CAPÍTULO 11

Banco de dados

Praticamente todo programa precisa ler ou armazenar dados. Para uma quantidade pequena de informação, arquivos simples resolvem o problema, mas, uma vez que os dados precisem ser atualizados, problemas de manutenção e dificuldade de acesso começam a aparecer.

Depois de algum tempo, o programador começa a ver que as operações realizadas com arquivos seguem alguns padrões, como: inserir, alterar, apagar e pesquisar. Ele também começa a perceber que as consultas podem ser feitas com critérios diferentes e que, à medida que o arquivo cresce, as operações se tornam mais lentas. Todos esses tipos de problema foram identificados e resolvidos há bastante tempo, com o uso de programas gerenciadores de banco de dados.

Programas gerenciadores de banco de dados foram desenvolvidos de forma a organizar e facilitar o acesso a grandes massas de informação. No entanto, para usarmos um banco de dados, precisamos saber como eles são organizados. Neste capítulo, abordaremos os conceitos básicos de banco de dados, bem como a utilização da linguagem SQL e o acesso via linguagem de programação, no caso, Python.

As listagens do capítulo 11 são bem maiores que as do restante do livro. É recomendado ler este capítulo perto de um computador e em alguns casos com as listagens impressas em papel ou facilmente acessíveis no site do livro.

11.1 Conceitos básicos

Para começarmos a utilizar os termos de banco de dados, é importante entender como as coisas funcionavam antes de usarmos computadores para controlar o registro de informações.

Não muito tempo atrás, as pessoas usavam a própria memória para armazenar informações importantes, como os números de telefones de amigos próximos e

até o próprio número de telefone. Hoje, com a proliferação de celulares, praticamente perdemos essa capacidade, por falta de uso. Mas, para entender banco de dados, precisamos compreender por que eles foram criados, quais problemas eles resolveram e como eram as coisas antes deles.

Imagine que você é uma pessoa extremamente popular e que precisa controlar mais de 100 contatos (nome e telefone) dos amigos mais próximos. Lembre-se de imaginar essa situação num mundo sem computadores. O que você faria para controlar todos os dados?

Provavelmente, você escreveria todos os nomes e telefones de seus novos contatos em folhas de papel. Usando um caderno, poder-se-ia facilmente anotar um nome abaixo do outro, com o número de telefone anotado mais à direita da folha.

Tabela 11.1 – Exemplo de nomes anotados numa folha de papel

Nome	Telefone
João	98901-0109
André	98902-8900
Maria	97891-3321

O problema com anotações em papel é que não conseguimos alterar os dados lá escritos, salvo se escrevermos a lápis, mas o resultado nunca é muito agradável. Outro problema é que não conhecemos novas pessoas em ordem alfabética, o que resulta numa lista de nomes e telefones desordenada. O papel também não ajuda quando temos mais amigos com nomes que começam por uma letra que outra, ou quando nossos amigos têm vários nomes, como João Carlos, e você nunca se lembra se registrou como João ou como Carlos!

Agora, imagine que estamos controlando não apenas nossos amigos, mas também contatos comerciais. No caso dos amigos, apenas nome e telefone já bastam para reestabelecer o contato. No caso de um contato comercial, a empresa onde a pessoa trabalha e a função que ela desempenha são importantes também. Com o tempo, precisaríamos de vários cadernos ou pastas, um para cada contato, organizados em armários, talvez uma gaveta para cada tipo de registro. Antes de os computadores se tornarem populares, as coisas eram organizadas dessa forma, e em muitos lugares são assim até hoje.

Uma vez que os problemas que podem ser resolvidos com banco de dados foram apresentados, nós já podemos aprender alguns conceitos importantíssimos para continuar o estudo, como: campos, registros, tabelas e tipos de dados.

Campos são a menor unidade de informação em um sistema gerenciador de banco de dados. Se fizermos uma comparação com nossa agenda no papel, nome e telefone seriam dois campos. O campo nome armazenaria o nome de cada contato; e o campo telefone, o número de telefone, respectivamente.

Cada linha de nossa agenda seria chamada de registro. Um registro é formado por um conjunto conhecido de campos. Em nosso exemplo, cada pessoa na agenda, com seu nome e telefone, formaria um registro.

Podemos pensar em tabelas do banco de dados como a unidade de armazenamento de registros do mesmo tipo. Imagine uma entrada da agenda telefônica, onde cada registro, contendo nome e telefone, seria armazenado. O conjunto de registros do mesmo tipo é organizado em tabelas, nesse caso, na tabela agenda ou lista telefônica.

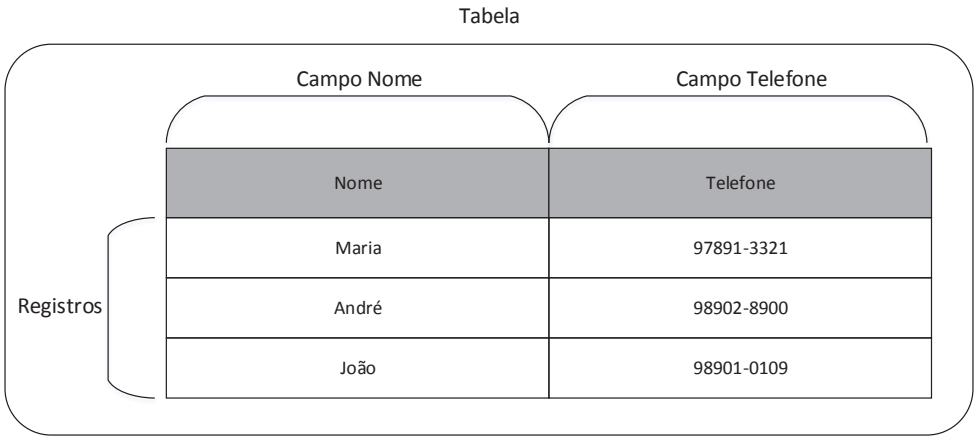

Figura 11.1 – Campos, Registros e tabela.

Os conceitos de campo, registro e tabela são fundamentais para o entendimento do resto de texto, vide figura 11.1. Não hesite em reler essa seção, caso algum desses conceitos ainda não estejam claros para você. Eles serão reapresentados nas seções seguintes, quando serão aplicados em um banco de dados demonstrativo.

11.2 SQL

Structured Query Language (SQL – Linguagem de Consulta Estruturada) é a linguagem usada para criar bancos de dados, gerar consultas, manipular (inserir, atualizar, alterar e apagar) registros e, principalmente, realizar consultas. É

uma linguagem de programação especializada na manipulação de dados, baseada na álgebra relacional e no modelo relacional criado por Edgar F. Codd (*http://pt.wikipedia.org/wiki/Edgar_Frank_Codd*).

Neste capítulo, nós veremos como escrever comandos SQL para o banco SQLite que vem pré-instalado com o interpretador Python e que é facilmente acessível de um programa. A linguagem SQL é definida por vários padrões, como SQL-92, mas cada banco de dados introduz modificações e adições ao padrão, embora o funcionamento básico continue o mesmo. Neste capítulo, nós veremos exclusivamente os comandos SQL no formato aceito pelo banco SQLite.

11.3 Python & SQLite

O SQLite é um gerenciador de banco de dados leve e completo, muito utilizado e presente mesmo em telefones celulares. Uma de suas principais características é não precisar de um servidor dedicado, sendo capaz de se iniciar a partir de seu programa. Nesta seção, nós veremos os comandos mais importantes e as etapas necessárias para utilizar o SQLite. Vejamos um programa Python que cria um banco de dados, uma tabela e um registro na listagem 11.1.

▶ Listagem 11.1 Exemplo de uso do SQLite em Python

```
import sqlite3 ❶
conexão = sqlite3.connect("agenda.db") ❷>
cursor = conexão.cursor() ❸
cursor.execute('''
        create table agenda(
            nome text,
            telefone text)
        ''') ❹
cursor.execute('''
        insert into agenda (nome, telefone)
            values(?, ?)
            ''', ("Nilo", "7788-1432")) ❺
conexão.commit() ❻
cursor.close() ❼
conexão.close() ❽
```

A primeira coisa a fazer é informar que utilizaremos um banco SQLite. Isso é feito em ❶. Depois do `import`, várias funções e objetos que acessam o banco de dados se tornam disponíveis ao seu programa. Antes de continuarmos, vamos criar o banco de dados em ❷. A conexão com o banco de dados se assemelha à manipulação de um arquivo, é a operação análoga a abrir um arquivo. O nome do banco de dados que estamos criando será gravado no arquivo `agenda.db`. A extensão `.db` é apenas uma convenção, mas é recomendado diferenciar o nome do arquivo de um arquivo normal, principalmente porque todos os seus dados serão guardados nesse arquivo. A grande vantagem de um banco de dados é que o registro de informações e toda a manutenção dos dados são feitos automaticamente para você com comandos SQL.

Em ❸, criamos um cursor. Cursores são objetos utilizados para enviar comandos e receber resultados do banco de dados. Um cursor é criado para uma conexão, chamando-se o método `cursor()`. Uma vez que obtivemos um cursor, nós podemos enviar comandos ao banco de dados. O primeiro deles é criar uma tabela para guardar nomes e telefones. Vamos chamá-la de agenda:

```
create table agenda(nome text, telefone text)
```

O comando SQL usado para criar uma tabela é `create table`. Esse comando precisa do nome da tabela a criar; nesse exemplo, `agenda` e uma lista de campos entre parênteses. `Nome` e `telefone` são nossos campos e `text` é o tipo. Embora em Python não precisemos declarar o tipo de uma variável, a maioria dos bancos de dados exige um tipo para cada campo. No caso do SQLite, o tipo não é exigido, mas vamos continuar a usá-lo para que você não tenha problemas com outros bancos, e para que a noção de tipo comece a fazer sentido. Um campo do tipo `text` pode armazenar dados como uma string do Python.

Em ❹, utilizamos o método `execute` de nosso cursor para enviar o comando ao banco de dados. Observe que escrevemos o comando em várias linhas, usando apóstrofos triplos do Python. A linguagem SQL não exige essa formatação, embora ela deixe o comando mais claro e simples de entender. Você poderia ter escrito tudo em uma só linha e mesmo utilizar uma string simples do Python.

Com a tabela criada, podemos começar a introduzir nossos dados. Vejamos o comando SQL usado para inserir um registro:

```
insert into agenda (nome, telefone) values (?, ?)
```

O comando `insert` precisa do nome da tabela, onde iremos inserir os dados, e também do nome dos campos e seus respectivos valores. `into` faz parte do comando `insert` e é escrito antes do nome da tabela. O nome dos campos é escrito logo a seguir, separados por vírgula e, dessa vez, não precisamos mais informar o

tipo dos campos, apenas a lista de nomes. Os valores que vamos inserir na tabela são especificados também entre parênteses, mas na segunda parte do comando `insert` que começa após a palavra `values`. Em nosso exemplo, a posição de cada valor foi marcada com interrogações, uma para cada campo. A ordem dos valores é a mesma dos campos; logo, a primeira interrogação se refere ao campo `nome`; a segunda, ao campo `telefone`. A linguagem SQL permite que escrevamos os valores diretamente no comando, como uma grande string, mas, hoje em dia, esse tipo de sintaxe não é recomendada, por ser insegura e facilmente utilizada para gerar um ataque de segurança de dados chamado SQLInjection (*http://pt.wikipedia.org/wiki/Inje%C3%A7%C3%A3o_de_SQL*). Você não precisa se preocupar com isso agora, principalmente porque, ao utilizarmos as interrogações, estamos utilizando parâmetros que evitam esse tipo de problema. Podemos entender as interrogações como um equivalente das máscaras de string do Python, mas que utilizaremos com comandos SQL.

