

Essential SQLAlchemy

MAPEANDO PYTHON PARA BANCOS DE DADOS



Jason Myers e Rick Copeland

Essential SQLAlchemy

Dive into SQLAlchemy, the popular, open source code library that helps Python programmers work with relational databases such as Oracle, MySQL, PostgreSQL, and SQLite. Using real-world examples, this practical guide shows you how to build a simple database application with SQLAlchemy, and how to connect to multiple databases simultaneously with the same metadata.

SQL is a powerful language for querying and manipulating data, but it's tough to integrate it with your application. SQLAlchemy helps you map Python objects to database tables without substantially changing your existing Python code. If you're an intermediate Python developer with knowledge of basic SQL syntax and relational theory, this book serves as both a learning tool and a handy reference.

Essential SQLAlchemy includes several sections:

- **SQLAlchemy Core:** Provide database services to your applications in a Pythonic way with the SQL Expression Language
- **SQLAlchemy ORM:** Use the object relational mapper to bind database schema and operations to data objects in your application
- **Alembic:** Use this lightweight database migration tool to handle changes to the database as your application evolves
- **Cookbook:** Learn how to use SQLAlchemy with web frameworks like Flask and libraries like SQLAlchemycodegen

Jason Myers is a Software Engineer at Cisco, where he works on OpenStack. Before moving to development, he worked as a systems architect building data centers and cloud architectures for a variety of large tech companies, hospitals, stadiums, and telecom providers.

Rick Copeland is the co-founder and CEO of Synapp.io, an Atlanta-based company that provides a SaaS solution for the email compliance and deliverability space. He is also an experienced Python developer with a focus on both relational and NoSQL databases.

PYTHON/DATABASES

US \$39.99 CAN \$45.99

ISBN: 978-1-491-91646-9



9 781491 916469

“Packed with clear, concise examples, *Essential SQLAlchemy* is required reading for anyone working with relational databases from Python. From low-level queries, to high-level ORM access and schema migrations, this book covers everything you need to connect your application to any relational database.”

—Doug Hellmann

OpenStack Contributor at Hewlett Packard Enterprise, author of *The Python Standard Library by Example* (Addison-Wesley)



Twitter: @oreillymedia
facebook.com/oreilly

SEGUNDA EDIÇÃO

SQLAlquimia Essencial

Jason Myers e Rick Copeland

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

SQLAlchemy Essencial

por Jason Myers e Rick Copeland

Copyright © 2016 Jason Myers e Rick Copeland. Todos os direitos reservados.

Impresso nos Estados Unidos da América.

Publicado por O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Os livros da O'Reilly podem ser adquiridos para uso educacional, comercial ou promocional de vendas. Edições online também estão disponíveis para a maioria dos títulos (<http://safaribooksonline.com>). Para mais informações, entre em contato com nosso departamento de vendas corporativo/institucional: 800-998-9938 ou corporate@oreilly.com.

Editores: Dawn Schanafelt e Meghan Blanchette

Indexador: Angela Howard

Editor de Produção: Shiny Kalapurakkal

Designer de Interiores: David Futato

Editor de texto: Charles Roumeliotis

Designer de capa: Karen Montgomery

Revisor: Jasmine Kwityn

Ilustrador: Rebecca Demarest

Junho de 2008: Primeira edição

Dezembro de 2015: Segunda edição

Histórico de revisões para a segunda edição

20/11/2015: Primeira versão

Consulte <http://oreilly.com/catalog/errata.csp?isbn=9781491916469> para detalhes do lançamento.

O logotipo O'Reilly é uma marca registrada da O'Reilly Media, Inc. Essential SQLAlchemy, a imagem da capa e a imagem comercial relacionada são marcas registradas da O'Reilly Media, Inc.

Embora o editor e os autores tenham feito esforços de boa fé para garantir que as informações e instruções contidas neste trabalho sejam precisas, o editor e os autores se isentam de qualquer responsabilidade por erros ou omissões, incluindo, sem limitação, responsabilidade por danos resultantes do uso ou confiança neste trabalho. O uso das informações e instruções contidas neste trabalho é por sua conta e risco. Se qualquer amostra de código ou outra tecnologia que este trabalho contém ou descreve está sujeita a licenças de código aberto ou direitos de propriedade intelectual de terceiros, é sua responsabilidade garantir que seu uso esteja em conformidade com tais licenças e/ou direitos.

978-1-491-91646-9

[LSI]

Índice

Prefácio você está vindo

Introdução ao SQLAlchemy xiii

Parte I. Núcleo SQLAlchemy

1. Esquema e Tipos	1
Tipos	1
Metadados	3
Tabelas	4
Colunas	5
Chaves e restrições	6
Índices	7
Relações e restrições de chave estrangeira	7
Persistindo as tabelas	9
2. Trabalhando com dados via SQLAlchemy Core	13
Inserindo dados	13
Consultando dados	17
ResultProxy	18
Controlando as colunas na consulta	20
Encomenda	20
Limitando	21
Funções e rótulos SQL integrados	22
Filtrando	24
ClauseElements	25
Operadores	26

Operadores booleanos	27
Conjunções	27
Atualizando dados	28
Excluindo dados	29
Associações	
Apelido	31
Agrupamento	32
Encadeamento	33
Consultas brutas	34
	35
3. Exceções e Transações.	37
Exceções	37
Erro de atributo	38
Erro de integridade	40
Manipulando Erros	41
Transações	43
4. Testes.	51
Testando com um banco de dados de teste	51
Usando simulações	58
5. Reflexão.	63
Refletindo Tabelas Individuais	63
Refletindo um banco de dados inteiro	66
Criação de consultas com objetos refletidos	66

Parte II. SQLAlchemy ORM

6. Definindo Schema com SQLAlchemy ORM.	71
Definindo Tabelas por meio de Classes	71
ORM Chaves, Restrições e Relações de	73
Índices que Persistem no Esquema	74
	75
7. Trabalhando com dados via SQLAlchemy ORM.	77
A sessão	77
Inserindo dados	80
Consultando dados	83
Controlando as colunas na consulta	86
Encomenda	86
Limitando	87

Funções e rótulos SQL integrados	88
Filtrando	89
Operadores	90
Operadores booleanos	92
Conjunções	92
Atualizando dados	93
Excluindo dados	94
Associações	97
Agrupamento	98
Encadeamento	99
Consultas brutas	101
8. Entendendo a Sessão e as Exceções.....	103
Os Estados da Sessão da Sessão	105
SQLAlchemy	105
Exceções	108
Exceção MultipleResultsFound	108
Erro de Instância Desanexada	110
Transações	112
9. Teste com SQLAlchemy ORM.....	121
Testando com um banco de dados de teste	121
Usando simulações	130
10. Reflexão com SQLAlchemy ORM e Automap.....	133
Refletindo um banco de dados com relacionamentos	133
refletidos do Automap	135
<hr/>	
Parte III. Alambique	
11. Introdução ao Alambique.....	139
Criando o ambiente de migração Configurando o	139
ambiente de migração	140
12. Construindo Migrações.....	143
Gerando uma migração de base vazia Gerando	143
automaticamente uma migração Criando uma	145
migração manualmente	149
13. Alambique de controle.....	151
Determinando o nível de migração de um banco de dados	151

Fazendo downgrade de migrações	152
Marcando o nível de migração do banco de dados	153
Gerando SQL	154
14. Livro de receitas.....	157
Hybrid Attributes	157
Association Proxy	160
Integrando SQLAlchemy com Flask	166
SQLAcodegen	169
15. Para onde ir a partir daqui.....	175
Índice.....	177

Prefácio

Estamos cercados por dados em todos os lugares, e sua capacidade de armazenar, atualizar e relatar esses dados é fundamental para todos os aplicativos que você cria. Esteja você desenvolvendo para a Web, desktop ou outros aplicativos, você precisa de acesso rápido e seguro aos dados.

Bancos de dados relacionais ainda são um dos lugares mais comuns para colocar esses dados.

O SQL é uma linguagem poderosa para consultar e manipular dados em um banco de dados, mas às vezes é difícil integrá-lo ao restante de seu aplicativo. Você pode ter usado a manipulação de strings para gerar consultas a serem executadas em uma interface ODBC ou usado uma API de banco de dados como programador Python. Embora essas possam ser maneiras eficazes de lidar com dados, elas podem dificultar muito a segurança e as alterações no banco de dados.

Este livro é sobre uma biblioteca Python muito poderosa e flexível chamada SQLAlchemy que preenche a lacuna entre bancos de dados relacionais e programação tradicional.

Embora o SQLAlchemy permita que você “desça” no SQL bruto para executar suas consultas, ele incentiva o pensamento de nível superior por meio de uma abordagem mais “Pythonic” e amigável para consultas e atualizações de banco de dados. Ele fornece as ferramentas que permitem mapear as classes e objetos de seu aplicativo para tabelas de banco de dados uma vez e depois “esquecê-lo” ou retornar ao seu modelo várias vezes para ajustar o desempenho.

SQLAlchemy é poderoso e flexível, mas também pode ser um pouco assustador.

Os tutoriais do SQLAlchemy expõem apenas uma fração do que está disponível nesta excelente biblioteca e, embora a documentação on-line seja extensa, geralmente é melhor como referência do que como forma de aprender a biblioteca inicialmente. Este livro pretende ser uma ferramenta de aprendizado e uma referência útil para quando você estiver no “modo de implementação” e precisar de uma resposta rápida.

Este livro se concentra na versão 1.0 do SQLAlchemy; no entanto, muito do que abordaremos está disponível para muitas das versões anteriores. Certamente funciona a partir de 0,8 em diante com pequenos ajustes, e a maioria a partir de 0,5.

Este livro foi escrito em três partes principais: SQLAlchemy Core, SQLAlchemy ORM e uma seção Alembic. As duas primeiras partes destinam-se a espelhar uma à outra como

de perto possível. Tivemos o cuidado de executar os mesmos exemplos em cada parte para que você possa comparar e contrastar as duas principais maneiras de usar o SQLAlchemy. O livro também foi escrito para que você possa ler as partes SQLAlchemy Core e ORM ou apenas uma que atenda às suas necessidades no momento.

Para quem é este livro

Este livro é destinado àqueles que desejam aprender mais sobre como usar bancos de dados relacionais em seus programas Python, ou já ouviram falar sobre SQLAlchemy e desejam mais informações sobre ele. Para tirar o máximo proveito deste livro, o leitor deve ter habilidades intermediárias em Python e pelo menos uma exposição moderada a bancos de dados SQL. Embora tenhamos trabalhado arduamente para tornar o material acessível, se você está apenas começando com o Python, recomendamos a leitura de [Introdução ao Python](#) por Bill Lubanovic ou assistindo aos vídeos “[Introduction to Python](#)” de Jessica McKellar, pois ambos são recursos fantásticos. Se você é novo em SQL e bancos de dados, confira [Aprendendo SQL](#) por Alan Beaulieu. Estes preencherão quaisquer lacunas que estiverem faltando enquanto você trabalha neste livro.

Como usar os exemplos

A maioria dos exemplos neste livro foi criada para ser executada em um loop read-eval-print (REPL). Você pode usar o Python REPL integrado digitando `python` no prompt de comando. Os exemplos também funcionam bem em um notebook iPython. Existem algumas partes do livro, como o [Capítulo 4](#), que orientarão a criar e usar arquivos em vez de um REPL. O código de exemplo fornecido é fornecido em notebooks IPython para a maioria dos exemplos, e arquivos Python para os capítulos que especificam para usá-los. Você pode aprender mais sobre o IPython em seu [site](#).

Suposições que este livro faz

Este livro pressupõe conhecimento básico sobre sintaxe e semântica do Python, particularmente as versões 2.7 e posteriores. Em particular, o leitor deve estar familiarizado com iteração e trabalhar com objetos em Python, pois eles são usados com frequência ao longo do livro. A segunda parte do livro trata extensivamente da programação orientada a objetos e do SQLAlchemy ORM. O leitor também deve conhecer a sintaxe básica de SQL e a teoria relacional, pois este livro pressupõe familiaridade com os conceitos SQL de definição de esquema e tabelas juntamente com a criação de instruções SELECT, INSERT, UPDATE e DELETE .

Convenções utilizadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

Indica novos termos, URLs, endereços de e-mail, nomes de arquivo e extensões de arquivo.

Largura constante

Usado para listagens de programas, bem como dentro de parágrafos para se referir a elementos de programa, como nomes de variáveis ou funções, bancos de dados, tipos de dados, variáveis de ambiente, instruções e palavras-chave.

Largura constante negrito

Mostra comandos ou outro texto que deve ser digitado literalmente pelo usuário.

Largura constante em itálico

Mostra o texto que deve ser substituído por valores fornecidos pelo usuário ou por valores determinados pelo contexto.



Este elemento significa uma dica ou sugestão.



Este elemento significa uma nota geral.



Este elemento indica um aviso ou cuidado.

Usando exemplos de código

Material suplementar (exemplos de código, exercícios, etc.) está disponível para download no link: <https://github.com/oreillymedia/essential-sqlalchemy-2e>.

Este livro está aqui para ajudá-lo a fazer o seu trabalho. Em geral, se um código de exemplo for oferecido com este livro, você poderá usá-lo em seus programas e documentação. Você não precisa entrar em contato conosco para obter permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que usa vários pedaços de código deste

livro não requer permissão. Vender ou distribuir um CD-ROM de exemplos dos livros da O'Reilly requer permissão. Responder a uma pergunta citando este livro e citando um código de exemplo não requer permissão. Incorporar uma quantidade significativa de código de exemplo deste livro na documentação do seu produto requer permissão.

Agradecemos, mas não exigimos, atribuição. Uma atribuição geralmente inclui o título, autor, editora e ISBN. Por exemplo: "Essential SQLAlchemy, Segunda Edição, por Jason Myers e Rick Copeland (O'Reilly). Copyright 2016 Jason Myers e Rick Copeland, 978-1-4919-1646-9."

Se você achar que o uso de exemplos de código está fora do uso justo ou da permissão dada acima, sinta-se à vontade para nos contatar em permissions@oreilly.com.

Livros on-line do Safari®



Livros on-line do Safari é uma biblioteca digital sob demanda que oferece conteúdo especializado em formato de livro e vídeo dos principais autores do mundo em tecnologia e negócios.

Profissionais de tecnologia, desenvolvedores de software, web designers e profissionais de negócios e criativos usam o Safari Books Online como seu principal recurso para pesquisa, solução de problemas, aprendizado e treinamento de certificação.

O Safari Books Online oferece uma variedade de [planos e preços](#) para [empresa, governo, Educação, e indivíduos](#).

Os membros têm acesso a milhares de livros, vídeos de treinamento e manuscritos de pré-publicação em um banco de dados totalmente pesquisável de editores como O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology e centenas [mais](#). Para obter mais informações sobre o Safari Books Online, visite-nos [online](#).

Como entrar em contato conosco

Envie comentários e perguntas sobre este livro à editora:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472 800-998-9938
(nos Estados Unidos ou Canadá) 707-829-0515
(internacional ou local) 707-829-0104 (fax)

Temos uma página web para este livro, onde listamos erratas, exemplos e qualquer informação adicional. Você pode acessar esta página no seguinte link: <http://oreil.ly/1lwEdiw>.

Para comentar ou fazer perguntas técnicas sobre este livro, envie um e-mail para bookquestions@oreilly.com.

Para obter mais informações sobre nossos livros, cursos, conferências e notícias, consulte nosso site em <http://www.oreilly.com>.

Encontre-nos no Facebook: <http://facebook.com/>

[oreilly](#) Siga-nos no Twitter: <http://twitter.com/oreillymedia>

Assista-nos no YouTube: <http://www.youtube.com/oreillymedia>

Agradecimentos

Muito obrigado a Patrick Altman, Eric Floehr e Alex Grönholm por seus comentários críticos de pré-publicação. Sem eles, este livro sem dúvida teria muitos problemas técnicos e seria muito mais difícil de ler.

Meu apreço vai para Mike Bayer, cuja recomendação levou a escrever este livro em primeiro lugar. Sou grata a Meghan Blanchette e Dawn Schana-sentiu por me empurrarem para completar o livro, me tornando uma escritora melhor e me aturando. Também gostaria de agradecer a Brian Dailey por ler alguns dos cortes mais difíceis do livro, fornecer ótimos comentários e rir comigo sobre isso.

Quero agradecer à comunidade de desenvolvimento de Nashville por me apoiar, especialmente Cal Evans, Jacques Woodcock, Luke Stokes e William Golden.

Agradeço ao meu empregador, Cisco Systems, por me conceder tempo e apoio para concluir o livro. Obrigado também ao Justin da Mechanical Keyboards por me manter abastecido com tudo que eu precisava para manter meus dedos digitando.

Mais importante, quero agradecer à minha esposa por me aturar lendo em voz alta para mim mesmo, desaparecendo para escrever e sendo minha fonte constante de apoio e esperança. Eu te amo, Denise.

Introdução ao SQLAlchemy

SQLAlchemy é uma biblioteca usada para interagir com uma ampla variedade de bancos de dados. Ele permite que você crie modelos de dados e consultas de uma maneira que pareça com classes e instruções normais do Python. Criado por Mike Bayer em 2005, o SQLAlchemy é usado por muitas empresas, grandes e pequenas, e é considerado por muitos como a maneira de fato de trabalhar com bancos de dados relacionais em Python.

Ele pode ser usado para se conectar aos bancos de dados mais comuns, como Postgres, MySQL, SQLite, Oracle e muitos outros. Ele também fornece uma maneira de adicionar suporte para outros bancos de dados relacionais. O Amazon Redshift, que usa um dialeto personalizado do PostgreSQL, é um ótimo exemplo de suporte a banco de dados adicionado pela comunidade.

Neste capítulo, exploraremos por que precisamos do SQLAlchemy, aprenderemos sobre seus dois modos principais e nos conectaremos a um banco de dados.

Por que usar SQLAlchemy?

A principal razão para usar SQLAlchemy é abstrair seu código do banco de dados subjacente e suas peculiaridades SQL associadas. SQLAlchemy aproveita poderosas instruções e tipos comuns para garantir que suas instruções SQL sejam elaboradas de forma eficiente e adequada para cada tipo de banco de dados e fornecedor sem que você precise pensar sobre isso.

Isso facilita a migração da lógica do Oracle para o PostgreSQL ou de um banco de dados de aplicativo para um data warehouse. Isso também ajuda a garantir que a entrada do banco de dados seja higienizada e escape adequadamente antes de ser enviada ao banco de dados. Isso evita problemas comuns, como ataques de injeção de SQL.

SQLAlchemy também oferece muita flexibilidade fornecendo dois modos principais de uso: SQL Expression Language (comumente referido como Core) e ORM. Esses modos podem ser usados separadamente ou em conjunto, dependendo de sua preferência e das necessidades de sua aplicação.

SQLAlchemy Core e a Linguagem de Expressão SQL A

Linguagem de Expressão SQL é uma forma Pythonic de representar declarações e expressões SQL comuns, e é apenas uma leve abstração da linguagem SQL típica.

Ele é focado no esquema real do banco de dados; no entanto, ele é padronizado de forma a fornecer uma linguagem consistente em um grande número de bancos de dados de back-end.

A SQL Expression Language também atua como base para o SQLAlchemy ORM.

ORM

O SQLAlchemy ORM é semelhante a muitos outros mapeadores relacionais de objeto (ORMs) que você pode ter encontrado em outras linguagens. Ele é focado no modelo de domínio do aplicativo e aproveita o padrão Unit of Work para manter o estado do objeto. Ele também fornece uma abstração de alto nível sobre a SQL Expression Language que permite ao usuário trabalhar de uma maneira mais idiomática. Você pode misturar e combinar o uso do ORM com a SQL Expression Language para criar aplicativos muito poderosos.

O ORM aproveita um sistema declarativo que é semelhante aos sistemas de registro ativo usados por muitos outros ORMs, como o encontrado em Ruby on Rails.

Embora o ORM seja extremamente útil, você deve ter em mente que há uma diferença entre a maneira como as classes podem ser relacionadas e como os relacionamentos de banco de dados subjacentes funcionam. Exploraremos mais detalhadamente as maneiras pelas quais isso pode afetar sua implementação no [Capítulo 6](#).

Escolhendo entre SQLAlchemy Core e ORM

Antes de começar a criar aplicativos com SQLAlchemy, você precisará decidir se usará principalmente o ORM ou o Core. A escolha de usar SQLAlchemy Core ou ORM como a camada de acesso a dados dominante para um aplicativo geralmente se resume a alguns fatores e preferências pessoais.

Os dois modos usam uma sintaxe ligeiramente diferente, mas a maior diferença entre Core e ORM é a visualização dos dados como esquema ou objetos de negócios. SQLAlchemy Core tem uma visão centrada no esquema, que como o SQL tradicional é focada em tabelas, chaves e estruturas de índice. O SQLAlchemy Core realmente brilha em data warehouse, relatórios, análises e outros cenários em que é útil poder controlar rigidamente a consulta ou operar em dados não modelados. O forte conjunto de conexões de banco de dados e as otimizações do conjunto de resultados são perfeitamente adequados para lidar com grandes quantidades de dados, mesmo em vários bancos de dados.

No entanto, se você pretende se concentrar mais em um design orientado a domínio, o ORM encapsulará grande parte do esquema e estrutura subjacentes em metadados e objetos de negócios. Esse encapsulamento pode tornar mais fácil fazer com que as interações do banco de dados pareçam mais com o código Python normal. As aplicações mais comuns podem ser modeladas desta forma. Também pode ser uma maneira altamente eficaz de injetar design orientado a domínio

em um aplicativo legado ou um com instruções SQL brutas espalhadas por toda parte.

Os microsserviços também se beneficiam da abstração do banco de dados subjacente, permitindo que o desenvolvedor se concentre apenas no processo que está sendo implementado.

No entanto, como o ORM é construído em cima do SQLAlchemy Core, você pode usar sua capacidade de trabalhar com serviços como Oracle Data Warehousing e Amazon Redshift da mesma maneira que interopera com o MySQL. Isso o torna um complemento maravilhoso para o ORM quando você precisa combinar objetos de negócios e dados armazenados.

Aqui está uma lista de verificação rápida para ajudá-lo a decidir qual opção é melhor para você:

- Se você estiver trabalhando com uma estrutura que já possui um ORM integrado, mas deseja para adicionar relatórios mais poderosos, use Core.
- Se você deseja visualizar seus dados em uma visualização mais centrada no esquema (como usado no SQL), usar Core.
- Se você tiver dados para os quais os objetos de negócios não são necessários, use o Core.
- Se você visualizar seus dados como objetos de negócios, use ORM.
- Se você estiver construindo um protótipo rápido, use ORM.
- Se você tem uma combinação de necessidades que realmente pode alavancar tanto o negócio objetos e outros dados não relacionados ao domínio do problema, use ambos!

Agora que você sabe como o SQLAlchemy está estruturado e a diferença entre Core e ORM, estamos prontos para instalar e começar a usar o SQLAlchemy para conectar a um banco de dados.

Instalando o SQLAlchemy e conectando-se a um banco de dados

SQLAlchemy pode ser usado com Python 2.6, Python 3.3 e Pypy 2.1 ou superior. Eu recomendo usar pip para realizar a instalação com o comando `pip install sqlalchemy`. Vale a pena notar que também pode ser instalado com `easy_install` e `distutils`; no entanto, `pip` é o método mais direto. Durante a instalação, o SQLAlchemy tentará construir algumas extensões C, que são aproveitadas para tornar o trabalho com conjuntos de resultados mais rápido e mais eficiente em termos de memória. Se você precisar desabilitar essas extensões devido à falta de um compilador no sistema em que está instalando, você pode usar `--global-option=--without-cextensions`. Observe que usar SQLAlchemy sem extensões C afetará negativamente o desempenho e você deve testar seu código em um sistema com as extensões C antes de otimizá-lo.

Instalando drivers de banco de

dados Por padrão, SQLAlchemy oferecerá suporte a SQLite3 sem drivers adicionais; entretanto, um driver de banco de dados adicional que usa a especificação padrão Python DBAPI (PEP-249) é necessário para conectar-se a outros bancos de dados. Essas DBAPIs fornecem a base para o dialeto que cada servidor de banco de dados fala, e geralmente habilitam os recursos exclusivos vistos em diferentes servidores e versões de banco de dados. Embora existam vários DBAPIs disponíveis para muitos dos bancos de dados, as instruções a seguir se concentram nos mais comuns:

PostgreSQL

[Psychog2](#) fornece amplo suporte para versões e recursos do PostgreSQL e pode ser instalado com pip install psycopg2.

MySQL

PyMySQL é minha biblioteca Python preferida para se conectar a um servidor de banco de dados MySQL. Ele pode ser instalado com um pip install pymysql. O suporte ao MySQL no SQLAlchemy requer o MySQL versão 4.1 e superior devido à forma como as senhas funcionavam antes dessa versão. Além disso, se um determinado tipo de instrução estiver disponível apenas em uma determinada versão do MySQL, SQLAlchemy não fornecerá um método para usar essas instruções em versões do MySQL em que a instrução não estiver disponível. É importante revisar a documentação do MySQL se um determinado componente ou função no SQLAlchemy não funcionar em seu ambiente.

Outros

SQLAlchemy também podem ser usados em conjunto com Drizzle, Firebird, Oracle, Sybase e Microsoft SQL Server. A comunidade também forneceu dialetos externos para muitos outros bancos de dados como IBM DB2, Informix, Amazon Redshift, EXASolution, SAP SQL Anywhere, Monet e muitos outros. A criação de um dialeto adicional é bem suportada pelo SQLAlchemy, e o [Capítulo 7](#) examinará o processo de fazer exatamente isso.

Agora que temos o SQLAlchemy e uma DBAPI instalados, vamos construir um mecanismo para se conectar a um banco de dados.

Conectando a um banco de dados

Para conectar a um banco de dados, precisamos criar um mecanismo SQLAlchemy. O mecanismo SQLAlchemy cria uma interface comum para o banco de dados para executar instruções SQL. Ele faz isso agrupando um conjunto de conexões de banco de dados e um dialeto de forma que eles possam trabalhar juntos para fornecer acesso uniforme ao banco de dados de back-end. Isso permite que nosso código Python não se preocupe com as diferenças entre bancos de dados ou DBAPIs.

SQLAlchemy fornece uma função para criar um mecanismo para nós, dada uma string de conexão e, opcionalmente, alguns argumentos de palavras-chave adicionais. Uma string de conexão é uma string especialmente formatada que fornece:

- Tipo de banco de dados (Postgres, MySQL, etc.) •
- Dialetos, exceto o padrão para o tipo de banco de dados (Psycopg2, PyMySQL, etc.) • Detalhes de autenticação opcionais (nome de usuário e senha) • Localização do banco de dados (arquivo ou nome de host do servidor de banco de dados) • Porta do servidor de banco de dados opcional •
- Nome do banco de dados opcional

As strings de conexão do banco de dados SQLite nos fazem representar um arquivo específico ou um local de armazenamento. [O exemplo P-1](#) define um arquivo de banco de dados SQLite chamado cookies.db armazenado no diretório atual por meio de um caminho relativo na segunda linha, um banco de dados na memória na terceira linha e um caminho completo para o arquivo na quarta (Unix) e quinta (Windows).

No Windows, a cadeia de conexão se pareceria com engine4; os \\ são necessários para o escape de string adequado, a menos que você use uma string bruta (r").

