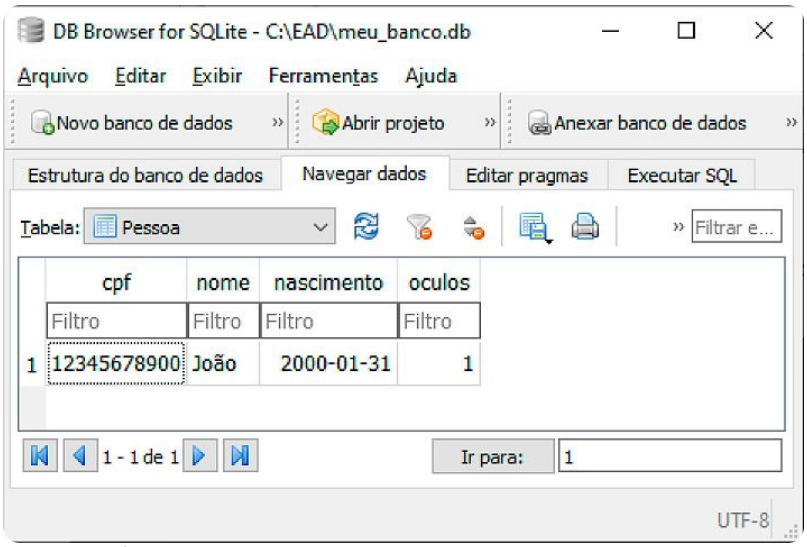
```
python

  import sqlite3 as conector

  # Abertura de conexão e aquisição de cursor
  conexao = conector.connect("./meu_banco.db")
  cursor = conexao.cursor()

  # Execução de um comando: SELECT... CREATE ...
  comando = '''INSERT INTO Pessoa (cpf, nome, nascimento, oculos)
       VALUES (12345678900, 'João', '2000-01-31', 1);'''

   cursor.execute(comando)

   # Efetivação do comando
   conexao.commit()

   # Fechamento das conexões
   cursor.close()
   conexao.close()
```

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 como ficou a tabela pelo DB Browser for SQLite.



DB Browser for SQLite.

Atividade 1

Questão

Inserir dados em uma tabela é uma tarefa comum no desenvolvimento de aplicações que interagem com bancos de dados. Qual comando SQL é utilizado para inserir um novo registro em uma tabela?

A

UPDATE INTO

B

INSERT INTO

C

ADD INTO

D

CREATE INTO

E

SELECT INTO

A alternativa B está correta.

O comando SQL INSERT INTO é utilizado para inserir novos registros em uma tabela. Ele permite especificar as colunas e os valores a serem adicionados, garantindo que os dados sejam armazenados corretamente.

Inserção de dados em tabela com queries dinâmicas

A inserção de dados com queries dinâmicas é uma técnica fundamental no desenvolvimento de aplicações flexíveis e seguras. Ela permite reutilizar comandos SQL, alterando apenas os valores a serem inseridos, o que facilita a manutenção do código e melhora a eficiência.

Neste vídeo, você aprenderá a usar delimitadores de parâmetros, como “?” no SQLite, para evitar SQL Injection e aumentar a flexibilidade do seu código, mantendo a integridade dos dados. Assista!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

É muito comum reutilizarmos uma mesma string para inserir diversos registros em uma tabela, alterando apenas os dados que serão adicionados.

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 forma geral, as APIs sqlite3, psycopg2 e PyMySQL fazem a concatenação da string e dos parâmetros antes de enviá-los ao banco de dados.

A concatenação da string 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.

```plain-text
>>> 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.

```plain-text
>>> cursor.execute(comando, (“Teste”, 123))
```

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

```plain-text
“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.

A definição dessa classe pode ser vista no script modelo.py.

```python
class Pessoa:
   def __init__(self, cpf, nome, data_nascimento, usa_oculos):
       self.cpf = cpf
       self.nome = nome
       self.data_nascimento = data_nascimento
       self.usa_oculos = usa_oculos
```

No script a seguir, temos os mesmos atributos que a entidade equivalente, mas com nomes diferentes. Fizemos isso para facilitar o entendimento dos exemplos dados aqui.

```python
import sqlite3 as conector
from modelo import Pessoa

# Abertura de conexão e aquisição de cursor
conexao = conector.connect("./meu_banco.db")
cursor = conexao.cursor()

# Criação de um objeto do tipo Pessoa
pessoa = Pessoa(10000000099, 'Maria', '1990-01-31', False)

# Definição de um comando com query parameter
comando = '''INSERT INTO Pessoa (cpf, nome, nascimento, oculos) VALUES (?, ?, ?, ?);'''
cursor.execute(comando, (pessoa.cpf, pessoa.nome, pessoa.data_nascimento,
pessoa.usa_oculos))

# Efetivação do comando
conexao.commit()

# Fechamento das conexões
cursor.close()
conexao.close()
```

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 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 (?)!

Comentário

O uso do delimitador de parâmetros, representado por uma interrogação (?), 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.

Veja a seguir o comando final enviado ao banco de dados pelo comando execute.

```
plain-text

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 19 e 20, 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.

Atividade 2

Questão

Escolher os delimitadores corretos para parâmetros em diferentes conectores é uma habilidade relevante para o desenvolvimento de aplicações. Qual delimitador é utilizado para parâmetros em queries dinâmicas com o conector sqlite3 em Python?

A

%s

B

%n

C

?

D

$

E

&

A alternativa C está correta.

O delimitador ? é utilizado no conector sqlite3 para indicar parâmetros em queries dinâmicas. Ele permite a inserção segura de valores, evitando vulnerabilidades como SQL Injection.

Inserção de dados em tabela com queries dinâmicas e nomes

Uma etapa essencial no desenvolvimento de aplicações eficazes é aprender a usar queries dinâmicas com nomes. Utilizar argumentos nomeados torna o código mais flexível e claro, facilita a manutenção e diminui a chance de erros. Além disso, essa prática melhora a segurança contra SQL Injection, protegendo a integridade dos dados.

Neste vídeo, demonstraremos como utilizar delimitadores e dicionários em Python para criar comandos SQL flexíveis e seguros, otimizando a inserção de registros e prevenindo erros, como o SQL Injection. Confira!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

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.