Em ❺, utilizamos o método `execute` para executar o comando `insert`, mas, dessa vez, passamos os dados logo após o comando. No exemplo, "Nilo" e "7788-1432" irão substituir a primeira e a segunda interrogação quando o comando for executado. É importante notar que os dois valores foram passados como uma tupla.

Uma vez que o comando é executado, os dados são enviados para o banco de dados, mas ainda não estão gravados definitivamente. Isso acontece, pois estamos usando uma transação. Transações serão apresentadas com mais detalhes em outra seção; por enquanto, considere o comando `commit` em ❻ como parte das operações necessárias para modificar o banco de dados.

Antes de terminarmos o programa, fechamos (`close`) o cursor e a conexão com o banco de dados, respectivamente em ❼ e ❽. Veremos mais adiante como usar a sentença **`with`** do Python para facilitar essas operações.

Execute o programa e verifique se o arquivo `agenda.db` foi criado. Se você executar o programa uma segunda vez, um erro será gerado com a mensagem:

```
Traceback (most recent call last):
  File "criatabela.py", line 9, in <module>
    ''')
sqlite3.OperationalError: table agenda already exists
```

Este erro acontece porque a tabela `agenda` já existe. Se você precisar executar o programa novamente, apague o arquivo `agenda.db`. Lembre-se de que todos os dados estão nesse arquivo e, ao apagá-lo, tudo é perdido. Você pode apagar esse arquivo sempre que quiser reinicializar o banco de dados.

Vejamos agora como ler os dados que gravamos no banco de dados, vamos fazer uma consulta (*query*). O programa da listagem 11.2 realiza a consulta e mostra os resultados na tela.

▶ Listagem 11.2 – Consulta

```
import sqlite3
conexão = sqlite3.connect("agenda.db")
cursor = conexão.cursor()
cursor.execute("select * from agenda") ❶
resultado=cursor.fetchone() ❷
print("Nome: %s\nTelefone: %s" % (resultado)) ❸
cursor.close()
conexão.close()
```

O programa é muito parecido com o anterior, uma vez que precisamos importar o módulo do SQLite, estabelecer uma conexão e criar um cursor. O comando SQL que realiza uma consulta é o comando `select`.

```
select * from agenda
```

O comando `select`, em sua forma mais simples, utiliza uma lista de campos e uma lista de tabelas. Em nosso exemplo, a lista de campos foi substituída por `*` (asterisco). O asterisco representa todos os campos da tabela sendo consultada, nesse caso `nome` e `telefone`. A palavra `from` é utilizada para separar a lista de campos da lista de tabelas. Em nosso exemplo, apenas a tabela `agenda`. O comando `select` é executado na linha ❶.

Para acessar os resultados do comando `select`, devemos utilizar o método `fetchone` de nosso cursor ❷. Esse método retorna uma tupla com os resultados de nossa consulta ou `None`, caso a tabela esteja vazia. Para simplificar nosso exemplo, o teste de `None` foi retirado.

A tupla retornada possui a mesma ordem dos campos de nossa consulta, nesse caso `nome` e `telefone`. Assim, `resultado[0]` é o primeiro campo, no caso `nome` e `resultado[1]` é o segundo, `telefone`. Em ❸, usamos uma string em Python e uma máscara com dois `%s`, um para cada campo na tupla `resultado`.

Execute o programa e verifique o resultado:

```
Nome: Nilo
Telefone: 7788-1432
```

Vejamos agora como incluir os outros telefones de nossa agenda. O programa da listagem 11.3 apresenta o método `executemany`. A principal diferença entre `executemany` e `execute` é que `executemany` trabalha com vários valores. Em nosso exemplo, utilizamos uma lista de tuplas, `dados`. Cada elemento da lista é uma tupla com dois valores, exatamente como fizemos no programa da listatem 11.1.

▶ Listagem 11.3 – Inserindo múltiplos registros

```
import sqlite3
dados =  [ ("João",  "98901-0109"),
           ("André", "98902-8900"),
           ("Maria", "97891-3321")]
conexão = sqlite3.connect("agenda.db")
cursor = conexão.cursor()
cursor.executemany('''
      insert into agenda (nome, telefone)
      values(?, ?)
      ''', dados)
conexão.commit()
cursor.close()
conexão.close()
```

Com os dados inseridos pelo programa, nossa agenda deve ter agora 4 registros. Vejamos como imprimir o conteúdo de nossa tabela, usando o mesmo comando SQL, mas, dessa vez, trabalhando com vários resultados.

▶ Listagem 11.4 – Consulta com múltiplos resultados

```
import sqlite3
conexão = sqlite3.connect("agenda.db")
cursor = conexão.cursor()
cursor.execute("select * from agenda")
resultado=cursor.fetchall() ❶
for registro in resultado:
   print("Nome: %s\nTelefone: %s" % (registro)) ❷
cursor.close()
conexão.close()
```

Veja o novo programa de consulta na Listagem 11.4. Em ❶, utilizamos o método `fetchall` de nosso cursor para retornar uma lista com os resultados de nossa

consulta. Em ❷, utilizamos a variável `registro` para exibir os dados. Assim como vimos o método `executemany`, que aceita uma lista de tuplas como parâmetro, `fetchall` retorna uma lista de tuplas. Cada elemento dessa lista é uma tupla contendo todos os campos retornados pela consulta. Uma vez que temos a lista `resultado`, utilizamos um simples **`for`** para trabalhar com cada registro.

O método `fetchall` retorna **`None`** caso o resultado da consulta seja vazio. Veremos isso em outros exemplos. Para consultas pequenas, contendo poucos registros como resultado, o método `fetchall` é muito interessante e fácil de utilizar. Para consultas maiores, onde mais de 100 registros são retornados, outros métodos de obter os resultados da consulta podem ser mais interessantes. Esses métodos evitam a criação de uma longa lista, que pode ocupar uma grande quantidade de memória e demorar muito tempo para executar.

A listagem 11.5 mostra o método `fetchone` ❶ sendo utilizado dentro de uma estrutura de repetição **`while`**. Como não sabemos quantos registros serão retornados, utilizamos um **`while True`**, que é interrompido quando o método `fetchone` retorna **`None`**, significando que todos os resultados da consulta já foram obtidos. Você pode ler *fetch* como obter; logo, `fetchone` seria obter um resultado e `fetchall` obter todos os resultados. A vantagem de `fetchone` nesse caso é que imprimimos o resultado da consulta tão logo obtemos um e mantemos a impressão à medida que outros resultados forem chegando. Esse tempo de entrega é um conceito importante a perceber, uma vez que os dados vêm do banco de dados para o nosso programa. Essa transferência é controlada pelo banco de dados, responsável por executar nossa consulta e gerar os resultados.

▶ Listagem 11.5 – Consulta, registro por registro

```
import sqlite3
conexão = sqlite3.connect("agenda.db")
cursor = conexão.cursor()
cursor.execute("select * from agenda")
while True:
    resultado=cursor.fetchone() ❶
    if resultado == None:
        break
    print("Nome: %s\nTelefone: %s" % (resultado))
cursor.close()
conexão.close()
```

Antes de passarmos para comandos SQL mais avançados, vejamos a estrutura `with` do Python que pode nos ajudar a não nos esquecermos de chamar os métodos `close` de nossos objetos. A listagem 11.6 mostra o programa equivalente ao da listagem 11.5, mas utilizando a cláusula `with`. Uma das vantagens de utilizarmos `with` é que criamos um bloco onde um objeto é tido como válido. Se algo acontecer dentro do bloco, como uma exceção, a estrutura `with` garante que o método `close` será chamado. Na realidade, `with` chama o método `__exit__` no fim do bloco e funciona muito bem com arquivos e conexões de banco de dados. Infelizmente, cursores não possuem o método `__exit__`, obrigando-nos a chamar manualmente o método `close`, ou a importar um módulo especial, `contextlib`, que oferece a função `closing` ❶, que adapta um cursor com um método `__exit__`, que chama `close`. Por enquanto, esse detalhe pode ficar apenas como uma curiosidade, mas falaremos mais de `with` no restante deste capítulo.

▶ Listagem 11.6 – Uso do with para fechar a conexão

```
import sqlite3
from contextlib import closing ❶

with sqlite3.connect("agenda.db") as conexão:
   with closing(conexão.cursor()) as cursor:
      cursor.execute("select * from agenda")
      while True:
         resultado=cursor.fetchone()
         if resultado == None:
            break
         print("Nome: %s\nTelefone: %s" % (resultado))
```

Exercício 11.1 Faça um programa que crie o banco de dados *preços.db* com a tabela `preços` para armazenar uma lista de preços de venda de produtos. A tabela deve conter o nome do produto e seu respectivo preço. O programa também deve inserir alguns dados para teste.

Exercício 11.2 Faça um programa para listar todos os preços do banco *preços.db*.

11.4 Consultando registros

Até agora, não fomos além do que poderíamos ter feito com simples arquivos texto. A facilidade de um sistema de banco de dados começa a aparecer quando precisamos procurar e alterar dados. Ao trabalharmos com arquivos, essas operações devem ser implementadas em nossos programas, mas com o SQLite, podemos realizá-las usando comandos SQL. Primeiramente, vamos utilizar uma variação do comando `select` para mostrar apenas alguns registros, implementando uma seleção de registros com base em uma pesquisa. Pesquisas em SQL são feitas com a cláusula `where`. Vejamos o comando SQL que seleciona todos os registros da agenda, cujo nome seja igual a "Nilo".

```
select * from agenda where nome = "Nilo"
```

Veja que apenas acrescentamos a cláusula `where` após o nome da tabela. O critério de seleção ou de pesquisa deve ser escrito como uma expressão, no caso `nome = "Nilo"`. A listagem 11.7 mostra o programa com essa modificação.

▶ Listagem 11.7 – Consulta com filtro de seleção

```
import sqlite3
conexão = sqlite3.connect("agenda.db")
cursor = conexão.cursor()
cursor.execute("select * from agenda where nome = 'Nilo'")
while True:
    resultado=cursor.fetchone()
    if resultado == None:
      break
    print("Nome: %s\nTelefone: %s" % (resultado))
cursor.close()
conexão.close()
```

Ao executarmos o programa da listagem 11.7, devemos ter apenas um resultado:

```
Nome: Nilo
Telefone: 7788-1432
```

Veja que escrevemos `'Nilo'` entre apóstrofos. Aqui, podemos usar um pouco do que já sabemos sobre strings em Python e escrever:

```
cursor.execute('select * from agenda where nome = "Nilo"')
```

Ou seja, poderíamos trocar as aspas por apóstrofos ou ainda usar aspas triplas:

```
cursor.execute("""select * from agenda where nome = "Nilo" """)
```

No caso de nosso exemplo, o nome 'Nilo' é uma constante e não há problemas em escrevê-lo diretamente em nosso comando `select`. No entanto, caso o nome a filtrar viesse de uma variável, ficaríamos tentados a escrever um programa, como o da listagem 11.8.