[Exemplo P-1.](#) Criando um mecanismo para um banco de dados SQLite

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///cookies.db') engine2 =
create_engine('sqlite:///memory:') engine3 =
create_engine('sqlite:///home/cookiemonster/cookies .db') engine4 =
create_engine('sqlite:///c:\\\\Users\\\\cookiemonster\\\\cookies.db')
```



A função `create_engine` retorna uma instância de um mecanismo; no entanto, ele não abre realmente uma conexão até que seja chamada uma ação que exija uma conexão, como uma consulta.

Vamos criar um mecanismo para um banco de dados PostgreSQL local chamado mydb. Começaremos importando a função `create_engine` do pacote base `sqlalchemy`. Em seguida, usaremos essa função para construir uma instância de mecanismo. No [Exemplo P-2](#), você notará que eu uso `postgresql+psycopg2` como os componentes de mecanismo e dialeto da string de conexão, mesmo que usar apenas `postgres` funcione. Isso ocorre porque prefiro ser explícito em vez de implícito, conforme recomendado no [Zen of Python](#).

Exemplo P-2. Criando um mecanismo para um banco de dados PostgreSQL local

```
from sqlalchemy import create_engine
engine = create_engine('postgresql+psycopg2://username:password@localhost:' \
'5432/mydb')
```

Agora vamos ver um banco de dados MySQL em um servidor remoto. Você notará, no [Exemplo P-3](#), que após a string de conexão temos um parâmetro de palavra-chave, pool_recycle, para definir com que frequência reciclar as conexões.

Exemplo P-3. Criando um mecanismo para um banco de dados MySQL remoto

```
from sqlalchemy import create_engine
engine = create_engine('mysql+pymysql://cookiemonster:chocolatechip' \
'@mysql01.monster.internal/cookies', pool_recycle=3600)
```



Por padrão, o MySQL fecha conexões ociosas por mais de oito horas. Para contornar esse problema, use pool_recycle=3600 ao criar um mecanismo, conforme mostrado no [Exemplo P-3](#).

Algumas palavras-chave opcionais para a função create_engine são:

`echo`

Isso registrará as ações processadas pelo mecanismo, como instruções SQL e seus parâmetros. O padrão é falso.

`encoding`

Isso define a codificação de string usada pelo SQLAlchemy. O padrão é utf-8, e a maioria das DBAPIs suporta essa codificação por padrão. Isso não define o tipo de codificação usado pelo próprio banco de dados backend.

`isolamento_level`

Isso instrui o SQLAlchemy a usar um nível de isolamento específico. Por exemplo, PostgreSQL com Psycopg2 tem READ COMMITTED, READ UNCOMMITTED, REPEATABLE READ, SERIALIZABLE e AUTOCOMMIT disponíveis com um padrão de READ COMMITTED. O PyMySQL tem as mesmas opções com um padrão de REPEATABLE READ para bancos de dados InnoDB.



Usar o argumento de palavra-chave de nível_isolamento definirá o nível de isolamento para qualquer DBAPI. Isso funciona da mesma forma que fazê-lo por meio de um par chave-valor na string de conexão em dialetos como Psycopg2 que suportam esse método.

pool_recycle

Isso recicla ou expira as conexões de banco de dados em intervalos regulares. Isso é importante para o MySQL devido aos tempos limite de conexão que mencionamos anteriormente. O padrão é -1, o que significa que não há tempo limite.

Uma vez que tenhamos um motor inicializado, estamos prontos para realmente abrir uma conexão com o banco de dados. Isso é feito chamando o método connect() no mecanismo, conforme mostrado aqui:

```
from sqlalchemy import create_engine
engine = create_engine('mysql+pymysql://cookiemonster:chocolatechip' \
                      '@mysql01.monster.internal/cookies', pool_recycle=3600)
conexão = engine.connect()
```

Agora que temos uma conexão com o banco de dados, podemos começar a usar o SQLAlchemy Core ou o ORM. Na Parte I, começaremos a explorar o SQLAlchemy Core e aprenderemos como definir e consultar seu banco de dados.

PARTE I

SQLAlchemy Core

Agora que podemos nos conectar aos bancos de dados, vamos começar a ver como usar o SQLAlchemy Core para fornecer serviços de banco de dados para nossos aplicativos. SQLAlchemy Core é uma maneira Pythonic de representar elementos de comandos SQL e estruturas de dados chamados SQL Expression Language. SQLAlchemy Core pode ser usado com o Django ou SQLAlchemy ORM, ou pode ser usado como uma solução independente.

CAPÍTULO 1

Esquema e Tipos

A primeira coisa que devemos fazer é definir quais dados nossas tabelas contêm, como esses dados estão inter-relacionados e quaisquer restrições sobre esses dados.

Para fornecer acesso ao banco de dados subjacente, SQLAlchemy precisa de uma representação das tabelas que devem estar presentes no banco de dados. Podemos fazer isso de três maneiras:

- Usando objetos Table definidos pelo usuário
- Usando classes declarativas que representam suas tabelas •

Inferindo-as do banco de dados

Este capítulo se concentra no primeiro deles, pois essa é a abordagem usada com SQLAlchemy Core; abordaremos as outras duas opções em capítulos posteriores, depois de compreendermos os fundamentos. Os objetos Table contêm uma lista de colunas digitadas e seus atributos, que estão associados a um contêiner de metadados comum. Começaremos nossa exploração das definições de esquema examinando os tipos disponíveis para construir tabelas no SQLAlchemy.

Tipos

Existem quatro categorias de tipos que podemos usar dentro do SQLAlchemy:

- Genérico
- Padrão SQL
- Específico do fornecedor
- Definido pelo usuário

SQLAlchemy define um grande número de tipos genéricos que são abstraídos de os tipos reais de SQL suportados por cada banco de dados de back-end. Esses tipos estão todos disponíveis disponíveis no módulo `sqlalchemy.types` e, por conveniência, também estão disponíveis em o módulo `sqlalchemy`. Então, vamos pensar em como esses tipos genéricos são úteis.

O tipo genérico Boolean normalmente usa o tipo SQL BOOLEAN e no Python lado lida em verdadeiro ou falso; no entanto, ele também usa SMALLINT em bancos de dados de back-end que não suporta um tipo BOOLEAN. Graças ao SQLAlchemy, este pequeno detalhe está oculto de você, e você pode confiar que quaisquer consultas ou declarações que você criar funcionarão corretamente em campos desse tipo, independentemente do tipo de banco de dados que está sendo usado. Você irá só tem que lidar com true ou false em seu código Python. Esse tipo de comportamento faz os tipos genéricos muito poderosos e úteis durante transições de banco de dados ou split back sistemas finais onde o data warehouse é um tipo de banco de dados e o transacional é outro. Os tipos genéricos e suas representações de tipo associados em Python e SQL podem ser vistos na [Tabela 1-1](#).

Tabela 1-1. Representações de tipo genérico

SQLAlchemy	Pitão	SQL
<code>BigInteger int</code>		BIGINT
<code>bool</code>	<code>bool</code>	BOOLEAN ou SMALLINT
<code>Encontro</code>	<code>datahora.data</code>	DATA (SQLite: STRING)
<code>Data hora</code>	<code>datetime.datetime DATETIME (SQLite: STRING)</code>	
<code>Enum</code>	<code>str</code>	ENUM ou VARCHAR
<code>Flutuador</code>	<code>flutuante ou decimal</code>	FLUTUANTE ou REAL
<code>inteiro</code>	<code>int</code>	INTEIRO
<code>Intervalo</code>	<code>datetime.timedelta INTERVAL ou DATE da época</code>	
<code>Grande byte binário</code>		BLOB ou BYTES
<code>Numérico</code>	<code>decimal.Decimal</code>	NUMÉRICO ou DECIMAL
<code>Unicode</code>	<code>código único</code>	UNICODE ou VARCHAR
<code>Texto</code>	<code>str</code>	CLOB ou TEXTO
<code>Tempo</code>	<code>datahora.hora</code>	DATA HORA



É importante aprender esses tipos genéricos, pois você precisará usá-los e defini-los regularmente.

Além dos tipos genéricos listados na [Tabela 1-1](#), os tipos padrão SQL e específicos do fornecedor estão disponíveis e são frequentemente usados quando um tipo genérico não funcionará conforme necessário no esquema do banco de dados devido ao seu tipo ou ao tipo específico especificado em um esquema existente. Algumas boas ilustrações disso são os tipos CHAR e NVARCHAR , que se beneficiam do uso do tipo SQL adequado em vez de apenas o tipo genérico. Se estivermos trabalhando com um esquema de banco de dados que foi definido antes de usar SQLAlchemy, gostaríamos de corresponder exatamente aos tipos. É importante ter em mente que o comportamento e a disponibilidade do tipo padrão SQL podem variar de banco de dados para banco de dados. Os tipos padrão SQL estão disponíveis no módulo sqlalchemy.types . Para ajudar a fazer uma distinção entre eles e os tipos genéricos, os tipos padrão estão em letras maiúsculas.

Os tipos específicos do fornecedor são úteis da mesma forma que os tipos padrão SQL; no entanto, eles estão disponíveis apenas em bancos de dados de back-end específicos. Você pode determinar o que está disponível na documentação do dialeto escolhido ou [no site do SQLAlchemy](#). Eles estão disponíveis no módulo sqlalchemy.dialects e existem submódulos para cada dialeto do banco de dados. Novamente, os tipos estão em letras maiúsculas para distinção dos tipos genéricos. Podemos querer aproveitar o poderoso campo JSON do PostgreSQL, o que podemos fazer com a seguinte instrução:

```
de sqlalchemy.dialects.postgresql importar JSON
```

Agora podemos definir campos JSON que podemos usar posteriormente com as muitas funções JSON específicas do PostgreSQL, como array_to_json, em nosso aplicativo.

Você também pode definir tipos personalizados que fazem com que os dados sejam armazenados da maneira de sua escolha. Um exemplo disso pode ser adicionar caracteres no texto armazenado em uma coluna VARCHAR quando colocado no registro do banco de dados e removê-los ao recuperar esse campo do registro. Isso pode ser útil ao trabalhar com dados legados ainda usados por sistemas existentes que executam esse tipo de prefixação que não é útil ou importante em seu novo aplicativo.

Agora que vimos as quatro variações de tipos que podemos usar para construir tabelas, vamos dar uma olhada em como a estrutura do banco de dados é mantida unida por metadados.

Metadados

Os metadados são usados para unir a estrutura do banco de dados para que possam ser acessados rapidamente dentro do SQLAlchemy. Muitas vezes é útil pensar em metadados como um tipo de catálogo de objetos Table com informações opcionais sobre o mecanismo e a conexão. Essas tabelas podem ser acessadas por meio de um dicionário, MetaData.tables. As operações de leitura são thread-safe;

no entanto, a construção da tabela não é completamente thread-safe. Os metadados precisam ser importados e inicializados antes que os objetos possam ser vinculados a eles. Vamos inicializar uma instância dos objetos MetaData que podemos usar no restante dos exemplos deste capítulo para manter nosso catálogo de informações:

```
de sqlalchemy importar metadados
MetaData = MetaData()
```

Assim que tivermos uma maneira de manter a estrutura do banco de dados, estaremos prontos para começar a definir as tabelas.

Tabelas

Objetos de tabela são inicializados no SQLAlchemy Core em um objeto MetaData fornecido chamando o construtor Table com o nome da tabela e metadados; quaisquer argumentos adicionais são assumidos como objetos de coluna. Há também alguns argumentos de palavras-chave adicionais que habilitam recursos que discutiremos mais adiante. Objetos de coluna representam cada campo na tabela. As colunas são construídas chamando Column com um nome, tipo e, em seguida, argumentos que representam quaisquer construções e restrições SQL adicionais. Para o restante deste capítulo, vamos construir um conjunto de tabelas que usaremos na Parte I. No [Exemplo 1-1](#), criamos uma tabela que pode ser usada para armazenar o inventário de cookies para nossa entrega de cookies online serviço.

Exemplo 1-1. Instanciando objetos e colunas de tabela

```
from sqlalchemy import Tabela, Coluna, Inteiro, Numérico, String, ForeignKey
```

```
cookies = Tabela('cookies', metadados,
    Column('cookie_id', Integer(), primary_key=True), ❶
    Column('cookie_name', String(50), index=True), ❷
    Column('cookie_recipe_url', String(255)),
    Column('cookie_sku', String(55)),
    Column('quantidade', Integer()),
    Column('custo_unidade', Numérico(12, 2)) ❸
)
```

- ❶ Observe a maneira como marcamos essa coluna como a chave primária da tabela. Mais sobre isso em um segundo.
- ❷ Estamos criando um índice de nomes de cookies para acelerar as consultas nesta coluna.
- ❸ Esta é uma coluna que recebe vários argumentos, comprimento e precisão, como 11.2, o que nos daria números de até 11 dígitos com duas casas decimais.

Antes de irmos muito longe nas tabelas, precisamos entender seus blocos de construção fundamentais: colunas.

Colunas

As colunas definem os campos que existem em nossas tabelas e fornecem o principal meio pelo qual definimos outras restrições por meio de seus argumentos de palavras-chave. Diferentes tipos de colunas têm diferentes argumentos primários. Por exemplo, colunas do tipo String têm comprimento como argumento principal, enquanto números com um componente fracionário terão precisão e comprimento. A maioria dos outros tipos não tem argumentos primários.



Às vezes, você verá exemplos que mostram apenas colunas String sem comprimento, que é o argumento principal. Esse comportamento não é universalmente suportado—por exemplo, MySQL e vários outros backends de banco de dados não permitem isso.

As colunas também podem ter alguns argumentos de palavras-chave adicionais que ajudam a moldar ainda mais seu comportamento. Podemos marcar as colunas conforme necessário e/ou forçá-las a serem únicas. Também podemos definir valores iniciais padrão e alterar valores quando o registro for atualizado. Um caso de uso comum para isso são os campos que indicam quando um registro foi criado ou atualizado para fins de log ou auditoria. Vamos dar uma olhada nesses argumentos de palavras-chave em ação no [Exemplo 1-2](#).

[Exemplo 1-2.](#) Outra Tabela com mais opções de Colunas

```
de datetime import datetime
de sqlalchemy import DateTime

users = Table('users', metadata,
    Column('user_id', Integer(), primary_key=True),
    Column('username', String(15), nullable=False, unique=True),
    Column('email_address', String(255), nullable=False), Column('phone',
    String(20), nullable=False), Column('password', String(25), nullable=False),
    Column('created_on', DateTime(), default=datetime.now),
    Column('updated_on', DateTime(), default=datetime.now,          ①
    onupdate=datetime.now)                                         ②
)
                                         ③
```

- ➊ Aqui estamos tornando esta coluna obrigatória (nullable=False) e também exigindo um valor único.
- ➋ O padrão define esta coluna para a hora atual se uma data não for especificada.
- ➌ Usar onupdate aqui redefinirá esta coluna para a hora atual toda vez que qualquer parte do registro for atualizada.



Você notará que definimos default e onupdate para a data que pode ser chamada time.now em vez da própria chamada de função, datetime.now(). Se tivéssemos usado a própria chamada de função, ela teria definido o padrão para o momento em que a tabela foi instanciada pela primeira vez. Ao usar o callable, obtemos a hora em que cada registro individual é instanciado e atualizado.

Temos usado argumentos de palavra-chave de coluna para definir construções e restrições de tabela; no entanto, também é possível declará-los fora de um objeto Column . Isso é crítico quando você está trabalhando com um banco de dados existente, pois você deve informar ao SQLAlchemy o esquema, as construções e as restrições presentes no banco de dados. Por exemplo, se você tiver um índice existente no banco de dados que não corresponde ao esquema de nomenclatura de índice padrão usado pelo SQLAlchemy, você deverá definir esse índice manualmente. As duas seções a seguir mostram como fazer exatamente isso.



Todos os comandos em “[Chaves e restrições](#)” na [página 6](#) e “[Índices](#)” na [página 7](#) são incluídos como parte do construtor Table ou adicionados à tabela por meio de métodos especiais. Eles serão persistidos ou anexados aos metadados como instruções independentes.

Chaves e restrições

Chaves e restrições são usadas como forma de garantir que nossos dados atendam a determinados requisitos antes de serem armazenados no banco de dados. Os objetos que representam chaves e restrições podem ser encontrados dentro do módulo SQLAlchemy base, e três dos mais comuns podem ser importados como mostrado aqui:

```
from sqlalchemy import PrimaryKeyConstraint, UniqueConstraint, CheckConstraint
```

O tipo de chave mais comum é a chave primária, que é usada como identificador exclusivo para cada registro em uma tabela de banco de dados e é usada para garantir um relacionamento adequado entre duas partes de dados relacionados em tabelas diferentes. Como você viu anteriormente no [Exemplo 1-1](#) e no [Exemplo 1-2](#), uma coluna pode se tornar uma chave primária simplesmente usando o argumento de palavra-chave primary_key . Você também pode definir chaves primárias compostas atribuindo a configuração primary_key a True em várias colunas. A chave será então essencialmente tratada como uma tupla na qual as colunas marcadas como chave estarão presentes na ordem em que foram definidas na tabela. As chaves primárias também podem ser definidas após as colunas no construtor de tabela, conforme mostrado no snippet a seguir. Você pode adicionar várias colunas separadas por vírgulas para criar uma chave composta. Se quiséssemos definir explicitamente a chave como mostrado no [Exemplo 1-2](#), ficaria assim:

```
PrimaryKeyConstraint('user_id', name='user_pk')
```

Outra restrição comum é a restrição exclusiva, usada para garantir que dois valores não sejam duplicados em um determinado campo. Para nosso serviço de entrega de cookies on-line, por exemplo, gostaríamos de garantir que cada cliente tivesse um nome de usuário exclusivo para registrar

em nossos sistemas. Também podemos atribuir restrições exclusivas às colunas, conforme mostrado anteriormente na coluna nome de usuário, ou podemos defini-las manualmente, conforme mostrado aqui:

```
UniqueConstraint('username', name='uix_username')
```

Não é mostrado no [Exemplo 1-2](#) o tipo de restrição de verificação. Esse tipo de restrição é usado para garantir que os dados fornecidos para uma coluna correspondam a um conjunto de critérios definidos pelo usuário. No exemplo a seguir, estamos garantindo que unit_cost nunca seja inferior a 0,00 porque cada cookie custa algo para ser feito (lembre-se de Economics 101: TINSTAAFC—ou seja, não existe um cookie grátis!):

```
CheckConstraint('unit_cost >= 0.00', name='unit_cost_positive')
```

Além de chaves e restrições, também podemos querer tornar as pesquisas em determinados campos mais eficientes. É aí que entram os índices.

Índices

Índices são usados para acelerar pesquisas de valores de campo e, no [Exemplo 1-1](#), criamos um índice na coluna cookie_name porque sabemos que pesquisaremos com frequência. Quando os índices são criados como mostrado nesse exemplo, você terá um índice chamado ix_cookies_cookie_name. Também podemos definir um índice usando um tipo de construção explícito. Várias colunas podem ser designadas separando-as por uma vírgula. Você também pode adicionar um argumento de palavra-chave de unique=True para exigir que o índice seja exclusivo também. Ao criar índices explicitamente, eles são passados para o construtor Table após as colunas. Para imitar o índice criado no [Exemplo 1-1](#), poderíamos fazê-lo explicitamente como mostrado aqui:

```
do índice de importação sqlalchemy  
Index('ix_cookies_cookie_name', 'cookie_name')
```

Também podemos criar índices funcionais que variam um pouco de acordo com o banco de dados de backend que está sendo usado. Isso permite criar um índice para situações em que você geralmente precisa consultar com base em algum contexto incomum. Por exemplo, e se quisermos selecionar por cookie SKU e nomear como um item associado, como SKU0001 Chocolate Chip? Poderíamos definir um índice como este para otimizar essa pesquisa:

```
Index('ix_test', mytable.c.cookie_sku, mytable.c.cookie_name))
```

Agora é hora de mergulhar na parte mais importante dos bancos de dados relacionais: relacionamentos de tabelas e como defini-los.

Relacionamentos e ForeignKeyConstraints

Agora que temos

uma tabela com colunas com todas as restrições e índices corretos, vamos ver como criamos relacionamentos entre tabelas. Precisamos de uma maneira de rastrear pedidos, incluindo itens de linha que representam cada cookie e quantidade pedida. Para ajudar a visualizar como essas tabelas devem ser relacionadas, dê uma olhada na [Figura 1-1](#).

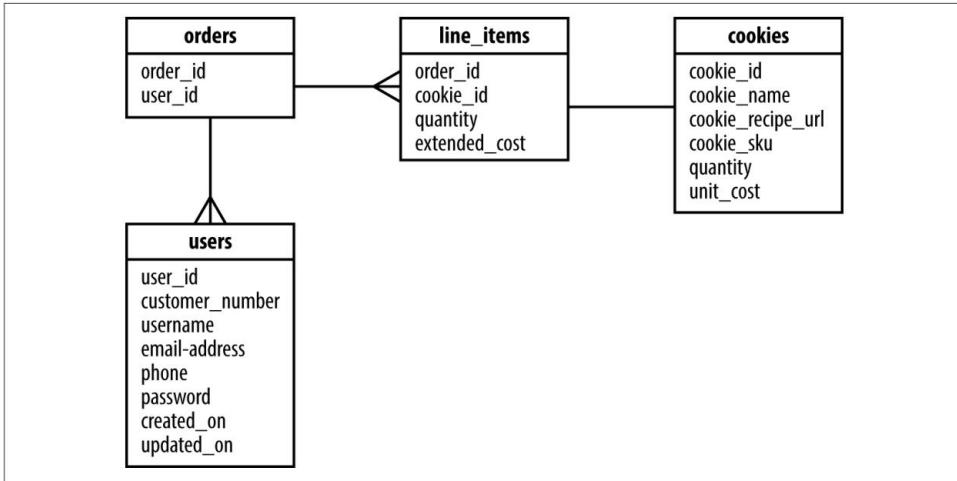


Figura 1-1. Visualização de relacionamento

Uma maneira de implementar um relacionamento é mostrada no [Exemplo 1-3](#) na tabela line_items na coluna order_id ; isso resultará em um ForeignKeyConstraint para definir o relacionamento entre as duas tabelas. Nesse caso, muitos itens de linha podem estar presentes em um único pedido. No entanto, se você se aprofundar na tabela line_items , verá que também temos um relacionamento com a tabela de cookies por meio do cookie_id ForeignKey.

Isso ocorre porque line_items é na verdade uma tabela de associação com alguns dados adicionais entre pedidos e cookies. As tabelas de associação são usadas para habilitar relacionamentos muitos para muitos entre duas outras tabelas. Uma única ForeignKey em uma tabela é tipicamente um sinal de um relacionamento um-para-muitos; no entanto, se houver vários relacionamentos ForeignKey em uma tabela, há uma forte possibilidade de que seja uma tabela de associação.

Exemplo 1-3. Mais tabelas com relacionamentos

```

from sqlalchemy import ForeignKey
orders = Table('orders', metadata,
    Column('order_id', Integer(), primary_key=True),
    Column('user_id', ForeignKey('users.user_id')),
    Column('shipped', Boolean(), default=False)
)

line_items = Table('line_items', metadados,
    Column('line_items_id', Integer(), primary_key=True),
    Column('order_id', ForeignKey('orders.order_id')),
    Column('cookie_id', ForeignKey(' cookies.cookie_id')),
    Column('quantity', Integer()), Column('extended_cost', Numeric(12,
    2))
)
    
```

- ① Observe que usamos uma string em vez de uma referência real à coluna.

Usar strings em vez de uma coluna real nos permite separar as definições de tabela em vários módulos e/ou não ter que nos preocupar com a ordem em que nossas tabelas são carregadas. Isso ocorre porque o SQLAlchemy só realizará a resolução dessa string para um nome de tabela e coluna na primeira vez que for acessado. Se usarmos referências físicas, como `cookies.c.cookie_id`, em nossas definições de ForeignKey ele realizará essa resolução durante a inicialização do módulo e poderá falhar dependendo da ordem em que as tabelas forem carregadas.

Você também pode definir uma ForeignKeyConstraint explicitamente, o que pode ser útil ao tentar corresponder a um esquema de banco de dados existente para que possa ser usado com SQLAlchemy. Isso funciona da mesma maneira que antes, quando criamos chaves, restrições e índices para corresponder a esquemas de nomes e assim por diante. Você precisará importar o ForeignKeyConstraint do módulo sqlalchemy antes de definir um em sua definição de tabela. O código a seguir mostra como criar a ForeignKeyConstraint para o campo `order_id` entre as tabelas `line_items` e `orders` :

```
ForeignKeyConstraint(['order_id'], [orders.order_id])
```

Até este ponto, definimos tabelas de forma que o SQLAlchemy possa entendê-las. Se seu banco de dados já existir e tiver o esquema já construído, você estará pronto para começar a escrever consultas. No entanto, se você precisar criar o esquema completo ou adicionar uma tabela, convém saber como mantê-los no banco de dados para armazenamento permanente.

Persistindo as tabelas

Todas as nossas tabelas e definições de esquema adicionais estão associadas a uma instância de metadados. Persistir o esquema para o banco de dados é simplesmente uma questão de chamar o método `create_all()` em nossa instância de metadados com o mecanismo onde ele deve criar essas tabelas:

```
metadata.create_all(engine)
```

Por padrão, `create_all` não tentará recriar tabelas que já existem no banco de dados e é seguro executar várias vezes. É mais sensato usar uma ferramenta de migração de banco de dados como o Alembic para lidar com quaisquer alterações em tabelas existentes ou esquemas adicionais do que tentar codificar manualmente as alterações diretamente no código do aplicativo (exploraremos isso mais detalhadamente no [Capítulo 11](#)). Agora que persistimos as tabelas no banco de dados, vamos dar uma olhada no [Exemplo 1-4](#), que mostra o código completo das tabelas nas quais trabalhamos neste capítulo.

Exemplo 1-4. Exemplo de código SQLite completo na memória

```
de datetime importação datetime

da importação sqlalchemy (MetaData, Table, Column, Integer, Numeric, String,
    DateTime, ForeignKey, create_engine)
metadados = MetaDados()

cookies = Table('cookies', metadados,
    Column('cookie_id', Integer(), primary_key=True),
    Column('cookie_name', String(50), index=True),
    Column('cookie_recipe_url', String(255)),
    Column('cookie_sku', String(55)),
    Column('quantidade', Integer()),
    Column('custo_unidade', Numérico(12, 2))
)
users = Table('users', metadata,
    Column('user_id', Integer(), primary_key=True),
    Column('customer_number', Integer(), autoincrement=True),
    Column('username', String(15) , nullable=False, unique=True),
    Column('email_address', String(255), nullable=False), Column('phone',
    String(20), nullable=False), Column('password', String( 25), nullable=False),
    Column('created_on', DateTime(), default=datetime.now), Column('updated_on',
    DateTime(), default=datetime.now, onupdate=datetime.now)
)
pedidos = Table('pedidos', metadados,
    Column('order_id', Integer(), primary_key=True),
    Column('user_id', ForeignKey('users.user_id'))
)
line_items = Table('line_items', metadados,
    Column('line_items_id', Integer(), primary_key=True),
    Column('order_id', ForeignKey('orders.order_id')), Column('cookie_id',
    ForeignKey(' cookies.cookie_id')), Column('quantity', Integer()),
    Column('extended_cost', Numeric(12, 2))
)
engine = create_engine('sqlite:///memory:')
metadata.create_all(engine)
```

Neste capítulo, vimos como os metadados são usados como um catálogo pelo SQLAlchemy para armazenar esquemas de tabela junto com outros dados diversos. Também podemos definir uma tabela com várias colunas e restrições. Exploramos os tipos de restrições e como construí-las explicitamente fora de um objeto de coluna para corresponder a um esquema ou esquema de nomenclatura existente. Em seguida, abordamos como definir valores padrão e valores onupdate para auditoria. Finalmente, agora sabemos como persistir ou salvar nosso esquema nos dados.

base para reaproveitamento. A próxima etapa é aprender a trabalhar com dados em nosso esquema por meio da SQL Expression Language.

CAPÍTULO 2

Trabalhando com dados via SQLAlchemy Core

Agora que temos tabelas em nosso banco de dados, vamos começar a trabalhar com os dados dentro dessas tabelas. Veremos como inserir, recuperar e excluir dados e, em seguida, aprenderemos como classificar, agrupar e usar relacionamentos em nossos dados. Usaremos a SQL Expression Language (SEL) fornecida pelo SQLAlchemy Core. Vamos continuar usando as tabelas que criamos no [Capítulo 1](#) para nossos exemplos neste capítulo. Vamos começar aprendendo como inserir dados.

Inserindo dados

Primeiro, construiremos uma instrução de inserção para colocar meu tipo favorito de biscoito (lascas de chocolate) na mesa de biscoitos . Para fazer isso, podemos chamar o método `insert()` na tabela de cookies e, em seguida, usar a instrução `values()` com argumentos de palavra-chave para cada coluna que estamos preenchendo com dados. O [Exemplo 2-1](#) faz exatamente isso.

Exemplo 2-1. Inserção única como método