```
plain-text
>>> 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 agora como poderia ficar a chamada do método execute para esse comando.

```
plain-text
>>> cursor.execute(comando, {“atrib1”:“Teste”, “atrib2”: 123})
```

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

```plain-text
INSERT INTO tabela1 (atributo1, atributo2) VALUES (‘Teste’, 123);
```

Observe o exemplo no script seguinte. Vamos criar um similar ao anterior, utilizando novamente a classe Pessoa. Porém, agora, o comando para inserir um registro no banco de dados utiliza os argumentos nomeados.

```python
import sqlite3 as conector
from modelo import Pessoa

# Abertura de conexão e aquisição de cursor
conexao = conector.connect("./meu_banco.db")
cursor = conexao.cursor()

# Criação de um objeto do tipo Pessoa
pessoa = Pessoa(20000000099, 'José', '1990-02-28', False)

# Definição de um comando com query parameter
comando = '''INSERT INTO Pessoa (cpf, nome, nascimento, oculos)
                 VALUES (:cpf,:nome,:data_nascimento,:usa_oculos);'''
cursor.execute(comando, {"cpf": pessoa.cpf,
                         "nome": pessoa.nome,
                         "data_nascimento": pessoa.data_nascimento,
                         "usa_oculos": pessoa.usa_oculos})

# Efetivação do comando
conexao.commit()

# Fechamento das conexões
cursor.close()
conexao.close()
```

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 9. Cada argumento nomeado do comando será substituído pelo valor da chave correspondente do dicionário.

Aqui está o comando final enviado ao banco de dados pelo comando execute.

```plain-text
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.

Agora 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, manteremos 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 o script.

```
python

 import sqlite3 as conector
 from modelo import Pessoa

 # Abertura de conexão e aquisição de cursor
 conexao = conector.connect("./meu_banco.db")
 cursor = conexao.cursor()

 # Criação de um objeto do tipo Pessoa
 pessoa = Pessoa(30000000099, 'Silva', '1990-03-30', True)

 # Definição de um comando com named parameter
 comando = '''INSERT INTO Pessoa VALUES (:cpf,:nome,:data_nascimento,:usa_oculos);'''
 cursor.execute(comando, vars(pessoa))
 print(vars(pessoa))

 # Efetivação do comando
 conexao.commit()

 # Fechamento das conexões
 cursor.close()
 conexao.close()
```

Veja o resultado/saída do script anterior.

```
D: \Banco de dados> &; C:/ python.exe "d/script12.py"
{'cpf': 30000000099, 'nome': 'Silva', 'data_nascimento': '1990-03-30', 'usa_oculos':
True'':}
```

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, no qual 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 a saída do console, onde imprimimos o resultado de vars(pessoa), linha 14.

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

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

```plain-text
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 para 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 Veículo. Confira essas classes no script a seguir.

```python
class Pessoa:
   def __init__(self, cpf, nome, data_nascimento, usa_oculos):
       self.cpf = cpf
       self.nome = nome
       self.data_nascimento = data_nascimento
       self.usa_oculos = usa_oculos
       self.veiculos = []

class Marca:
   def __init__(self, id, nome, sigla):
       self.id = id
       self.nome = nome
       self.sigla = sigla


class Veiculo:
   def __init__(self, placa, ano, cor, motor, proprietario, marca):
       self.placa = placa
       self.ano = ano
       self.cor = cor
       self.motor = motor
       self.proprietario = proprietario
       self.marca = marca
```

Para inserir os registros, vamos criar este script.

```python
import sqlite3 as conector
from modelo import Marca, Veiculo
conexao = conector.connect("./meu_banco.db")
conexao.execute("PRAGMA foreign_keys = on")
cursor = conexao.cursor()

# Inserção de dados na tabela Marca
comando1 = '''INSERT INTO Marca (nome, sigla) VALUES (:nome, :sigla);'''

marca1 = Marca(1, "Marca A", "MA")
cursor.execute(comando1, vars(marca1))
marca1.id = cursor.lastrowid

marca2 = Marca(2, "Marca B", "MB")
cursor.execute(comando1, vars(marca2))
marca2.id = cursor.lastrowid

# Inserção de dados na tabela Veiculo
comando2 = '''INSERT INTO Veiculo
              VALUES (:placa, :ano, :cor, :motor, :proprietario, :marca);'''
veiculo1 = Veiculo("AAA0001", 2001, "Prata", 1.0, 10000000099, marca1.id)
veiculo2 = Veiculo("BAA0002", 2002, "Preto", 1.4, 10000000099, marca1.id)
veiculo3 = Veiculo("CAA0003", 2003, "Branco", 2.0, 20000000099, marca2.id)
veiculo4 = Veiculo("DAA0004", 2004, "Azul", 2.2, 30000000099, marca2.id)
cursor.execute(comando2, vars(veiculo1))
cursor.execute(comando2, vars(veiculo2))
cursor.execute(comando2, vars(veiculo3))
cursor.execute(comando2, vars(veiculo4))

# Efetivação do comando
conexao.commit()
# Fechamento das conexões
cursor.close()
conexao.close()
```

No script anterior, 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 exclusiva do SQLite, usada para ajustar certos comportamentos internos do banco de dados. Por padrão, o SQLite não aplica a verificação de restrições de chave estrangeira. Isso acontece por razões históricas, já que versões anteriores do SQLite não suportavam chaves estrangeiras.

No comando da linha 4, habilitamos a flag foreign_keys, para 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 Veículo no banco.