▶ Listagem 11.8 – Consulta com filtro de seleção vindo de variável

```
import sqlite3
nome=input("Nome a selecionar: ")
conexão = sqlite3.connect("agenda.db")
cursor = conexão.cursor()
cursor.execute('select * from agenda where nome = "%s"' % nome)
while True:
    resultado=cursor.fetchone()
    if resultado == None:
      break
    print("Nome: %s\nTelefone: %s" % (resultado))
cursor.close()
conexão.close()
```

Execute o programa da listagem 11.8 com vários valores: Nilo, João e Maria. Experimente também com um nome que não existe. A cláusula `where` funciona de forma parecida a um filtro. Imagine que o comando `select` cria uma lista e que a expressão lógica definida no `where` é avaliada para cada elemento. Quando o resultado dessa avaliação é verdadeiro, a linha é copiada para uma outra lista, a lista de resultados, retornada pela nossa consulta.

Veja que o programa funciona relativamente muito bem, exceto quando nada encontramos e o programa termina sem dizer muita coisa. Nós vamos corrigir esse problema logo a seguir, mas execute o programa da listagem 11.8 mais uma vez e digite a seguinte sequência como nome:

```
X" or "1"="1
```

Surpreso com o resultado? Esse é o motivo por não utilizarmos variáveis em nossas consultas. Esse tipo de vulnerabilidade é um exemplo de SQLInjection, um ataque bem conhecido. Isso acontece por que o comando SQL resultante é:

```
select * from agenda where nome = "X" or "1"="1"
```

Para evitar este tipo de ataque, sempre utilize parâmetros com valores variáveis.

O `or` da linguagem SQL funciona de forma semelhante ao **`or`** do Python. Dessa forma, nossa entrada de dados foi modificada por um valor digitado no programa. Esse tipo de erro é muito grave e pode ficar muito tempo em nossos programas sem ser percebido. Isso acontece porque a consulta é uma string como outra qualquer, e o valor passado para o método `execute` é a string resultante. Dessa forma, o valor digitado pelo usuário pode introduzir elementos que nós não desejamos. Os operadores relacionais **`and`** e **`not`** funcionam exatamente como em Python, e você também pode usá-los em expressões SQL.

Para não cairmos nesse tipo de armadilha, utilizaremos sempre parâmetros em nossas consultas.

▶ Listagem 11.9 – Consulta utilizando parâmetros

```
import sqlite3

nome=input("Nome a selecionar: ")
conexão = sqlite3.connect("agenda.db")
cursor = conexão.cursor()
cursor.execute('select * from agenda where nome = ?', (nome,))
x=0
while True:
    resultado=cursor.fetchone()
    if resultado == None:
      if x == 0:
         print("Nada encontrado.")
      break
    print("Nome: %s\nTelefone: %s" % (resultado))
    x+=1
cursor.close()
conexão.close()
```

Na listagem 11.9, utilizamos um parâmetro, como fizemos antes para inserir nossos registros. Um detalhe importante é que escrevemos `(nome,)`, repare a vírgula após `nome`. Esse detalhe é importante, pois o segundo parâmetro do método `execute` é uma tupla, e, em Python, tuplas com apenas um elemento são escritas com uma vírgula após o primeiro valor. Veja também que utilizamos a variável `x` para contar quantos resultados obtivemos. Como o método `fetchone` retorna **`None`**

quando todos os registros foram recebidos, verificamos se `x == 0`, para saber se algo já havia sido obtido anteriormente ou se devemos imprimir uma mensagem dizendo que nada foi encontrado.

Exercício 11.3 Escreva um programa que realize consultas do banco de dados preços.db, criado no exercício 11.1. O programa deve perguntar o nome do produto e listar seu preço.

Exercício 11.4 Modifique o programa do exercício 11.3 de forma a perguntar dois valores e listar todos os produtos com preços entre esses dois valores.

11.5 Atualizando registros

Já sabemos como criar tabelas, inserir registros e fazer consultas simples. Vamos começar a usar o comando `update` para alterar nossos registros. Por exemplo, vamos alterar o registro com o telefone de "Nilo" para "12345-6789":

```
update agenda set telefone = "12345-6789" where nome = 'Nilo'
```

A cláusula `where` funciona como no comando `select`, ou seja, ela avalia uma expressão lógica que, quando verdadeira, inclui o registro na lista de registros a modificar. A segunda parte do comando `update` é a cláusula `set`. Essa cláusula é usada para indicar o que fazer nos registros selecionados pela expressão do `where`. No exemplo, `set telefone = "12345-6789"` muda o conteúdo do campo `telefone` para "12345-6789". O comando inteiro poderia ser lido como: atualize os registros da tabela agenda, alterando o telefone para "12345-6789" em todos os registros onde o campo `nome` é igual a "Nilo". Vejamos o programa da listagem 11.10.

▶ Listagem 11.10 – Atualizando o telefone

```
import sqlite3
conexão = sqlite3.connect("agenda.db")
cursor = conexão.cursor()
cursor.execute("""update agenda
                  set telefone = '12345-6789'
                  where nome = 'Nilo'""")
conexão.commit()
conexão.close()
```

Nesse exemplo, utilizamos constantes, logo não precisamos usar parâmetros. As mesmas regras que aprendemos para o comando `select` se aplicam ao comando `update`. Se os valores não forem constantes, você tem que utilizar parâmetros.

O comando `update` pode alterar mais de um registro de uma só vez. Faça uma cópia do arquivo `agenda.db` e experimente modificar o programa da Listagem 11.10, retirando a cláusula `where`:

```
update agenda set telefone = "12345-6789"
```

Você verá que todos os registros foram modificados:

```
Nome: Nilo
Telefone: 12345-6789
Nome: João
Telefone: 12345-6789
Nome: André
Telefone: 12345-6789
Nome: Maria
Telefone: 12345-6789
```

Sem a cláusula `where`, todos os registros serão selecionados e alterados. Vamos utilizar a propriedade `rowcount` de nosso cursor para saber quantos registros foram alterados por nosso `update`. Veja o programa da listagem 11.11 com essas alterações.

▶ Listagem 11.11 – Exemplo de update sem where e com rowcount

```
import sqlite3
conexão = sqlite3.connect("agenda.db")
cursor = conexão.cursor()
cursor.execute("""update agenda
                  set telefone = '12345-6789' """)
print("Registros alterados: ", cursor.rowcount)
conexão.commit()
conexão.close()
```

Não se esqueça de que, após modificar o banco de dados, precisamos chamar o método `commit`, como fizemos ao inserir os registros. Caso nos esqueçamos, as alterações serão perdidas.

A propriedade `rowcount` é muito interessante para confirmarmos o resultado de comandos de atualização, como `update`. Essa propriedade não funciona com `select`, retornando sempre `-1`. Por isso, na listagem 11.9, contamos os registros retornados

por nosso `select` em vez de usarmos `rowcount`. No caso de `update`, poderíamos fazer uma verificação de quantos registros seriam alterados antes de chamarmos o `commit`. Vejamos o programa da listagem 11.12.

▶ Listagem 11.12 – update com rollback

```
import sqlite3
conexão = sqlite3.connect("agenda.db")
cursor = conexão.cursor()
cursor.execute("""update agenda
                  set telefone = '12345-6789' """)
print("Registros alterados: ", cursor.rowcount)
if cursor.rowcount == 1:
   conexão.commit()
   print("Alterações gravadas")
else:
   conexão.rollback()
   print("Alterações abortadas")
conexão.close()
```

No programa da listagem 11.12, utilizamos o valor de `rowcount` para decidir se as alterações deveriam ser registradas ou ignoradas. Como já sabemos, o método `commit` grava as alterações. O método `rollback` faz o inverso, abortando as alterações e deixando o banco de dados como antes. Os métodos `commit` e `rollback` fazem o controle de transações do banco de dados. Podemos entender uma transação como um conjunto de operações que deve ser executado completamente. Isso significa operações que não fazem sentido, salvo se realizadas em um só grupo. Se a execução do grupo falhar, todas as alterações causadas durante a transação corrente devem ser revertidas (`rollback`). Caso tudo ocorra como planejado, as operações serão armazenadas definitivamente no banco de dados (`commit`). Veremos outros exemplos mais adiante.

Exercício 11.5 Escreva um programa que aumente o preço de todos os produtos do banco *preços.db* em 10%.

Exercício 11.6 Escreva um programa que pergunte o nome do produto e um novo preço. Usando o banco *preços.db*, atualize o preço deste produto no banco de dados.

11.6 Apagando registros

Além de inserir, consultar e alterar registros, podemos também apagá-los. O comando `delete` apaga registros com base em um critério de seleção, especificado na cláusula `where` que já conhecemos. Faça outra cópia do arquivo *agenda.db*. Copie o antigo banco de dados, com os registros antes de executarmos o programa da listagem 11.11.

A sintaxe do comando `delete` é:

```
delete from agenda where nome = 'Maria'
```

Ou seja, apague da tabela `agenda` todos os registros com nome igual a "Maria". Vejamos o programa da listagem 11.13.

▶ Listagem 11.13 – Apagando registros

```
import sqlite3
conexão = sqlite3.connect("agenda.db")
cursor = conexão.cursor()
cursor.execute("""delete from agenda
                  where nome = 'Maria' """)
print("Registros apagados: ", cursor.rowcount)
if cursor.rowcount == 1:
   conexão.commit()
   print("Alterações gravadas")
else:
   conexão.rollback()
   print("Alterações abortadas")
conexão.close()
```

Utilizamos o método `rowcount` para ter certeza de que estávamos apagando apenas um registro. Assim como no comando `insert` e `update`, você precisa chamar `commit` para gravar as alterações ou `rollback`,caso contrário.

11.7 Simplificando o acesso sem cursores

A interface de banco de dados do Python nos permite executar alguns comandos utilizando diretamente o objeto da conexão, sem criarmos explicitamente um cursor. Vejamos a listagem 11.14, que é uma versão simplificada do programa da listagem 11.4.

▶ **Listagem 11.14 – Consulta vários registros, acesso simplificado**

```
import sqlite3
with sqlite3.connect("agenda.db") as conexão:
    for registro in conexão.execute("select * from agenda"): ❶
        print("Nome: %s\nTelefone: %s" % (registro))
```

Na listagem 11.14, utilizamos a estrutura **`with`** para facilitar o fechamento da conexão. Em ❶, `conexão.execute` retorna um cursor que pode ser usado com **`for`**. Você pode também utilizar o método `executemany` diretamente com o objeto conexão. Essa utilização simplificada funciona muito bem com SQLite, mas não faz parte da interface padrão de banco de dados do Python, a DB-API 2.0. Ao utilizar cursores, você obedece a DB-API 2.0 que é implementada por outros bancos de dados, simplificando a migração de seu código para outros bancos de dados, como o MySQL ou MariaDB.

11.8 Acessando os campos como em um dicionário

Acessar os campos por posição nem sempre é tão fácil. Em Python, usando SQLite, podemos acessá-los pelo nome, adicionando uma linha:

```
conexão.row_factory = sqlite3.Row
```

Vejamos o programa completo na Listagem 11.15.