```
ins = cookies.insert().values(  
    cookie_name="chocolate ",  
    cookie_recipe_url="http://some.aweso.me/cookie/recipe.html",  
    cookie_sku="CC01", quantidade="12", unit_cost="0.50"  
  
) print(str(ins))
```

No [Exemplo 2-1](#), `print(str(ins))` nos mostra a instrução SQL real que será executada:

INSERIR EM cookies
(cookie_name, cookie_recipe_url, cookie_sku, quantidade, unit_cost)

VALORES

```
(:cookie_name, :cookie_recipe_url, :cookie_sku, :quantity, :unit_cost)
```

Nossos valores fornecidos foram substituídos por :column_name nesta instrução SQL, que é como SQLAlchemy representa os parâmetros exibidos por meio da função str() .

Os parâmetros são usados para ajudar a garantir que nossos dados tenham sido escapados adequadamente, o que mitiga problemas de segurança, como ataques de injeção de SQL. Ainda é possível visualizar os parâmetros observando a versão compilada de nossa instrução insert, porque cada backend de banco de dados pode manipular os parâmetros de uma maneira ligeiramente diferente (isso é controlado pelo dialeto). O método compile() no objeto ins retorna um objeto de pilha SQLCom que nos dá acesso aos parâmetros reais que serão enviados com a consulta através do atributo params :

```
ins.compile().params
```

Isso compila a instrução por meio de nosso dialeto, mas não a executa, e acessamos o atributo params dessa instrução.

Isto resulta em:

```
{
    'cookie_name': 'chocolate chip',
    'cookie_recipe_url': 'http://some.aweso.me/cookie/recipe.html', 'cookie_sku':
    'CC01', 'quantity': '12', 'unit_cost': '0,50'

}
```

Agora que temos uma visão completa da instrução insert e entendemos o que será inserido na tabela, podemos usar o método execute() em nossa conexão para enviar a instrução para o banco de dados, que inserirá o registro em a tabela ([Exemplo 2-2](#)).

Exemplo 2-2. Executando a instrução de inserção

```
resultado = connection.execute(ins)
```

Também podemos obter o ID do registro que acabamos de inserir acessando o atributo insert_primary_key :

```
result.inserted_primary_key
[1]
```

Vamos fazer um desvio rápido aqui para discutir o que acontece quando chamamos o método execute() . Quando estamos construindo uma instrução SQL Expression Language como a instrução insert que usamos até agora, na verdade estamos criando uma estrutura semelhante a uma árvore que pode ser rapidamente percorrida de maneira descendente. Quando chamamos o método execute , ele usa a instrução e quaisquer outros parâmetros passados para compilar a instrução com o

compilador de dialeto de banco de dados apropriado. Esse compilador constrói uma instrução SQL parametrizada normal percorrendo essa árvore. Essa instrução é retornada ao método `execute`, que envia a instrução SQL para o banco de dados por meio da conexão na qual o método foi chamado. O servidor de banco de dados então executa a instrução e retorna os resultados da operação.

Além de ter `insert` como um método de instância fora de um objeto `Table`, ele também está disponível como uma função de nível superior para aqueles momentos em que você deseja construir uma instrução “gerativamente” (um passo de cada vez) ou quando a tabela pode não ser inicialmente conhecida. Por exemplo, nossa empresa pode ter duas divisões separadas, cada uma com suas próprias tabelas de estoque separadas. Usar a função `insert` mostrada no [Exemplo 2-3](#) nos permitiria usar uma instrução e apenas trocar as tabelas.

Exemplo 2-3. Inserir função

```
from sqlalchemy import insert
ins = insert(cookies).values(
    cookie_name="chocolate ", ①
    cookie_recipe_url="http://some.aweso.me/cookie/recipe.html",
    cookie_sku="CC01", quantidade="12", unit_cost="0.50"

)
```

- ① Observe que a tabela agora é o argumento para a função de inserção .



Enquanto `insert` funciona igualmente bem como um método de um objeto `Table` e como uma função autônoma mais generativa, eu prefiro a abordagem generativa porque ela espelha mais de perto as instruções SQL que as pessoas estão acostumadas a ver.

O método `execute` do objeto de conexão pode receber mais do que apenas instruções. Também é possível fornecer os valores como argumentos de palavras-chave para o método `execute` após nossa instrução. Quando a instrução é compilada, ela adiciona cada uma das chaves de argumento de palavra-chave à lista de colunas e adiciona cada um de seus valores à parte `VALUES` da instrução SQL ([Exemplo 2-4](#)).

Exemplo 2-4. Valores na instrução de execução

```
ins = cookies.insert()
resultado = connection.execute( ins,
    cookie_name='dark chocolate
    chip', cookie_recipe_url='http://
    some.aweso.me/cookie/recipe_dark.html', cookie_sku='CC02', ②
```

```

quantidade='1',
custo_unidade='0,75'

) result.inserted_primary_key

```

- ① Nossa instrução de inserção é o primeiro argumento para a função de execução , como antes.

- ② Adicionamos nossos valores como argumentos de palavra-chave à função de execução .

Isto resulta em:

[2]

Embora isso não seja usado com frequência na prática para inserções únicas, ele fornece uma boa ilustração de como uma instrução é compilada e montada antes de ser enviada ao servidor de banco de dados. Podemos inserir vários registros de uma vez usando uma lista de dicionários com os dados que vamos enviar. Vamos usar esse conhecimento para inserir dois tipos de cookies na tabela de cookies ([Exemplo 2-5](#)).

Exemplo 2-5. Várias inserções

```

lista_inventário = [ { ❶
    'cookie_name': 'manteiga de
    amendoim', 'cookie_recipe_url': 'http://some.aweso.me/cookie/peanut.html',
    'cookie_sku': 'PB01', 'quantity': '24', 'unit_cost' : '0,25'

},
{
    'cookie_name': 'oatmeal raisin',
    'cookie_recipe_url': 'http://some.okay.me/cookie/raisin.html', 'cookie_sku':
    'EWW01', 'quantity': '100', 'unit_cost' : '1,00'

}
] resultado = connection.execute(ins, lista_de_inventário) ❷

```

- ❶ Construa nossa lista de cookies.

- ❷ Use a lista como o segundo parâmetro a ser executado.



Os dicionários na lista devem ter exatamente as mesmas chaves. SQLAlchemy compilará a instrução no primeiro dicionário da lista e a instrução falhará se os dicionários subsequentes forem diferentes, pois a instrução já foi construída com as colunas anteriores.

Agora que temos alguns dados em nossa tabela de cookies , vamos aprender como consultar as tabelas e recuperar esses dados.

Consultando dados

Para começar a construir uma consulta, começamos usando a função select , que é análoga à instrução SQL SELECT padrão . Inicialmente, vamos selecionar todos os registros em nossa tabela de cookies ([Exemplo 2-6](#)).

Exemplo 2-6. Função de seleção simples

```
from sqlalchemy.sql import select s =
select([cookies]) rp = connection.execute(s)
results = rp.fetchall()
```

②

- ①** Lembre-se de que podemos usar str(s) para ver a instrução SQL que o banco de dados verá, que neste caso é SELECT cookies.cookie_id, cookies.cookie_name, cookies.cookie_recipe_url, cookies.quantity, cookies.unit_cost FROM cookies
- ②** Isso diz ao rp, o ResultProxy, para retornar todas as linhas.

A variável results agora contém uma lista representando todos os registros em nossa tabela de cookies :

```
[(1, u'chocolate', u'http://some.aweso.me/cookie/recipe.html', u'CC01',
 12, Decimal('0.50')), (2,
u'dark chocolate chip', u'http://some.aweso.me/cookie/recipe_dark.html', u'CC02', 1,
Decimal('0.75 ')), (3, u'manteiga de amendoim', u'http://some.aweso.me/cookie/peanut.html',
u'PB01', 24, Decimal('0.25')), (4, u'oatmeal raisin', u'http://some.okay.me/cookie/raisin.html',
u'EW01', 100, Decimal('1.00'))]
```

No exemplo anterior, passei uma lista contendo a tabela de cookies . O método select espera uma lista de colunas para selecionar; no entanto, por conveniência, ele também aceita objetos Table e seleciona todas as colunas da tabela. Também é possível usar o método select no objeto Table para fazer isso, conforme mostrado no [Exemplo 2-7](#). Novamente, prefiro vê-lo escrito mais como o [Exemplo 2-6](#).

Exemplo 2-7. Método de seleção simples

```
from sqlalchemy.sql import select s =
cookies.select() rp = connection.execute(s)
results = rp.fetchall()
```

Antes de nos aprofundarmos nas consultas, precisamos saber um pouco mais sobre os objetos ResultProxy .

ResultProxy Um

ResultProxy é um wrapper em torno de um objeto de cursor DBAPI e seu objetivo principal é facilitar o uso e a manipulação dos resultados de uma instrução. Por exemplo, facilita o manuseio dos resultados da consulta, permitindo o acesso usando um índice, nome ou objeto Column. O Exemplo 2-8 demonstra todos esses três métodos. É muito importante se sentir confortável usando cada um desses métodos para obter os dados de coluna desejados.

Exemplo 2-8. Manipulando linhas com um ResultProxy

```
first_row = results[0]      ❶
first_row[1]    ❷
first_row.cookie_name   ❸
first_row[cookies.c.cookie_name] ❹
```

- ❶ Obtenha a primeira linha do ResultProxy do Exemplo 2-7.
- ❷ Coluna de acesso por índice.
- ❸ Coluna de acesso por nome.
- ❹ Coluna de acesso por objeto Column .

Todos eles resultam em 'chocolate chip' e cada um deles faz referência ao mesmo elemento de dados no primeiro registro de nossa variável de resultados . Essa flexibilidade no acesso é apenas parte do poder do ResultProxy. Também podemos aproveitar o ResultProxy como um iterável e executar uma ação em cada registro retornado sem criar outra variável para armazenar os resultados. Por exemplo, podemos querer imprimir o nome de cada cookie em nosso banco de dados (Exemplo 2-9).

Exemplo 2-9. Iterando em um ResultProxy

```
rp = connection.execute(s) para ❶
registro em rp:
    print(record.cookie_name)
```

- ❶ Estamos reutilizando a mesma instrução select anterior.

Isso retorna:

```
pepitas de chocolate
pepitas de chocolate escuro
```

passas de aveia
manteiga de amendoim

Além de usar o ResultProxy como um iterável ou chamar o método `fetchall()`, muitas outras maneiras de acessar dados por meio do ResultProxy estão disponíveis. Na verdade, todas as variáveis de resultado em “[Inserindo dados](#)” na [página 13](#) eram, na verdade, ResultProxys. Ambos os métodos `rowcount()` e `insert_primary_key()` que usamos nessa seção são apenas algumas das outras maneiras de obter informações de um ResultProxy. Você também pode usar os seguintes métodos para buscar resultados:

`primeiro()`

Retorna o primeiro registro se houver e fecha a conexão.

`buscar ()`

Retorna uma linha e deixa o cursor aberto para você fazer chamadas de busca adicionais.

`escalar()`

Retorna um único valor se uma consulta resultar em um único registro com uma coluna.

Se você quiser ver as colunas que estão disponíveis em um conjunto de resultados, você pode usar o método `keys()` para obter uma lista dos nomes das colunas. Usaremos os métodos `first`, `scalar`, `fetchone` e `fetchall`, bem como o ResultProxy como um iterável no restante deste capítulo.

Dicas para um bom código de produção

Ao escrever o código de produção, você deve seguir estas diretrizes:

- Use o primeiro método para obter um único registro sobre o `fetchone` e métodos escalares, porque é mais claro para nossos colegas codificadores.
- Use a versão iterável do ResultProxy sobre os métodos `fetchall` e `fetchone`. É mais eficiente em termos de memória e tendemos a operar os dados um registro de cada vez.
- Evite o método `fetchone`, pois ele deixa as conexões abertas se você não for cuidadoso.
- Use o método `escalar` com moderação, pois ele gera erros se uma consulta retornar mais de uma linha com uma coluna, o que geralmente é perdido durante o teste.

Toda vez que consultamos o banco de dados nos exemplos anteriores, todas as colunas eram retornadas para cada registro. Muitas vezes, precisamos apenas de uma parte dessas colunas para realizar nosso trabalho. Se os dados nessas colunas extras forem grandes, isso poderá fazer com que nossos aplicativos fiquem lentos e consumam muito mais memória do que deveriam.

SQLAlchemy não adiciona muita sobrecarga às consultas ou ResultProxys; no entanto, contabilizando o

os dados que você obtém de uma consulta geralmente são o primeiro lugar a ser verificado se uma consulta estiver consumindo muita memória. Vejamos como limitar as colunas retornadas em uma consulta.

Controlando as Colunas na Consulta

Para limitar os campos

que são retornados de uma consulta, precisamos passar as colunas que queremos para o construtor do método select() como uma lista. Por exemplo, talvez você queira executar uma consulta que retorne apenas o nome e a quantidade de cookies, conforme mostrado no [Exemplo 2-10](#).

Exemplo 2-10. Selecione apenas cookie_name e quantidade

```
s = select([cookies.c.cookie_name, cookies.c.quantity]) rp =
connection.execute(s) print(rp.keys()) resultado = rp.first()
1
2
```

- 1** Retorna a lista de colunas, que é [u'cookie_name', u'quantity'] neste exemplo (usado apenas para demonstração, não é necessário para resultados).
- 2** Observe que isso retorna apenas o primeiro resultado.

Resultado:

```
(u'chocolate', 12),
```

Agora que podemos construir uma instrução select simples, vamos ver outras coisas que podemos fazer para alterar como os resultados são retornados em uma instrução select. Começaremos alterando a ordem na qual os resultados são retornados.

Ordenação

Se você observar todos os resultados do [Exemplo 2-10](#) em vez de apenas o primeiro registro, verá que os dados não estão realmente em nenhuma ordem específica. No entanto, se quisermos que a lista seja retornada em uma ordem específica, podemos encadear uma instrução order_by() ao nosso select, conforme mostrado no [Exemplo 2-11](#). Nesse caso, queremos que os resultados sejam ordenados pela quantidade de biscoitos que temos à mão.

Exemplo 2-11. Ordem por quantidade crescente

```
s = select([cookies.c.cookie_name, cookies.c.quantity]) s =
s.order_by(cookies.c.quantity) rp = connection.execute(s) for
cookie in rp: print('{} - {}'.format(cookie.quantity,
cookie.cookie_name))
```

Isto resulta em:

1 - pepitas de chocolate negro
 12 - pepitas de chocolate 24
 - pasta de amendoim 100 -
 passas de aveia

Salvamos a instrução select na variável s , usamos essa variável s e adicionamos a instrução order_by a ela, e então a reatribuímos à variável s . Este é um exemplo de como compor declarações de forma generativa ou passo a passo. Isso é o mesmo que combinar as instruções select e order_by em uma linha, conforme mostrado aqui:

```
s = select([...]).order_by(...)
```

No entanto, quando temos a lista completa de colunas no select e as colunas order na instrução order_by , ela excede o limite de 79 caracteres por linha do Python (que foi estabelecido no PEP8). Usando a instrução de tipo generativo, podemos permanecer abaixo desse limite. Ao longo do livro, veremos alguns exemplos em que esse estilo generativo pode introduzir benefícios adicionais, como adicionar coisas condicionalmente à instrução. Por enquanto, tente quebrar suas instruções nesses limites de 79 caracteres e isso ajudará a tornar o código mais legível.

Se você quiser classificar em ordem reversa ou decrescente, use a instrução desc() . A função desc() envolve a coluna específica que você deseja classificar de maneira decrescente, conforme mostrado no [Exemplo 2-12](#).

Exemplo 2-12. Ordem por quantidade decrescente

```
from sqlalchemy import desc
= select([cookies.c.cookie_name, cookies.c.quantity]) s =
s.order_by(desc(cookies.c.quantity)) ❶
```

- ❶ Observe que estamos envolvendo a coluna cookies.c.quantity na função desc() .



A função desc() também pode ser usada como um método em um objeto Column , como cookies.c.quantity.desc(). No entanto, isso pode ser um pouco mais confuso para ler em instruções longas, então sempre uso desc() como uma função.

Também é possível limitar o número de resultados retornados se precisarmos apenas de um certo número deles para nossa aplicação.

Limitando

Nos exemplos anteriores, usamos os métodos first() ou fetchone() para obter apenas uma única linha de volta. Enquanto nosso ResultProxy nos deu a única linha que pedimos, a consulta real passou e acessou todos os resultados, não apenas o único registro. Se quisermos limitar a

consulta, podemos usar a função `limit()` para realmente emitir uma instrução de limite como parte de nossa consulta. Por exemplo, suponha que você só tenha tempo para fazer dois lotes de cookies e queira saber quais dois tipos de cookies você deve fazer. Você pode usar nossa consulta ordenada anterior e adicionar uma instrução de limite para retornar os dois tipos de cookies que mais precisam ser reabastecidos ([Exemplo 2-13](#)).

Exemplo 2-13. Dois menores estoques de biscoitos

```
s = select([cookies.c.cookie_name, cookies.c.quantity]) s =
s.order_by(cookies.c.quantity) s = s.limit(2) rp = connection.execute(s)
print([resultado.cookie_name para resultado em rp])
```

①

- ① Aqui estamos usando os recursos iteráveis do `ResultsProxy` em uma compreensão de lista.

Isto resulta em:

[u'chocolate amargo', u'chocolate']

Agora que você sabe que tipo de biscoitos você precisa assar, provavelmente está começando a ficar curioso sobre quantos biscoitos restam em seu inventário. Muitos bancos de dados incluem funções SQL projetadas para disponibilizar certas operações diretamente no servidor de banco de dados, como `SUM`; vamos explorar como usar essas funções a seguir.

Funções e rótulos SQL integrados

`SQLAlchemy` também pode aproveitar as funções SQL encontradas no banco de dados de back-end. Duas funções de banco de dados muito usadas são `SUM()` e `COUNT()`. Para usar essas funções, precisamos importar o módulo `sqlalchemy.sql.func` onde elas são encontradas. Essas funções são agrupadas em torno da(s) coluna(s) em que estão operando. Assim, para obter uma contagem total de cookies, você usaria algo como o [Exemplo 2-14](#).

Exemplo 2-14. Resumindo nossos cookies

```
from sqlalchemy.sql import func s =
select([func.sum(cookies.c.quantity)]) rp =
connection.execute(s) print(rp.scalar())
```

①

- ① Observe o uso de escalar, que retornará apenas a coluna mais à esquerda no primeiro registro.

Isto resulta em:

137



Eu costumo sempre importar o módulo func , pois importar sum diretamente pode causar problemas e confusão com a função sum integrada do Python .

Agora vamos usar a função count para ver quantos registros de inventário de cookies temos em nossa tabela de cookies ([Exemplo 2-15](#)).

Exemplo 2-15. Contando nossos registros de inventário

```
s = select([func.count(cookies.c.cookie_name)]) rp =
connection.execute(s) record = rp.first() print(record.keys())
print(record.count_1)
```

❶ ❷

- ❶ Isso nos mostrará as colunas no ResultProxy.
- ❷ O nome da coluna é gerado automaticamente e geralmente é <func_name>_<position>.

Isto resulta em:

[u'count_1'] 4

Este nome de coluna é irritante e complicado. Além disso, se tivermos várias contagens em uma consulta, teríamos que saber o número da ocorrência na instrução e incorporá-lo ao nome da coluna, de modo que a quarta função count() seria count_4. Isso simplesmente não é tão explícito e claro quanto deveríamos ser em nossa nomenclatura, especialmente quando cercado por outro código Python. Felizmente, SQLAlchemy fornece uma maneira de corrigir isso por meio da função label() . O [Exemplo 2-16](#) executa a mesma consulta que o [Exemplo 2-15](#); no entanto, ele usa label para nos dar um nome mais útil para acessar essa coluna.

Exemplo 2-16. Renomeando nossa coluna de contagem

```
s = select([func.count(cookies.c.cookie_name).label('inventory_count')]) rp =
connection.execute(s) record = rp.first() print(record.keys()) print(record .contagem de
inventário)
```

❶

- ❶ Observe que apenas usamos a função label() no objeto de coluna que queremos alterar.

Isto resulta em:

```
[u'inventory_count'] 4
```

Vimos exemplos de como restringir as colunas ou o número de linhas retornadas do banco de dados, então agora é hora de aprender sobre consultas que filtram dados com base em critérios que especificamos.

Filtragem

A filtragem de consultas é feita adicionando instruções where() exatamente como no SQL. Uma cláusula where() típica tem uma coluna, um operador e um valor ou coluna. É possível encadear várias cláusulas where() juntas, e elas agirão como ANDs em instruções SQL tradicionais. No Exemplo 2-17, encontraremos um cookie chamado “chocolate chip”.

Exemplo 2-17. Filtrando por nome de cookie

```
s = select([cookies]).where(cookies.c.cookie_name == 'chocolate') rp =
connection.execute(s) record = rp.first() print(record.items())
```

①

- ① Aqui estou chamando o método items() no objeto row, que me dará uma lista de colunas e valores.

Isto resulta em:

```
[
    (u'cookie_id', 1),
    (u'cookie_name', u'chocolate chip'),
    (u'cookie_recipe_url', u'http://some.aweso.me/cookie/recipe.html'), (u'cookie_sku',
    u'CC01'), (u'quantity', 12), (u'unit_cost', Decimal('0.50'))
```

]

Também podemos usar uma instrução where() para encontrar todos os nomes de cookies que contêm a palavra “chocolate” (Exemplo 2-18).

Exemplo 2-18. Encontrar nomes com chocolate neles

```
s = select([cookies]).where(cookies.c.cookie_name.like('%chocolate%')) rp =
connection.execute(s) para registro em rp.fetchall():
```

```
print(record.cookie_name)
```

Isto resulta em:

```
pepitas de chocolate
pepitas de chocolate escuro
```

Na instrução where() do [Exemplo 2-18](#), estamos usando cookies.c.cookie_name coluna dentro de uma instrução where() como um tipo de ClauseElement para filtrar nossos resultados. Devemos tomar um breve momento e falar mais sobre ClauseElements e os addições capacidades cionais que eles fornecem.

ClauseElements

ClauseElements são apenas uma entidade que usamos em uma cláusula e normalmente são colunas em uma mesa; no entanto, ao contrário das colunas, ClauseElements vem com muitas capacidades. No [Exemplo 2-18](#), estamos aproveitando o método like() que é disponível em ClauseElements. Existem muitos outros métodos disponíveis, que são listados na [Tabela 2-1](#). Cada um desses métodos é análogo a uma instrução SQL padrão construir. Você encontrará vários exemplos destes usados ao longo do livro.

Tabela 2-1. Métodos ClauseElement

Método	Objetivo
between(cleft, cright)	Encontre onde a coluna está entre cleft e cright
concat(columna_dois)	Concatenar coluna com column_two
distinto()	Encontre apenas valores exclusivos para a coluna
na lista[])	Encontre onde a coluna está na lista
é_(Nenhum)	Encontre onde a coluna é Nenhum (comumente usado para verificações de Nulos com Nenhum)
contém(cadeia)	Encontre onde a coluna tem <i>string</i> (diferencia maiúsculas de minúsculas)
termina com (string)	Encontre onde a coluna termina com <i>string</i> (diferencia maiúsculas de minúsculas)
como (cadeia)	Encontre onde a coluna é como <i>string</i> (diferencia maiúsculas de minúsculas)
começa com(string)	Encontre onde a coluna começa com <i>string</i> (diferencia maiúsculas de minúsculas)
ilike (string)	Encontre onde a coluna é como <i>string</i> (isso não diferencia maiúsculas de minúsculas)



Existem também versões negativas desses métodos, como notlike e notin_(). A única exceção à nomenclatura *not<method>* convenção é o método isnot() , que elimina o sublinhado.

Se não usarmos um dos métodos listados na [Tabela 2-1](#), teremos um operador em nossas cláusulas where . A maioria dos operadores funciona como você poderia esperar; no entanto, devemos discutir os operadores com um pouco mais de detalhes, pois existem algumas diferenças.

Operadores

Até agora, exploramos apenas onde uma coluna era igual a um valor ou usamos um dos métodos ClauseElement como like(); no entanto, também podemos usar muitos outros operadores comuns para filtrar dados. SQLAlchemy fornece sobrecarga para a maioria dos operadores padrão do Python. Isso inclui todos os operadores de comparação padrão (==, !=, <, >, <=, >=), que agem exatamente como você esperaria em uma instrução Python. O operador == também recebe uma sobrecarga adicional quando comparado a None, que o converte em uma instrução IS NULL . Os operadores aritméticos (+, -, *, / e %) também são suportados com recursos adicionais para concatenação de string independente do banco de dados, conforme mostrado no [Exemplo 2-19](#).

Exemplo 2-19. Concatenação de strings com \+

```
s = select([cookies.c.cookie_name, 'SKU-' + cookies.c.cookie_sku]) para linha em
connection.execute(s): print(row)
```

Isto resulta em:

```
(u'chocolate chip', u'SKU-CC01') (u'dark
chocolate chip', u'SKU-CC02') (u'manteiga de
amendoim', u'SKU-PB01') (u'oatmeal raisin',
u'SKU-EWW01')
```

Outro uso comum de operadores é calcular valores de várias colunas.

Você fará isso com frequência em aplicativos e relatórios que lidam com dados ou estatísticas financeiras.

O [Exemplo 2-20](#) mostra um cálculo de valor de estoque comum.

Exemplo 2-20. Valor do inventário por cookie

```
from sqlalchemy import cast s ❶
= select([cookies.c.cookie_name,
         cast((cookies.c.quantity * cookies.c.unit_cost),
              Numeric(12,2).label('inv_cost'))]) para ❷
linha em connection.execute(s):
    print('{} - {}'.format(row.cookie_name, row.inv_cost))
```

- ❶ Cast() é outra função que nos permite converter tipos. Nesse caso, obteremos resultados como 6,0000000000, portanto, ao convertê-lo, podemos fazer com que pareça moeda. Também é possível realizar a mesma tarefa em Python com print('{} - {:.2f}'.format(row.cookie_name, row.inv_cost)).

- ② Observe que estamos novamente usando a função label() para renomear a coluna. Sem essa renomeação, a coluna seria nomeada anon_1, pois a operação não resulta em um nome.

Isto resulta em:

```
pepitas de chocolate -
6,00 pepitas de chocolate negro
- 0,75 pasta de amendoim -
6,00 aveia passas - 100,00
```

Operadores booleanos

SQLAlchemy também permite os operadores booleanos SQL AND, OR e NOT por meio dos operadores lógicos bit a bit (&, | e ~). Cuidado especial deve ser tomado ao usar as sobrecargas AND, OR e NOT devido às regras de precedência do operador Python. Por exemplo, & liga mais de perto do que <, então quando você escreve A < B & C < D, o que você está realmente escrevendo é A < (B&C) < D, quando você provavelmente pretendia obter (A < B) & (C < D). Por favor, use conjunções em vez dessas sobrecargas, pois elas tornarão seu código mais expressivo.

Freqüentemente queremos encadear várias cláusulas where() juntas de maneira inclusiva e excludente; isso deve ser feito através de conjunções.

Conjunções

Embora seja possível encadear várias cláusulas where() juntas, geralmente é mais legível e funcional usar conjunções para obter o efeito desejado. As conjunções em SQLAlchemy são and_(), or_() e not_(). Portanto, se quisermos obter uma lista de cookies com um custo inferior a um valor e acima de uma determinada quantidade, podemos usar o código mostrado no Exemplo 2-21.

Exemplo 2-21. Usando a conjunção e()

```
from sqlalchemy import and_, or_, not_
s = select([cookies]).where( and_( cookies.c.quantity
> 23, cookies.c.unit_cost < 0.40
)
) para linha em connection.execute(s):
    print(row.cookie_name)
```

A função or_() funciona como o oposto de and_() e inclui resultados que correspondem a qualquer uma das cláusulas fornecidas. Se quiséssemos pesquisar em nosso inventário por cookies

tipos que temos entre 10 e 50 em estoque ou onde o nome contém chip, poderíamos usar o código mostrado no Exemplo 2-22.

Exemplo 2-22. Usando a conjunção or()