A entidade Marca é um requisito para criação de registros na tabela Veículo, visto que a tabela Veículo 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.

Comentário

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 Veículo, 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 Veículo. 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.

Atenção

Lembre-se de que como os atributos proprietário 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:

```
plain-text

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. Veja!

Tabela: Marca

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

Tabela: Pessoa

	cpf	nome	nascimento	oculos
	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: Veiculo

	placa	ano	cor	motor	proprietario	marca
	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

DB Browser for SQLite.

Atividade 3

Questão

O uso de queries dinâmicas com nomes é uma prática avançada que melhora a legibilidade e a segurança do código em aplicações que interagem com bancos de dados. Qual é o delimitador correto para usar argumentos nomeados em queries dinâmicas com o conector sqlite3?

A

%s

B

¶m

C

?param

D

:param

E

#param

A alternativa D está correta.

No conector sqlite3, o delimitador :param é utilizado para argumentos nomeados em queries dinâmicas. Isso facilita a substituição de parâmetros de forma clara e segura, evitando erros e melhorando a legibilidade do código.

Atualização e remoção de dados em uma tabela

Para manter a integridade e a relevância das informações em um banco de dados, precisamos entender as operações de atualização e remoção de dados em uma tabela. Utilizando comandos SQL como UPDATE e DELETE, é possível modificar registros existentes e remover dados desnecessários ou obsoletos, garantindo que o banco de dados reflita com precisão o estado atual da aplicação. Dominar essas habilidades é fundamental para a administração eficiente de dados e para a implementação de funcionalidades dinâmicas em sistemas de informação.

Neste vídeo, vamos aprender a atualizar e remover dados em tabelas usando os comandos SQL UPDATE e DELETE. Veremos a sintaxe correta, boas práticas e exemplos para garantir que suas operações sejam seguras e mantenham a integridade do banco de dados. Confira!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Ao trabalharmos com banco de dados, duas operações importantes e bastante usadas são a atualização de dados e a remoção de dados.

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. Aqui está sua sintaxe:

```
plain-text

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, vamos mostrar como atualizar registros de uma tabela utilizando os três métodos.

```
python

 import sqlite3 as conector

 # Abertura de conexão e aquisição de cursor
 conexao = conector.connect("./meu_banco.db")
 conexao.execute("PRAGMA foreign_keys = on")
 cursor = conexao.cursor()

 # Definição dos comandos
 comando1 = '''UPDATE Pessoa SET oculos= 1;'''
 cursor.execute(comando1)

 comando2 = '''UPDATE Pessoa SET oculos= ? WHERE cpf=30000000099;'''
 cursor.execute(comando2, (False,))

 comando3 = '''UPDATE Pessoa SET oculos= :usa_oculos WHERE cpf= :cpf;'''
 cursor.execute(comando3, {"usa_oculos": False, "cpf": 20000000099})

 # Efetivação do comando
 conexao.commit()

 # Fechamento das conexões
 cursor.close()
 conexao.close()
```

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, em que 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 óculos para 1 (verdadeiro) para TODOS os registros da tabela. Isso ocorreu porque 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 óculos e explicitamos o valor do CPF na cláusula WHERE.

Observe o comando executado pela linha 13.

```
plain-text

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

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

Na linha 15, criamos mais um comando de UPDATE, dessa vez utilizando o argumento nomeado tanto para o atributo óculos quanto para o CPF da cláusula WHERE.

Veja a seguir o comando final enviado pela linha 16 ao banco de dados.

```
plain-text

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

Vamos alterar o valor do atributo óculos 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 Veículo, seria lançada uma exceção do tipo IntegrityError, devido à restrição da chave estrangeira.

Atenção

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

Vamos agora aprender a remover registros de uma tabela.

Para remover um registro, utilizamos o comando SQL DELETE. Confira a sintaxe!

```plain-text
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 no script a seguir.

```python
import sqlite3 as conector

# Abertura de conexão e aquisição de cursor
conexao = conector.connect("./meu_banco.db")
conexao.execute("PRAGMA foreign_keys = on")
cursor = conexao.cursor()

# Definição dos comandos
comando = '''DELETE FROM Pessoa WHERE cpf= 12345678900;'''
cursor.execute(comando)

# Efetivação do comando
conexao.commit()

# Fechamento das conexões
cursor.close()
conexao.close()
```

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.

Aqui está o comando final enviado ao banco de dados pelo comando execute:

```
plain-text

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 Veículo por meio de seu CPF, seria gerada uma exceção do tipo IntegrityError caso tentássemos removê-las.

Atividade 4

Questão

A atualização e a remoção de dados são operações comuns em bancos de dados, com o objetivo de manter a precisão e a integridade das informações. Qual comando SQL é utilizado para atualizar registros em uma tabela?

A

DELETE

B

INSERT

C

UPDATE

D

SELECT

E

ALTER

A alternativa C está correta.

O comando SQL UPDATE é utilizado para modificar os valores dos registros existentes em uma tabela, especificando as colunas a serem alteradas e as novas informações.

Populando as tabelas do banco de dados na prática

Vamos aprender a preencher tabelas em bancos de dados usando Python, o que permite inserir dados de forma segura, mantendo a integridade e consistência das informações. Dominar essa técnica é essencial para automatizar a carga de dados, facilitar a migração entre sistemas e realizar testes eficazes durante o desenvolvimento de software.

Vamos mostrar agora como conectar-se ao banco, criar tabelas para um sistema de e-commerce e inserir dados iniciais utilizando queries dinâmicas. Em seguida, faremos uma alteração nas queries para utilizar parâmetros nomeados. Confira no vídeo!

> Conteúdo interativo
>
> Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Você foi contratado por uma empresa de e-commerce para desenvolver um sistema de gerenciamento de produtos e pedidos capaz de armazenar informações sobre produtos, clientes e pedidos. Sua tarefa é criar um script em Python que se conecte a um banco de dados SQLite, criar as tabelas necessárias e inserir dados iniciais.