▶ **Listagem 11.15 – Acessando os campos pelo nome**

```
import sqlite3
conexão = sqlite3.connect("agenda.db")
conexão.row_factory = sqlite3.Row
cursor = conexão.cursor()
for registro in cursor.execute("select * from agenda"):
    print("Nome: %s\nTelefone: %s" % (registro["nome"], registro["telefone"]))
cursor.close()
conexão.close()
```

Dessa forma, registro pode ser acessado como se fosse um dicionário, onde o nome do campo é usado como chave. Outra facilidade que essa linha traz é que as chaves são aceitas independentemente se escrevermos o nome dos campos em maiúsculas ou minúsculas. Por exemplo:

```
    print("Nome: %s\nTelefone: %s" % (registro["NOME"], registro["Telefone"]))
```

11.9 Gerando uma chave primária

Até agora, trabalhamos apenas com campos normais, ou seja, campos que contêm dados. Conforme nossas tabelas crescem, trabalhar com os dados pode não ser a melhor solução, e precisaremos acrescentar campos para manter o banco de dados. Uma dessas necessidades é identificar cada registro de maneira única. Nós podemos utilizar dados que não se repetem, ou que não deveriam se repetir, como o nome da pessoa, como uma chave primária. Podemos entender uma chave primária como a chave de um dicionário, mas, nesse caso, para tabelas em nosso banco de dados. Qualquer campo ou um conjunto de campos podem servir de chave primária. Uma alternativa oferecida pelo SQLite é a geração automática de chaves. Nesse caso, o banco se encarrega de criar números únicos para cada registro.

Vamos implementar outro banco de dados, com a população da cada estado do Brasil. Veremos como deixar o SQLite gerar uma chave primária automaticamente:

```
create table estados(
                id integer primary key autoincrement,
                nome text,
                população integer)
```

Ao criarmos a tabela estados, estamos especificando três campos: `id`, `nome` e `população`. Veja que `id` e `população` são do tipo `integer`, ou seja, números inteiros (`int`). `id` é o campo que escolhemos para ser a chave primária dessa tabela, e escrevemos `primary key autoincrement` para que o SQLite gere eses números automaticamente. Entenda `id` como a abreviação de identificador único ou identidade. *Primary key* significa chave primária.

O programa da listagem 11.16 cria o banco de dados `brasil.db`, a tabela estados e também inclui o nome e a população de todos os estados brasileiros. Os dados foram extraídos da Wikipédia (*http://pt.wikipedia.org/wiki/Anexo:Lista_de_unidades_federativas_do_Brasil_por_popula%C3%A7%C3%A3o*).

▶ Listagem 11.16 – Criação do banco de dados com a população dos estados brasileiros

```
import sqlite3

dados = [["São Paulo",43663672], ["Minas Gerais",20593366], ["Rio de Janeiro",
16369178], ["Bahia",15044127], ["Rio Grande do Sul",11164050], ["Paraná",10997462],
["Pernambuco",9208511], ["Ceará",8778575], ["Pará",7969655], ["Maranhão",6794298],
["Santa Catarina",6634250], ["Goiás",6434052], ["Paraíba", 3914418], ["Espírito
Santo",3838363], ["Amazonas",3807923], ["Rio Grande do Norte", 3373960], ["Alagoas",
3300938], ["Piauí",3184165], ["Mato Grosso",3182114], ["Distrito Federal",2789761],
["Mato Grosso do Sul",2587267], ["Sergipe",2195662], ["Rondônia",1728214],
["Tocantins",1478163], ["Acre",776463], ["Amapá",734995], ["Roraima",488072]]
```

```
conexão = sqlite3.connect("brasil.db")
conexão.row_factory = sqlite3.Row
cursor = conexão.cursor()
cursor.execute("""create table estados(
                  id integer primary key autoincrement,
                  nome text,
                  população integer
                  )""")
cursor.executemany("insert into estados(nome, população) values(?,?)", dados)
conexão.commit()
cursor.close()
conexão.close()
```

O valor do campo `id` será gerado automaticamente. Uma vez que temos a população dos estados, vamos fazer uma consulta para listar os estados em ordem alfabética. Veja o programa completo na listagem 11.17. Ao executar esse programa, observe os valores gerados no campo `id`, no caso valores numéricos de 1 a 27.

▶ Listagem 11.17 – Consulta dos estados brasileiros, ordenados por nome

```
import sqlite3
conexão = sqlite3.connect("brasil.db")
conexão.row_factory = sqlite3.Row
print("%3s %-20s %12s" % ("Id","Estado","População"))
print("="*37)
for estado in conexão.execute("select * from estados order by nome"):
   print("%3d %-20s %12d" %
         (estado["id"],
          estado["nome"],
          estado["população"]))
conexão.close()
```

A grande diferença é que estamos utilizando a cláusula `order by` para ordenar os resultados de nossa consulta; neste caso, pelo campo `nome`.

```
   select * from estados order by nome
```

Modifique o programa para que os estados sejam impressos pela população, usando a consulta:

```
select * from estados order by população
```

Execute o programa novamente e veja que os estados foram agora impressos pela população, mas da menor para a maior. Embora essa seja a ordem normal, quando trabalhamos com lista de estados por população, esperamos ver do estado mais populoso para o menos populoso, ou seja, na ordem inversa (decrescente) dos valores. Vejamos esse resultado ao adicionarmos `desc` após o nome do campo:

```
select * from estados order by população desc
```

11.10 Alterando a tabela

Vamos acrescentar mais alguns campos a nossa tabela de estados. Um campo para a região do Brasil e outro para a sigla do estado. Em SQL, o comando utilizado para alterar os campos de uma tabela é o `alter table`.

```
alter table estados add sigla text
alter table estados add add região text
```

O comando `alter table` do SQLite é limitado se comparado com outros bancos de dados. Em outros bancos, pode-se alterar vários campos com um só `alter table`, mas no SQLite, somos obrigados a alterar um campo de cada vez. As limitações do alter table do SQLite não param por aí. Por isso, planeje suas tabelas com cuidado e, caso precise realizar grandes mudanças, prefira criar uma outra tabela com as alterações e copiar os dados da tabela antiga. Execute o programa da listagem 11.18 para alterar a tabela estados e adicionar os campos sigla e região.

▶ Listagem 11.18 – Alterando a tabela

```
import sqlite3
with sqlite3.connect("brasil.db") as conexão:
    conexão.execute("""alter table estados
                       add sigla text""")
    conexão.execute("""alter table estados
                       add região text""")
```

Agora que a tabela possui os novos campos, vamos alterar nossos registros e preencher a região e sigla de cada estado. Execute o programa da listagem 11.19.

▶ Listagem 11.19 – Preenchendo a sigla e a região de cada estado

```
import sqlite3
dados = [["SP", "SE", "São Paulo"], ["MG", "SE", "Minas Gerais"], ["RJ", "SE", "Rio
de Janeiro"], ["BA", "NE", "Bahia"], ["RS", "S", "Rio Grande do Sul"], ["PR", "S",
"Paraná"], ["PE", "NE", "Pernambuco"], ["CE", "NE", "Ceará"], ["PA", "N", "Pará"],
["MA", "NE", "Maranhão"], ["SC", "S", "Santa Catarina"], ["GO", "CO", "Goiás"],
["PB", "NE", "Paraíba"], ["ES", "SE", "Espírito Santo"], ["AM", "N", "Amazonas"],
["RN", "NE", "Rio Grande do Norte"], ["AL", "NE", "Alagoas"], ["PI", "NE", "Piauí"],
["MT", "CO", "Mato Grosso"], ["DF", "CO", "Distrito Federal"], ["MS", "CO", "Mato
Grosso do Sul"], ["SE", "NE", "Sergipe"], ["RO", "N", "Rondônia"], ["TO", "N",
"Tocantins"], ["AC", "N", "Acre"], ["AP", "N", "Amapá"], ["RR", "N", "Roraima"] ]


with sqlite3.connect("brasil.db") as conexão:
    conexão.executemany("""update estados
                           set sigla = ?,
                           região = ?
                           where nome = ?""", dados)
```

Agora nosso banco de dados possui uma tabela estados com a população, sigla e região de cada estado. Esses novos campos permitirão utilizarmos funções de agregação da linguagem SQL: `count`, `min`, `max`, `avg` e `sum`.

11.11 Agrupando dados

Um banco de dados pode realizar operações de agrupamento de dados facilmente. Podemos, por exemplo, solicitar o valor mínimo de um grupo de registros, assim como também o máximo ou a média desses valores. No entanto, temos que modificar nossos comandos SQL para indicar uma cláusula de agrupamento, ou seja, devemos indicar como o banco de dados deve agrupar nossos registros.

Vejamos como realizar um grupo simples e exibir quantos registros fazem parte desse grupo, usando a função `count`. A cláusula SQL que indica agrupamento é `group by`, seguida do nome dos campos que compõem o grupo. Imagine que o banco vai concatenar cada um desses campos, criando um valor para cada registro. Vamos chamar esse valor de "chave de grupo". Todos os registros com a mesma chave de grupo fazem parte do mesmo grupo e serão representados por apenas um registro na consulta de seleção. Essa consulta com grupo só pode conter os campos utilizados para compor a chave do grupo e funções de agrupamento de dados, como `min` (mínimo), `max` (máximo), `avg` (média), `sum` (soma) e `count` (contagem).

Um exemplo concreto como nosso banco de dados é agrupar os estados por região. A consulta seria algo como:

```
select região, count(*) from estados group by região
```

Esse comando utilizada a cláusula `group by` região para especificar a chave de grupo. Dessa forma, todos os registros que pertencem a mesma região são agrupados. Observe que os campos após o `select` incluem região e `count(*)`. O campo região pôde ser incluído, pois faz parte da chave de grupo especificada na `group by`. A função `count(*)` retorna quantos registros fazem parte do grupo. Vejamos o resultado do programa da listagem 11.20.

▶ Listagem 11.20 – Agrupando e contando estados por região

```
import sqlite3
print("Região Número de Estados")
print("====== ================")
with sqlite3.connect("brasil.db") as conexão:
    for região in conexão.execute("""
        select região, count(*)
        from estados
        group by região"""):
        print("{0:6} {1:17}".format(*região))
```

Resultado da execução do programa da listagem 11.20:

```
Região Número de Estados
====== ================
CO                    4
N                     7
NE                    9
S                     3
SE                    4
```

Vamos adicionar as funções `min`, `max`, `sum`, `avg` no campo população. Veja o programa da listagem 11.21.

▶ Listagem 11.21 – Usando as funções de agregação

```
import sqlite3
print("Região Estados População  Mínima    Máxima      Média    Total (soma)")
print("====== =======          ========= ========== ==========  ============")
```

```
with sqlite3.connect("brasil.db") as conexão:
   for região in conexão.execute("""
      select região, count(*), min(população),
             max(população), avg(população), sum(população)
      from estados
      group by região"""):
      print("{0:6} {1:7} {2:18,} {3:10,} {4:10,.0f} {5:13,}".format(*região))
   print ("\nBrasil: {0:6} {1:18,} {2:10,} {3:10,.0f} {4:13,}".format(
      *conexão.execute("""
         select count(*), min(população), max(população),
                avg(população), sum(população) from estados""").fetchone()))
```

Resultado do programa da listagem 11.21:

```
Região Estados População  Mínima    Máxima      Média    Total (soma)
====== =======          ========= ========== ==========  ============
CO           4          2,587,267  6,434,052  3,748,298    14,993,194
N            7            488,072  7,969,655  2,426,212    16,983,485
NE           9          2,195,662 15,044,127  6,199,406    55,794,654
S            3          6,634,250 11,164,050  9,598,587    28,795,762
SE           4          3,838,363 43,663,672 21,116,145    84,464,579

Brasil:     27            488,072 43,663,672  7,445,618   201,031,674
```

Com o programa da listagem 11.21, conseguimos calcular a população mínima, máxima, média e total de cada região e também para o Brasil. Veja que na segunda consulta, a que calcula os dados para o Brasil, não utilizamos a cláusula `group by`, fazendo com que todos os registros façam parte do grupo.