```
from sqlalchemy import and_, or_, not_
select

([cookies]).where( or_( cookies.c.quantity.between(10,
    50), cookies.c.cookie_name.contains('chip')
)
) para linha em connection.execute(s):
    print(row.cookie_name)
```

Isto resulta em:

```
chocolate amargo
com gotas de chocolate
manteiga de amendoim
```

A função not_() funciona de maneira semelhante a outras conjunções e simplesmente é usada para selecionar registros onde um registro não corresponde à cláusula fornecida. Agora que podemos consultar dados confortavelmente, estamos prontos para atualizar os dados existentes.

Atualizando dados

Muito parecido com o método insert que usamos anteriormente, também existe um método update com sintaxe quase idêntica a inserts, exceto que ele pode especificar uma cláusula where que indica quais linhas devem ser atualizadas. Assim como as instruções de inserção, as instruções de atualização podem ser criadas pela função update() ou pelo método update() na tabela que está sendo atualizada.

Você pode atualizar todas as linhas em uma tabela deixando de fora a cláusula where .

Por exemplo, suponha que você terminou de assar os biscoitos de chocolate que precisávamos para nosso inventário. No Exemplo 2-23, vamos adicioná-los ao nosso inventário existente com uma consulta de atualização e, em seguida, verificar quantos temos atualmente em estoque.

Exemplo 2-23. Atualizando dados

```
from sqlalchemy import update
= update(cookies).where(cookies.c.cookie_name == "chocolate") u =
u.values(quantity=(cookies.c.quantity + 120)) result = connection.execute(u )
print(result.rowcount) s = select([cookies]).where(cookies.c.cookie_name ==
"chocolate") resultado = connection.execute(s.first()) for chave in resultado.keys():
    print('{:>20}: {}'.format(chave, resultado[chave]))
```

① Usando o método generativo de construir nossa declaração.

② Imprimindo quantas linhas foram atualizadas.

Isso retorna:

```
1
cookie_id: 1
cookie_name: pepitas de chocolate
cookie_recipe_url: http://some.aweso.me/cookie/recipe.html cookie_sku:
CC01 quantidade: 132 unit_cost: 0,50
```

Além de atualizar os dados, em algum momento vamos querer remover os dados de nossas tabelas. A seção a seguir explica como fazer isso.

Excluindo dados

Para criar uma instrução delete, você pode usar a função `delete()` ou o método `delete()` na tabela da qual você está excluindo os dados. Ao contrário de `insert()` e `update()`, `delete()` não aceita parâmetros de valores, apenas uma cláusula `where` opcional (omitir a cláusula `where` excluirá todas as linhas da tabela). Veja o [Exemplo 2-24](#).

Exemplo 2-24. Excluindo dados

```
from sqlalchemy import delete
delete(cookies).where(cookies.c.cookie_name == "dark chocolate chip") resultado =
connection.execute(u) print(resultado.rowcount)

s = select([cookies]).where(cookies.c.cookie_name == "chocolate amargo") resultado =
connection.execute(s).fetchall() print(len(resultado))
```

Isso retorna:

```
1
0
```

OK, neste ponto, vamos carregar alguns dados usando o que já aprendemos para as tabelas `users`, `orders` e `line_items`. Você pode copiar o código mostrado aqui, mas também deve tirar um momento para brincar com as diferentes maneiras de inserir os dados:

```
lista_cliente = [ {
    'username': 'cookiemon',
    'email_address': 'mon@cookie.com',
    'phone': '111-111-1111', 'password':
    'password'}
```

```

},
{
    'username': 'cakeeater',
    'email_address': 'cakeeater@cake.com', 'phone':
    '222-222-2222', 'password': 'password'

},
{
    'username': 'pieguy',
    'email_address': 'guy@pie.com',
    'phone': '333-333-3333', 'password':
    'password'
}

] ins = users.insert()
resultado = connection.execute(ins, customer_list)

```

Agora que temos clientes, podemos começar a inserir seus pedidos e itens de linha no sistema também:

```

ins = insert(orders).values(user_id=1, order_id=1) result =
connection.execute(ins) ins = insert(line_items) order_items =
[ {

    'order_id': 1,
    'cookie_id': 1,
    'quantity': 2,
    'extended_cost': 1,00
},
{
    'order_id': 1,
    'cookie_id': 3,
    'quantity': 12,
    'extended_cost': 3,00
}

] resultado = connection.execute(ins, order_items) ins =
insert(orders).values(user_id=2, order_id=2) result =
connection.execute(ins) ins = insert(line_items) order_items =
[ {

    'order_id': 2,
    'cookie_id': 1,
    'quantity': 24,
    'extended_cost': 12,00
},
{
    'order_id': 2,
    'cookie_id': 4,
    'quantity': 6,
    'extended_cost': 6,00
}

```

```

    }

] resultado = connection.execute(ins, order_items)

```

No Capítulo 1, você aprendeu como definir ForeignKeys e relacionamentos, mas não os usamos para realizar consultas até este ponto. Vamos dar uma olhada nos relacionamentos a seguir.

Associações

Agora vamos usar os métodos `join()` e `outerjoin()` para dar uma olhada em como consultar dados relacionados. Por exemplo, para atender ao pedido feito pelo usuário do cookiemon, precisamos determinar quantos de cada tipo de cookie foram pedidos. Isso requer que você use um total de três junções para chegar até o nome dos cookies. Também vale a pena notar que, dependendo de como as junções são usadas em um relacionamento, você pode querer reorganizar a parte de uma instrução; uma maneira de fazer isso no SQLAlchemy é por meio da cláusula `select_from()`. Com `select_from()`, podemos substituir toda a cláusula `from` que SQLAlchemy geraria por uma que especificamos ([Exemplo 2-25](#)).

Exemplo 2-25. Usando join para selecionar entre várias tabelas

```

colunas = [orders.c.order_id, users.c.username, users.c.phone,
           cookies.c.cookie_name, line_items.c.quantity,
           line_items.c.extended_cost]
cookiemon_orders =
select(columns).cookiemon_orders =
cookiemon_orders.select_from(orders.join(users).join(
    line_items).join(cookies)).where(users.c.username ==
                                     'cookiemon') resultado =
connection.execute(cookiemon_orders).fetchall() para linha no resultado:

imprimir (linha)

```

① Observe que estamos dizendo ao SQLAlchemy para usar as junções de relacionamento como a cláusula `from`.

Isto resulta em:

```
(u'1', u'cookiemon', u'111-111-1111', u'chocolate', 2, Decimal('1.00')) (u'1', u'cookiemon', u'111-111-1111', u'manteiga de amendoim', 12, Decimal('3.00'))
```

O SQL fica assim:

```

SELECT orders.order_id, users.username, users.phone, cookies.cookie_name,
line_items.quantity, line_items.extended_cost FROM users JOIN orders ON users.user_id
= orders.user_id JOIN line_items ON orders.order_id = line_items.order_id JOIN cookies
ON cookies.cookie_id = line_items.cookie_id WHERE users.username = :username_1

```

Também é útil obter uma contagem de pedidos de todos os usuários, incluindo aqueles que não possuem pedidos presentes. Para fazer isso, temos que usar o método `outerjoin()`, e ele

requer um pouco mais de cuidado na ordenação da junção, pois a tabela em que usamos o método `outer join()` será aquela da qual todos os resultados serão retornados ([Exemplo 2-26](#)).

Exemplo 2-26. Usando `outerjoin` para selecionar entre várias tabelas

```
colunas = [users.c.username, func.count(orders.c.order_id)] all_orders =
select(columns) all_orders = all_orders.select_from(users.outerjoin(orders))
all_orders = all_orders.group_by(users.c.username) resultado =
connection.execute(all_orders).fetchall() para linha no resultado: print(row)
```

❶

- SQLAlchemy sabe como juntar as tabelas de usuários e pedidos por causa da chave estrangeira definida na tabela de pedidos .

Isto resulta em:

```
(u'cakeeater', 1)
(u'cookiemon', 1)
(u'pieguy', 0)
```

Até agora, usamos e juntamos diferentes tabelas em nossas consultas. No entanto, e se tivermos uma tabela auto-referencial como uma tabela de funcionários e seus chefes? Para tornar isso fácil de ler e entender, SQLAlchemy usa aliases.

Apelido

Ao usar junções, muitas vezes é necessário fazer referência a uma tabela mais de uma vez. No SQL, isso é feito usando aliases na consulta. Por exemplo, suponha que temos o seguinte esquema (parcial) que rastreia a estrutura de relatórios dentro de uma organização:

```
tabela_funcionário = Tabela(
    'employee', metadados,
    Column('id', Integer, primary_key=True),
    Column('manager', None, ForeignKey('employee.id')),
    Column('name', String(255)))
```

Agora suponha que queremos selecionar todos os funcionários gerenciados por um funcionário chamado Fred. Em SQL, podemos escrever o seguinte:

```
SELECT funcionário.nome
FROM funcionário, funcionário COMO
gerente WHERE funcionário.gerente_id =
gerente.id E gerente.nome = 'Fred'
```

SQLAlchemy também permite o uso de alias selecionáveis neste tipo de situação através da função ou método `alias()` :

```
>>> gerente = employee_table.alias('mgr') >>>
stmt = select([employee_table.c.name],
...             and_(employee_table.c.manager_id==manager.c.id,
...                   manager.c.name=='Fred'))
>>> imprimir(stmt)
SELECT funcionário.nome
FROM funcionário,funcionário AS
gerente ONDE funcionário.gerente_id = gerente.id E gerente.nome = ?
```

O SQLAlchemy também pode escolher o nome do alias automaticamente, o que é útil para garantir que não haja colisões de nomes:

```
>>> gerente = employee_table.alias() >>>
stmt = select([employee_table.c.name],
...             and_(employee_table.c.manager_id==manager.c.id,
...                   manager.c.name=='Fred' ))
>>> imprimir(stmt)
SELECT funcionário.nome
FROM funcionário,funcionário AS
funcionário_1 WHERE funcionário.gerente_id = funcionários_1.id E funcionários_1.nome = ?
```

Também é útil poder agrupar dados quando queremos relatar dados, então vamos analisar isso a seguir.

Agrupamento

Ao usar o agrupamento, você precisa de uma ou mais colunas para agrupar e uma ou mais colunas que faz sentido agregar com contagens, somas, etc., como faria no SQL normal. Vamos obter uma contagem de pedidos por cliente ([Exemplo 2-27](#)).

[Exemplo 2-27.](#) Dados de agrupamento

```
colunas = [users.c.username, func.count(orders.c.order_id)] all_orders = ①
select(columns) all_orders = all_orders.select_from(users.outerjoin(orders))
all_orders = all_orders.group_by(users.c.username ) resultado =
connection.execute(all_orders).fetchall() para linha no resultado ② print(row)
```

① Agregação por contagem

② Agrupamento pela coluna incluída não agregada

Isto resulta em:

```
(u'cakeeater', 1)
(u'cookiemon', 1)
(u'pieguy', 0)
```

Mostramos a construção generativa de declarações ao longo dos exemplos anteriores, mas quero focar nela especificamente por um momento.

Encadeamento

Usamos o encadeamento várias vezes ao longo deste capítulo e simplesmente não o reconhecemos diretamente. Onde o encadeamento de consultas é particularmente útil é quando você está aplicando lógica ao construir uma consulta. Então, se quisermos ter uma função que tenha uma lista de pedidos para nós, ela pode se parecer com o [Exemplo 2-28](#).

Exemplo 2-28. Encadeamento

```
def get_orders_by_customer(cust_name):
    columns = [orders.c.order_id, users.c.username, users.c.phone,
               cookies.c.cookie_name, line_items.c.quantity,
               line_items.c.extended_cost]
    cust_orders = select(columns)
    cust_orders =
        cust_orders.select_from( users.join(orders).join(line_items).join(cookies) )
    cust_orders = cust_orders.where(users.c.username == cust_name) result =
    connection.execute( cust_orders).fetchall() retorna o resultado

get_orders_by_customer('comedor de bolo')
```

Isto resulta em:

```
[(u'2', u'comedor de bolo', u'222-222-2222', u'chocolate', 24, Decimal('12.00')), (u'2', u'comedor de
bolo', u' 222-222-2222', u'oatmeal raisin', 6, Decimal('6.00'))]
```

No entanto, e se quiséssemos obter apenas os pedidos que foram enviados ou ainda não foram enviados? Teríamos que escrever funções adicionais para oferecer suporte a essas opções de filtro adicionais desejadas, ou podemos usar condicionais para construir cadeias de consulta. Outra opção que podemos querer é incluir ou não detalhes. Essa capacidade de encadear consultas e cláusulas permite relatórios bastante poderosos e construção de consultas complexas ([Exemplo 2-29](#)).

Exemplo 2-29. Encadeamento condicional

```
def get_orders_by_customer(cust_name, shipping =Nenhum, details=False):
    columns = [orders.c.order_id, users.c.username, users.c.phone] joins =
    users.join(orders) if details:

        columns.extend([cookies.c.cookie_name, line_items.c.quantity,
                        line_items.c.extended_cost]) joins =
        joins.join(line_items).join(cookies) cust_orders = select(columns) cust_orders
        = cust_orders.select_from(joins )
```

```

cust_orders = cust_orders.where(users.c.username == cust_name) se enviado
não for Nenhum:
    cust_orders = cust_orders.where(orders.c.shipped == enviado)
    resultado = connection.execute(cust_orders).fetchall() retorna
    resultado

get_orders_by_customer('comedor de bolo') ①

get_orders_by_customer('cakeeater', details=True) ②

get_orders_by_customer('cakeeater', enviado=True) ③

get_orders_by_customer('cakeeater', enviado=False) ④

get_orders_by_customer('cakeeater', enviado=False, detalhes=Verdadeiro) ⑤

```

- ① Recebe todos os pedidos.
- ② Obtém todos os pedidos com detalhes.
- ③ Obtém apenas os pedidos que foram enviados.
- ④ Recebe pedidos que ainda não foram enviados.
- ⑤ Obtém pedidos que ainda não foram enviados com detalhes.

Isto resulta em:

```

[(u'2 ', u'comedor de bolo', u'222-222-2222 ')]

[(u'2', u'comedor de bolo', u'222-222-2222', u'chocolate', 24, Decimal('12.00')), (u'2', u'comedor de
bolo', u' 222-222-2222', u'oatmeal raisin', 6, Decimal('6.00'))]

```

[]

```

[(u'2 ', u'comedor de bolo', u'222-222-2222 ')]

[(u'2', u'comedor de bolo', u'222-222-2222', u'chocolate', 24, Decimal('12.00')), (u'2', u'comedor de
bolo', u' 222-222-2222', u'oatmeal raisin', 6, Decimal('6.00'))]

```

Até agora neste capítulo, usamos a SQL Expression Language para todos os exemplos; no entanto, você pode estar se perguntando se também pode executar instruções SQL padrão. Vamos dar uma olhada.

Consultas brutas

Também é possível executar instruções SQL brutas ou usar SQL bruto em parte de uma consulta SQLAlchemy Core. Ele ainda retorna um ResultProxy e você pode continuar interagindo com ele como faria com uma consulta criada usando a sintaxe SQL Expression de

SQLAlchemy Core. Recomendo que você use apenas consultas brutas e texto quando necessário, pois isso pode levar a resultados imprevistos e vulnerabilidades de segurança. Primeiro, vamos querer executar uma instrução select simples ([Exemplo 2-30](#)).

Exemplo 2-30. Consultas brutas completas

```
resultado = connection.execute("seleccione * dos pedidos").fetchall()
print(result)
```

Isto resulta em:

```
[(1, 1, 0), (2, 2, 0)]
```

Embora eu raramente use uma instrução SQL bruta completa, geralmente uso pequenos trechos de texto para ajudar a tornar uma consulta mais clara. [O Exemplo 2-31](#) é de uma cláusula where do SQL bruto usando a função text() .

Exemplo 2-31. Consulta de texto parcial

```
from sqlalchemy import text
stmt = select([users]).where(text("username='cookieemon'"))
print(connection.execute(stmt).fetchall())
```

Isto resulta em:

```
[(1, Nenhum, u'cookieemon', u'mon@cookie.com', u'111-111-1111', u'password',
  datetime.datetime(2015, 3, 30, 13, 48, 25, 536450),
  datetime.datetime(2015, 3, 30, 13, 48, 25, 536457))]
```

Agora você deve ter uma compreensão de como usar a SQL Expression Language para trabalhar com dados no SQLAlchemy. Exploramos como criar, ler, atualizar e excluir operações. Este é um bom ponto para parar e explorar um pouco por conta própria. Tente criar mais cookies, pedidos e itens de linha e use cadeias de consulta para agrupá-los por pedido e usuário. Agora que você explorou um pouco mais e esperançosamente quebrou alguma coisa, vamos investigar como reagir a exceções levantadas no SQLAlchemy e como usar transações para agrupar instruções que devem ter sucesso ou falhar como um grupo.

CAPÍTULO 3

Exceções e transações

No capítulo anterior, trabalhamos muito com dados em instruções únicas e evitamos fazer qualquer coisa que pudesse resultar em erro. Neste capítulo, vamos propositadamente realizar algumas ações incorretamente para que possamos ver os tipos de erros que ocorrem e como devemos responder a eles. Concluiremos o capítulo aprendendo como agrupar as instruções que precisam ser bem-sucedidas em transações para que possamos garantir que o grupo seja executado corretamente ou seja limpo corretamente. Vamos começar por explodir as coisas!

Exceções

Existem inúmeras exceções que podem ocorrer no SQLAlchemy, mas vamos nos concentrar nas mais comuns: AttributeErrors e IntegrityErrors. Ao aprender a lidar com essas exceções comuns, você estará mais bem preparado para lidar com as que ocorrem com menos frequência.

Para acompanhar este capítulo, certifique-se de iniciar um novo shell Python e carregar as tabelas que construímos no [Capítulo 1](#) em seu shell. O [Exemplo 3-1](#) contém essas tabelas e a conexão novamente para referência.

Exemplo 3-1. Configurando nosso ambiente de shell

```
de datetime importação datetime  
da importação sqlalchemy (MetaData, Table, Column, Integer, Numeric, String,  
                         DateTime, ForeignKey, Boolean, create_engine,  
                         Verificar Restrição)  
metadados = MetaDados()  
  
cookies = Table('cookies', metadados,  
               Column('cookie_id', Integer(), primary_key=True),
```

```

        Column('cookie_name', String(50), index=True),
        Column('cookie_recipe_url', String(255)),
        Column('cookie_sku', String(55)),
        Column('quantidade', Integer()),
        Column('custo_unidade', Numérico(12, 2)),
        CheckConstraint('quantity > 0', name='quantity_positive')
    )

users = Table('users', metadata,
    Column('user_id', Integer(), primary_key=True),
    Column('username', String(15), nullable=False, unique=True),
    Column('email_address', String(255), nullable=False), Column('phone',
    String(20), nullable=False), Column('password', String(25), nullable=False),
    Column('created_on', DateTime (), default=datetime.now),
    Column('updated_on', DateTime(), default=datetime.now,
    onupdate=datetime.now)
)

pedidos = Table('orders', metadata,
    Column('order_id', Integer()),
    Column('user_id', ForeignKey('users.user_id')),
    Column('shipped', Boolean(), default=False )
)

line_items = Table('line_items', metadados,
    Column('line_items_id', Integer(), primary_key=True),
    Column('order_id', ForeignKey('orders.order_id')),
    Column('cookie_id', ForeignKey(' cookies.cookie_id')),
    Column('quantity', Integer()), Column('extended_cost', Numeric(12,
    2)))
)

engine = create_engine('sqlite:///memory:')
metadata.create_all(engine) connection =
engine.connect()

```

O primeiro erro que vamos aprender é o `AttributeError`; este é o erro mais comumente encontrado no código que estou depurando.

Erro de atributo

Começaremos com um `AttributeError` que ocorre quando você tenta acessar um atributo que não existe. Isso geralmente ocorre quando você está tentando acessar uma coluna em um `ResultProxy` que não está presente. `AttributeErrors` ocorrem quando você tenta acessar um atributo de um objeto que não está presente nesse objeto. Você provavelmente já se deparou com isso no código Python normal. Estou destacando isso porque esse é um erro comum no SQLAlchemy e é muito fácil perder o motivo pelo qual está ocorrendo. Para demonstrar esse erro, vamos inserir um registro em nossa tabela de usuários e executar uma consulta nele. Então

tentaremos acessar uma coluna dessa tabela que não selecionamos na consulta ([Exemplo 3-2](#)).

Exemplo 3-2. Causando um AttributeError

```
de sqlalchemy import select, insira ins =
insert(users).values( username="cookieemon",
                      email_address="mon@cookie.com",
                      phone="111-111-1111", password="password"

) resultado = connection.execute(ins)      ❶

s = select([users.c.username]) results
= connection.execute(s) for result in
results:
    print(result.username)
    print(result.password)      ❷
```

- ❶ Inserindo um registro de teste.
- ❷ A senha não existe, pois consultamos apenas a coluna nome de usuário.

O código no [Exemplo 3-2](#) faz com que o Python lance um `AttributeError` e interrompa a execução do nosso programa. Vamos examinar a saída de erro e aprender a interpretar o que aconteceu ([Exemplo 3-3](#)).

Exemplo 3-3. Saída de erro do [Exemplo 3-2](#)

```
biscoito mon
```

```
AttributeError                                     Traceback (última chamada mais recente) ❶
<ipython-input-37-c4520631a10a> em <module>()
  3 para resultado nos
  resultado.print(result.username)
----> 5      print(result.password)      ❷

AttributeError: Não foi possível localizar a coluna na linha da coluna 'senha'      ❸
```

- ❶ Isso nos mostra o tipo de erro e que um traceback está presente.
- ❷ Esta é a linha real onde ocorreu o erro.
- ❸ Esta é a parte interessante que precisamos focar.

No [Exemplo 3-3](#), temos o formato típico para um `AttributeError` em Python. Começa com uma linha que indica o tipo de erro. Em seguida, há um traceback mostrando-nos

onde ocorreu o erro. Como tentamos acessar a coluna em nosso código, ela nos mostra a linha real que falhou. O bloco final de linhas é onde os detalhes importantes podem ser encontrados. Ele especifica novamente o tipo de erro e logo depois mostra por que isso ocorreu. Nesse caso, é porque nossa linha do ResultProxy não possui uma coluna de senha. Consultamos apenas o nome de usuário. Embora este seja um erro comum do Python que podemos causar por meio de um bug em nosso uso de objetos SQLAlchemy, também existem erros específicos do SQLAlchemy que fazem referência a bugs que causamos com as próprias instruções SQLAlchemy. Vejamos um exemplo: o IntegrityError.

IntegrityError Outro

erro comum do SQLAlchemy é o IntegrityError, que ocorre quando tentamos fazer algo que violaria as restrições configuradas em uma Coluna ou Tabela. Esse tipo de erro é comumente encontrado nos casos em que você exige que algo seja exclusivo—por exemplo, se você tentar criar dois usuários com o mesmo nome de usuário, um IntegrityError será lançado porque os nomes de usuário em nossa tabela de usuários devem ser exclusivos. O Exemplo 3-4 mostra algum código que causará tal erro.

Exemplo 3-4. Causando um erro de integridade

```
s = select([users.c.username])
connection.execute(s).fetchall()      ❶

[(u'cookiemon',)]

ins =
    insert(users).values( username="cookiemon",
                           email_address="damon@cookie.com",
                           phone="111-111-1111", password="password"
                           )

) resultado = connection.execute(ins)      ❷
```

❶ Visualize os registros atuais na tabela de usuários .

❷ Tente inserir o segundo registro, o que resultará no erro.

O código no Exemplo 3-4 faz com que SQLAlchemy crie um IntegrityError. Vamos examinar a saída de erro e aprender a interpretar o que aconteceu (Exemplo 3-5).

Exemplo 3-5. Saída de erro de integridade

Erro de integridade <ipython-input-7-6ecafb68a8ab> em <module>() 5 password="password" 6)	Traceback (última chamada mais recente)
--	---

```
----> 7 resultado = connection.execute(ins) ①
...
②
IntegrityError: (sqlite3.IntegrityError) Falha na restrição UNIQUE: users.username
[SQL: u'INSERT INTO users (username, email_address, phone, password, created_on,
updated_on) VALUES (?, ?, ?, ?, ?, ?)' ] [parâmetros: ('cookieemon', 'damon@cookie.com',
'111-111-1111', 'senha', '2015-04-26 10:52:24.275082', '2015-04-26 10: 52:24.275099')] ③
```

- ① Esta é a linha que acionou o erro.
- ② Há um longo rastreamento aqui que eu omiti.
- ③ Esta é a parte interessante que precisamos focar.

O [Exemplo 3-5](#) mostra o formato típico para uma saída IntegrityError em SQLAlchemy. Começa com a linha que indica o tipo de erro. Em seguida, inclui os detalhes de rastreamento. No entanto, isso normalmente é apenas nossa instrução execute e código SQLAlchemy interno. Normalmente, o traceback pode ser ignorado para o tipo IntegrityError. O bloco final de linhas é onde os detalhes importantes são encontrados. Ele especifica novamente o tipo de erro e informa o que o causou. Neste caso, mostra:

Falha na restrição UNIQUE: users.username

Isso nos aponta para o fato de que há uma restrição exclusiva na coluna de nome de usuário na tabela de usuários que tentamos violar. Em seguida, ele nos fornece os detalhes da instrução SQL e seus parâmetros compilados semelhantes ao que vimos no [Capítulo 2](#). Os novos dados que tentamos inserir na tabela não foram inseridos devido ao erro. Este erro também interrompe a execução do nosso programa.

Embora existam muitos outros tipos de erros, os dois que abordamos são os mais comuns. A saída para todos os erros no SQLAlchemy seguirá o mesmo formato dos dois que acabamos de ver. A documentação do SQLAlchemy contém informações sobre os outros tipos de erros.

Como não queremos que nossos programas falhem sempre que encontrarem um erro, precisamos aprender a lidar com os erros corretamente.

Lidando com Erros

Para evitar que um erro falhe ou interrompa nosso programa, os erros precisam ser tratados de forma limpa. Podemos fazer isso como faríamos para qualquer erro do Python, com um bloco try/except. Por exemplo, podemos usar um bloco try/except para capturar o erro e imprimir uma mensagem de erro, então continuar com o resto do nosso programa; O [Exemplo 3-6](#) mostra os detalhes.

Exemplo 3-6. Capturando uma exceção

```
from sqlalchemy.exc import IntegrityError ins =      ①
insert(users).values( username="cookiemon",
                     email_address="damon@cookie.com",
                     phone="111-111-1111", password="password"