Objetivo

Desenvolver um script em Python que:

1. Conecte-se a um banco de dados SQLite.
2. Crie três tabelas: Produtos, Clientes e Pedidos.
3. Insira dados iniciais nessas tabelas.
4. Selecione e exiba os dados armazenados.

Passo 1: Conectar-se ao banco de dados

1. No arquivo gerenciamento_ecommerce.py, importe a biblioteca sqlite3.
2. Crie uma função para conectar-se ao banco de dados.

```
python

import sqlite3

def conectar_banco(nome_banco):
    conexao = sqlite3.connect(nome_banco)
    return conexao
```

Passo 2: Criar tabelas

Defina uma função para criar as tabelas Produtos, Clientes e Pedidos.

```python
def criar_tabelas(conexao):
    cursor = conexao.cursor()

    cursor.execute('''CREATE TABLE IF NOT EXISTS Produtos (
                      id INTEGER PRIMARY KEY AUTOINCREMENT,
                      nome TEXT NOT NULL,
                      preco REAL NOT NULL,
                      estoque INTEGER NOT NULL)''')

    cursor.execute('''CREATE TABLE IF NOT EXISTS Clientes (
                      id INTEGER PRIMARY KEY AUTOINCREMENT,
                      nome TEXT NOT NULL,
                      email TEXT NOT NULL)''')

    cursor.execute('''CREATE TABLE IF NOT EXISTS Pedidos (
                      id INTEGER PRIMARY KEY AUTOINCREMENT,
                      cliente_id INTEGER NOT NULL,
                      produto_id INTEGER NOT NULL,
                      quantidade INTEGER NOT NULL,
                      data_pedido TEXT NOT NULL,
                      FOREIGN KEY (cliente_id) REFERENCES Clientes(id),
                      FOREIGN KEY (produto_id) REFERENCES Produtos(id))''')

    conexao.commit()
```

Passo 3: Inserir dados iniciais

Defina uma função para inserir dados nas tabelas.

```python
def inserir_dados(conexao):
    cursor = conexao.cursor()

    produtos = [('Notebook', 2999.99, 10),
                ('Smartphone', 1999.99, 20),
                ('Tablet', 999.99, 30)]

    clientes = [('Alice', 'alice@example.com'),
                ('Bob', 'bob@example.com'),
                ('Charlie', 'charlie@example.com')]

    pedidos = [(1, 1, 2, '2023-06-15'),
               (2, 2, 1, '2023-06-16'),
               (3, 3, 3, '2023-06-17')]

    cursor.executemany('INSERT INTO Produtos (nome, preco, estoque) VALUES (?, ?, ?)',
produtos)
    cursor.executemany('INSERT INTO Clientes (nome, email) VALUES (?, ?)', clientes)
    cursor.executemany('INSERT INTO Pedidos (cliente_id, produto_id, quantidade,
data_pedido) VALUES (?, ?, ?, ?)', pedidos)

    conexao.commit()
```

Passo 4: Montar o script principal

No final do arquivo gerenciamento_ecommerce.py, adicione o código principal para executar as funções definidas.

```
python

if __name__ == '__main__':
    conexao = conectar_banco('ecommerce.db')
    criar_tabelas(conexao)
    inserir_dados(conexao)
    conexao.close()
```

Passo 5: Executar o script

- Abra um terminal ou prompt de comando.
- Navegue até o diretório onde o arquivo gerenciamento_ecommerce.py está salvo.
- Execute o script com o comando:

```
bash

python gerenciamento_ecommerce.py
```

Resultado esperado

O script deve conectar-se ao banco de dados ecommerce.db, criar as tabelas Produtos, Clientes e Pedidos, inserir os dados fornecidos e exibir os dados dessas tabelas no console.

Atividade 5

Questão

Durante a inserção de dados nas tabelas de um banco de dados SQLite, é importante usar corretamente os delimitadores de parâmetros. Sendo possível, inclusive, utilizar parâmetros nomeados. O que aconteceria se o desenvolvedor substituísse os "?" pelos parâmetros :nome, :preco, :estoque na inserção de Produtos mantendo o restante do código da mesma forma?

A

A inserção falharia devido a um erro de sintaxe.

B

A inserção funcionaria perfeitamente sem nenhuma alteração adicional.

C

O Python geraria uma exceção de tipo.

D

O código precisaria ser adaptado para passar um dicionário com os valores.

E

A ordem dos valores se tornaria irrelevante.

A alternativa D está correta.

Ao usar parâmetros nomeados como :nome, :preco, :estoque, é necessário passar um dicionário com os valores correspondentes ao método execute para que a inserção funcione corretamente.

Seleção de registros de uma tabela

Para recuperar informações específicas e relevantes para uma aplicação, é fundamental saber selecionar registros em um banco de dados. Usando o comando SQL SELECT, desenvolvedores podem realizar consultas e filtrar dados com base em condições específicas, possibilitando a criação de relatórios e o fornecimento de dados necessários para uma aplicação dinâmica e responsiva.

Neste vídeo, você aprenderá a usar o comando SELECT, incluindo a cláusula WHERE para filtrar resultados, e a recuperar informações de forma eficiente. Assista!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Aprenderemos agora 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. Vamos lá!

Para selecionar e recuperar registros de um banco de dados, utilizamos o comando SQL SELECT. Aqui está sua sintaxe:

```
plain-text

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, vamos mostrar como recuperar todos os registros da tabela Pessoa.