Ao utilizarmos as funções de agregação e a cláusula `group by`, podemos continuar usando tudo que já aprendemos em SQL, como as cláusulas `where` e `order by`. Vejamos o mesmo programa da listagem 11.21, mas com as linhas ordenadas pela população total de cada região em ordem decrescente.

▶ Listagem 11.22 – Funções de agregação com order by

```
import sqlite3
print("Região Estados População  Mínima    Máxima      Média    Total (soma)")
print("====== =======          ========= ========== ==========  ============")
```

```
with sqlite3.connect("brasil.db") as conexão:
    for região in conexão.execute("""
        select região, count(*), min(população),
               max(população), avg(população), sum(população)
        from estados
        group by região
        order by sum(população) desc"""):
        print("{0:6} {1:7} {2:18,} {3:10,} {4:10,.0f} {5:13,}".format(*região))
    print ("\nBrasil: {0:6} {1:18,} {2:10,} {3:10,.0f} {4:13,}".format(
        *conexão.execute("""
            select count(*), min(população), max(população),
                   avg(população), sum(população) from estados""").fetchone()))
```

No programa da listagem 11.22, apenas acrescentamos a linha `order by sum(população) desc` no final de nossa consulta. Veja que repetimos a função de agregação `sum(população)` para indicar que a ordenação será feita pela soma da população. Você pode utilizar a cláusula **as** do SQL para dar nomes às colunas de uma consulta. Veja a consulta modificada para usar **as** e criar uma coluna `tpop` para a soma da população:

```
select região, count(*), min(população), max(população), avg(população),
       sum(população) as tpop
    from estados group by região
    order by tpop desc
```

Veja que escrevemos `sum(população) as tpop`, dando o nome `tpop` à soma. Depois, utilizamos o nome `tpop` na cláusula do `order by`. Esse tipo de construção evita a repetição da função na consulta e facilita a leitura.

Resultado da execução do programa da listagem 11.22:

```
Região Estados População  Mínima    Máxima      Média    Total (soma)
====== ======= ========== ========= ========== ========== ============
SE           4          3,838,363 43,663,672 21,116,145    84,464,579
NE           9          2,195,662 15,044,127  6,199,406    55,794,654
S            3          6,634,250 11,164,050  9,598,587    28,795,762
N            7            488,072  7,969,655  2,426,212    16,983,485
CO           4          2,587,267  6,434,052  3,748,298    14,993,194

Brasil:     27            488,072 43,663,672  7,445,618   201,031,674
```

Podemos também filtrar os resultados após o agrupamento, usando a cláusula `having`. Para entender a diferença entre `where` e `having`, imagine que `where` é executada antes do agrupamento, selecionando os registros que farão parte do resultado, antes do agrupamento ser realizado. A cláusula `having`, avalia o resultado do agrupamento e decide quais farão parte do resultado final. Por exemplo, podemos escolher apenas as regiões com mais de 5 estados. Como a quantidade de estados por região só é conhecida após o agrupamento (`group by`), essa condição deve aparecer em uma cláusula `having`.

```
select região, count(*), min(população),
         max(população), avg(população), sum(população) as tpop
         from estados group by região
       having count(*)>5
         order by tpop desc
```

Veja o programa completo na listagem 11.23.

▶ Listagem 11.23 – Utilizando having para listar apenas as regiões com mais de 5 estados

```
import sqlite3
print("Região Estados População  Mínima    Máxima      Média    Total (soma)")
print("====== =======          ========= ========== ==========  ============")

with sqlite3.connect("brasil.db") as conexão:
   for região in conexão.execute("""
      select região, count(*), min(população),
             max(população), avg(população), sum(população) as tpop
      from estados
      group by região
      having count(*)>5
      order by tpop desc"""):
      print("{0:6} {1:7} {2:18,} {3:10,} {4:10,.0f} {5:13,}".format(*região))
```

Resultando em:

```
Região Estados População  Mínima    Máxima      Média    Total (soma)
====== =======          ========= ========== ==========  ============
NE           9          2,195,662 15,044,127  6,199,406    55,794,654
N            7            488,072  7,969,655  2,426,212    16,983,485
```

Uma vez que apenas as regiões Norte (N) e Nordeste (NE) possuem mais de 5 estados.

11.12 Trabalhando com datas

Embora o SQLite trabalhe com datas, o tipo DATE não é suportado diretamente, gerando uma certa confusão entre datas e strings. Vamos criar uma tabela com um campo do tipo data.

▶ Listagem 11.24 – Criando uma tabela de feriados nacionais

```
import sqlite3

feriados = [["2014-01-01", "Confraternização Universal"], ["2014-04-21", "Tiradentes"],
["2014-05-01", "Dia do trabalhador"], ["2014-09-07", "Independência"],
["2014-10-12", "Padroeira do Brasil"], ["2014-11-02", "Finados"], ["2014-11-15",
"Proclamação da República"], ["2014-12-25", "Natal"] ]

with sqlite3.connect("brasil.db") as conexão:
    conexão.execute("create table feriados(id integer primary key autoincrement,
       data date, descrição text)")
    conexão.executemany("insert into feriados(data,descrição) values (?,?)", feriados)
```

No programa da listagem 11.24, criamos a tabela feriados e inserimos algumas datas. Observe que escrevemos as datas no formato ISO 8601 (*http://pt.wikipedia.org/wiki/ISO_8601*), ou seja: ANO-MÊS-DIA. Nesse formato, a data do Natal (25/12/2014) é escrita como 2014-12-25. Escrever as datas nesse formato é uma característica do SQLite. Sempre escreva suas datas no formato ISO ao trabalhar com esse gerenciador de banco de dados. Observe que utilizamos o tipo `date` (data) na coluna `data`. Modifique o ano de 2014 para o ano corrente, caso necessário.

Vejamos como acessar esses valores no programa da listagem 11.25.

▶ Listagem 11.25 – Acessando um campo do tipo data

```
import sqlite3
with sqlite3.connect("brasil.db") as conexão:
    for feriado in conexão.execute("select * from feriados"):
        print(feriado)
```

Que resulta em:

```
(1, '2014-01-01', 'Confraternização Universal')
(2, '2014-04-21', 'Tiradentes')
(3, '2014-05-01', 'Dia do trabalhador')
(4, '2014-09-07', 'Independência')
(5, '2014-10-12', 'Padroeira do Brasil')
```

```
(6, '2014-11-02', 'Finados')
(7, '2014-11-15', 'Proclamação da República')
(8, '2014-12-25', 'Natal')
```

No programa da listagem 11.25, acessamos o campo data como fazemos até então, sem algum procedimento especial. Veja que, ao imprimirmos a tupla `feriado` com o resultado de nossa seleção, o campo `data` foi impresso como uma string qualquer. Nada impede que utilizemos strings para representar datas, como fizemos ao criar o banco de dados, mas campos datas são mais interessantes, pois podemos facilmente consultar o dia da semana e também realizar operações com datas.

Vamos modificar nossa conexão com o SQLite de forma a solicitar o processamento dos tipos de campo em nossas consultas. Ao solicitarmos a conexão, devemos passar `detect_types=sqlite3.PARSE_DECLTYPES` como parâmetro. Vejamos a listagem 11.26 com essa modificação.

▶ Listagem 11.26 – Solicitando o tratamento do tipo dos campos

```
import sqlite3


with sqlite3.connect("brasil.db",detect_types=sqlite3.PARSE_DECLTYPES) as conexão:
    for feriado in conexão.execute("select * from feriados"):
        print(feriado)
```

O resultado do programa da listagem 11.26 é bem diferente:

```
(1, datetime.date(2014, 1, 1), 'Confraternização Universal')
(2, datetime.date(2014, 4, 21), 'Tiradentes')
(3, datetime.date(2014, 5, 1), 'Dia do trabalhador')
(4, datetime.date(2014, 9, 7), 'Independência')
(5, datetime.date(2014, 10, 12), 'Padroeira do Brasil')
(6, datetime.date(2014, 11, 2), 'Finados')
(7, datetime.date(2014, 11, 15), 'Proclamação da República')
(8, datetime.date(2014, 12, 25), 'Natal')
```

Veja que os valores do campo `data` agora são objetos da classe `datetime.date`. Isso evita termos que fazer a conversão manualmente de string para `datetime.date`. No programa da listagem 11.27, utilizamos o método `strftime` do objeto da classe `datetime.date` para exibir apenas o dia e o mês da data, sem o ano.

▶ **Listagem 11.27 – Trabalhando com datas**

```
import sqlite3
with sqlite3.connect("brasil.db",detect_types=sqlite3.PARSE_DECLTYPES) as conexão:
    conexão.row_factory = sqlite3.Row
    for feriado in conexão.execute("select * from feriados"):
        print("{0} {1}".format(feriado["data"].strftime("%d/%m"), feriado["descrição"]))
```

No programa da listagem 11.27, voltamos a utilizar `row_factory` para acessarmos os campos por nome, como em um dicionário. O método `strftime` foi utilizado com a máscara `"%d/%m"` para exibir apenas o dia e o mês. Você pode verificar os formatos aceitos por strftime na tabela 9.3.

Vejamos o resultado do programa da listagem 11.27:

```
01/01 Confraternização Universal
21/04 Tiradentes
01/05 Dia do trabalhador
07/09 Independência
12/10 Padroeira do Brasil
02/11 Finados
15/11 Proclamação da República
25/12 Natal
```

Vejamos um pouco o que podemos fazer com os objetos do módulo `datetime`.

▶ **Listagem 11.28 – Feriados nos próximos 60 dias**

```
import sqlite3
import datetime

hoje = datetime.date.today()
hoje60dias = hoje + datetime.timedelta(days=60)

with sqlite3.connect("brasil.db",detect_types=sqlite3.PARSE_DECLTYPES) as conexão:
    conexão.row_factory = sqlite3.Row
    for feriado in conexão.execute("select * from feriados where data >= ? and data
<= ?", (hoje, hoje60dias)):
        print("{0} {1}".format(feriado["data"].strftime("%d/%m"), feriado["descrição"]))
```

O programa da listagem 11.28 utiliza objetos do módulo `datetime`. Em `hoje`, guardamos a data atual (`datetime.date.today()`). Em `hoje60dias`, utilizamos um objeto do tipo `datetime.timedelta` para acrescentar 60 dias à data atual. Com esses dois

objetos `date`, podemos utilizar a cláusula `where` do SQLite para selecionar os feriados entre `hoje` e `hoje60dias`. Consulte a documentação do Python para saber mais sobre o módulo `datetime`. Os objetos das classes `timedelta` e `datetime.datetime` são bastante úteis, caso você precise realizar operações com datas e guardar a hora com a data (`datetime`).

11.13 Chaves e relações

Agora que já sabemos o básico de como manipular registros em nosso banco de dados, veremos conceitos mais avançados que permitirão trabalharmos com várias tabelas. Para selecionarmos nossos registros, vimos que precisamos construir expressões lógicas que identifiquem ou que permitam a seleção desses registros. No caso de nossa agenda, o campo `nome` foi usado em nossas expressões, mas utilizar dados como critério de seleção não é uma boa ideia a longo termo. Dados podem mudar e se repetir entre várias tabelas. Por exemplo, imagine a situação onde nossa agenda teria vários telefones por pessoa, como na tabela 11.2:

Tabela 11.2 – Agenda com uma tabela

nome	número	tipo
Nilo	12345-6789	Casa
Nilo	98745-4321	Celular

Nesse exemplo, poderíamos simplesmente adicionar dois registros com o mesmo nome, mas estaríamos complicando nosso trabalho mais tarde, pois, se quiséssemos mudar o telefone de um dos registros, não poderíamos utilizar `nome = "Nilo"` como critério de seleção, seríamos obrigados a utilizar uma condição composta por `nome` e `telefone`.