)
tente: resultado = connection.execute(ins)
excepto IntegrityError como erro:          ②
    print(error.orig.message, error.params)
```

- ❶ Todas as exceções SQLAlchemy estão disponíveis no módulo sqlalchemy.exc .
- ❷ Capturamos a exceção IntegrityError como erro para que possamos acessar as propriedades da exceção.

No [Exemplo 3-6](#), estamos executando a mesma instrução do [Exemplo 3-4](#), mas envolvendo a execução da instrução em um bloco try/except que captura um IntegrityError e imprime uma mensagem com a mensagem de erro e os parâmetros da instrução. Embora este exemplo tenha demonstrado como imprimir uma mensagem de erro, podemos escrever qualquer código Python que desejarmos na cláusula de exceção. Isso pode ser útil para retornar uma mensagem de erro aos usuários do aplicativo para informá-los de que a operação falhou. Ao lidar com o erro com um bloco try/except , nosso aplicativo continua a ser executado e executado.

Enquanto o [Exemplo 3-6](#) mostra um IntegrityError, este método de tratamento de erros funcionará para qualquer tipo de erro gerado pelo SQLAlchemy. Para obter mais informações sobre outras exceções do SQLAlchemy, consulte a documentação do SQLAlchemy em <http://docs.sqlalchemy.org/en/latest/core/exceptions.html>.



Lembre-se de que é uma prática recomendada envolver o mínimo de código possível em um bloco try/except e apenas capturar erros específicos. Isso evita a captura de erros inesperados que realmente deveriam ter um comportamento diferente da captura do erro específico que você está observando.

Embora tenhamos conseguido lidar com as exceções de uma única instrução usando as ferramentas tradicionais do Python, esse método sozinho não funcionará se tivermos várias instruções de banco de dados que dependem umas das outras para serem completamente bem-sucedidas. Nesses casos, precisamos agrupar essas instruções em uma transação de banco de dados, e o SQLAlchemy fornece um wrapper simples de usar para esse propósito embutido no objeto de conexão: transações.

Transações

Em vez de aprender a teoria profunda do banco de dados por trás das transações, pense nas transações como uma maneira de garantir que várias instruções de banco de dados sejam bem-sucedidas ou falhem como um grupo. Quando iniciamos uma transação, registramos o estado atual de nosso banco de dados; então podemos executar várias instruções SQL. Se todas as instruções SQL na transação forem bem-sucedidas, o banco de dados continuará funcionando normalmente e descartaremos o estado anterior do banco de dados. A [Figura 3-1](#) mostra o fluxo de trabalho normal da transação.

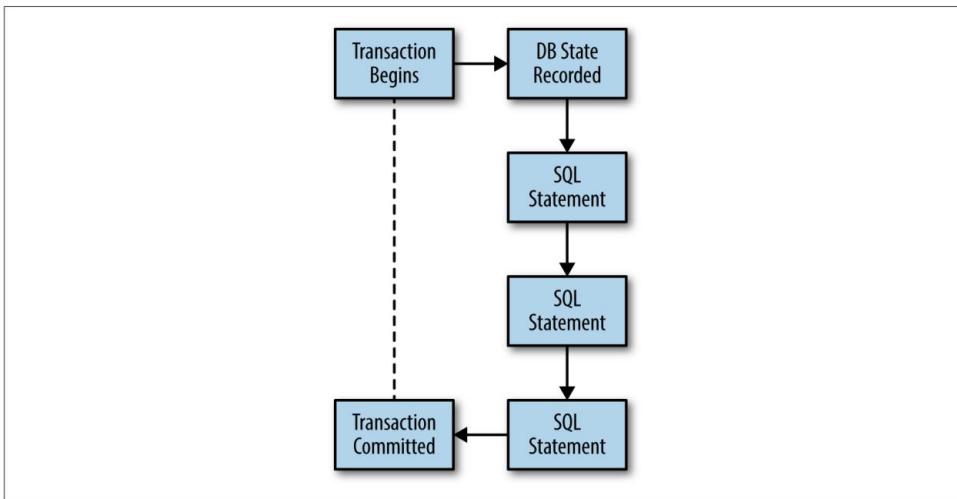


Figura 3-1. Fluxo de transação bem-sucedido

No entanto, se uma ou mais dessas instruções falharem, podemos detectar esse erro e usar o estado anterior para reverter todas as instruções que foram bem-sucedidas. A [Figura 3-2](#) mostra um fluxo de trabalho de transação com erro.

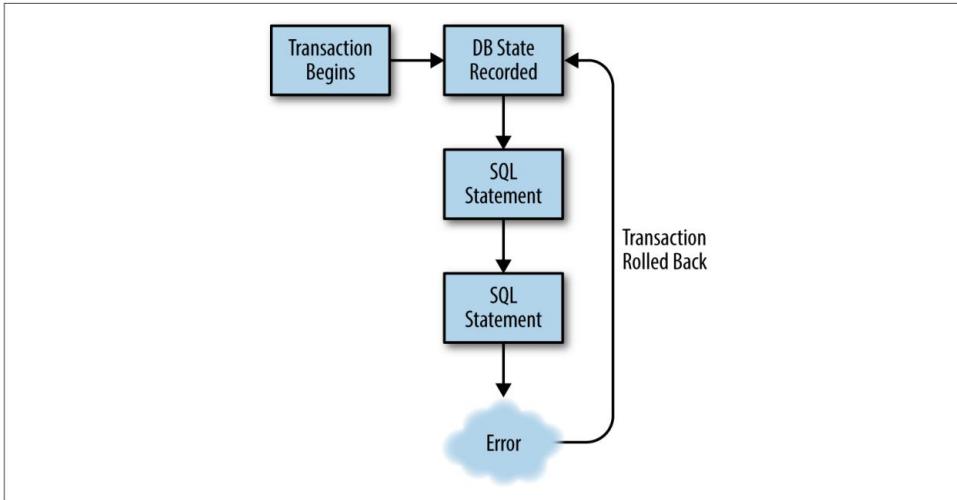


Figura 3-2. Fluxo de transação com falha

Já existe um bom exemplo de quando podemos querer fazer isso em nosso banco de dados existente. Depois que um cliente nos encomendar cookies, precisamos enviar esses cookies para o cliente e removê-los de nosso inventário. No entanto, e se não tivermos cookies suficientes para atender um pedido? Precisaremos detectar isso e não enviar esse pedido. Podemos resolver isso com transações.

Precisaremos de um novo shell Python com as tabelas do [Capítulo 2](#); no entanto, precisamos adicionar um CheckConstraint à coluna de quantidade para garantir que ela não fique abaixo de 0, pois não podemos ter cookies negativos no inventário. Em seguida, recrie o usuário cookiemon, bem como os registros de biscoito de chocolate e chocolate amargo. Defina a quantidade de biscoitos de chocolate para 12 e os biscoitos de chocolate amargo para 1.

O [Exemplo 3-7](#) mostra o código completo para configurar as tabelas com CheckConstraint, adicionar o usuário cookiemon e adicionar os cookies.

Exemplo 3-7. Configurando o ambiente de transações

```

de datetime importação datetime

da importação sqlalchemy (MetaData, Table, Column, Integer, Numeric, String,
                        DateTime, ForeignKey, Boolean, create_engine,
                        Verificar Restrição)
metadados = MetaDados()

cookies = Table('cookies', metadados,
                Column('cookie_id', Integer(), primary_key=True),
                Column('cookie_name', String(50), index=True),
                Column('cookie_recipe_url', String(255)),
                Column('cookie_sku', String(55)),
  
```

```

        Column('quantidade', Integer()),
        Column('custo_unidade', Numérico(12, 2)),
        CheckConstraint('quantidade >= 0', nome='quantidade_positiva')
    )

users = Table('users', metadata,
    Column('user_id', Integer(), primary_key=True),
    Column('username', String(15), nullable=False, unique=True),
    Column('email_address', String(255), nullable=False), Column('phone',
    String(20), nullable=False), Column('password', String(25), nullable=False),
    Column('created_on', DateTime (), default=datetime.now),
    Column('updated_on', DateTime(), default=datetime.now,
    onupdate=datetime.now)
)

pedidos = Table('orders', metadata,
    Column('order_id', Integer()),
    Column('user_id', ForeignKey('users.user_id')),
    Column('shipped', Boolean(), default=False )
)

line_items = Table('line_items', metadados,
    Column('line_items_id', Integer(), primary_key=True),
    Column('order_id', ForeignKey('orders.order_id')), Column('cookie_id',
    ForeignKey(' cookies.cookie_id')), Column('quantity', Integer()),
    Column('extended_cost', Numeric(12, 2))

)

engine = create_engine('sqlite:///memory:')
metadata.create_all(engine) connection =
engine.connect() from sqlalchemy import select,
insert, update ins =
insert(users).values( username="cookiemon" ,
    email_address="mon@cookie.com",
    phone="111-111-1111", password="senha"

) resultado = connection.execute(ins)
ins = cookies.insert() lista_de_inventário
= [ {

    'cookie_name': 'chocolate chip',
    'cookie_recipe_url': 'http://some.aweso.me/cookie/recipe.html', 'cookie_sku':
    'CC01', 'quantity': '12', 'unit_cost' : '0,50'

},
{
    'cookie_name': 'dark chocolate chip',
    'cookie_recipe_url': 'http://some.aweso.me/cookie/recipe_dark.html',

```

```

    'cookie_sku': 'CC02',
    'quantity': '1', 'unit_cost':
    '0,75'
}

] resultado = connection.execute(ins, lista_de_inventário)

```

Agora vamos definir dois pedidos para o usuário cookiemon. O primeiro pedido será de nove biscoitos com gotas de chocolate, e o segundo pedido será de quatro biscoitos com gotas de chocolate e um biscoito com gotas de chocolate amargo. Faremos isso usando as instruções de inserção discutidas no capítulo anterior. O [Exemplo 3-8](#) mostra os detalhes.

Exemplo 3-8. Adicionando os pedidos

```
ins = insert(orders).values(user_id=1, order_id='1') result =
connection.execute(ins) ins = insert(line_items) order_items = [ {
```

```

    'order_id': 1,
    'cookie_id': 1,
    'quantity': 9,
    'extended_cost': 4,50
}

] resultado = connection.execute(ins, order_items) ❶
```

```
ins = insert(orders).values(user_id=1, order_id='2') result =
connection.execute(ins) ins = insert(line_items) order_items = [ {
```

```

    'order_id': 2,
    'cookie_id': 1,
    'quantity': 4,
    'extended_cost': 1,50
},
{
    'order_id': 2,
    'cookie_id': 2,
    'quantity': 1,
    'extended_cost': 4,50
}
```

```
] resultado = connection.execute(ins, order_items) ❷
```

❶ Adicionando o primeiro pedido.

❷ Adicionando a segunda ordem.

Isso nos dará todos os dados de pedidos que precisamos para explorar como as transações funcionam; agora precisamos definir uma função chamada `ship_it`. Nossa função `ship_it` aceitará um `order_id`, removerá os cookies do inventário e marcará o pedido como enviado.

O Exemplo 3-9 mostra como isso funciona.

Exemplo 3-9. Definindo a função `ship_it`

`def ship_it(order_id):`

```
s = select([line_items.c.cookie_id, line_items.c.quantity]) s =
s.where(line_items.c.order_id == order_id) cookies_to_ship =
connection.execute(s) para cookie em cookies_to_ship:
    1
    u = update(cookies).where(cookies.c.cookie_id==cookie.cookie_id) u =
    u.values(quantity = cookies.c.quantity - cookie.quantity) result =
    connection.execute(u)
    u = update(orders).where(orders.c.order_id == order_id) u =
    u.values(shipped=True) result = connection.execute(u) print("ID do
pedido enviado: {}".format(order_id)) 2
```

- ➊ Para cada tipo de cookie que encontramos no pedido, removemos a quantidade pedida da tabela de cookies para saber quantos cookies ainda temos.
- ➋ Atualizamos o pedido para marcá-lo como enviado.

A função `ship_it` realizará todas as ações necessárias quando enviarmos um pedido.

Vamos executá-lo em nosso primeiro pedido e, em seguida, consultar a tabela de cookies para ter certeza de que reduziu a contagem de cookies corretamente. O Exemplo 3-10 mostra como fazer isso.

Exemplo 3-10. Executando `ship_it` no primeiro pedido

```
ship_it(1) s ➊
= select([cookies.c.cookie_name, cookies.c.quantity])
connection.execute(s).fetchall() 2
```

- ➊ Execute `ship_it` no primeiro `order_id`.
- ➋ Veja nosso inventário de cookies.

A execução do código no Exemplo 3-10 resulta em:

```
[(u'chocolate', 3), (u'chocolate amargo', 1)]
```

Excelente! Funcionou. Podemos ver que não temos cookies suficientes em nosso estoque para atender o segundo pedido; no entanto, em nosso armazém de ritmo acelerado, esses pedidos podem ser processados ao mesmo tempo. Agora tente enviar nosso segundo pedido com a função `ship_it` e observe o que acontece (como mostrado no Exemplo 3-11).

Exemplo 3-11. Executando ship_it no segundo pedido

```
ship_it(2)
```

Esse comando nos dá este resultado:

```
Erro de integridade                                         Traceback (última chamada mais recente)
<ipython-input-9-47771be6653b> em <module>() ----> 1
ship_it(2)

<ipython-input-6-301c0ed7c4a1> em ship_it(order_id)
  7      u = update(cookies).where(cookies.c.cookie_id ==
  8          cookie.cookie_id)
----> 9      u = u.values(quantidade = cookies.c.quantity-cookie.quantity) resultado =
 10         connection.execute(u)
 11      u = update(orders).where(orders.c.order_id == order_id) u =
...
IntegrityError: (sqlite3.IntegrityError) Falha na restrição CHECK: quantidade_positiva
[SQL: u'UPDATE cookies SET quantidade=(cookies.quantity - ?) WHERE
cookies.cookie_id = ?'] [parâmetros: (4, 1)]
```

Recebemos um IntegrityError porque não tínhamos biscoitos de chocolate suficientes para enviar o pedido. No entanto, vamos ver o que aconteceu com nossa tabela de cookies usando as duas últimas linhas do [Exemplo 3-10](#):

```
[(u'chocolate', 3), (u'chocolate amargo', 0)]
```

Ele não removeu os biscoitos de chocolate por causa do IntegrityError, mas removeu os biscoitos de chocolate amargo. Isso não é bom! Nós só queremos enviar pedidos inteiros para nossos clientes. Usando o que você aprendeu sobre try/except em “[Lidando com Erros](#)” na [página 41](#) anteriormente, você pode criar um método exceto complicado que reverteria a remessa parcial. No entanto, as transações nos fornecem uma maneira melhor de lidar apenas com esse tipo de evento.

As transações são iniciadas chamando o método begin() no objeto de conexão. O resultado dessa chamada é um objeto de transação que podemos usar para controlar o resultado de todas as nossas instruções. Se todas as nossas instruções forem bem-sucedidas, confirmamos a transação chamando o método commit() no objeto da transação. Caso contrário, chamamos o método rollback() nesse mesmo objeto. Vamos reescrever a função ship_it para usar uma transação para executar nossas instruções com segurança; [O Exemplo 3-12](#) mostra o que fazer.

Exemplo 3-12. Transacional ship_it

```
de sqlalchemy.exc import IntegrityError def
  ①
ship_it(order_id):
    s = select([line_items.c.cookie_id, line_items.c.quantity])
```

```

s = s.where(line_items.c.order_id == order_id) transação
= connection.begin() cookies_to_ship = ②
connection.execute(s).fetchall() ③

tentativa: para cookie em cookies_to_ship:
    u = update(cookies).where(cookies.c.cookie_id == cookie.cookie_id) u =
    u.values(quantity = cookies.c.quantity-cookie.quantity) result = connection.execute(u)

    u = update(orders).where(orders.c.order_id == order_id) u =
    u.values(shipped=True) result = connection.execute(u) print("ID do
pedido enviado: {}".format(order_id)) transação.commit()

④

exceto IntegrityError como erro:
    transaction.rollback() print(er⑤)

```

- ➊ Importando o IntegrityError para que possamos tratar sua exceção.
- ➋ Iniciando a transação.
- ➌ Buscando todos os resultados apenas para facilitar o acompanhamento do que está acontecendo.
- ➍ Confirmando a transação se não ocorrerem erros.
- ➎ Revertendo a transação se ocorrer um erro.

Agora vamos redefinir a quantidade de biscoitos de chocolate amargo de volta para 1:

```

u = update(cookies).where(cookies.c.cookie_name == "chocolate amargo") u =
u.values(quantity = 1) result = connection.execute(u)

```

Precisamos executar novamente nosso `ship_it` baseado em transações no segundo pedido. O programa não é interrompido pelo erro e nos imprime a mensagem de erro sem o traceback. Vamos verificar o estoque como fizemos no [Exemplo 3-10](#) para ter certeza de que ele não atrapalhou nosso estoque com uma remessa parcial:

```
[(u'chocolate', 3), (u'chocolate amargo', 1)]
```

Excelente! Nossa função transacional não atrapalhou nosso inventário ou travou nosso aplicativo. Também não tivemos que fazer muita codificação para reverter manualmente as instruções que tiveram sucesso. Como você pode ver, as transações podem ser realmente úteis em situações como essa e podem economizar muita codificação manual.

Neste capítulo, vimos como lidar com exceções tanto em instruções simples quanto em grupos de instruções. Usando um bloco try/except normal em uma única instrução, podemos evitar que nosso aplicativo falhe caso ocorra um erro na instrução do banco de dados.

Também analisamos como as transações podem nos ajudar a evitar bancos de dados inconsistentes e falhas de aplicativos em grupos de instruções. No próximo capítulo, aprenderemos como testar nosso código para garantir que ele se comporte da maneira que esperamos.

CAPÍTULO 4

Teste

A maioria dos testes dentro de aplicativos consiste em testes unitários e funcionais; no entanto, com SQLAlchemy, pode ser muito trabalhoso simular corretamente uma instrução de consulta ou um modelo para teste de unidade. Esse trabalho geralmente não leva a muito ganho em relação ao teste em um banco de dados durante o teste funcional. Isso leva as pessoas a criar funções de wrapper para suas consultas que podem ser facilmente simuladas durante os testes de unidade ou apenas testar em um banco de dados em seus testes de unidade e função. Eu pessoalmente gosto de usar pequenas funções de wrapper quando possível ou – se isso não fizer sentido por algum motivo ou se eu estiver em código legado – zombar disso.

Este capítulo aborda como realizar testes funcionais em um banco de dados e como simular consultas e conexões SQLAlchemy.

Testando com um banco de dados de teste

Para nosso aplicativo de exemplo, teremos um arquivo app.py que contém nossa lógica de aplicativo e um arquivo db.py que contém nossas tabelas e conexões de banco de dados.

Esses arquivos podem ser encontrados na pasta CH05/ do código de exemplo.

Como um aplicativo é estruturado é um detalhe de implementação que pode ter um grande efeito sobre como você precisa fazer seus testes. Em db.py, você pode ver que nosso banco de dados é configurado por meio da classe DataAccessLayer . Estamos usando essa classe de acesso a dados para nos permitir inicializar um esquema de banco de dados e conectá-lo a um mecanismo sempre que quisermos. Você verá esse padrão comumente usado em estruturas da Web quando acoplado ao SQLAlchemy.

A classe DataAccessLayer é inicializada sem um mecanismo e uma conexão na variável dal . O

Exemplo 4-1 mostra um trecho do nosso arquivo db.py.

Exemplo 4-1. Classe DataAccessLayer

```

from datetime import datetime
from sqlalchemy import (MetaData, Table, Column, Integer, Numeric, String,
                        DateTime, ForeignKey, Boolean, create_engine)

class DataAccessLayer:
    connection = Nenhum
    engine = Nenhum
    conn_string = Nenhum
    metadata = MetaData()
    cookies = Table('cookies',
                    metadata,
                    Column('cookie_id', Integer(), primary_key=True),
                    Column('cookie_name', String(50), index=True),
                    Column('cookie_recipe_url', String(255)),
                    Column('cookie_sku', String(55)),
                    Column('quantidade', Integer()),
                    Column('custo_unidade', Numérico(12, 2)))
    ) ①

    def db_init(self, conn_string): ②
        self.engine = create_engine(conn_string ou self.conn_string)
        self.metadata.create_all(self.engine) self.connection = self.engine.connect()

dal = DataAccessLayer() ③

```

- ① No arquivo completo, criamos todas as tabelas que usamos desde o [Capítulo 1](#), não apenas cookies.
- ② Isso fornece uma maneira de inicializar uma conexão com uma string de conexão específica, como uma fábrica.
- ③ Isso fornece uma instância da classe DataAccessLayer que pode ser importada em todo o nosso aplicativo.

Vamos escrever testes para a função `get_orders_by_customer` que construímos no [Capítulo 2](#), que se encontra no arquivo `app.py`, mostrado no [Exemplo 4-2](#).

Exemplo 4-2. app.py para testar

```

de db import dal de ①
sqlalchemy.sql import selecione

```

```

def get_orders_by_customer(cust_name, enviado=Nenhum, detalhes=False):
    colunas = [dal.orders.c.order_id, dal.users.c.username, dal.users.c.phone] joins =
        dal.users.join(dal.orders) ②

    se detalhes:
        columns.extend([dal.cookies.c.cookie_name,
                        dal.line_items.c.quantity,
                        dal.line_items.c.extended_cost]) joins
        = joins.join(dal.line_items).join(dal.cookies)

    cust_orders = select(columns)
    cust_orders = cust_orders.select_from(joins).where(dal.users.c.username
                                                       == cust_name)

    se enviado não for Nenhum:
        cust_orders = cust_orders.where(dal.orders.c.shipped == enviado)

    return dal.connection.execute(cust_orders).fetchall()

```

① Esta é a nossa instância DataAccessLayer do arquivo db.py.

② Como nossas tabelas estão dentro do objeto dal , nós as acessamos de lá.

Vejamos todas as maneiras pelas quais a função get_orders_by_customer pode ser usada. Vamos supor para este exercício que já validamos que as entradas para a função são do tipo correto. No entanto, em seus testes, seria muito sensato certificar-se de testar com dados que funcionarão corretamente e dados que possam causar erros.

Aqui está uma lista das variáveis que nossa função pode aceitar e seus possíveis valores:

- cust_name pode estar em branco, uma string contendo o nome de um cliente válido ou um string que não contém o nome de um cliente válido.
- enviado pode ser Nenhum, Verdadeiro ou Falso.
- os detalhes podem ser Verdadeiros ou Falso.

Se quisermos testar todas as combinações possíveis, precisaremos de 12 (essas 3 para testar completamente esta função.

* 3 * 2) testes



É importante não testar coisas que são apenas parte da funcionalidade básica do SQLAlchemy, pois o SQLAlchemy já vem com uma grande coleção de testes bem escritos. Por exemplo, não gostaríamos de testar uma instrução simples de inserção, seleção, exclusão ou atualização, pois elas são testadas no próprio projeto SQLAlchemy. Em vez disso, procure testar coisas que seu código manipula que podem afetar como a instrução SQLAlchemy é executada ou os resultados retornados por ela.

Para este exemplo de teste, usaremos o módulo unittest integrado . Não se preocupe se você não estiver familiarizado com este módulo; vamos explicar os pontos-chave. Primeiro, precisamos configurar a classe de teste e inicializar a conexão do dal , que é mostrada no [Exemplo 4-3](#) criando um novo arquivo chamado test_app.py.

Exemplo 4-3. Configurando os testes

```
importar teste unitário
```

```
class TestApp(unittest.TestCase):
```

①

```
    @classmethod
    def setUpClass(cls): ②
        dal.db_init('sqlite:///memory:') ③
```

① unittest requer classes de teste herdadas de unittest.TestCase.

② O método setUpClass é executado uma vez para toda a classe de teste.

③ Essa linha inicializa uma conexão com um banco de dados na memória para teste.

Agora precisamos carregar alguns dados para usar durante nossos testes. Não vou incluir o código completo aqui, pois são as mesmas inserções com as quais trabalhamos no [Capítulo 2](#), modificadas para usar o DataAccessLayer; ele está disponível no arquivo de exemplo db.py. Com nossos dados carregados, estamos prontos para escrever alguns testes. Vamos adicionar esses testes como funções dentro da classe TestApp , conforme mostrado no [Exemplo 4-4](#).

Exemplo 4-4. Os primeiros seis testes para nomes de usuário em branco

```
def test_orders_by_customer_blank(self): results ①
    = get_orders_by_customer("")
    self.assertEqual(results, []) ②

def test_orders_by_customer_blank_shipped(self): results
    = get_orders_by_customer("", True)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_notshipped(self): results =
    get_orders_by_customer("", False) self.assertEqual(results,
    [])

def test_orders_by_customer_blank_details(self):
    resultados = get_orders_by_customer("", details=True)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_shipped_details(self): results =
    get_orders_by_customer("", True, True) self.assertEqual(results,
    [])
```

```
def test_orders_by_customer_blank_notshipped_details(self): results =
    get_orders_by_customer("", False, True) self.assertEqual(results, [])
```

- ➊ unittest espera que cada teste comece com o teste de letras.
- ➋ unittest usa assertEquals para verificar se o resultado corresponde ao que você espera.
Como um usuário não foi encontrado, você deve obter uma lista vazia de volta.

Agora salve o arquivo de teste como `test_app.py` e execute os testes de unidade com o seguinte comando:

```
# python -m unittest test_app
.....
```

Executou 6 testes em 0,018s



Você pode receber um aviso sobre os tipos SQLite e decimal; apenas ignore isso, pois é normal para nossos exemplos. Isso ocorre porque o SQLite não tem um tipo decimal verdadeiro, e o SQLAlchemy quer que você saiba que pode haver algumas esquisitices devido à conversão do tipo float do SQLite. É sempre aconselhável investigar essas mensagens, porque no código de produção elas normalmente indicarão a maneira correta de fazer algo. Estamos acionando este aviso propositalmente aqui para que você veja como é.

Agora precisamos carregar alguns dados e garantir que nossos testes ainda funcionem. Novamente, vamos reutilizar o trabalho que fizemos no [Capítulo 2](#) e inserir os mesmos usuários, pedidos e itens de linha. No entanto, desta vez vamos envolver as inserções de dados em uma função chamada `db_prep`. Isso nos permitirá inserir esses dados antes de um teste com uma simples chamada de função. Para simplificar, coloquei esta função dentro do arquivo `db.py` (veja o [Exemplo 4-5](#)); no entanto, em situações do mundo real, ele geralmente estará localizado em um arquivo de ferramentas ou utilitários de teste.

Exemplo 4-5. Inserindo alguns dados de teste

```
def prep_db():
    ins = dal.cookies.insert()
    dal.connection.execute(ins, cookie_name=' chocolate amargo',
                          cookie_recipe_url='http://some.aweso.me/cookie/recipe_dark.html',
                          cookie_sku='CC02', quantidade='1', unit_cost='0.75') Inventory_list = [ {
        'cookie_name': 'manteiga de amendoim',
```

```
'cookie_recipe_url': 'http://some.aweso.me/cookie/peanut.html', 'cookie_sku':  
'PB01', 'quantity': '24', 'unit_cost': '0,25'  
  
},  
{  
    'cookie_name': 'oatmeal raisin',  
    'cookie_recipe_url': 'http://some.okay.me/cookie/raisin.html', 'cookie_sku':  
    'EWW01', 'quantity': '100', 'unit_cost' : '1,00'  
  
}  
]  
dal.connection.execute(ins, lista_de_inventário )  
  
lista_cliente = [ {  
  
    'username': "cookiemon",  
    'email_address': "mon@cookie.com",  
    'phone': "111-111-1111", 'password':  
    "password"  
},  
{  
    'username': "cakeeater",  
    'email_address': "cakeeater@cake.com",  
    'phone': "222-222-2222", 'password': "password"  
}  
,  
{  
    'username': "pieguy",  
    'email_address': "guy@pie.com",  
    'phone': "333-333-3333", 'password':  
    "password"  
}  
]  
ins = dal.users.insert()  
dal.connection.execute(ins, customer_list) ins =  
insert(dal.orders).values(user_id=1, order_id='wlk001') dal.connection.execute(ins)  
ins = insert(dal.line_items) order_items = [ {  
  
    'order_id': 'wlk001',  
    'cookie_id': 1, 'quantity':  
    2, 'extended_cost': 1,00  
},  
{  
    'order_id': 'wlk001',  
    'cookie_id': 3, 'quantity':  
    12, 'extended_cost': 3,00  
}
```

```

    }

] dal.connection.execute(ins, order_items) ins =
insert(dal.orders).values(user_id=2, order_id='ol001')
dal.connection.execute(ins) ins = insert(dal.line_items) order_items = [{

    'order_id': 'ol001',
    'cookie_id': 1, 'quantity':
24, 'extended_cost':
12,00
},
{
    'order_id': 'ol001',
    'cookie_id': 4, 'quantity':
6, 'extended_cost': 6,00

}

] dal.connection.execute(ins, order_items)

```

Agora que temos uma função prep_db , podemos usá-la em nosso método test_app.py setUpClass para carregar dados no banco de dados antes de executar nossos testes. Então agora nosso método setUp Class se parece com isso:

```

@classmethod
def setUpClass(cls):
    dal.db_init('sqlite:///memory') prep_db()

```

Podemos usar esses dados de teste para garantir que nossa função faça a coisa certa quando recebe um nome de usuário válido. Esses testes vão para dentro da nossa classe TestApp como novas funções, como mostra o Exemplo 4-6 .

Exemplo 4-6. Testes para um usuário válido