```python
import sqlite3 as conector

# Abertura de conexão e aquisição de cursor
conexao = conector.connect("./meu_banco.db")
cursor = conexao.cursor()

# Definição dos registros
comando = '''SELECT nome, oculos FROM Pessoa;'''
cursor.execute(comando)

# Recuperação dos dados
registros = cursor.fetchall()
print("Tipo retornado pelo fetchall():", type(registros))

for registro in registros:
    print("Tipo:", type(registro), "- Conteúdo:", registro)

# Fechamento das conexões
cursor.close()
conexao.close()
```

Veja o resultado/saída do script anterior.

```
D:\Banco de dados> &; C:python.exe "d:Banco de dados/script16.py"


Tipo retornado pelo fetchall():
Tipo: - Conteúdo: ('Maria', 0)
Tipo: - Conteúdo: ('João', 1)
Tipo: - Conteúdo: ('Silva', 1)
```

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

```plain-text
SELECT nome, oculos FROM Pessoa;
```

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

Acompanhe agora o passo a passo da recupeção de todos os registros da tabela Pessoa.

- 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.

- Ao final, fechamos o cursor e a conexão.

Observe que cada registro é uma tupla, composta pelos atributos nome e óculos da entidade Pessoa. Os registros são sempre retornados em forma de tupla, mesmo que contenham apenas um atributo.

Atenção

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

Vamos criar agora uma consulta para retornar às pessoas que usam óculos. Observe como ficou o exemplo.

```
python

 import sqlite3 as conector
 from modelo import Pessoa
 # Abertura de conexão e aquisição de cursor
 conexao = conector.connect("./meu_banco.db")
 cursor = conexao.cursor()

 # Definição dos comandos
 comando = '''SELECT * FROM Pessoa WHERE oculos=:usa_oculos;'''
 cursor.execute(comando, {"usa_oculos": True})

 # Recuperação dos registros
 registros = cursor.fetchall()
 for registro in registros:
      pessoa = Pessoa(*registro)
      print("cpf:", type(pessoa.cpf), pessoa.cpf)
      print("nome:", type(pessoa.nome), pessoa.nome)
      print("nascimento:", type(pessoa.data_nascimento), pessoa.data_nascimento)
      print("oculos:", type(pessoa.usa_oculos), pessoa.usa_oculos)

 # Fechamento das conexões
 cursor.close()
 conexao.close()
```

Observe o resultado/saída do script anterior.

```
D:\Banco de dados> &; C:python.exe "d:/Banco de dados/script17.py"
cpf: 12345678900
nome: João
nascimento: 2000-01-31
oculos: 1
cpf: 30000000099
nome: Silva
```

```
nascimento: 1990-03-30
oculos: 1
```

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:

```
plain-text

SELECT * FROM Pessoa WHERE oculos=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 Óculos, nessa ordem, criamos um objeto do tipo Pessoa na linha 14.

> Foi utilizado 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.

Veja que os dados dos atributos Nascimento e Óculos estão exatamente como no banco de dados: nascimento é uma string e óculos 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 permite realizar essa conversão automaticamente, mas exige algumas configurações adicionais.

Vamos mostrar agora como fazer a conversão de datas e booleanos.

```
python
```

```
 import sqlite3 as conector
 from modelo import Pessoa
 # Abertura de conexão e aquisição de cursor
 conexao = conector.connect("./meu_banco.db", detect_types=conector.PARSE_DECLTYPES)
 cursor = conexao.cursor()

 # Funções conversoras
 def conv_bool(dado):
    return True if dado == 1 else False

 # Registro de conversores
 conector.register_converter("BOOLEAN", conv_bool)

 # Definição dos comandos
 comando = '''SELECT * FROM Pessoa WHERE oculos=:usa_oculos;'''
 cursor.execute(comando, {"usa_oculos": True})

# Recuperação dos registros
registros = cursor.fetchall()
for registro in registros:
    pessoa = Pessoa(*registro)
    print("cpf:", type(pessoa.cpf), pessoa.cpf)
    print("nome:", type(pessoa.nome), pessoa.nome)
    print("nascimento:", type(pessoa.data_nascimento), pessoa.data_nascimento)
    print("oculos:", type(pessoa.usa_oculos), pessoa.usa_oculos)

# Fechamento das conexões
cursor.close()
conexao.close()
```

Confira o resultado/saída do script anterior.

```
D:\Banco de dados> &; C:/python.exe "d:/Banco de dados/script18.py"
cpf: 12345678900
nome: João
nascimento: 2000-01-31
oculos: False
cpf: 30000000099
nome: Silva
nascimento: 1990-03-30
oculos: False
```

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 Óculos. Agora são das classes date e bool!

Atividade 1

Questão

Selecionar registros em um banco de dados permite recuperar informações necessárias. Qual cláusula é utilizada em conjunto com o comando SELECT para filtrar registros específicos em uma tabela?

A

GROUP BY

B

ORDER BY

C

WHERE

D

HAVING

E

JOIN

A alternativa C está correta.

A cláusula WHERE é utilizada em conjunto com o comando SELECT para filtrar registros específicos em uma tabela com base em condições definidas, permitindo a recuperação de dados relevantes.

Seleção de registros utilizando junção e de registros relacionados

É importante entender como usar junções para selecionar registros de diferentes tabelas, criando consultas mais complexas e úteis. Com o JOIN, desenvolvedores podem combinar dados de várias tabelas, oferecendo uma visão mais completa. Isso permite, por exemplo, listar pessoas junto com seus veículos e marcas. A seleção de registros relacionados é essencial para analisar dados interconectados, gerar relatórios detalhados, alimentar dashboards e implementar funcionalidades avançadas em aplicações.

Neste vídeo, vamos explorar como utilizar junções em SQL para selecionar registros relacionados em bancos de dados. Vamos ver como o comando JOIN permite combinar dados de várias tabelas, facilitando a criação de consultas avançadas que revelam as conexões entre diferentes entidades e oferecem uma visão detalhada das informações armazenadas. Confira!