Esse problema poderia ser resolvido adicionando-se uma chave que identificasse cada pessoa de forma única. Essa chave pode ser um simples número, desde que único, vamos chamá-la de identificador, ou simplesmente `id`. A tabela 11.3 mostra nossos dados com esse novo campo.

Tabela 11.3 – Agenda com uma tabela e uma chave

id	nome	número	tipo
1	Nilo	12345-6789	Casa
1	Nilo	98745-4321	Celular

Embora nossos dados estejam em melhor forma, ainda estamos repetindo o campo `nome` várias vezes. E o campo `id` não identifica unicamente cada registro. Esse tipo de problema é chamado de redundância de dados. Em uma base de dados, quanto menos redundância tivermos em nossos dados, mais fácil será de mantê-los. Por exemplo, imagine que queiramos mudar o nome Nilo para adicionar também Menezes. Teríamos que atualizar os dois registros, pois guardamos a mesma informação em dois lugares (registros) diferentes.

Uma melhor forma de representar esses dados é dividindo nossos dados em várias tabelas. Por exemplo, uma tabela para nome e outra para telefone. Vejamos as tabelas 11.4 e 11.5 com essa nova divisão. Veja que o campo `id` na tabela 11.4 pode ser usado como chave primária. A chave primária (`id`) da tabela nomes foi copiada na tabela 11.5, no campo `id_nome`.

Tabela 11.4 – Tabela nomes

id	Nome
1	Nilo

Tabela 11.5 – Tabela Telefones

id_nome	número	tipo
1	12345-6789	Casa
1	98745-4321	Celular

Dessa forma, armazenamos o `nome` em apenas um lugar. Ainda temos outros problemas, pois agora nossos telefones não possuem um identificador único. Veja como ficaria nossa tabela, acrescentando-se uma chave a Telefones, tabela 11.6:

Tabela 11.6 – Tabela telefones

id	id_nome	número	tipo
1	1	12345-6789	Casa
2	1	98745-4321	Celular

Assim, uma chave primária é um campo de um registro que o identifica de forma única na tabela. Resolvemos nosso problema de redundância, mas como acessar esses dados em tabelas diferentes com comandos SQL? Bem, vamos utilizar o comando `select`, mas com várias tabelas e especificar uma forma de as interligar.

```
select * from nomes, telefones where nomes.id = telefones.id_nomes
```

Esse comando difere de nossos outros exemplos por utilizar mais de uma tabela, após a cláusula `from`. Uma vez que utilizamos várias tabelas, somos obrigados a especificar como essas tabelas se relacionam; caso contrário, obteremos o que é chamado de produto cartesiano, onde nosso resultado conterá a combinação de cada registro da primeira tabela, com cada registro da segunda. É esse relacionamento que é especificado em `nomes.id = telefones.id_nomes`, ou seja, especificamos um critério que liga as duas tabelas; no caso, quando o campo `id` da tabela nomes (`nomes.id`) for igual ao campo `id_nomes` da tabela telefones (`telefones.id_nomes`).

Mas o tipo do telefone ainda se repete, o que pode levar a resultados indesejáveis em nossa agenda. Vamos criar outra tabela para guardar os tipos de telefone de forma a não repeti-los. Veja o resultado nas tabelas 11.7 e 11.8.

Tabela 11.7 Tabela telefones com o campo id_tipo

id	id_nome	número	id_tipo
1	1	12345-6789	1
2	1	98745-4321	2

Tabela 11.8 Tabela Tipos

id	descrição
1	Casa
2	Celular

Vejamos os comandos SQL para criar essas tabelas:

```
create table tipos(id integer primary key autoincrement,
                   descrição text);
create table nomes(id integer primary key autoincrement,
                   nome text);
create table telefones(id integer primary key autoincrement,
                   id_nome integer,
                   número text,
                   id_tipo integer);
```

Com o uso de `primary key autoincrement`, como já vimos anteriormente, o SQLite se encarregará de gerar os números que utilizaremos em nossas chaves primárias. Agora, podemos revistar a agenda do capítulo 10 e convertê-la para utilizar um banco de dados em vez de um simples arquivo-texto.

11.14 Convertendo a agenda para utilizar um banco de dados

Converter a agenda para um banco de dados nos levará a enfrentar um problema de mapeamento entre objetos e os bancos de dados relacionais, como o SQLite. Um dos maiores problemas nesse tipo de mapeamento é manter os dados entre nosso programa e o banco de dados sincronizados. Existem bibliotecas inteiras escritas apenas para resolver esse tipo de problema, usando o que é chamado de Mapeamento Objeto Relacional (*Object-Relational Mapping*, ORM). Em nossa agenda, temos uma lista de registros, cada um com um nome e uma lista de telefones. Cada telefone possui um tipo pré-cadastrado.

Primeiramente, vamos criar uma subclasse de `ListaÚnica` para controlar os registros apagados de nossas listas. Depois, criaremos métodos em uma outra classe, chamada `DBAgenda`, responsável por manter o banco de dados e executar as operações da agenda. Uma mudança nessa nova versão da agenda é que carregamos o registro apenas quando precisamos carregar. Ao voltarmos ao menu principal, todas as mudanças já estarão salvas no banco de dados, fazendo as opções Lê e Grava inúteis.

▶ Listagem 11.29 – Novas classes – listagem parcial

```
class DBListaÚnica(ListaÚnica):
    def __init__(self, elem_class):
        super().__init__(elem_class)
        self.apagados = []
    def remove(self, elem):
        if elem.id is not None:
            self.apagados.append(elem.id)
        super().remove(elem)
    def limpa(self):
        self.apagados = []
class DBNome(Nome):
    def __init__(self, nome, id_=None):
        super().__init__(nome)
        self.id = id_
class DBTipoTelefone(TipoTelefone):
    def __init__(self, id_, tipo):
        super().__init__(tipo)
        self.id = id_
```

```
class DBTelefone(Telefone):
    def __init__(self, número, tipo=None, id_=None, id_nome=None):
        super().__init__(número, tipo)
        self.id = id_
        self.id_nome = id_nome
class DBTelefones(DBListaÚnica):
    def __init__(self):
        super().__init__(DBTelefone)
class DBTiposTelefone(ListaÚnica):
    def __init__(self):
        super().__init__(DBTipoTelefone)
class DBDadoAgenda:
    def __init__(self, nome):
        self.nome = nome
        self.telefones = DBTelefones()
    @property
    def nome(self):
        return self.__nome
    @nome.setter
    def nome(self, valor):
        if type(valor)!=DBNome:
            raise TypeError("nome deve ser uma instância da classe DBNome")
        self.__nome = valor
    def pesquisaTelefone(self, telefone):
        posição = self.telefones.pesquisa(DBTelefone(telefone))
        if posição == -1:
            return None
        else:
            return self.telefones[posição]
```

A classe `DBListaÚnica` herda de nossa classe `ListaÚnica` e sua principal função é manter uma lista de `id` apagados. Isso nos permitirá apagar os elementos de nossas listas e, numa fase seguinte, apagá-los do banco de dados. A classe `DBListaÚnica` só pode trabalhar com classes que possuam um atributo `id`.

As classes `DBNome` e `DBTelefone` derivam de `Nome` e `Telefone` respectivamente. A principal diferença é que agora elas incluem o atributo `id`. Veja que esse atributo é um

parâmetro opcional, pois, ao criarmos nossos objetos, eles não terão ainda suas chaves primárias. Além disso, usaremos o fato de que objetos sem `id` provavelmente acabaram de ser criados e precisam ser inseridos no banco de dados. Isso ficará mais claro no programa completo.

Já na classe `DBDadosAgenda` modificamos o tipo da lista de telefones de `Telefones` para `DBTelefones`. A classe `DBTelefones` é uma derivação de `DBListaÚnica` que aceita apenas elementos do tipo `DBTelefone`. Fizemos o mesmo entre `DBTipoTelefone` e `DBTiposTelefones`.

Até agora, fizemos apenas a mudança dos tipos, em preparação para trabalhar com o banco de dados. A principal mudança foi o novo atributo `id` que acrescentamos em todas as nossas classes. É o valor desse campo que utilizaremos em nossas consultas, alterações e remoções.

▶ Listagem 11.30 – Listagem parcial – Classe DBAgenda

```
BANCO = """
create table tipos(id integer primary key autoincrement,
                   descrição text);
create table nomes(id integer primary key autoincrement,
                   nome text);
create table telefones(id integer primary key autoincrement,
                   id_nome integer,
                   número text,
                   id_tipo integer);
insert into tipos(descrição) values ("Celular");
insert into tipos(descrição) values ("Fixo");
insert into tipos(descrição) values ("Fax");
insert into tipos(descrição) values ("Casa");
insert into tipos(descrição) values ("Trabalho");
"""


class DBAgenda:
    def __init__(self, banco):
        self.tiposTelefone = DBTiposTelefone()
        self.banco = banco
        novo = not os.path.isfile(banco)
        self.conexão = sqlite3.connect(banco)
```

```
        self.conexão.row_factory = sqlite3.Row
        if novo:
            self.cria_banco()
        self.carregaTipos()
    def carregaTipos(self):
        for tipo in self.conexão.execute("select * from tipos"):
            id_ = tipo["id"]
            descrição = tipo["descrição"]
            self.tiposTelefone.adiciona(DBTipoTelefone(id_, descrição))
    def cria_banco(self):
        self.conexão.executescript(BANCO)
    def pesquisaNome(self, nome):
        if not isinstance(nome, DBNome):
            raise TypeError("nome deve ser do tipo DBNome")
        achado = self.conexão.execute("""select count(*)
                                         from nomes where nome = ?""",
                                         (nome.nome,)).fetchone()
        if(achado[0]>0):
            return self.carrega_por_nome(nome)
        else:
            return None
    def carrega_por_id(self, id):
        consulta = self.conexão.execute(
                "select * from nomes where id = ?", (id,))
        return carrega(consulta.fetchone())
    def carrega_por_nome(self, nome):
        consulta = self.conexão.execute(
                "select * from nomes where nome = ?", (nome.nome,))
        return self.carrega(consulta.fetchone())
    def carrega(self, consulta):
        if consulta is None:
            return None
        novo = DBDadoAgenda(DBNome(consulta["nome"], consulta["id"]))
        for telefone in self.conexão.execute(
            "select * from telefones where id_nome = ?",
            (novo.nome.id,)):
```

```
            ntel = DBTelefone(telefone["número"], None,
                              telefone["id"], telefone["id_nome"])
            for tipo in self.tiposTelefone:
                if tipo.id == telefone["id_tipo"]:
                    ntel.tipo = tipo
                    break
            novo.telefones.adiciona(ntel)
        return novo
    def lista(self):
        consulta = self.conexão.execute(
                "select * from nomes order by nome")
        for registro in consulta:
            yield self.carrega(registro)
def novo(self, registro):
        try:
            cur = self.conexão.cursor()
            cur.execute("insert into nomes(nome) values (?)",
                        (str(registro.nome),))
            registro.nome.id = cur.lastrowid
            for telefone in registro.telefones:
                cur.execute("""insert into telefones(número,
                               id_tipo, id_nome) values (?,?,?)""",
                            (telefone.número, telefone.tipo.id,
                             registro.nome.id))
                telefone.id = cur.lastrowid
            self.conexão.commit()
        except:
            self.conexão.rollback()
            raise
        finally:
            cur.close()
    def atualiza(self, registro):
        try:
            cur = self.conexão.cursor()
            cur.execute("update nomes set nome=? where id = ?",
                (str(registro.nome), registro.nome.id))
```

```
            for telefone in registro.telefones:
                if telefone.id is None:
                    cur.execute("""insert into telefones(número,
                                   id_tipo, id_nome)
                                   values (?,?,?)""",
                            (telefone.número, telefone.tipo.id,
                            registro.nome.id))
                    telefone.id = cur.lastrowid
                else:
                    cur.execute("""update telefones set número=?,
                                          id_tipo=?, id_nome=?
                                          where id = ?""",
                            (telefone.número, telefone.tipo.id,
                            registro.nome.id, telefone.id))
            for apagado in registro.telefones.apagados:
                cur.execute("delete from telefones where id = ?", (apagado,))
            self.conexão.commit()
            registro.telefones.limpa()
        except:
            self.conexão.rollback()
            raise
        finally:
            cur.close()
    def apaga(self, registro):
        try:
            cur = self.conexão.cursor()
            cur.execute("delete from telefones where id_nome = ?",
                            (registro.nome.id,))
            cur.execute("delete from nomes where id = ?",
                        (registro.nome.id,))
            self.conexão.commit()
        except:
            self.conexão.rollback()
            raise
        finally:
            cur.close()
```

A listagem 11.30 apresenta a classe `DBAgenda`, que substituirá a classe `Agenda`. A classe `DBAgenda` mantém o banco de dados em sincronia com as classes e objetos em memória, sendo responsável por todas as operações com o banco. Esse tipo de construção em camadas evita termos que escrever o código de manipulação e criação do banco na classe `AppAgenda`.