```

def test_orders_by_customer(self):
    resultados_esperados = [(u'wlk001', u'cookiemon', u'111-111-1111')] resultados
    = get_orders_by_customer('cookiemon') self.assertEqual(resultados,
    resultados_esperados )

def test_orders_by_customer_shipped_only(self):
    resultados = get_orders_by_customer('cookiemon', True)
    self.assertEqual(resultados, [])

def test_orders_by_customer_unshipped_only(self):
    expect_results = [(u'wlk001', u'cookiemon', u'111-111-1111')] resultados =
    get_orders_by_customer('cookiemon', False) self.assertEqual(results,
    expect_results )

```

```

def test_orders_by_customer_with_details(self):
    expect_results = [ (u'wlk001', u'cookiemon',
                       u'111-111-1111', u'dark chocolate chip', 2, Decimal('1.00')), (u'wlk001',
                                                                       u'cookiemon', u'111-111-1111', u'oatmeal raisin', 12, Decimal('3.00'))]

    ] resultados = get_orders_by_customer('cookiemon', details=True)
    self.assertEqual(resultados, expect_results)

def test_orders_by_customer_shipped_only_with_details(self):
    resultados = get_orders_by_customer('cookiemon', True, True)
    self.assertEqual(resultados, [])

def test_orders_by_customer_unshipped_only_details(self):
    esperado_results = [ (u'wlk001', u'cookiemon', u'111-111-1111',
                          u'dark chocolate chip', 2, Decimal('1.00')), (u'wlk001', u'cookiemon',
                                                                       u'111-111-1111', u'oatmeal raisin', 12, Decimal('3.00'))]

    ] resultados = get_orders_by_customer('cookiemon', False, True)
    self.assertEqual(resultados, resultados_esperados )

```

Usando os testes do [Exemplo 4-6](#) como orientação, você pode concluir os testes para ver o que acontece com um usuário diferente, como cakeeater? Que tal os testes para um nome de usuário que ainda não existe no sistema? Ou se obtivermos um inteiro em vez de uma string para o nome de usuário, qual será o resultado? Compare seus testes com os do código de exemplo fornecido quando terminar para ver se seus testes são semelhantes aos usados neste livro.

Aprendemos como podemos usar SQLAlchemy em testes funcionais para determinar se uma função se comporta conforme o esperado em um determinado conjunto de dados. Também vimos como configurar um arquivo unittest e como preparar o banco de dados para uso em nossos testes. Em seguida, estamos prontos para examinar os testes sem atingir o banco de dados.

Usando simulações

Essa técnica pode ser uma ferramenta poderosa quando você tem um ambiente de teste onde a criação de um banco de dados de teste não faz sentido ou simplesmente não é viável. Se você tiver uma grande quantidade de lógica que opera no resultado da consulta, pode ser útil simular o código SQLAlchemy para retornar os valores desejados para que você possa testar apenas a lógica circundante. Normalmente, quando vou zombar de alguma parte da consulta, ainda crio o banco de dados na memória, mas não carrego nenhum dado nele e zombou da própria conexão do banco de dados. Isso me permite controlar o que é retornado pelos métodos execute e fetch. Vamos explorar como fazer isso nesta seção.

Para aprender a usar mocks em nossos testes, vamos fazer um único teste para um usuário válido. Desta vez, usaremos a poderosa biblioteca simulada do Python para controlar o que é

retornado pela conexão. O Mock faz parte do módulo unittest no Python 3. No entanto, se você estiver usando o Python 2, precisará instalar a biblioteca de mock usando pip para obter os recursos de mock mais recentes. Para fazer isso, execute este comando:

```
simulação de instalação do pip
```

Agora que temos o mock instalado, podemos usá-lo em nossos testes. O Mock tem uma função de patch que nos permite substituir um determinado objeto em um arquivo Python por um MagicMock que podemos controlar em nosso teste. Um MagicMock é um tipo especial de objeto Python que rastreia como ele é usado e nos permite definir como ele se comporta com base em como está sendo usado.

Primeiro, precisamos importar a biblioteca simulada. No Python 2, precisamos fazer o seguinte:

```
importação simulada
```

No Python 3, precisamos fazer o seguinte:

```
da simulação de importação unittest
```

Com o mock importado, vamos usar o método patch como um decorador para substituir a parte de conexão do objeto dal . Um decorador é uma função que encapsula outra função e altera o comportamento da função encapsulada. Como o objeto dal é importado por nome no arquivo app.py, precisaremos corrigi-lo dentro do módulo app . Isso será passado para a função de teste como um argumento. Agora que temos um objeto simulado, podemos definir um valor de retorno para o método execute , que neste caso não deve ser nada além de um método fetchall encadeado cujo valor de retorno são os dados com os quais queremos testar. O Exemplo 4-7 mostra o código necessário para usar o mock no lugar do objeto dal .

Exemplo 4-7. Teste de conexão simulado

```
importar teste
unitário do decimal importar decimal

importação simulada

do db import dal, prep_db do app
import get_orders_by_customer

class TestApp(unittest.TestCase):
    cookie_orders = [(u'wlk001', u'cookiemon', u'111-111-1111')] cookie_details =
        [ (u'wlk001', u'cookiemon', u'111-111-1111', u'dark chocolate chip', 2, Decimal('1.00')),
          (u'wlk001', u'cookiemon', u'111-111-1111', u'oatmeal raisin', 12, Decimal('3.00'))]

    @mock.patch('app.dal.connection')
    def test_get_orders(self, connection):
        connection.cursor.return_value.fetchall.return_value = self.cookie_orders
        result = get_orders_by_customer()
        self.assertEqual(result, self.cookie_orders)
```

①

```
def test_orders_by_customer(self, mock_conn):          ②
    mock_conn.execute.return_value.fetchall.return_value = self.cookie_orders resultados = ③
    get_orders_by_customer('cookieemon') self.assertEqual(results, self.cookie_orders) ④
```

- ➊ Corrigindo o dal.connection no módulo do aplicativo com uma simulação.
- ➋ Esse mock é passado para a função de teste como mock_conn.
- ➌ Definimos o valor de retorno do método execute para o valor retornado encadeado de o método fetchall , que definimos como self.cookie_order.
- ➍ Agora chamamos a função de teste onde o dal.connection será simulado e retornamos o valor que definimos na etapa anterior.

Você pode ver que uma consulta complicada ou ResultProxy como a do [Exemplo 4-7](#) pode ficar entediante rapidamente ao tentar simular a consulta ou conexão completa.

Não fuja do trabalho; pode ser muito útil para encontrar bugs obscuros.

Se você quisesse simular a consulta, seguiria o mesmo padrão de usar o decorador mock.patch e simular o objeto selecionado no módulo do aplicativo .

Vamos tentar isso com uma das consultas de teste vazias. Temos que simular todos os valores de retorno do elemento de consulta encadeado, que nesta consulta são as cláusulas select, select_from e where . O [Exemplo 4-8](#) demonstra como fazer isso.

Exemplo 4-8. Zombando da consulta também

```
@mock.patch('app.select')           ①
@mock.patch('app.dal.connection') def
test_orders_by_customer_blank(self, mock_conn, mock_select):           ②
    mock_select.return_value.select_from.return_value.\
        where.return_value = ③
    mock_conn.execute.return_value.fetchall.return_value = [] resultados ④
    = get_orders_by_customer("") self.assertEqual(results, [])
```

- ➊ Zombando do método select , pois ele inicia a cadeia de consulta.
- ➋ Os decoradores são passados para a função em ordem. À medida que avançamos na pilha de decoradores da função, os argumentos são adicionados à esquerda.
- ➌ Temos que simular o valor de retorno para todas as partes da consulta encadeada.
- ➍ Ainda precisamos zombar da conexão ou o código SQLAlchemy do módulo do aplicativo tentaria fazer a consulta.

Como exercício, você deve trabalhar na construção do restante dos testes que construímos com o banco de dados na memória com os tipos de teste simulados. Recomendo que você zombe da consulta e da conexão para se familiarizar com o processo de simulação.

Agora você deve se sentir confortável em como testar uma função que contém funções SQLAlchemy dentro dela. Você também deve entender como preencher previamente os dados no banco de dados de teste para uso em seu teste. Por fim, você deve entender como zombar dos objetos de consulta e de conexão. Embora este capítulo tenha usado um exemplo simples, vamos mergulhar mais fundo em testes no [Capítulo 14](#), que analisa Flask, Pyramid e pytest.

A seguir, veremos como lidar com um banco de dados existente com SQLAlchemy sem a necessidade de recriar todo o esquema em Python por meio de reflexão.

CAPÍTULO 5

Reflexão

A reflexão é uma técnica que nos permite preencher um objeto SQLAlchemy de um banco de dados existente. Você pode refletir tabelas, exibições, índices e chaves estrangeiras. Este capítulo explorará como usar a reflexão em um banco de dados de exemplo.

Para testar, recomendo usar o banco de dados Chinook. Você pode aprender mais sobre isso em <http://chinookdatabase.codeplex.com/>. Usaremos a versão SQLite, que está disponível na pasta CH06/ do código de exemplo deste livro. Essa pasta também contém uma imagem do esquema do banco de dados para que você possa visualizar o esquema com o qual trabalharemos ao longo deste capítulo. Começaremos refletindo uma única tabela.

Refletindo Tabelas Individuais

Para nossa primeira reflexão, vamos gerar a tabela Artista . Precisaremos de um objeto de metadados para manter as informações do esquema da tabela refletida e um mecanismo anexado ao banco de dados Chinook. O Exemplo 5-1 demonstra como configurar essas duas coisas; o processo deve ser muito familiar para você agora.

Exemplo 5-1. Configurando nossos objetos iniciais

```
de sqlalchemy import MetaData, metadados  
create_engine = MetaData() engine = create_engine('sqlite:///  
Chinook_Sqlite.sqlite')
```

①

- ① Essa cadeia de conexão pressupõe que você esteja no mesmo diretório que o banco de dados de exemplo.

Com os metadados e o mecanismo configurados, temos tudo o que precisamos para refletir uma tabela. Vamos criar um objeto de tabela usando um código semelhante ao código de criação de tabela no Capítulo 1; no entanto, em vez de definir as colunas manualmente, vamos usar o

autoload e autoload_with argumentos de palavra-chave. Isso refletirá as informações do esquema no objeto de metadados e armazenará uma referência à tabela na variável artist . O Exemplo 5-2 demonstra como realizar a reflexão.

Exemplo 5-2. Refletindo a tabela Artista

```
from sqlalchemy import Table
artist = Table('Artist', metadata, autoload=True, autoload_with=engine)
```

Essa última linha é tudo o que precisamos para refletir a tabela Artista ! Podemos usar essa tabela exatamente como fizemos no Capítulo 2 com nossas tabelas definidas manualmente. O Exemplo 5-3 mostra como realizar uma consulta simples com a tabela para ver que tipo de dados está nela.

Exemplo 5-3. Usando a tabela Artista

```
artist.columns.keys() de ❶
sqlalchemy import select s =
select([artist]).limit(10) ❷
engine.execute(s).fetchall()
```

❶ Listando as colunas.

❷ Selecionando os 10 primeiros registros.

Isto resulta em:

```
['ArtistId', 'Nome']
[(1, u'AC/DC'), (2,
 u'Accept'), (3,
 u'Aerosmith'), (4,
 u'Alanis Morissette'), (5, u'Alice
 In Chains') , (6, u'Ant\xf4nio
 Carlos Jobim'), (7, u'Apocalyptica'), (8,
 u'Audioslave'), (9, u'BackBeat'), (10,
 u'Billy Cobham') ]
```

Observando o esquema do banco de dados, podemos ver que existe uma tabela Álbum que está relacionada à tabela Artista . Vamos refletir da mesma forma que fizemos para a tabela Artista :

```
album = Table('Album', metadata, autoload=True, autoload_with=engine)
```

Agora vamos verificar os metadados da tabela Álbum para ver o que foi refletido. O Exemplo 5-4 mostra como fazer isso.

Exemplo 5-4. Como visualizar os metadados

```
metadata.tables['álbum']

Table('álbum',
    MetaData(bind=Nenhum),
    Column('AlbumId', INTEGER(), table=<álbum>, primary_key=True, nullable=False),
    Column('Título', NVARCHAR(comprimento=160), tabela=<álbum>, anulável=False),
    Column('ArtistId', INTEGER(), table=<álbum>, nullable=False), schema=Nenhum)

)
```

Curiosamente, a chave estrangeira para a tabela Artista não parece ter sido refletida. Vamos verificar o atributo estrangeira_chaves da tabela Álbum para ter certeza de que não está presente:

```
album.foreign_keys

definir()
```

Então realmente não se refletiu. Isso ocorreu porque as duas tabelas não foram refletidas ao mesmo tempo e o destino da chave estrangeira não estava presente durante a reflexão. Em um esforço para não deixá-lo em um estado semi-quebrado, SQLAlchemy descartou o relacionamento unilateral. Podemos usar o que aprendemos no [Capítulo 1](#) para adicionar a ForeignKey ausente e restaurar o relacionamento:

```
from sqlalchemy import ForeignKeyConstraint
album.append_constraint(ForeignKeyConstraint(['ArtistId'],
    [artist.ArtistId])
)
```

Agora, se executarmos novamente o [Exemplo 5-4](#), podemos ver que a coluna ArtistId é uma ForeignKey:

```
Table('álbum',
    MetaData(bind=Nenhum),
    Column('AlbumId', INTEGER(), table=<álbum>, primary_key=True, nullable=False),
    Column('Title', NVARCHAR(length=160), table=<álbum>, nullable=False), Column('ArtistId',
    INTEGER(), ForeignKey('artist.ArtistId'), tabela=<álbum>, anulável=False),
    esquema=Nenhum)
```

Agora vamos ver se podemos usar o relacionamento para unir as tabelas corretamente. Podemos executar o seguinte código para testar o relacionamento:

```
str(artist.join(álbum))

"artist JOIN album ON artist." ArtistId " = album." ArtistId "
```

Excelente! Agora podemos realizar consultas que usam esse relacionamento. Funciona exatamente como as consultas discutidas em ["Joins"](#) na página 31.

Seria um pouco trabalhoso repetir o processo de reflexão para cada tabela individual em nosso banco de dados. Felizmente, SQLAlchemy permite refletir um banco de dados inteiro em uma vez.

Refletindo um banco de dados inteiro

Para refletir um banco de dados inteiro, podemos usar o método `reflect` no objeto de metadados . O método `reflect` examinará tudo o que estiver disponível no mecanismo fornecido e refletirá tudo o que puder. Vamos usar nossos metadados e objetos de mecanismo existentes para refletir todo o banco de dados Chinook:

```
metadata.reflect(bind=engine)
```

Essa instrução não retorna nada se for bem-sucedida; no entanto, podemos executar o seguinte código para recuperar uma lista de nomes de tabelas para ver o que foi refletido em nossos metadados:

```
metadata.tables.keys()
```

```
dict_keys(['Linha da fatura', 'Funcionário', 'Fatura', 'álbum', 'Gênero',
          'PlaylistTrack', 'Album', 'Customer', 'MediaType', 'Artist',
          'Faixa', 'artista', 'Playlist'])
```

As tabelas que refletimos manualmente são listadas duas vezes, mas com letras maiúsculas e minúsculas diferentes. Isso se deve ao fato de SQLAlchemy refletir as tabelas conforme são nomeadas e, no banco de dados Chinook, elas são maiúsculas. Devido ao tratamento de diferenciação de maiúsculas e minúsculas do SQLite, os nomes em letras minúsculas e maiúsculas apontam para as mesmas tabelas no banco de dados.



Tenha cuidado, pois a distinção entre maiúsculas e minúsculas pode atrapalhar outros bancos de dados, como Oracle.

Agora que temos nosso banco de dados refletido, estamos prontos para discutir o uso de tabelas refletidas nas consultas.

Criação de consultas com objetos refletidos

Como você viu no [Exemplo 5-3](#), consultar tabelas que refletimos e armazenamos em uma variável funciona exatamente como no [Capítulo 2](#). No entanto, para o restante das tabelas que foram refletidas quando refletimos todo o banco de dados, Precisaremos de uma maneira de se referir a eles em nossa consulta. Podemos fazer isso atribuindo-os a uma variável do atributo `tables` dos metadados, conforme mostrado no [Exemplo 5-5](#).

Exemplo 5-5. Usando uma tabela refletida na consulta

```
playlist = metadata.tables['Playlist']          ①

from sqlalchemy import select s
= select([playlist]).limit(10)                  ②
engine.execute(s).fetchall()
```

① Estabeleça uma variável para ser uma referência à tabela.

② Use essa variável na consulta.

A execução do código no [Exemplo 5-5](#) nos dá este resultado:

```
engine.execute(s).fetchall()
[(1, 'Música'), (2,
'Filmes'), (3,
'Programas de
TV'), (4, 'Audiolivros'),
(5, 'Música dos anos
90'), (6, 'Audiolivros'),
(7, 'Filmes'), (8,
'Música'), (9, 'Vídeos
de música'), (10,
'Programas de TV')]
```

Ao atribuir as tabelas refletidas que queremos usar em uma variável, elas podem ser usadas da mesma forma que os capítulos anteriores.

A reflexão é uma ferramenta muito útil; no entanto, a partir da versão 1.0 do SQLAlchemy, não podemos refletir CheckConstraints, comentários ou gatilhos. Você também não pode refletir os padrões do lado do cliente ou uma associação entre uma sequência e uma coluna. No entanto, é possível adicioná-los manualmente usando os métodos descritos no [Capítulo 1](#).

Agora você entende como refletir tabelas individuais e reparar problemas como relacionamentos ausentes nessas tabelas. Além disso, você aprendeu como refletir um banco de dados inteiro e usar tabelas individuais em consultas das tabelas refletidas.

Este capítulo resume as partes essenciais do SQLAlchemy Core e da SQL Expression Language. Espero que você tenha percebido o quanto poderosas são essas partes do SQLAlchemy. Eles geralmente são ignorados devido ao ORM SQLAlchemy, mas usá-los pode adicionar ainda mais recursos aos seus aplicativos. Vamos aprender sobre o SQLAlchemy ORM.

PARTE II

SQLAlchemy ORM

O SQLAlchemy ORM é o que a maioria das pessoas pensa quando você menciona SQLAlchemy. Ele fornece uma maneira muito eficaz de vincular o esquema e as operações do banco de dados aos mesmos objetos de dados usados em seu aplicativo. Ele oferece uma maneira de criar aplicativos rapidamente e colocá-los nas mãos dos clientes. Usar o ORM é tão simples que a maioria das pessoas não considera os possíveis efeitos colaterais de seu código. Ainda é importante pensar em como o banco de dados será usado pelo seu código. No entanto, a coisa maravilhosa sobre o ORM é que você pode pensar nisso quando surgir a necessidade e não para todas as consultas. Vamos começar a usar o ORM.

CAPÍTULO 6

Definindo esquema com SQLAlchemy ORM

Você define o esquema ligeiramente diferente ao usar o SQLAlchemy ORM porque ele se concentra em objetos de dados definidos pelo usuário em vez do esquema do banco de dados subjacente. No SQLAlchemy Core, criamos um contêiner de metadados e declaramos um objeto Table associado a esses metadados. No SQLAlchemy ORM, vamos definir uma classe que herda de uma classe base especial chamada declarative_base. O declarative_base combina um contêiner de metadados e um mapeador que mapeia nossa classe para uma tabela de banco de dados. Ele também mapeia instâncias da classe para registros nessa tabela, caso tenham sido salvas. Vamos nos aprofundar na definição de tabelas dessa maneira.

Definindo tabelas por meio de classes ORM

Uma classe adequada para uso com o ORM deve fazer quatro coisas:

- Herdar do objeto declarative_base . • Conter __tablename__, que é o nome da tabela a ser usada no banco de dados.
- Contém um ou mais atributos que são objetos Column . • Certifique-se de que um ou mais atributos constituam uma chave primária.

Precisamos examinar os dois últimos requisitos relacionados a atributos um pouco mais de perto. Primeiro, definir colunas em uma classe ORM é muito semelhante a definir colunas em um objeto Table , que discutimos no Capítulo 2; no entanto, há uma diferença muito importante. Ao definir colunas em uma classe ORM, não precisamos fornecer o nome da coluna como o primeiro argumento para o construtor Column . Em vez disso, o nome da coluna será definido como o nome do atributo de classe ao qual está atribuído. Todo o resto que abordamos em “Tipos” na página 1 e “Colunas” na página 5 também se aplica aqui e funciona conforme o esperado. Em segundo lugar, o requisito de uma chave primária pode parecer estranho a princípio; Como as-

sempre, o ORM precisa ter uma maneira de identificar e associar exclusivamente uma instância da classe a um registro específico na tabela de banco de dados subjacente. Vejamos nossa tabela de cookies definida como uma classe ORM ([Exemplo 6-1](#)).

Exemplo 6-1. Tabela de cookies definida como uma classe ORM

```
from sqlalchemy import Table, Column, Integer, Numeric, String from
sqlalchemy.ext.declarative import declarative_base
```

```
Base = declarative_base()      ❶

class Cookie(Base):    ❷
    __tablename__ = 'cookies' ❸

    cookie_id = Column(Integer(), primary_key=True)      ❹
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255))
    cookie_sku =
    Column(String(55))
    quantidade = Column(Integer())
    unit_cost = Column(Numeric(12, 2))
```

- ❶ Crie uma instância do declarative_base.
- ❷ Herdar da Base.
- ❸ Defina o nome da tabela.
- ❹ Defina um atributo e defina-o como uma chave primária.

Isso resultará na mesma tabela mostrada no [Exemplo 2-1](#), e isso pode ser verificado observando a propriedade `Cookie.__table__`:

```
>>> Cookie.__table__
Table('cookies', MetaData(bind=None),
      Column('cookie_id', Integer(), table=<cookies>, primary_key=True, nullable=False),
      Column('cookie_name', String(length=50), table=<cookies>),
      Column('cookie_recipe_url', String(length=255), table=<cookies>),
      Column('cookie_sku', String(length=15), table=<cookies>),
      Column('quantidade', Integer(), table=<cookies>),
      Column('unit_cost', Numeric(precision=12, scale=2),
             table=<cookies>), schema=Nenhum)
```

Vejamos outro exemplo. Desta vez, quero recrivar nossa tabela de usuários do [Exemplo 2-2](#). O [Exemplo 6-2](#) demonstra como as palavras-chave adicionais funcionam da mesma forma nos esquemas ORM e Core.

Exemplo 6-2. Outra tabela com mais opções de colunas

```
de datetime import datetime de
sqlalchemy import DateTime

class User(Base):
    __tablename__ = 'users'

    user_id = Column(Integer(), primary_key=True)
    username = Column(String(15), nullable=False, unique=True) ①
    email_address = Column(String(255), nullable=False) phone =
    Column(String(20) , nullable=False) senha = Column(String(25),
    nullable=False) created_on = Column(DateTime(), default=datetime.now)
    updated_on = Column(DateTime(), default=datetime.now,
    onupdate=datetime. agora) ② ③
```

- ➊ Aqui estamos tornando esta coluna obrigatória (nullable=False) e exigindo que os valores sejam exclusivos.
- ➋ O padrão define esta coluna para a hora atual se uma data não for especificada.
- ➌ Usar onupdate aqui redefinirá esta coluna para a hora atual toda vez que qualquer parte do registro for atualizada.

Chaves, Restrições e Índices Em

“Chaves e Restrições” na página 6, discutimos que é possível definir chaves e restrições no construtor de tabela, além de poder fazê-lo nas próprias colunas, como mostrado anteriormente. No entanto, ao usar o ORM, estamos construindo classes e não usando o construtor de tabelas. No ORM, eles podem ser adicionados usando o atributo `__table_args__` em nossa classe. `__table_args__` espera obter uma tupla de argumentos de tabela adicionais, conforme mostrado aqui:

```
class SomeDataClass(Base):
    __tablename__ = 'somedatabable'
    __table_args__ = (ForeignKeyConstraint(['id'], ['other_table.id']),
                    CheckConstraint(custo_unidade >= 0,00',
                                    nome='custo_unidade_positivo'))
```

A sintaxe é a mesma de quando os usamos em um construtor `Table()`. Isso também se aplica ao que você aprendeu em “Índices” na página 7.

Nesta seção, abordamos como definir algumas tabelas e seus esquemas com o ORM. Outra parte importante da definição de modelos de dados é estabelecer relacionamentos entre várias tabelas e objetos.

Relacionamentos

Tabelas e objetos relacionados são outro local onde há diferenças entre SQLAlchemy Core e ORM. O ORM usa uma coluna ForeignKey semelhante para restringir e vincular os objetos; no entanto, ele também usa uma diretiva de relacionamento para fornecer uma propriedade que pode ser usada para acessar o objeto relacionado. Isso adiciona algum uso extra de banco de dados e sobrecarga ao usar o ORM; no entanto, as vantagens de ter essa capacidade superam em muito as desvantagens. Eu adoraria fornecer uma estimativa aproximada da sobrecarga adicional, mas ela varia de acordo com seus modelos de dados e como você os usa.

Na maioria dos casos, nem vale a pena considerar. O [Exemplo 6-3](#) mostra como definir um relacionamento usando os métodos relacionamento e backref .

Exemplo 6-3. Tabela com um relacionamento

```
da importação sqlalchemy ForeignKey, booleano
da relação de importação sqlalchemy.orm , backref ①

class Order(Base):
    __tablename__ = 'pedidos'

    order_id = Column(Integer(), primary_key=True) user_id
    = Column(Integer(), ForeignKey('users.user_id')) enviado =
    Column(Boolean(), default=False) ②

    usuário = relacionamento("Usuário", backref=backref('pedidos', order_by=order_id)) ③
```

- ➊ Observe como importamos os métodos de relacionamento e backref do sqlalchemy.orm.
- ➋ Estamos definindo uma ForeignKey assim como fizemos com SQLAlchemy Core.
- ➌ Isso estabelece uma relação um-para-muitos.

Observando o relacionamento do usuário definido na classe Order , ele estabelece um relacionamento um para muitos com a classe User . Podemos obter o Usuário relacionado a este Pedido acessando a propriedade do usuário . Esse relacionamento também estabelece uma propriedade orders na classe User por meio do argumento de palavra-chave backref , que é ordenado pelo order_id. A diretiva de relacionamento precisa de uma classe de destino para o relacionamento e, opcionalmente, pode incluir uma referência inversa a ser adicionada à classe de destino. SQLAlchemy sabe usar a ForeignKey que definimos que corresponde à classe que definimos no relacionamento. No exemplo anterior, a ForeignKey(users.user_id), que tem a coluna user_id da tabela users , mapeia para a classe User por meio do atributo __tablename__ de usuários e forma o relacionamento.

Também é possível estabelecer um relacionamento um-para-um: no [Exemplo 6-4](#), a classe LineItem tem um relacionamento um-para-um com a classe Cookie . O argumento de palavra-chave uselist=False o define como um relacionamento um-para-um. Também usamos uma referência inversa mais simples, pois não nos importamos em controlar o pedido.

Exemplo 6-4. Mais tabelas com relacionamentos

```
class LineItem(Base):
    __tablename__ = 'line_items'

    line_item_id = Column(Integer(), primary_key=True) order_id =
    = Column(Integer(), ForeignKey('orders.order_id')) cookie_id =
    Column(Integer(), ForeignKey('cookies.cookie_id')) quantidade =
    Column( Integer()) extended_cost = Column(Numeric(12, 2))

    pedido = relacionamento("Pedido", backref=backref('line_items',
                                                       order_by=line_item_id))
    cookie = relacionamento("Cookie", uselist=False) ❶
```

- ❶ Isso estabelece uma relação de um para um.

Esses dois exemplos são bons pontos de partida para os relacionamentos; no entanto, os relacionamentos são uma parte extremamente poderosa do ORM. Nós mal arranhemos a superfície do que pode ser feito com relacionamentos, e vamos explorá-los mais no Cookbook no [Capítulo 14](#). Com nossas classes todas definidas e nossos relacionamentos estabelecidos, estamos prontos para criar nossas tabelas no banco de dados.

Persistindo o esquema

Para criar nossas tabelas de banco de dados, vamos usar o método create_all nos metadados dentro de nossa instância Base . Requer uma instância de um mecanismo, assim como no SQLAlchemy Core:

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///memory:')

Base.metadata.create_all(engine)
```

Antes de discutirmos como trabalhar com nossas classes e tabelas por meio do ORM, precisamos aprender como as sessões são usadas pelo ORM. É disso que trata o próximo capítulo.