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Seleção de registros utilizando junção

Agora, vamos aprender a buscar registros em tabelas relacionadas. No exemplo a seguir, vamos buscar os veículos e suas respectivas marcas, utilizando junção de tabelas.

```
python

 import sqlite3 as conector
 from modelo import Veiculo

 # Abertura de conexão e aquisição de cursor
 conexao = conector.connect("./meu_banco.db")
 cursor = conexao.cursor()

 # Definição dos comandos
 comando = '''SELECT * FROM Veiculo;'''
 cursor.execute(comando)

 # Recuperação dos registros
 reg_veiculos = cursor.fetchall()
 for reg_veiculo in reg_veiculos:
    veiculo = Veiculo(*reg_veiculo)
    print("Placa:", veiculo.placa, ", Marca:", veiculo.marca)

 # Fechamento das conexões
 cursor.close()
 conexao.close()
```

Veja o resultado/saída do script anterior.

```
D:Banco de dados> &; C:python.exe "d:/Banco de dados/script19_1.py"
Placa: AAA0001 , Marca: 1
Placa: BAA0002 , Marca: 1
```

```
Placa: CAA0003 , Marca: 2
Placa: DAA0004 , Marca: 2
```

Após abrir a conexão e obter um cursor, selecionamos todos os registros da tabela Veículo, 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 Veículo 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 porque 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 Veículo e Marca no comando SELECT.

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

```plain-text
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, vamos criar um script de forma que o Veículo tenha acesso ao nome da Marca, não ao id.

```python
 import sqlite3 as conector
 from modelo import Veiculo

 # Abertura de conexão e aquisição de cursor
 conexao = conector.connect("./meu_banco.db")
 cursor = conexao.cursor()

 # Definição dos comandos
 comando = '''
 SELECT Veiculo.placa, Veiculo.ano, Veiculo.cor, Veiculo.motor, Veiculo.proprietario,
Marca.nome
 FROM Veiculo
 JOIN Marca ON Marca.id = Veiculo.marca;'''
 cursor.execute(comando)

 # Recuperação dos registros
 reg_veiculos = cursor.fetchall()
 for reg_veiculo in reg_veiculos:
    veiculo = Veiculo(*reg_veiculo)
    print("Placa:", veiculo.placa, ", Marca:", veiculo.marca)

 # Fechamento das conexões
 cursor.close()
 conexao.close()
```

Confira a seguir o resultado/saída do script anterior.

```
D:\Banco de dados> &; C:/python.exe "d:/Banco de dados/script19_2.py" Placa: AAA0001 , Marca: Marca A
Placa: BAA0002 , Marca: Marca A
Placa: CAA0003 , Marca: Marca B
Placa: DAA0004 , Marca: Marca B
```

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.

```
plain-text

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 veículo 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 Veículo 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 Veículo contém o nome da Marca. No próximo exemplo, vamos dar um passo além. Vamos atribuir ao atributo Marca, do veículo, um objeto do tipo Marca. Dessa forma, vamos ter acesso a todos os atributos da marca de um veículo.

Para isso, vamos fazer alguns ajustes no nosso modelo. Observe os códigos a seguir.

```python
class Pessoa:
   def __init__(self, cpf, nome, data_nascimento, usa_oculos):
       self.cpf = cpf
       self.nome = nome
       self.data_nascimento = data_nascimento
       self.usa_oculos = usa_oculos
       self.veiculos = []

class Marca:
   def __init__(self, id, nome, sigla):
       self.id = id
       self.nome = nome
       self.sigla = sigla

class Veiculo:
   def __init__(self, placa, ano, cor, motor, proprietario, marca):
       self.placa = placa
       self.ano = ano
       self.cor = cor
       self.motor = motor
       self.proprietario = proprietario
       self.marca = marca
```

```python
import sqlite3 as conector
from modelo import Veiculo, Marca

# Abertura de conexão e aquisição de cursor
conexao = conector.connect("./meu_banco.db")
cursor = conexao.cursor()

# Definição dos comandos
comando = '''SELECT * FROM
               Veiculo JOIN Marca ON Marca.id = Veiculo.marca;'''
cursor.execute(comando)

# Recuperação dos registros
registros = cursor.fetchall()
for registro in registros:
   print(registro)
   marca = Marca(*registro[6:])
   veiculo = Veiculo(*registro[:5], marca)
   print("Placa:", veiculo.placa, ", Marca:", veiculo.marca.nome)

# Fechamento das conexões
cursor.close()
conexao.close()
```

Veja o resultado/saída do script anterior.

```
D:\Banco de dados> &; C:python.exe "d:/Banco de dados/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

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, Script modelo.py , 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')

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'), que inclui os atributos Id, Nome e Sigla. Essa tupla é utilizada em conjunto com o operador * para criar 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:

```plain-text
marca = Marca(1, 'Marca A', 'MA')
```

Na linha 18, utilizamos array slice para selecionar apenas os atributos do Veículo, que retornou a tupla ('AAA0001', 2001, 'Prata', 1.0, 10000000099), que incluiu 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:

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

Na linha 19, imprimimos a placa do veículo e o nome da marca. Note que o atributo marca, do objeto Veiculo, faz referência ao objeto Marca, permitindo acessar veiculo.marca.nome. Ao final do script, encerramos 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, permitindo sua reutilização no nosso script final. Confira os scripts a seguir.

python

```
from modelo import Veiculo, Marca


def recuperar_veiculos(conexao, cpf):
   # Aquisição de cursor
   cursor = conexao.cursor()

   # Definição dos comandos
   comando = '''SELECT * FROM Veiculo
               JOIN Marca ON Marca.id = Veiculo.marca
               WHERE Veiculo.proprietario = ?;'''
   cursor.execute(comando, (cpf,))

   # Recuperação dos registros
   veiculos = []
   registros = cursor.fetchall()
   for registro in registros:
       marca = Marca(*registro[6:])
       veiculo = Veiculo(*registro[:5], marca)
       veiculos.append(veiculo)