A primeira coisa que a classe `DBAgenda` faz é verificar se banco de dados já existe. Essa verificação é feita antes do pedido de conexão no método `__init__`. Para saber se o banco já existe, utilizamos a função `os.path.isfile(banco)`. Caso o arquivo não exista, ele será criado pelo método `cria_banco`, chamado logo após a conexão com o banco de dados. Veja que guardamos o objeto conexão como um atributo de `DBAgenda`.

O método `cria_banco` é muito simples, utilizando o método `executescript` da conexão. Esse método executa vários comandos de uma só vez. Para simplificar a criação do banco, escrevemos o código que cria todas as tabelas e popula a tabela tipos na variável global `BANCO`. A execução de vários comandos só é possível porque eles estão separados por `;`.

O método `carrega_tipos` realiza a leitura de todos os tipos de telefone no banco de dados e os guarda na lista `tiposTelefone`. Observe o cuidado em manter os `id`. Esse valor será utilizado depois para obter o tipo correto de cada telefone.

O método `pesquisa_nome` também traz novidades. Nele, executamos uma consulta usando a função `count(*)`. Se um registro for encontrado, o método `carrega_por_nome` é chamado para transformar o resultado de nossa consulta em uma coleção de objetos, da mesma forma que trabalhamos com a agenda no capítulo 10. Veja também que utilizamos a cláusula `where` para pesquisar na tabela nomes diretamente, uma vez que não carregamos todos os registros do banco de dados para a memória.

Em `carrega`, o resultado de nossa consulta é transformado em um objeto `DBDadoAgenda`, após criarmos uma instância de `DBNome` com os campos `nome` e `id` vindos de nossa consulta. Esse tipo de acesso é possível, pois registramos `self.conexão.row_factory = sqlite3.Row` no método `__init__`. Uma vez que o nome foi carregado, utilizamos seu `id` como `id_nome` em nossa próxima consulta, que carregará os valores de telefone. Observe o cuidado durante a leitura dos telefones e a transformação do resultado em `DBTelefone`. Como o tipo ainda não foi carregado e temos todos eles em memória, fazemos uma pesquisa em `self.TiposTelefone` para converter o `id` em uma instância de `TipoTelefone`. É esse tipo de mapeamento que não é tão simples de fazer e que é bastante trabalhoso pela grande quantidade de código necessária para mantê-lo. É nessas horas que um ORM ajuda. Uma

vez que tudo foi convertido, `novo` é retornado com todos os dados pré-carregados.

O método `lista` executa uma consulta total da tabela nomes. Porém, para evitar a criação de uma grande lista, utilizamos a instrução `yield` do Python que retorna cada valor carregado, um de cada vez. Na realidade, o método `lista` retorna um gerador (*generator*) que pode ser utilizado em um `for` do Python. Isso evita termos que carregar e converter todos os valores antes de ter os primeiros resultados na tela.

Em `novo`, convertemos um objeto do tipo `DBDadoAgenda` em registros das tabelas nomes e telefones. Essa conversão é realizada dentro de uma transação, daí o porquê de utilizamos explicitamente um cursor (`cur`). Realizar essa operação dentro de uma transação permitirá melhorar a consistência do banco de dados, pois se um erro acontecer antes de completarmos todas as operações, o estado do banco de dados será revertido ao estado anterior ao início de nossas operações. Como o `id` de nomes é gerado automaticamente, veja que utilizamos a propriedade `lastrowid` de nosso cursor, logo após a execução do `insert`. Desta forma, podemos utilizar o novo id para popular a tabela de telefones. Realizamos então o mesmo processo a cada novo telefone, atribuindo o valor de `lastrowid` ao `id` do telefone.

Já o método `atualiza` é bem mais complexo. Durante a atualização de um registro, precisamos atualizar o campo `nome` na tabela de nomes. Em nossa solução caseira, não temos como saber se nome foi alterado ou não. Poderíamos alterar a classe para saber quando o nome foi alterado e executar o `update` apenas quando necessário. Esta é outra característica das bibliotecas de ORM que trazem esse tipo de funcionalidade em suas classes. Para mantermos a agenda o mais simples possível, `nome` sempre será atualizado. Para cada `telefone`, fazemos a verificação do valor de `telefone.id`. Se um telefone já possui um valor em `id`, provavelmente ele já está registrado no banco de dados e precisa ser atualizado. Caso ainda não possua um valor em `id`, provavelmente se trata de um telefone inserido durante a alteração. Dessa forma, escolhemos entre fazer um `insert` ou um `update` na tabela telefones. Por último, verificamos se a lista de apagados possui uma lista de `id` a apagar. Essa lista foi construída pela classe `DBListaÚnica`, onde guardamos o `id` de cada elemento apagado. Essa mudança foi necessária, pois diferente dos novos registros e dos registros alterados, os registros apagados são removidos de nossa lista. Se não mantivermos a lista dos `id` apagados, ficaríamos sem saber quais telefones foram removidos, e nosso banco ficaria inconsistente. Para cada `id` na lista de apagados, executamos um `delete`. Logo após, terminamos a transação com um `commit` para nos assegurarmos de que todas as operações foram registradas no banco de dados e, só então, apagamos a lista de telefones removidos com o método `limpa`.

O método `apaga` é um pouco mais simples. Nele, utilizamos a mesma estrutura de proteção e uma transação. O interessante é que, primeiramente, apagamos os telefones e, depois, os nomes. Da forma que geramos nosso banco de dados, essa ordem não importa, uma vez que não estamos utilizando os recursos de integridade referencial do banco.

O programa da agenda é apresentado por inteiro na listagem 11.31.