CAPÍTULO 7

Trabalhando com dados via SQLAlchemy ORM

Agora que definimos as classes que representam as tabelas em nosso banco de dados e as persistimos, vamos começar a trabalhar com dados por meio dessas classes. Neste capítulo, veremos como inserir, recuperar, atualizar e excluir dados. Em seguida, aprenderemos como classificar e agrupar esses dados e observar como os relacionamentos funcionam. Começaremos aprendendo sobre a sessão SQLAlchemy, uma das partes mais importantes do ORM SQLAlchemy.

A sessão

A sessão é a maneira como o SQLAlchemy ORM interage com o banco de dados. Ele envolve uma conexão de banco de dados por meio de um mecanismo e fornece um mapa de identidade para objetos que você carrega por meio da sessão ou associa à sessão. O mapa de identidade é uma estrutura de dados semelhante a um cache que contém uma lista exclusiva de objetos determinados pela tabela e chave primária do objeto. Uma sessão também encerra uma transação, e essa transação ficará aberta até que a sessão seja confirmada ou revertida, muito semelhante ao processo descrito em “[Transações](#) na página 43.”

Para criar uma nova sessão, SQLAlchemy fornece a classe sessionmaker para garantir que as sessões possam ser criadas com os mesmos parâmetros em todo o aplicativo. Ele faz isso criando uma classe Session que foi configurada de acordo com os argumentos passados para a fábrica do criador de sessões . A fábrica do criador de sessões deve ser usada apenas uma vez no escopo global do aplicativo e tratada como uma definição de configuração. Vamos criar uma nova sessão associada a um banco de dados SQLite na memória:

```
do sqlalchemy import create_engine
sqlalchemy.orm import sessionmaker
engine = create_engine('sqlite:///memory:')
```

```
Sessão = criador de sessões(bind=motor) ❷
```

```
sessão = Sessão() ❸
```

- ❶ Importa a classe do criador de sessões .
- ❷ Define uma classe Session com a configuração de ligação fornecida pelo sessionmaker.
- ❸ Cria uma sessão para nosso uso de nossa classe Session gerada .

Agora temos uma sessão que podemos usar para interagir com o banco de dados. Embora a sessão tenha tudo o que precisa para se conectar ao banco de dados, ela não se conectará até que dermos algumas instruções que exijam isso. Continuaremos a usar as classes que criamos no [Capítulo 6](#) para nossos exemplos no capítulo. Vamos adicionar alguns métodos `__repr__` para facilitar a visualização e recriação de instâncias de objetos; no entanto, esses métodos não são necessários:

```
de datetime importação datetime
```

```
da importação sqlalchemy (Tabela, Coluna, Inteiro, Numérico, String, DateTime,
Foreignkey)
de sqlalchemy.ext.declarative import declarative_base do
relacionamento de importação sqlalchemy.orm , backref
```

```
Base = declarative_base()
```

```
class Cookie(Base):
    __tablename__ = 'cookies'

    cookie_id = Column(Integer(), primary_key=True)
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255))
    cookie_sku =
    Column(String(55))
    quantidade = Column(Integer())
    unit_cost = Column(Numeric(12, 2))
```

```
def __repr__(self): ❶
    return "Cookie(cookie_name='{self.cookie_name}', " \
           "cookie_recipe_url='{self.cookie_recipe_url}', " \
           "cookie_sku='{self.cookie_sku}', " \
           "quantidade ={self.quantity}, " \
           "unit_cost={self.unit_cost})".format(self=self)
```

```
class User(Base):
    __tablename__ = 'users'

    user_id = Column(Integer(), primary_key=True)
```

```

nome_de_usuario = Column(String(15), nullable=False, unique=True)
email_address = Column(String(255), nullable=False) phone =
Column(String(20), nullable=False) senha = Column(String(25) ),
nullable=False) created_on = Column(DateTime(), default=datetime.now)
updated_on = Column(DateTime(), default=datetime.now,
onupdate=datetime.now)

def __repr__(self):
    return "User(username='{self.username}', " \
           "email_address='{self.email_address}', " \
           "phone='{self.phone}', " \
           "senha ='{self.password}')".format(self=self)

class Order(Base):
    __tablename__ = 'orders'
    order_id = Column(Integer(), primary_key=True) user_id =
    Column(Integer(), ForeignKey('users.user_id'))

    usuário = relacionamento("Usuário", backref=backref('pedidos', order_by=order_id))

    def __repr__(self):
        return "Order(user_id={self.user_id}, " \
               "enviado={self.shipped})".format(self=self)

class LineItems(Base):
    __tablename__ = 'line_items'
    line_item_id = Column(Integer(), primary_key=True) order_id =
    Column(Integer(), ForeignKey('orders.order_id')) cookie_id =
    Column(Integer(), ForeignKey('cookies.cookie_id')) quantidade =
    Column(Integer()) extended_cost = Column(Numeric(12, 2)) pedido =
    relacionamento("Pedido", backref=backref('line_items', order_by=line_item_id))
    cookie = relacionamento ("Cookie", uselist=False, order_by=id)

    def __repr__(self):
        return "LineItems(order_id={self.order_id}, " \
               "cookie_id={self.cookie_id}, " \
               "quantity={self.quantity}, " \
               "extended_cost={self.extended_cost})".format( self=self)

```

Base.metadata.create_all(motor)

②

- ➊ Um método `__repr__` define como o objeto deve ser representado. Normalmente, é a chamada do construtor necessária para reciar a instância. Isso aparecerá mais tarde em nossa saída de impressão.

- ② Cria as tabelas no banco de dados definido pelo mecanismo.

Com nossas classes recriadas, estamos prontos para começar a aprender a trabalhar com dados em nosso banco de dados e vamos começar a inserir dados.

Inserindo dados

Para criar um novo registro de cookie em nosso banco de dados, inicializamos uma nova instância da classe Cookie que contém os dados desejados. Em seguida, adicionamos essa nova instância do Objeto de cookie para a sessão e confirme a sessão. Isso é ainda mais fácil de fazer porque a herança do declarative_base fornece um construtor padrão que podemos usar ([Exemplo 7-1](#)).

[Exemplo 7-1.](#) Inserindo um único objeto

```
cc_cookie = Cookie(cookie_name='chocolate',  
                  cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',  
                  cookie_sku='CC01', quantidade=12, unit_cost=0.50) session.add(cc_cookie)  
session.commit()  
  
①  
②  
③
```

- ① Criando uma instância da classe Cookie .
- ② Adicionando a instância à sessão.
- ③ Confirmando a sessão.

Quando commit() é chamado na sessão, o cookie é realmente inserido no banco de dados. Ele também atualiza cc_cookie com a chave primária do registro no banco de dados.

Podemos ver isso fazendo o seguinte:

```
print(cc_cookie.cookie_id)
```

1

Vamos tomar um momento para discutir o que acontece com o banco de dados quando executamos o código no [Exemplo 7-1](#). Quando criamos a instância da classe Cookie e a adicionamos à sessão, nada é enviado ao banco de dados. Não é até que chamamos commit() na sessão que qualquer coisa é enviada para o banco de dados. Quando commit() é chamado, acontece o seguinte:

INFO:sqlalchemy.engine.base.Engine:BEGIN (implícito) ①

INFO:sqlalchemy.engine.base.Engine:INSERT INTO cookies (cookie_name,
cookie_recipe_url, cookie_sku, quantidade, unit_cost) VALUES (?, ?, ?, ?, ?) ②

```
INFO:sqlalchemy.engine.base.Engine:(chocolate chip, 'http://
some.aweso.me/cookie/recipe.html', 'CC01', 12, 0.5)
```

③

```
INFO:sqlalchemy.engine.base.Engine:COMMIT
```

④

- ➊ Inicie uma transação.
- ➋ Insira o registro no banco de dados.
- ➌ Os valores para a inserção.
- ➍ Confirme a transação.



Se você quiser ver os detalhes do que está acontecendo aqui, você pode adicionar echo=True à sua instrução create_engine como um argumento de palavra-chave após a string de conexão. Certifique-se de fazer isso apenas para teste e não use echo=True em produção!

Primeiro, uma nova transação é iniciada e o registro é inserido no banco de dados. Em seguida, o mecanismo envia os valores de nossa instrução de inserção. Finalmente, a transação é confirmada no banco de dados e a transação é fechada. Esse método de processamento é frequentemente chamado de padrão Unit of Work.

Em seguida, vamos ver algumas maneiras de inserir vários registros. Se você for fazer trabalho adicional com os objetos depois de inseri-los, você vai querer usar o método mostrado no [Exemplo 7-2](#). Nós simplesmente criamos várias instâncias e, em seguida, adicionamos ambas à sessão antes de confirmar a sessão.

Exemplo 7-2. Várias inserções

```
dcc = Cookie(cookie_name=' chocolate amargo',
            cookie_recipe_url='http://some.aweso.me/cookie/recipe_dark.html',
            cookie_sku='CC02', quantidade=1, custo_unidade=0,75)
```

```
mol = Cookie(cookie_name='melaço',
            cookie_recipe_url='http://some.aweso.me/cookie/recipe_molasses.html',
            cookie_sku='MOL01', quantidade=1, custo_unidade=0,80) session.add(dcc)
session.add(mol) session.flush() print(dcc.cookie_id) print(mol.cookie_id)
```

- ➊
- ➋
- ➌

- ➊ Adiciona o biscoito de chocolate amargo.
- ➋ Acrescenta o biscoito de melado.
- ➌ Limpa a sessão.

Observe que usamos o método `flush()` na sessão em vez de `commit()` no [Exemplo 7-2](#). Um flush é como um `commit`; no entanto, ele não realiza uma confirmação do banco de dados e encerra a transação. Por causa disso, as instâncias `dcc` e `mol` ainda estão conectadas à sessão e podem ser usadas para executar tarefas de banco de dados adicionais sem acionar consultas de banco de dados adicionais. Também emitimos a instrução `session.flush()` uma vez, embora tenhamos adicionado vários registros ao banco de dados. Na verdade, isso resulta em duas instruções de inserção sendo enviadas ao banco de dados dentro de uma única transação.

O [Exemplo 7-2](#) resultará no seguinte:

```
2
3
```

O segundo método de inserir vários registros no banco de dados é ótimo quando você deseja inserir dados na tabela e não precisa realizar trabalho adicional nesses dados. Ao contrário do método que usamos no [Exemplo 7-2](#), o método no [Exemplo 7-3](#) não associa os registros à sessão.

[Exemplo 7-3](#). Inserção em massa de vários registros

```
c1 = Cookie(cookie_name='manteiga de amendoim',
            cookie_recipe_url='http://some.aweso.me/cookie/peanut.html', cookie_sku='PB01',
            quantidade=24, unit_cost=0.25) c2 = Cookie(cookie_name='oatmeal raisin',
            cookie_recipe_url='http://some.okay.me/cookie/raisin.html', cookie_sku='EWW01',
            quantidade=100, unit_cost=1.00)
```

```
session.bulk_save_objects([c1,c2])
❶ session.commit() print(c1.cookie_id)
```

- ❶ Adiciona os cookies a uma lista e usa o método `bulk_save_objects`.

O [Exemplo 7-3](#) não resultará na impressão de nada na tela porque o objeto `c1` não está associado à sessão e não pode atualizar seu `cookie_id` para impressão. Se observarmos o que foi enviado ao banco de dados, podemos ver que apenas uma única instrução de inserção estava na transação:

```
INFO:sqlalchemy.engine.base.Engine:INSERT INTO cookies (cookie_name,
cookie_recipe_url, cookie_sku, quantidade, unit_cost) VALUES (?, ?, ?, ?, ?)
```

❶

```
INFO:sqlalchemy.engine.base.Engine:
    (('manteiga de amendoim', 'http://some.aweso.me/cookie/peanut.html', 'PB01', 24, 0.25), ('passa
    de aveia', 'http://some.okay.me/cookie/raisin.html', 'EWW01', 100, 1.0))
INFO:sqlalchemy.engine.base.Engine:COMMIT
```

① Uma única inserção

O método demonstrado no [Exemplo 7-3](#) é substancialmente mais rápido do que realizar múltiplas adições e inserções individuais como fizemos no [Exemplo 7-2](#). Essa velocidade vem à custa de alguns recursos que obtemos no add e commit normal, como:

- As configurações e ações de relacionamento não são respeitadas ou acionadas.
- Os objetos não estão conectados à sessão.
- A busca de chaves primárias não é feita por padrão.
- Nenhum evento será acionado.

Além de bulk_save_objects, existem métodos adicionais para criar e atualizar objetos por meio de um dicionário, e você pode aprender mais sobre operações em massa e seu desempenho na [documentação do SQLAlchemy](#).



Se você estiver inserindo vários registros e não precisar acessar relacionamentos ou a chave primária inserida, use bulk_save_objects ou seus métodos relacionados. Isso é especialmente verdadeiro se você estiver ingerindo dados de uma fonte de dados externa, como um CSV ou um grande documento JSON com matrizes aninhadas.

Agora que temos alguns dados em nossa tabela de cookies , vamos aprender como consultar as tabelas e recuperar esses dados.

Consultando dados

Para começar a construir uma consulta, começamos usando o método query() na instância da sessão. Inicialmente, vamos selecionar todos os registros em nossa tabela de cookies passando a classe Cookie para o método query() , conforme mostrado no [Exemplo 7-4](#).

Exemplo 7-4. Pegue todos os biscoitos

```
cookies = session.query(Cookie).all()
print(cookies)
```

①

- ①** Retorna uma lista de instâncias de Cookie que representam todos os registros na tabela de cookies .

O Exemplo 7-4 produzirá o seguinte:

```
[Cookie(cookie_name='chocolate',
       cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
       cookie_sku='CC01', quantidade=12, unit_cost=0.50),
Cookie(cookie_name='dark chocolate chip',
       cookie_recipe_url='http://some.aweso.me/cookie/recipe_dark.html',
       cookie_sku='CC02', quantidade=1, unit_cost=0.75), Cookie(cookie_name='molasses',
       cookie_recipe_url='http:// some.aweso.me/cookie/recipe_molasses.html',
       cookie_sku='MOL01', quantidade=1, custo_unidade=0.80),

Cookie(cookie_name=' manteiga de amendoim ',
       cookie_recipe_url='http://some.aweso.me/cookie/peanut.html',
       cookie_sku='PB01', quantidade=24, custo_unidade=0.25),
Cookie(cookie_name='oatmeal raisin',
       cookie_recipe_url='http://some.okay.me/cookie/raisin.html',
       cookie_sku='EWW01', quantidade=100, unit_cost=1.00)
]
```

Como o valor retornado é uma lista de objetos, podemos usar esses objetos normalmente. Esses objetos estão conectados à sessão, o que significa que podemos alterá-los ou excluí-los e persistir essa alteração no banco de dados, conforme mostrado mais adiante neste capítulo.

Além de podermos recuperar uma lista de objetos de uma só vez, podemos usar a consulta como um iterável, conforme mostrado no Exemplo 7-5.

Exemplo 7-5. Usando a consulta como um iterável

```
para cookie em session.query(Cookie):
    print(cookie) ❶
```

❶ Não anexamos um all() ao usar um iterável.

Usar a abordagem iterável nos permite interagir com cada objeto de registro individualmente, liberá-lo e obter o próximo objeto.

O Exemplo 7-5 exibiria a seguinte saída:

```
Cookie(cookie_name='chocolate',
       cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
       cookie_sku='CC01', quantidade=12, unit_cost=0.50),
Cookie(cookie_name='dark chocolate chip',
       cookie_recipe_url='http://some.aweso.me/cookie/recipe_dark.html',
       cookie_sku='CC02', quantidade=1, unit_cost=0.75), Cookie(cookie_name='molasses',
       cookie_recipe_url='http:// some.aweso.me/cookie/recipe_molasses.html',
       cookie_sku='MOL01', quantidade=1, custo_unidade=0.80),

Cookie(cookie_name=' manteiga de amendoim ',
       cookie_recipe_url='http://some.aweso.me/cookie/peanut.html',
```

```

        cookie_sku='PB01', quantidade=24, custo_unidade=0,25),
Cookie(cookie_name='oatmeal raisin',
       cookie_recipe_url='http://some.okay.me/cookie/raisin.html',
       cookie_sku='EWW01', quantidade=100, unit_cost=1.00)

```

Além de usar a consulta como iterável ou chamar o método `all()` , existem muitas outras maneiras de acessar os dados. Você pode usar os seguintes métodos para buscar resultados:

`primeiro()`

Retorna o primeiro objeto de registro, se houver.

`1()`

Consulta todas as linhas e gera uma exceção se algo diferente de um único resultado for retornado.

`escalar()`

Retorna o primeiro elemento do primeiro resultado, Nenhum se não houver resultado ou um erro se houver mais de um resultado.

Dicas para um bom código de produção

Ao escrever o código de produção, você deve seguir estas diretrizes:

- Use a versão iterável da consulta sobre o método `all()` . É mais eficiente em termos de memória do que lidar com uma lista completa de objetos e tendemos a operar os dados um registro por vez de qualquer maneira.
- Para obter um único registro, use o método `first()` (em vez de `one()` ou `scalar()`) porque é mais claro para nossos colegas codificadores. A única exceção a isso é quando você deve garantir que haja um e apenas um resultado de uma consulta; nesse caso, use `one()`.
- Use o método `scalar()` com moderação, pois ele gera erros se uma consulta retornar mais de uma linha com uma coluna. Em uma consulta que seleciona registros inteiros, ela retornará o objeto de registro inteiro, o que pode ser confuso e causar erros.

Toda vez que consultamos o banco de dados nos exemplos anteriores, todas as colunas eram retornadas para cada registro. Muitas vezes, precisamos apenas de uma parte dessas colunas para realizar nosso trabalho. Se os dados nessas colunas extras forem grandes, isso poderá fazer com que nossos aplicativos fiquem lentos e consumam muito mais memória do que deveriam. SQLAlchemy não adiciona muita sobrecarga às consultas ou objetos; no entanto, a contabilização dos dados que você obtém de uma consulta geralmente é o primeiro lugar para verificar se uma consulta está consumindo muita memória. Vejamos como limitar as colunas retornadas em uma consulta.

Controlando as Colunas na Consulta

Para limitar os campos que são retornados de uma consulta, precisamos passar as colunas que queremos no construtor do método query() separadas por colunas. Por exemplo, talvez você queira executar uma consulta que retorne apenas o nome e a quantidade de cookies, conforme mostrado no [Exemplo 7-6](#).

Exemplo 7-6. Selecione apenas cookie_name e quantidade

```
print(session.query(Cookie.cookie_name, Cookie.quantity).first())
```

➊

- ➊ Seleciona o cookie_name e a quantidade da tabela de cookies e retorna o primeiro resultado.

Quando executamos o [Exemplo 7-6](#), ele gera o seguinte:

```
(u'chocolate', 12),
```

A saída de uma consulta onde fornecemos os nomes das colunas é uma tupla desses valores das colunas.

Agora que podemos construir uma instrução select simples, vamos ver outras coisas que podemos fazer para alterar como os resultados são retornados em uma consulta. Começaremos alterando a ordem na qual os resultados são retornados.

Ordenação

Se você observar todos os resultados do [Exemplo 7-6](#) em vez de apenas o primeiro registro, verá que os dados não estão realmente em nenhuma ordem específica. No entanto, se quisermos que a lista seja retornada em uma ordem específica, podemos encadear uma instrução order_by() ao nosso select, conforme mostrado no [Exemplo 7-7](#). Neste caso, queremos que os resultados sejam ordenados pela quantidade de cookies que temos em mãos.

Exemplo 7-7. Ordem por quantidade crescente

```
para cookie em session.query(Cookie).order_by(Cookie.quantity):
    print('{:3} - {}'.format(cookie.quantity, cookie.cookie_name))
```

O [Exemplo 7-7](#) imprimirá a seguinte saída:

```
1 - pepitas de chocolate
      preto 1 - melado 12 - pepitas
      de chocolate 24 - pasta de
      amendoim 100 - passas de
      aveia
```

Se você quiser classificar em ordem reversa ou decrescente, use a instrução `desc()`. A função `desc()` envolve a coluna específica que você deseja classificar de maneira decrescente, conforme mostrado no [Exemplo 7-8](#).

Exemplo 7-8. Ordem por quantidade decrescente

```
de sqlalchemy import desc
para cookie em session.query(Cookie).order_by(desc(Cookie.quantity)): print('{:3} - ❶
{}'.format(cookie.quantity, cookie.cookie_name))
```

- ❶ Envolvemos a coluna que queremos classificar de forma decrescente na função `desc()`.



A função `desc()` também pode ser usada como um método em um objeto de coluna, como `Cookie.quantity.desc()`. No entanto, isso pode ser um pouco mais confuso para ler em instruções longas e, portanto, sempre uso `desc()` como uma função.

Também é possível limitar o número de resultados retornados se precisarmos apenas de um certo número deles para nossa aplicação. A seção a seguir explica como.

Limitando

Nos exemplos anteriores, usamos o método `first()` para obter apenas uma única linha de volta. Enquanto nossa `query()` nos deu a única linha que pedimos, a query real passou e acessou todos os resultados, não apenas o único registro. Se quisermos limitar a consulta, podemos usar a notação de fatia de matriz para emitir uma instrução de limite como parte de nossa consulta. Por exemplo, suponha que você só tenha tempo para assar dois lotes de biscoitos e queira saber quais dois tipos de biscoitos você deve fazer. Você pode usar nossa consulta ordenada anterior e adicionar uma instrução de limite para retornar os dois tipos de cookies que mais precisam ser reabastecidos. O [Exemplo 7-9](#) mostra como isso pode ser feito.

Exemplo 7-9. Dois menos inventários de cookies

```
query = session.query(Cookie).order_by(Cookie.quantity)[:2]
print([result.cookie_name for result in query]) ❶
```

- ❶ Isso executa a consulta e divide a lista retornada. Isso pode ser muito ineficiente com um grande conjunto de resultados.

A saída do [Exemplo 7-9](#) é assim:

`[u'chocolate amargo', u'melaço']`

Além de usar a notação de fatia de array, também é possível usar o `limit()` demonstração.

Exemplo 7-10. Dois menos inventários de cookies com limite

```
query = session.query(Cookie).order_by(Cookie.quantity).limit(2)
print([result.cookie_name for result in query])
```

Agora que você sabe que tipo de biscoitos você precisa assar, provavelmente está começando a ficar curioso sobre quantos biscoitos restam em seu inventário. Muitos bancos de dados incluem funções SQL projetadas para disponibilizar certas operações diretamente no servidor de banco de dados, como SUM; vamos explorar como usar essas funções.

Funções e rótulos SQL integrados

SQLAlchemy também pode aproveitar as funções SQL encontradas no banco de dados de back-end. Duas funções de banco de dados muito usadas são SUM() e COUNT(). Para usar essas funções, precisamos importar o gerador do módulo sqlalchemy.func que as disponibiliza. Essas funções são agrupadas em torno da(s) coluna(s) em que estão operando. Assim, para obter uma contagem total de cookies, você usaria algo como o [Exemplo 7-11](#).

Exemplo 7-11. Resumindo nossos cookies

```
from sqlalchemy import func
inv_count = session.query(func.sum(Cookie.quantity)).scalar() print(inv_count)❶
```

- ❶ Observe o uso de escalar, que retornará apenas a coluna mais à esquerda no primeiro registro.



Eu costumo sempre importar o gerador de módulo func , pois importar sum diretamente pode causar problemas e confusão com o Python construído na função soma

Quando executamos o [Exemplo 7-11](#), resulta em:

138

Agora vamos usar a função count para ver quantos registros de inventário de cookies temos em nossa tabela de cookies ([Exemplo 7-12](#)).

Exemplo 7-12. Contando nossos registros de inventário

```
rec_count = session.query(func.count(Cookie.cookie_name)).first() print(rec_count)
```

Ao contrário do [Exemplo 7-11](#), onde usamos escalar e obtivemos um único valor, o [Exemplo 7-12](#) resulta em uma tupla para usarmos, pois usamos o primeiro método em vez de escalar:

(5,)

O uso de funções como count() e sum() acabará retornando tuplas ou resultados com nomes de coluna como count_1. Esses tipos de retorno muitas vezes não são o que queremos.

Além disso, se tivermos várias contagens em uma consulta, teríamos que saber o número da ocorrência na instrução e incorporá-lo ao nome da coluna, de modo que a quarta função count() seria count_4. Isso simplesmente não é tão explícito e claro quanto deveríamos ser em nossa nomenclatura, especialmente quando cercado por outro código Python.

Felizmente, SQLAlchemy fornece uma maneira de corrigir isso por meio da função label().

O [Exemplo 7-13](#) executa a mesma consulta que o [Exemplo 7-12](#); no entanto, ele usa label() para nos dar um nome mais útil para acessar essa coluna.

Exemplo 7-13. Renomeando nossa coluna de contagem

```
rec_count = session.query(func.count(Cookie.cookie_name)
                           \.label('inventory_count')).first()
print(rec_count.keys())
print(rec_count.inventory_count)
```

❶

❶ Eu usei a função label() no objeto de coluna que quero alterar.

O [Exemplo 7-13](#) resulta em:

[u'inventory_count'] 5

Vimos exemplos de como restringir as colunas ou o número de linhas retornadas do banco de dados, então agora é hora de aprender sobre consultas que filtram dados com base em critérios que especificamos.

Filtragem A

filtragem de consultas é feita anexando instruções filter() à nossa consulta. Uma cláusula filter() típica tem uma coluna, um operador e um valor ou coluna. É possível encadear várias cláusulas filter() juntas ou separar por vírgula várias expressões ClauseElement em um único filtro, e elas agirão como ANDs em instruções SQL tradicionais. No [Exemplo 7-14](#), encontraremos um biscoito chamado “chocolate”.

Exemplo 7-14. Filtrando por nome de cookie com filtro

```
record = session.query(Cookie).filter(Cookie.cookie_name == 'chocolate').first() print(record)
```

O Exemplo 7-14 imprime o registro de biscoito de chocolate:

```
Cookie(cookie_name='chocolate',
       cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
       cookie_sku='CC01', quantidade=12, unit_cost=0,50)
```

Existe também um método filter_by() que funciona de forma semelhante ao método filter() , exceto que em vez de fornecer explicitamente a classe como parte da expressão de filtro, ele usa expressões de palavra-chave de atributo da entidade primária da consulta ou da última entidade que foi unida a a declaração. Ele também usa uma atribuição de palavra-chave em vez de um booleano. O Exemplo 7-15 executa exatamente a mesma consulta que o Exemplo 7-14.

Exemplo 7-15. Filtrando por nome de cookie com filter_by

```
registro = session.query(Cookie).filter_by(cookie_name='chocolate').first() print(record)
```

Também podemos usar uma instrução where para localizar todos os nomes de cookies que contêm a palavra “chocolate”, conforme mostrado no Exemplo 7-16.

Exemplo 7-16. Encontrar nomes com “chocolate” neles

```
query = session.query(Cookie).filter(Cookie.cookie_name.like('%chocolate%')) para registro
na consulta: print(record.cookie_name)
```

O código no Exemplo 7-16 retornará:

```
pepitas de chocolate
pepitas de chocolate escuro
```

No Exemplo 7-16, estamos usando a coluna Cookie.cookie_name dentro de uma instrução de filtro como um tipo de ClauseElement para filtrar nossos resultados e estamos aproveitando o método like() que está disponível em ClauseElements. Existem muitos outros métodos disponíveis, listados na Tabela 2-1.

Se não usarmos um dos métodos ClauseElement , teremos um operador em nossas cláusulas de filtro. A maioria dos operadores funciona como você poderia esperar, mas como a seção a seguir explica, existem algumas diferenças.