   # Fechamento do cursor
   cursor.close()
   return veículos
```

python

```
import sqlite3 as conector
from modelo import Pessoa
from script19_4 import recuperar_veiculos

# Abertura de conexão e aquisição de cursor
conexao = conector.connect("./meu_banco.db")
cursor = conexao.cursor()

# Definição dos comandos
comando = '''SELECT * FROM Pessoa;'''
cursor.execute(comando)

# Recuperação dos registros
pessoas = []
reg_pessoas = cursor.fetchall()
for reg_pessoa in reg_pessoas:
   pessoa = Pessoa(*reg_pessoa)
   pessoa.veiculos = recuperar_veiculos(conexao, pessoa.cpf)
   pessoas.append(pessoa)

for pessoa in pessoas:
   print(pessoa.nome)
   for veiculo in pessoa.veiculos:
       print('\t', veiculo.placa, veiculo.marca.nome)
# Fechamento das conexões
cursor.close()
conexao.close()
```

Aqui está o resultado/saída do script anterior:

```
```

```
D:Banco de dados> &; C:python.exe "d:Banco de dados/script20.py"
Maria
AAA0001 Marca A
BAA0002 Marca A

José
CAA0003 Marca B

Silva
DAA0004 Marca B
```

No script 19_4 utilizamos como base o script do exemplo anterior para criar uma função que retorne uma lista dos veículos de determinada pessoa.

Essa função tem como parâmetro 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 veículos. 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 veículos.

> O script20 é o nosso script principal, destinado a gerar uma lista das pessoas cadastradas no banco de dados, incluindo seus veículos e marcas.

Após estabelecer a conexão e o cursor, criamos o comando SQL na linha 10 para recuperar as pessoas, e o executamos na linha 11. Na linha 14, criamos a variável do tipo lista chamada Pessoas, e na linha 15, utilizamos a função fetchall do cursor para recuperar todos os registros das pessoas.

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 Pessoas criada anteriormente. Na linha 21, iteramos sobre a lista Pessoas e imprimimos seu nome e a lista de veículos que possui. Ao final, fechamos a conexão e o cursor.

Recomendação

Verifique a saída do console que está logo após a imagem.

Atividade 2

Questão

A seleção de registros usando junções é uma técnica avançada que permite combinar dados de várias tabelas. Qual comando SQL é utilizado para mesclar registros de duas ou mais tabelas?

A

UNION

B

JOIN

C

INTERSECT

D

CONNECT

E

MERGE

A alternativa B está correta.

O comando SQL JOIN é utilizado para combinar registros de duas ou mais tabelas com base em uma condição de relação, sendo essencial para recuperar dados de diferentes tabelas, possibilitando consultas complexas e detalhadas.

Demonstrando como selecionar dados no banco de dados

Dominar a seleção de dados relacionados e o uso de JOIN é fundamental para quem trabalha com bancos de dados, especialmente em aplicações complexas. O comando JOIN permite combinar dados de várias tabelas, oferecendo uma visão completa das informações. Isso facilita a geração de relatórios detalhados, análises abrangentes e o desenvolvimento de funcionalidades que dependem de dados interconectados. Além disso, o uso eficiente de JOINs melhora o desempenho das consultas, tornando o sistema mais eficiente.

Aprenda a usar JOIN para selecionar dados relacionados em bancos de dados assistindo ao vídeo a seguir. Revisaremos como criar tabelas e inserir dados iniciais. Em seguida, faremos consultas avançadas para obter informações mais completas, essenciais para relatórios e análises.

Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Você foi contratado por uma livraria on-line para desenvolver um sistema de gerenciamento de pedidos capaz de armazenar informações sobre livros, clientes e pedidos.

Sua tarefa é desenvolver um script em Python que se conecte a um banco de dados SQLite, crie as tabelas necessárias, insira dados iniciais utilizando parâmetros nomeados e demonstre a seleção de dados relacionados usando JOIN, para obter informações completas sobre os pedidos, incluindo detalhes dos livros e clientes.

Objetivo

Desenvolver um script em Python que:

- Conecte-se a um banco de dados SQLite.
- Crie três tabelas: Livros, Clientes e Pedidos.
- Insira dados iniciais nessas tabelas usando parâmetros nomeados.
- Utilize JOIN para selecionar e exibir dados relacionados sobre os pedidos.

Passo 1: Definir as classes

No arquivo gerenciamento_livraria.py, defina as classes Livro, Cliente e Pedido.

```
python

class Livro:
    def __init__(self, titulo, autor, preco):
        self.titulo = titulo
        self.autor = autor
        self.preco = preco

class Cliente:
    def __init__(self, nome, email):
        self.nome = nome
        self.email = email

class Pedido:
    def __init__(self, cliente_id, livro_id, quantidade, data_pedido):
        self.cliente_id = cliente_id
        self.livro_id = livro_id
        self.quantidade = quantidade
        self.data_pedido = data_pedido
```

Passo 2: Conectar-se ao banco de dados

Importe a biblioteca sqlite3 e crie uma função para conectar-se ao banco de dados.

```
python

import sqlite3

def conectar_banco(nome_banco):
    conexao = sqlite3.connect(nome_banco)
    return conexao
```

Passo 3: Criar tabelas

Defina uma função para criar as tabelas Livros, Clientes e Pedidos.