▶ Listagem 11.31 – Agenda com banco de dados completo

```
import sys
import sqlite3
import os.path
from functools import total_ordering
BANCO = """
create table tipos(id integer primary key autoincrement,
                   descrição text);
create table nomes(id integer primary key autoincrement,
                   nome text);
create table telefones(id integer primary key autoincrement,
                   id_nome integer,
                   número text,
                   id_tipo integer);
insert into tipos(descrição) values ("Celular");
insert into tipos(descrição) values ("Fixo");
insert into tipos(descrição) values ("Fax");
insert into tipos(descrição) values ("Casa");
insert into tipos(descrição) values ("Trabalho");
"""

def nulo_ou_vazio(texto):
    return texto == None or not texto.strip()
def valida_faixa_inteiro(pergunta, inicio, fim, padrão = None):
    while True:
        try:
            entrada = input(pergunta)
            if nulo_ou_vazio(entrada) and padrão != None:
                entrada = padrão
```

```
            valor = int(entrada)
            if inicio <= valor <= fim:
                return(valor)
        except ValueError:
            print("Valor inválido, favor digitar entre %d e %d" %
                 (inicio, fim))
def valida_faixa_inteiro_ou_branco(pergunta, inicio, fim):
    while True:
        try:
            entrada = input(pergunta)
            if nulo_ou_vazio(entrada):
                return None
            valor = int(entrada)
            if inicio <= valor <= fim:
                return(valor)
        except ValueError:
            print("Valor inválido, favor digitar entre %d e %d" %
                 (inicio, fim))
class ListaÚnica:
    def __init__(self, elem_class):
        self.lista = []
        self.elem_class = elem_class
    def __len__(self):
        return len(self.lista)
    def __iter__(self):
        return iter(self.lista)
    def __getitem__(self, p):
        return self.lista[p]
    def indiceVálido(self, i):
        return i>=0 and i<len(self.lista)
    def adiciona(self, elem):
        if self.pesquisa(elem) == -1:
            self.lista.append(elem)
    def remove(self, elem):
        self.lista.remove(elem)
```

```
    def pesquisa(self, elem):
        self.verifica_tipo(elem)
        try:
            return self.lista.index(elem)
        except ValueError:
            return -1
    def verifica_tipo(self, elem):
        if type(elem)!=self.elem_class:
            raise TypeError("Tipo inválido")
    def ordena(self, chave = None):
        self.lista.sort(key= chave)
class DBListaÚnica(ListaÚnica):
    def __init__(self, elem_class):
        super().__init__(elem_class)
        self.apagados = []
    def remove(self, elem):
        if elem.id is not None:
            self.apagados.append(elem.id)
        super().remove(elem)
    def limpa(self):
        self.apagados = []

@total_ordering
class Nome:
    def __init__(self, nome):
        self.nome = nome
    def __str__(self):
        return self.nome
    def __repr__(self):
        return "<Classe {3} em 0x{0:x} Nome: {1} Chave: {2}>".format(
                id(self), self.__nome, self.__chave,
                type(self).__name__)
    def __eq__(self, outro):
        return self.nome == outro.nome
    def __lt__(self, outro):
        return self.nome < outro.nome
```

```
    @property
    def nome(self):
        return self.__nome
    @nome.setter
    def nome(self, valor):
        if nulo_ou_vazio(valor):
            raise ValueError("Nome não pode ser nulo nem em branco")
        self.__nome = valor
        self.__chave = Nome.CriaChave(valor)
    @property
    def chave(self):
        return self.__chave
    @staticmethod
    def CriaChave(nome):
        return nome.strip().lower()
class DBNome(Nome):
    def __init__(self, nome, id_=None):
        super().__init__(nome)
        self.id = id_
@total_ordering
class TipoTelefone:
    def __init__(self, tipo):
        self.tipo = tipo
    def __str__(self):
        return "({0})".format(self.tipo)
    def __eq__(self, outro):
        if outro is None:
            return False
        return self.tipo == outro.tipo
    def __lt__(self, outro):
        return self.tipo < outro.tipo
class DBTipoTelefone(TipoTelefone):
    def __init__(self, id_, tipo):
        super().__init__(tipo)
        self.id = id_
```

```
class Telefone:
    def __init__(self, número, tipo=None):
        self.número = número
        self.tipo = tipo
    def __str__(self):
        if self.tipo!=None:
           tipo = self.tipo
        else:
            tipo = ""
        return "{0} {1}".format(self.número, tipo)
    def __eq__(self, outro):
        return self.número == outro.número and (
               (self.tipo == outro.tipo) or (
                self.tipo == None or outro.tipo == None))
    @property
    def número(self):
        return self.__número
    @número.setter
    def número(self, valor):
        if  nulo_ou_vazio(valor):
            raise ValueError("Número não pode ser None ou em branco")
        self.__número = valor
class DBTelefone(Telefone):
    def __init__(self, número, tipo=None, id_=None, id_nome=None):
        super().__init__(número, tipo)
        self.id = id_
        self.id_nome = id_nome
class DBTelefones(DBListaÚnica):
    def __init__(self):
        super().__init__(DBTelefone)
class DBTiposTelefone(ListaÚnica):
    def __init__(self):
        super().__init__(DBTipoTelefone)
```

```
class DBDadoAgenda:
    def __init__(self, nome):
        self.nome = nome
        self.telefones = DBTelefones()
    @property
    def nome(self):
        return self.__nome
    @nome.setter
    def nome(self, valor):
        if type(valor)!=DBNome:
            raise TypeError("nome deve ser uma instância da classe DBNome")
        self.__nome = valor
    def pesquisaTelefone(self, telefone):
        posição = self.telefones.pesquisa(DBTelefone(telefone))
        if posição == -1:
            return None
        else:
            return self.telefones[posição]
class DBAgenda:
    def __init__(self, banco):
        self.tiposTelefone = DBTiposTelefone()
        self.banco = banco
        novo = not os.path.isfile(banco)
        self.conexão = sqlite3.connect(banco)
        self.conexão.row_factory = sqlite3.Row
        if novo:
            self.cria_banco()
        self.carregaTipos()
    def carregaTipos(self):
        for tipo in self.conexão.execute("select * from tipos"):
            id_ = tipo["id"]
            descrição = tipo["descrição"]
            self.tiposTelefone.adiciona(DBTipoTelefone(id_, descrição))
    def cria_banco(self):
        self.conexão.executescript(BANCO)
```

```
    def pesquisaNome(self, nome):
        if not isinstance(nome, DBNome):
            raise TypeError("nome deve ser do tipo DBNome")
        achado = self.conexão.execute("""select count(*)
                                        from nomes where nome = ?""",
                                        (nome.nome,)).fetchone()
        if(achado[0]>0):
            return self.carrega_por_nome(nome)
        else:
            return None
    def carrega_por_id(self, id):
        consulta = self.conexão.execute(
                "select * from nomes where id = ?", (id,))
        return carrega(consulta.fetchone())
    def carrega_por_nome(self, nome):
        consulta = self.conexão.execute(
                "select * from nomes where nome = ?", (nome.nome,))
        return self.carrega(consulta.fetchone())
    def carrega(self, consulta):
        if consulta is None:
            return None
        novo = DBDadoAgenda(DBNome(consulta["nome"], consulta["id"]))
        for telefone in self.conexão.execute(
            "select * from telefones where id_nome = ?",
            (novo.nome.id,)):
            ntel = DBTelefone(telefone["número"], None,
                              telefone["id"], telefone["id_nome"])
            for tipo in self.tiposTelefone:
                if tipo.id == telefone["id_tipo"]:
                    ntel.tipo = tipo
                    break
            novo.telefones.adiciona(ntel)
        return novo
```

```
    def lista(self):
        consulta = self.conexão.execute(
                "select * from nomes order by nome")
        for registro in consulta:
            yield self.carrega(registro)
    def novo(self, registro):
        try:
            cur = self.conexão.cursor()
            cur.execute("insert into nomes(nome) values (?)",
                        (str(registro.nome),))
            registro.nome.id = cur.lastrowid
            for telefone in registro.telefones:
                cur.execute("""insert into telefones(número,
                               id_tipo, id_nome) values (?,?,?)""",
                            (telefone.número, telefone.tipo.id,
                             registro.nome.id))
                telefone.id = cur.lastrowid
            self.conexão.commit()
        except:
            self.conexão.rollback()
            raise
        finally:
            cur.close()
    def atualiza(self, registro):
        try:
            cur = self.conexão.cursor()
            cur.execute("update nomes set nome=? where id = ?",
                (str(registro.nome), registro.nome.id))
            for telefone in registro.telefones:
                if telefone.id is None:
                    cur.execute("""insert into telefones(número,
                                   id_tipo, id_nome)
                                   values (?,?,?)""",
                            (telefone.número, telefone.tipo.id,
                            registro.nome.id))
                    telefone.id = cur.lastrowid
```

```
                else:
                    cur.execute("""update telefones set número=?,
                                        id_tipo=?, id_nome=?
                                        where id = ?""",
                                (telefone.número, telefone.tipo.id,
                                registro.nome.id, telefone.id))
            for apagado in registro.telefones.apagados:
                cur.execute("delete from telefones where id = ?",
                            (apagado,))
            self.conexão.commit()
            registro.telefones.limpa()
        except:
            self.conexão.rollback()
            raise
        finally:
            cur.close()
    def apaga(self, registro):
        try:
            cur = self.conexão.cursor()
            cur.execute("delete from telefones where id_nome = ?",
                            (registro.nome.id,))
            cur.execute("delete from nomes where id = ?",
                        (registro.nome.id,))
            self.conexão.commit()
        except:
            self.conexão.rollback()
            raise
        finally:
            cur.close()
class Menu:
    def __init__(self):
        self.opções = [ ["Sair", None] ]
    def adicionaopção(self, nome, função):
        self.opções.append([nome, função])
    def exibe(self):
        print("====")
```

```
        print("Menu")
        print("====\n")
        for i, opção in enumerate(self.opções):
            print("[{0}] - {1}".format(i, opção[0]))
        print()
    def execute(self):
        while True:
            self.exibe()
            escolha = valida_faixa_inteiro("Escolha uma opção: ",
                        0, len(self.opções)-1)
            if escolha == 0:
                break
            self.opções[escolha][1]()
class AppAgenda:
    @staticmethod
    def pede_nome():
         return(input("Nome: "))
    @staticmethod
    def pede_telefone():
         return(input("Telefone: "))
    @staticmethod
    def mostra_dados(dados):
        print("Nome: %s" % dados.nome)
        for telefone in dados.telefones:
            print("Telefone: %s" % telefone)
        print()
    @staticmethod
    def mostra_dados_telefone(dados):
        print("Nome: %s" % dados.nome)
        for i, telefone in enumerate(dados.telefones):
           print("{0} - Telefone: {1}".format(i, telefone))
        print()
    def __init__(self, banco):
        self.agenda = DBAgenda(banco)
        self.menu = Menu()
        self.menu.adicionaopção("Novo", self.novo)
```

```
        self.menu.adicionaopção("Altera", self.altera)
        self.menu.adicionaopção("Apaga", self.apaga)
        self.menu.adicionaopção("Lista", self.lista)
        self.ultimo_nome = None
    def pede_tipo_telefone(self, padrão = None):
        for i,tipo in enumerate(self.agenda.tiposTelefone):
            print(" {0} - {1} ".format(i,tipo),end=None)
        t = valida_faixa_inteiro("Tipo: ",0,
            len(self.agenda.tiposTelefone)-1, padrão)
        return self.agenda.tiposTelefone[t]
    def pesquisa(self, nome):
        if type(nome)==str:
            nome = DBNome(nome)
        dado = self.agenda.pesquisaNome(nome)
        return dado
    def novo(self):
        novo = AppAgenda.pede_nome()
        if nulo_ou_vazio(novo):
            return
        nome = DBNome(novo)
        if self.pesquisa(nome) != None:
            print("Nome já existe!")
            return
        registro = DBDadoAgenda(nome)
        self.menu_telefones(registro)
        self.agenda.novo(registro)
    def apaga(self):
        nome = AppAgenda.pede_nome()
        if(nulo_ou_vazio(nome)):
           return
        p = self.pesquisa(nome)
        if p != None:
            self.agenda.apaga(p)
        else:
            print("Nome não encontrado.")
```

```
    def altera(self):
        nome = AppAgenda.pede_nome()
        if(nulo_ou_vazio(nome)):
           return
        p = self.pesquisa(nome)
        if p != None:
            print("\nEncontrado:\n")
            AppAgenda.mostra_dados(p)
            print("Digite enter caso não queira alterar o nome")
            novo = AppAgenda.pede_nome()
            if not nulo_ou_vazio(novo):
                p.nome.nome = novo
            self.menu_telefones(p)
            self.agenda.atualiza(p)
        else:
            print("Nome não encontrado!")
    def menu_telefones(self, dados):
        while True:
            print("\nEditando telefones\n")
            AppAgenda.mostra_dados_telefone(dados)
            if(len(dados.telefones)>0):
                print("\n[A] - alterar\n[D] - apagar\n", end="")
            print("[N] - novo\n[S] - sair\n")
            operação = input("Escolha uma operação: ")
            operação = operação.lower()
            if operação not in ["a","d","n", "s"]:
                print("Operação inválida. Digite A, D, N ou S")
                continue
            if operação == 'a' and len(dados.telefones)>0:
                self.altera_telefones(dados)
            elif operação == 'd' and len(dados.telefones)>0:
                self.apaga_telefone(dados)
            elif operação == 'n':
                self.novo_telefone(dados)
            elif operação == "s":
                break
```

```
    def novo_telefone(self, dados):
        telefone = AppAgenda.pede_telefone()
        if nulo_ou_vazio(telefone):
            return
        if dados.pesquisaTelefone(telefone) != None:
            print("Telefone já existe")
        tipo = self.pede_tipo_telefone()
        dados.telefones.adiciona(DBTelefone(telefone, tipo))
    def apaga_telefone(self, dados):
        t = valida_faixa_inteiro_ou_branco(
            "Digite a poisção do número a apagar, enter para sair: ",
            0, len(dados.telefones)-1)
        if t == None:
            return
        dados.telefones.remove(dados.telefones[t])
    def altera_telefones(self,dados):
        t = valida_faixa_inteiro_ou_branco(
            "Digite a poisção do número a alterar, enter para sair: ",
            0, len(dados.telefones)-1)
        if t == None:
            return
        telefone = dados.telefones[t]
        print("Telefone: %s" % telefone)
        print("Digite enter caso não queira alterar o número")
        novotelefone = AppAgenda.pede_telefone()
        if not nulo_ou_vazio(novotelefone):
            telefone.número = novotelefone
        print("Digite enter caso não queira alterar o tipo")
        telefone.tipo = self.pede_tipo_telefone(
            self.agenda.tiposTelefone.pesquisa(telefone.tipo))
```

```
    def lista(self):
        print("\nAgenda")
        print("-"*60)
        for e in self.agenda.lista():
            AppAgenda.mostra_dados(e)
        print("-"*60)
    def execute(self):
        self.menu.execute()
if __name__ == "__main__":
    if len(sys.argv) > 1:
        app = AppAgenda(sys.argv[1])
        app.execute()
    else:
        print("Erro: nome do banco de dados não informado")
        print("      agenda.py nome_do_banco")
```

Na classe `AppAgenda`, modificamos as opções do menu e os tipos usados nas pesquisas. Veja que, com a utilização da classe `DBAgenda`, conseguimos isolar as operações de banco de dados da classe `AppAgenda`. Como não temos operações de leitura e gravação, o nome do banco de dados deve ser passado obrigatoriamente na linha de comando. Uma mensagem de erro será exibida caso você se esqueça desse detalhe.