```python
def criar_tabelas(conexao):
    cursor = conexao.cursor()

    cursor.execute('''CREATE TABLE IF NOT EXISTS Livros (
                      id INTEGER PRIMARY KEY AUTOINCREMENT,
                      titulo TEXT NOT NULL,
                      autor TEXT NOT NULL,
                      preco REAL NOT NULL)''')

    cursor.execute('''CREATE TABLE IF NOT EXISTS Clientes (
                      id INTEGER PRIMARY KEY AUTOINCREMENT,
                      nome TEXT NOT NULL,
                      email TEXT NOT NULL)''')

    cursor.execute('''CREATE TABLE IF NOT EXISTS Pedidos (
                      id INTEGER PRIMARY KEY AUTOINCREMENT,
                      cliente_id INTEGER NOT NULL,
                      livro_id INTEGER NOT NULL,
                      quantidade INTEGER NOT NULL,
                      data_pedido TEXT NOT NULL,
                      FOREIGN KEY (cliente_id) REFERENCES Clientes(id),
                      FOREIGN KEY (livro_id) REFERENCES Livros(id))''')

    conexao.commit()
```

Passo 4: Inserir dados iniciais com parâmetros nomeados

Defina uma função para inserir dados nas tabelas usando parâmetros nomeados.

```
python
```

```
def inserir_dados(conexao):
    cursor = conexao.cursor()

    livros = [Livro('Python para Iniciantes', 'John Doe', 39.99),
              Livro('Algoritmos e Estruturas de Dados', 'Jane Smith', 49.99),
              Livro('Inteligência Artificial', 'Alan Turing', 59.99)]

    clientes = [Cliente('Alice', 'alice@example.com'),
                Cliente('Bob', 'bob@example.com'),
                Cliente('Charlie', 'charlie@example.com')]

    pedidos = [Pedido(1, 1, 2, '2023-06-15'),
               Pedido(2, 2, 1, '2023-06-16'),
               Pedido(3, 3, 3, '2023-06-17')]

    for livro in livros:
        cursor.execute('INSERT INTO Livros (titulo, autor, preco) VALUES
(:titulo, :autor, :preco)', vars(livro))

    for cliente in clientes:
        cursor.execute('INSERT INTO Clientes (nome, email) VALUES (:nome, :email)',
vars(cliente))

    for pedido in pedidos:
        cursor.execute('INSERT INTO Pedidos (cliente_id, livro_id, quantidade,
data_pedido) VALUES (:cliente_id, :livro_id, :quantidade, :data_pedido)',
vars(pedido))
    conexao.commit()
```

Passo 5: Selecionar e exibir dados relacionados utilizando JOIN

Defina uma função para selecionar e exibir os dados relacionados das tabelas utilizando JOIN.

```
python
```

```
def exibir_pedidos(conexao):
    cursor = conexao.cursor()
    query = '''
    SELECT Pedidos.id, Clientes.nome, Livros.titulo, Pedidos.quantidade,
Pedidos.data_pedido
    FROM Pedidos
    JOIN Clientes ON Pedidos.cliente_id = Clientes.id
    JOIN Livros ON Pedidos.livro_id = Livros.id
    '''
    cursor.execute(query)
    pedidos = cursor.fetchall()
    print('Pedidos:')

    for pedido in pedidos:
        print(pedido)
```

Passo 6: Montar o script principal

No final do arquivo gerenciamento_livraria.py, adicione o código principal para executar as funções definidas.

```
python

if __name__ == '__main__':
    conexao = conectar_banco('livraria.db')
    criar_tabelas(conexao)
    inserir_dados(conexao)
    exibir_pedidos(conexao)
    conexao.close()
```

Passo 7: Executar o script

1. Abra um terminal ou prompt de comando.
2. Navegue até o diretório onde o arquivo gerenciamento_livraria.py está salvo.
3. Execute o script com o comando:

```
bash

python gerenciamento_livraria.py
```

Resultado esperado

O script deve conectar-se ao banco de dados livraria.db, criar as tabelas Livros, Clientes e Pedidos, inserir os dados fornecidos usando parâmetros nomeados e exibir os dados dos pedidos no console, mostrando informações detalhadas sobre os clientes e os livros relacionados a cada pedido.

Atividade 3

Questão

Geralmente, o JOIN é acompanhado pela palavra ON e pela definição de uma igualdade que representa a relação. No exemplo estudado, o que aconteceria se o desenvolvedor invertesse a ordem para JOIN Clientes ON Clientes.id = Pedidos.cliente_id na função exibir_pedidos?

A

A consulta falharia devido a um erro de sintaxe.

B

A consulta retornaria resultados diferentes.

C

A consulta retornaria os mesmos resultados corretamente.

D

A consulta executaria mais lentamente.

E

A consulta geraria uma exceção de tipo.

A alternativa C está correta.

A ordem dos elementos em uma cláusula JOIN (tabela.coluna = tabela.coluna) não afeta o resultado final da consulta. Ambas as formas são sintaticamente corretas e retornam os mesmos resultados.

Considerações finais

- Principais conectores de SGBD em Python.
- Métodos e exceções de conectores.
- Conexão, acesso e criação de bancos de dados.
- Operações CRUD em bancos de dados.

Explore +

Confira as indicações que separamos para você!

Leia a documentação da **biblioteca sqlite3**, fundamental para a conexão do Python com banco de dados SQLite. Disponível na página do desenvolvedor.

Leia a documentação da **biblioteca psycopg2**, disponível no site do desenvolvedor, uma das bibliotecas mais usadas para a conexão do Python com banco de dados PostgreSQL.

Visite o site do projeto **mysql-connector-python**, um dos conectores necessários para que a conexão de uma aplicação Python com o banco de dados MySQL seja possível.

Confira também o projeto **mysqlclient**, outro conector necessário para realizar a conexão de uma aplicação Python com o banco de dados MySQL.

Explore o site do projeto **PyMySQL**, outro conector utilizado para a conexão de uma aplicação Python com o banco de dados MySQL.

Referências

HIPP, R. D. SQLite. **SQLite**. Consultado na internet em: 8 out. 2020.

MYSQL. **MySQL Connector**. Consultado na internet em: 8 out. 2020.

PSYCOPG. **Psycopg2**. Consultado na internet em: 8 out. 2020.

PYTHON. **Python Software Foundation**. Consultado na internet em: 8 out. 2020.

SQLITEBROWSER. **DB Browser for SQLite**. Consultado na internet em: 8 out. 2020.