Operadores

Até agora, exploramos apenas situações em que uma coluna era igual a um valor ou usamos um dos métodos ClauseElement como like(); no entanto, também podemos usar muitos outros operadores comuns para filtrar dados. SQLAlchemy fornece sobrecarga para a maioria dos operadores padrão do Python. Isso inclui todos os operadores de comparação padrão (==, !=, <, >, <=, >=), que agem exatamente como você esperaria em uma instrução Python. O operador == também recebe uma sobrecarga adicional quando comparado a None,

que o converte em uma instrução IS NULL . Operadores aritméticos (+, -, *, / e %) também são suportados com recursos adicionais para concatenação de strings independente de banco de dados, conforme mostrado no [Exemplo 7-17](#).

Exemplo 7-17. Concatenação de strings com +

```
resultados = session.query(Cookie.cookie_name, 'SKU-' + Cookie.cookie_sku).all() for row in
results: print(row)
```

O Exemplo 7-17 resulta em:

```
('chocolate chip', 'SKU-CC01')
('chocolate chip escuro', 'SKU-CC02')
('melaço', 'SKU-MOL01') ('manteiga de
amendoim', 'SKU-PB01') (' passas de
aveia', 'SKU-EWW01')
```

Outro uso comum de operadores é calcular valores de várias colunas.

Você fará isso com frequência em aplicativos e relatórios que lidam com dados ou estatísticas.

O Exemplo 7-18 mostra um cálculo de valor de estoque comum.

Exemplo 7-18. Valor do inventário por cookie

```
from sqlalchemy import cast      ❶
query = session.query(Cookie.cookie_name,
                      cast((Cookie.quantity * Cookie.unit_cost),
                           Numeric(12,2)).label('inv_cost'))           ❷
para resultado na
    consulta: print('{0} - {1}'.format(resultado.cookie_name, resultado.inv_cost))
```

- ❶ cast é uma função que nos permite converter tipos. Neste caso, obteremos resultados como 6,0000000000, portanto, ao convertê-lo, podemos fazer com que pareça moeda. Também é possível realizar a mesma tarefa em Python com print('{0} - {1:.2f}'.format(row.cookie_name, row.inv_cost)).
- ❷ Estamos usando a função label() para renomear a coluna. Sem essa renomeação, a coluna não estaria listada nas chaves do objeto resultado , pois a operação não tem nome.

O Exemplo 7-18 resulta em:

```
pepitas de chocolate -
6,00 pepitas de chocolate negro
- 0,75 melado - 0,80 pasta de
amendoim - 6,00 aveia passas
- 100,00
```

Se precisarmos combinar as instruções where, podemos usar alguns métodos diferentes. Um desses métodos é conhecido como operadores booleanos.

Operadores booleanos

SQLAlchemy também permite os operadores booleanos SQL AND, OR e NOT por meio dos operadores lógicos bit a bit (&, | e ~). Cuidado especial deve ser tomado ao usar as sobrecargas AND, OR e NOT devido às regras de precedência do operador Python. Por exemplo, & liga mais de perto do que <, então quando você escreve A < B & C < D, o que você está realmente escrevendo é A < (B&C) < D, quando você provavelmente pretendia obter (A < B) & (C) <D).

Muitas vezes, queremos encadear várias cláusulas where juntas de maneiras inclusivas e excludentes; isso deve ser feito através de conjunções.

Conjunções

Embora seja possível encadear várias cláusulas filter() juntas, geralmente é mais legível e funcional usar conjunções para obter o efeito desejado. Também prefiro usar conjunções em vez de operadores booleanos, pois as conjunções tornarão seu código mais expressivo. As conjunções em SQLAlchemy são and_(), or_() e not_(). Eles têm sublinhados para separá-los das palavras-chave internas. Portanto, se quisermos obter uma lista de cookies com um custo inferior a um valor e acima de uma determinada quantidade, podemos usar o código mostrado no [Exemplo 7-19](#).

Exemplo 7-19. Usando filtro com várias expressões ClauseElement para executar um AND

```
query = session.query(Cookie).filter( Cookie.quantity
> 23, Cookie.unit_cost < 0,40

) para resultado na
consulta: print(result.cookie_name)
```

A função or_() funciona como o oposto de and_() e inclui resultados que correspondem a qualquer uma das cláusulas fornecidas. Se quisermos pesquisar em nosso inventário por tipos de cookies que temos entre 10 e 50 em estoque ou onde o nome contém chip, podemos usar o código mostrado no [Exemplo 7-20](#).

Exemplo 7-20. Usando a conjunção or()

```
from sqlalchemy import and_, or_, not_
query = session.query (Cookie).filter(
or_
( Cookie.quantity.between(10, 50),
Cookie.cookie_name.contains('chip')
```

)

) para resultado na
 consulta: print(result.cookie_name)

O Exemplo 7-20 resulta em:

chocolate amargo
 com gotas de chocolate
 manteiga de amendoim

A função `not_()` funciona de maneira semelhante a outras conjunções e é usada para selecionar registros onde um registro não corresponde à cláusula fornecida.

Agora que podemos consultar dados confortavelmente, estamos prontos para atualizar os dados existentes.

Atualizando dados

Muito parecido com o método `insert` que usamos anteriormente, há também um método de atualização com sintaxe quase idêntica às inserções, exceto que eles podem especificar uma cláusula `where` que indica quais linhas devem ser atualizadas. Assim como as instruções de inserção, as instruções de atualização podem ser criadas usando a função `update()` ou o método `update()` na tabela que está sendo atualizada. Você pode atualizar todas as linhas em uma tabela deixando de fora a cláusula `where`.

Por exemplo, suponha que você terminou de assar os biscoitos de chocolate que precisávamos para nosso inventário. No Exemplo 7-21, vamos adicioná-los ao nosso inventário existente com uma consulta de atualização e, em seguida, verificar quantos temos atualmente em estoque.

Exemplo 7-21. Atualizando dados via objeto

```
query = session.query(Cookie)      ❶
cc_cookie = query.filter(Cookie.cookie_name == "chocolate").first() cc_cookie.quantity = ❷
cc_cookie.quantity + 120 session.commit() print(cc_cookie.quantity)
```

❶ As linhas 1 e 2 estão usando o método generativo para construir nossa declaração.

❷ Estamos consultando para obter o objeto aqui; no entanto, se você já o tiver, poderá editá-lo diretamente sem consultá-lo novamente.

O Exemplo 7-21 retorna:

132

Também é possível atualizar os dados no local sem ter o objeto originalmente (Exemplo 7-22).

Exemplo 7-22. Atualizando dados no local

```
query = session.query(Cookie)
query = query.filter(Cookie.cookie_name == "chocolate")
query.update({Cookie.quantity: Cookie.quantity - 20})  
①
cc_cookie = query.first()  
②
print(cc_cookie.quantity)
```

- ① O método update() faz com que o registro seja atualizado fora da sessão e retorna o número de linhas atualizadas.
- ② Estamos reutilizando a consulta aqui porque ela tem os mesmos critérios de seleção que precisamos.

O Exemplo 7-22 retorna:

112

Além de atualizar os dados, em algum momento vamos querer remover os dados de nossas tabelas. A seção a seguir explica como fazer isso.

Excluindo dados

Para criar uma instrução delete, você pode usar a função delete() ou o método delete() na tabela da qual você está excluindo os dados. Ao contrário de insert() e update(), delete() não aceita parâmetros de valores, apenas uma cláusula where opcional (omitar a cláusula where excluirá todas as linhas da tabela). Veja o Exemplo 7-23.

Exemplo 7-23. Excluindo dados

```
query = session.query(Cookie)
query = query.filter(Cookie.cookie_name == "chocolate amargo") dcc_cookie =
query.one() session.delete(dcc_cookie) session.commit() dcc_cookie = query.first()
print(dcc_cookie)
```

O Exemplo 7-23 retorna:

Nenhum

Também é possível excluir dados no local sem ter o objeto (Exemplo 7-24).

Exemplo 7-24. Excluindo dados

```
query = session.query(Cookie)
query = query.filter(Cookie.cookie_name == "melaço") query.delete()
```

```
mol_cookie = query.first()
print(mol_cookie)
```

O Exemplo 7-24 retorna:

Nenhum

OK, neste ponto, vamos carregar alguns dados usando o que já aprendemos para as tabelas users, orders e line_items . Você pode copiar o código mostrado aqui, mas também deve tirar um momento para brincar com as diferentes maneiras de inserir os dados:

```
cookiemon = User(username='cookiemon',
                  email_address='mon@cookie.com',
                  phone='111-111-1111',
                  password='password') cakeeater =
User(username='cakeeater',
      email_address='cakeeater@cake.com',
      phone='222-222-2222', password='password')
pieperson = User(username='pieperson',
                  email_address='person@pie.com',
                  phone='333-333-3333',
                  password='password')
session.add(cookiemon) session.add(cakeeater)
session.add(pieperson) session.commit()
```

Agora que temos clientes, podemos começar a inserir seus pedidos e itens de linha no sistema também. Vamos aproveitar a relação entre esses dois objetos ao inseri-los na tabela. Se tivermos um objeto que queremos associar a outro, podemos fazer isso atribuindo-o à propriedade de relacionamento como faríamos com qualquer outra propriedade. Vamos ver isso em ação algumas vezes no Exemplo 7-25.

Exemplo 7-25. Adicionando objetos relacionados

```
o1 = Order()
o1.user = cookiemon    ❶
sessão.adicionar(o1)

cc = session.query(Cookie).filter(Cookie.cookie_name ==
                                  "chocolate ").one()
line1 = LineItem(cookie=cc, quantidade=2, extended_cost=1,00)    ❷

pb = session.query(Cookie).filter(Cookie.cookie_name == " manteiga
                                  de amendoim ").one()
line2 = LineItem(quantity=12, extended_cost=3,00)    ❸
line2.cookie = pb line2.order = o1

o1.line_items.append(line1)    ❹
```

```
o1.line_items.append(line2)
```

```
session.commit()
```

- ➊ Define cookiemon como o usuário que fez o pedido.
- ➋ Cria um item de linha para o pedido e o associa ao cookie.
- ➌ Cria um item de linha uma parte de cada vez.
- ➍ Adiciona o item de linha ao pedido por meio do relacionamento.

No [Exemplo 7-25](#), criamos uma instância Order vazia e definimos sua propriedade user para a instância cookiemon . Em seguida, adicionamos à sessão. Em seguida, consultamos o cookie de chocolate e criamos um LineItem e configuramos o cookie para ser o cookie de chocolate que acabamos de consultar. Repetimos esse processo para o segundo item de linha do pedido; no entanto, construímos-o aos poucos. Por fim, adicionamos os itens de linha ao pedido e o confirmamos.

Antes de prosseguirmos, vamos adicionar mais um pedido para outro usuário:

```
o2 = Order()
o2.user = cakeeater

cc = session.query(Cookie).filter(Cookie.cookie_name ==
                                    "chocolate").one() line1 =
LineItem(cookie=cc, quantidade=24, extended_cost=12,00)

aveia = session.query(Cookie).filter(Cookie.cookie_name == "oatmeal
                                      raisin").one()
line2 = LineItem(cookie=aveia, quantidade=6, extended_cost=6,00)

o2.line_items.append(line1)
o2.line_items.append(line2)

session.add(o2) ❶
session.commit()
```

- ➊ Movi esta linha para baixo com os outros acréscimos, e o SQLAlchemy determinou a ordem adequada para criá-los para garantir que fosse bem-sucedido.

No [Capítulo 6](#), você aprendeu como definir ForeignKeys e relacionamentos, mas não os usamos para realizar nenhuma consulta até este ponto. Vamos dar uma olhada nos relacionamentos.

Associações

Agora vamos usar os métodos `join()` e `outerjoin()` para dar uma olhada em como consultar dados relacionados. Por exemplo, para atender ao pedido feito pelo usuário do cookiemon, precisamos determinar quantos de cada tipo de cookie foram pedidos. Isso requer que você use um total de três junções para chegar até o nome dos cookies. O ORM torna essa consulta muito mais fácil do que normalmente seria com SQL bruto ([Exemplo 7-26](#)).

Exemplo 7-26. Usando join para selecionar entre várias tabelas

```
query = session.query(Order.order_id, User.username, User.phone,
                      Cookie.cookie_name, LineItem.quantity,
                      LineItem.extended_cost)
query.join(User).join(LineItem).join(Cookie)
results = query.filter(User.username == 'cookiemon').all()
print(results)
```

- ➊ Diz ao SQLAlchemy para unir os objetos relacionados.

O [Exemplo 7-26](#) produzirá o seguinte:

```
[ (u'1', u'cookiemon', u'111-111-1111', u'chocolate', 2, Decimal('1.00')) (u'1', u'cookiemon',
  u'111-111-1111', u'manteiga de amendoim', 12, Decimal('3.00'))
]
```

Também é útil obter uma contagem de pedidos de todos os usuários, incluindo aqueles que não possuem pedidos presentes. Para fazer isso, temos que usar o método `outerjoin()`, e isso requer um pouco mais de cuidado na ordenação da junção, pois a tabela em que usamos o método `outer join()` será aquela da qual todos os resultados serão retornados ([Exemplo 7-27](#)).

Exemplo 7-27. Usando outerjoin para selecionar entre várias tabelas

```
query = session.query(User.username, func.count(Order.order_id))
query.outerjoin(Order).group_by(User.username) para linha na consulta:
print(row)
```

O [Exemplo 7-27](#) resulta em:

```
(u'cakeeater', 1)
(u'cookiemon', 1)
(u'pieperson', 0)
```

Até agora, usamos e juntamos diferentes tabelas em nossas consultas. No entanto, e se tivermos uma tabela auto-referencial como uma tabela de gerentes e seus relatórios? O ORM

permite estabelecer uma relação que aponta para a mesma mesa; no entanto, precisamos especificar uma opção chamada `remote_side` para tornar o relacionamento muitos para um:

```
class Funcionário(Base):
    __tablename__ = 'funcionários'

    id = Column(Integer(), primary_key=True)
    manager_id = Column(Integer(), ForeignKey('employees.id')) name =
        Column(String(255), nullable=False)

    gerente = relacionamento("Funcionário", backref=backref('relatórios'),
        remote_side=[id]) ①

Base.metadata.create_all(motor)
```

- ① Estabelece um relacionamento de volta para a mesma tabela, especifica o `remote_side` e torna o relacionamento muitos para um.

Vamos adicionar um funcionário e outro funcionário subordinado a ela:

```
marsha = Funcionário(nome='Marsha')
fred = Funcionário(nome='Fred')

marsha.reports.append(fred)

session.add(marsha)
session.commit()
```

Agora se quisermos imprimir os funcionários que se reportam a Marsha, faríamos isso acessando a propriedade `reports` da seguinte forma:

```
para relatório em marsha.reports:
    print(report.name)
```

Também é útil poder agrupar dados quando queremos relatar dados, então vamos analisar isso a seguir.

Agrupamento

Ao usar o agrupamento, você precisa de uma ou mais colunas para agrupar e uma ou mais colunas que faz sentido agregar com contagens, somas, etc., como faria no SQL normal. Vamos obter uma contagem de pedidos por cliente ([Exemplo 7-28](#)).

Exemplo 7-28. Dados de agrupamento

```
query = session.query(User.username, func.count(Order.order_id)) query =
query.outerjoin(Order).group_by(User.username) para linha na consulta:
print(row) ① ②
```

- ➊ Agregação por contagem
- ➋ Agrupamento pela coluna incluída não agregada

O Exemplo 7-28 resulta em:

```
(u'cakeeater', 1)
(u'cookieemon', 1)
(u'pieguy', 0)
```

Mostramos a construção generativa de declarações ao longo dos exemplos anteriores, mas quero focar nela especificamente por um momento.

Encadeamento

Usamos o encadeamento várias vezes ao longo deste capítulo e simplesmente não o reconhecemos diretamente. Onde o encadeamento de consultas é particularmente útil é quando você está aplicando lógica ao construir uma consulta. Então, se quisermos ter uma função que tenha uma lista de pedidos para nós, ela pode se parecer com o Exemplo 7-29.

Exemplo 7-29. Encadeamento

```
def get_orders_by_customer(cust_name):
    query = session.query(Order.order_id, User.username, User.phone,
                          Cookie.cookie_name, LineItem.quantity,
                          LineItem.extended_cost)
    query = query.join(User).join(LineItem).join(Cookie) results =
    query.filter(User.username == cust_name).all() retorna resultados

get_orders_by_customer('comedor de bolo')
```

O Exemplo 7-29 resulta em:

```
[(u'2', u'comedor de bolo', u'222-222-2222', u'chocolate', 24, Decimal('12.00')), (u'2', u'comedor de bolo', u' 222-222-2222', u'oatmeal raisin', 6, Decimal('6.00'))]
```

No entanto, e se quiséssemos obter apenas os pedidos que foram enviados ou ainda não foram enviados? Teríamos que escrever funções adicionais para oferecer suporte a essas opções de filtro adicionais, ou podemos usar condicionais para construir cadeias de consulta. Outra opção que podemos querer é incluir ou não detalhes. Essa capacidade de encadear consultas e cláusulas permite relatórios bastante poderosos e construção de consultas complexas (Exemplo 7-30).

Exemplo 7-30. Encadeamento condicional

```
def get_orders_by_customer(cust_name, enviado=Nenhum, detalhes=False):
    query = session.query(Order.order_id, User.username, User.phone)
```

```

query = query.join(User) se
detalhes:
    query = query.add_columns(Cookie.cookie_name, LineItem.quantity,
                               LineItem.extended_cost)
    query = query.join(LineItem).join(Cookie)
se enviado não for Nenhum:
    query = query.where(Order.shipped == enviado)
resultados = query.filter(User.username == cust_name).all() retornar
resultados

```

get_orders_by_customer('comedor de bolo')	1
get_orders_by_customer('cakeeater', details=True)	2
get_orders_by_customer('cakeeater', enviado=True)	3
get_orders_by_customer('cakeeater', enviado=False)	4
get_orders_by_customer('cakeeater', enviado=False, detalhes=Verdadeiro)	5

- 1** Recebe todos os pedidos.
- 2** Obtém todos os pedidos com detalhes.
- 3** Obtém apenas os pedidos que foram enviados.
- 4** Recebe pedidos que ainda não foram enviados.
- 5** Obtém pedidos que ainda não foram enviados com detalhes.

O Exemplo 7-30 resulta em:

```

[(u'2 ', u'comedor de bolo', u'222-222-2222 ')]
[(u'2 ', u'comedor de bolo', u'222-222-2222', u'chocolate', 24, Decimal('12.00')), (u'2 ', u'comedor de
bolo', u' 222-222-2222', u'oatmeal raisin', 6, Decimal('6.00'))]

```

[]

```

[(u'2 ', u'comedor de bolo', u'222-222-2222 ')]

```

```

[(u'2 ', u'comedor de bolo', u'222-222-2222', u'chocolate', 24, Decimal('12.00')), (u'2 ', u'comedor de
bolo', u' 222-222-2222', u'oatmeal raisin', 6, Decimal('6.00'))]

```

Até agora neste capítulo, usamos o ORM para todos os exemplos; no entanto, também é possível usar SQL literal às vezes. Vamos dar uma olhada.

Consultas brutas

Embora eu raramente use uma instrução SQL bruta completa, geralmente uso pequenos trechos de texto para ajudar a tornar uma consulta mais clara. O Exemplo 7-31 mostra uma cláusula where do SQL bruto usando a função text().

Exemplo 7-31. Consulta de texto parcial

```
from sqlalchemy import text
query = session.query(User).filter(text("username='cookieemon'")) print(query.all())
```

O Exemplo 7-31 resulta em:

```
[User(username='cookieemon', email_address='mon@cookie.com',
      telefone='111-111-1111', senha='senha')]
```

Agora você deve ter uma compreensão de como usar o ORM para trabalhar com dados no SQLAlchemy. Exploramos como criar, ler, atualizar e excluir operações. Este é um bom ponto para parar e explorar um pouco por conta própria. Tente criar mais cookies, pedidos e itens de linha e use cadeias de consulta para agrupá-los por pedido e usuário.

Agora que você explorou um pouco mais e esperançosamente quebrou algo, vamos investigar como reagir a exceções levantadas no SQLAlchemy e como usar transações para agrupar instruções que devem ter sucesso ou falhar como um grupo.

CAPÍTULO 8

Entendendo a Sessão e as Exceções

No capítulo anterior, trabalhamos muito com a sessão e evitamos fazer qualquer coisa que pudesse resultar em uma exceção. Neste capítulo, aprenderemos um pouco mais sobre como nossos objetos e a sessão SQLAlchemy interagem. Concluiremos este capítulo realizando propositalmente algumas ações incorretamente para que possamos ver os tipos de exceções que ocorrem e como devemos responder a elas. Vamos começar aprendendo mais sobre a sessão. Primeiro, vamos configurar um banco de dados SQLite na memória usando as tabelas do [Capítulo 6](#):

```
do sqlalchemy import create_engine do
sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///memory:')

Sessão = criador de sessões(bind=engine)

sessão = Sessão()

de datetime importação datetime

da importação sqlalchemy (Tabela, Coluna, Inteiro, Numérico, String, DateTime,
Foreignkey, Boolean)
da importação sqlalchemy.ext.declarative declarative_base da
relação de importação sqlalchemy.orm , backref

Base = declarative_base()

class Cookie(Base):
    __tablename__ = 'cookies'

    cookie_id = Column(Integer, primary_key=True)
    cookie_name = Column(String(50), index=True)
```

```

cookie_recipe_url = Column(String(255))
cookie_sku = Column(String(55)) quantidade =
Column(Integer()) unit_cost = Column(Numeric(12,
2))

def __init__(self, nome, recipe_url=Nenhum, sku=Nenhum, quantidade=0,
            unit_cost=0.00):
    self.cookie_name = nome
    self.cookie_recipe_url = receita_url
    self.cookie_sku = sku self.quantity =
    quantidade self.unit_cost = unit_cost

def __repr__(self):
    return "Cookie(cookie_name='{self.cookie_name}', " \
           "cookie_recipe_url='{self.cookie_recipe_url}', " \
           "cookie_sku='{self.cookie_sku}', " \
           "quantidade ={self.quantity}, " \
           "unit_cost={self.unit_cost})".format(self=self)

class User(Base):
    __tablename__ = 'users'

    user_id = Column(Integer(), primary_key=True)
    username = Column(String(15), nullable=False, unique=True)
    email_address = Column(String(255), nullable=False) phone =
    Column(String(20), nullable=False) senha = Column(String(25),
    nullable=False) created_on = Column(DateTime(), default=datetime.now)
    updated_on = Column(DateTime(), default=datetime.now,
    onupdate=datetime. agora)

    def __init__(self, username, email_address, phone, password):
        self.username = nome de usuário self.email_address = email_address
        self.phone = telefone self.password = password

    def __repr__(self):
        return "User(username='{self.username}', " \
               "email_address='{self.email_address}', " \
               "phone='{self.phone}', " \
               "senha ='{self.password}')".format(self=self)

class Order(Base):
    __tablename__ = 'orders'
    order_id = Column(Integer(), primary_key=True) user_id
    = Column(Integer(), ForeignKey('users.user_id')) enviado =
    Column(Boolean(), default= Falso)

    usuário = relacionamento("Usuário", backref=backref('pedidos', order_by=order_id))

```

```

def __repr__(self):
    return "Order(user_id={self.user_id}, " \
           "enviado={self.shipped})".format(self=self)

class LineItem(Base):
    __tablename__ = 'line_items'
    line_item_id = Column(Integer(), primary_key=True) order_id =
    = Column(Integer(), ForeignKey('orders.order_id')) cookie_id =
    Column(Integer(), ForeignKey( 'cookies.cookie_id')) quantidade =
    Column(Integer()) extended_cost = Column(Numeric(12, 2))

pedido = relacionamento("Pedido", backref=backref('line_items',
                                                 order_by=line_item_id))
cookie = relacionamento("Cookie", uselist=False)

def __repr__(self):
    return "LineItems(order_id={self.order_id}, " \
           "cookie_id={self.cookie_id}, " \
           "quantity={self.quantity}, " \
           "extended_cost={self.extended_cost})".format( self=self)

Base.metadata.create_all(motor)

```

Agora que temos nosso banco de dados inicial e objetos definidos, estamos prontos para aprender mais sobre como a sessão funciona com objetos.

A sessão SQLAlchemy

Cobrimos um pouco sobre a sessão SQLAlchemy no [Capítulo 7](#); no entanto, aqui quero focar em como nossos objetos de dados e a sessão interagem.

Quando usamos uma consulta para obter um objeto, recebemos de volta um objeto que está conectado a uma sessão. Esse objeto pode se mover por vários estados em relação à sessão.

Estados da Sessão

Compreender os estados de sessão pode ser útil para solucionar problemas de exceções e lidar com comportamentos inesperados. Existem quatro estados possíveis para instâncias de objetos de dados. Essa:

Transiente

A instância não está em sessão e não está no banco de dados.

Pendente

A instância foi adicionada à sessão com `add()`, mas não foi liberada ou confirmada.

Persistente

O objeto em sessão possui um registro correspondente no banco de dados.

Detached

A instância não está mais conectada à sessão, mas possui um registro no banco de dados.

Podemos observar uma instância se mover por esses estados enquanto trabalhamos com ela.

Começaremos criando uma instância de um cookie:

```
cc_cookie = Cookie('chocolate',
    'http://some.aweso.me/cookie/recipe.html', 'CC01', 12,
    0,50)
```

Para ver o estado da instância, podemos usar o poderoso método `inspect()` fornecido pelo SQLAlchemy.

Quando usamos `inspect()` em uma instância, ganhamos acesso a várias informações úteis. Vamos fazer uma inspeção da nossa instância `cc_cookie` :

```
from sqlalchemy import inspect
insp = inspect(cc_cookie)
```

Nesse caso, estamos interessados nas propriedades transitórias, pendentes, persistentes e desanexadas que indicam o estado atual. Vamos percorrer essas propriedades como mostrado no [Exemplo 8-1](#).

Exemplo 8-1. Obtendo o estado da sessão de uma instância

```
for state in ['transient', 'pending', 'persistent', 'detached']: print('{:>10}: {}'.format(state,
    getattr(insp, state)))
```

O [Exemplo 8-1](#) resulta em uma lista de estados e um booleano indicando se essa é a estado do aluguel:

```
transiente: Verdadeiro
pendente: Falso
persistente: Falso
desanexado: Falso
```

Como você pode ver na saída, o estado atual de nossa instância de cookie é transitório, que é o estado em que os objetos recém-criados estão antes de serem liberados ou confirmados no banco de dados. Se adicionarmos `cc_cookie` à sessão atual e executarmos novamente o [Exemplo 8-1](#), obteremos a seguinte saída:

```
transitório: falso
pendente:
Verdadeiro persistente:
Falso desanexado: Falso
```

Agora que nossa instância de cookie foi adicionada à sessão atual, podemos ver que ela foi movida para pendente. Se confirmarmos a sessão e executarmos novamente o [Exemplo 8-1](#), podemos ver que o estado é atualizado para persistente:

```
transiente: Falso
pendente: Falso
persistente: Verdadeiro
destacado: Falso
```

Finalmente, para colocar cc_cookie no estado desanexado, queremos chamar o método expunge() na sessão. Você pode fazer isso se estiver movendo dados de uma sessão para outra. Um caso em que você pode querer mover dados de uma sessão para outra é quando você está arquivando ou consolidando dados de seu banco de dados primário para seu data warehouse:

```
session.expurge(cc_cookie)
```

Se executarmos novamente o [Exemplo 8-1](#) uma última vez após eliminar cc_cookie, veremos que agora ele está em um estado desanexado:

```
transiente: Falso
pendente: Falso
persistente: Falso
desanexado: Verdadeiro
```

Se você estiver usando o método inspect() no código normal, provavelmente desejará usar insp.transient, insp.pending, insp.persistent e insp.detached. Usamos get attr para acessá-los para que pudéssemos percorrer os estados em vez de codificar cada estado individualmente.

Agora que vimos como um objeto se move através dos estados de sessão, vamos ver como podemos usar o inspetor para ver o histórico de uma instância antes de confirmá-la. Primeiro, adicionaremos nosso objeto de volta à sessão e alteraremos o atributo cookie_name :

```
session.add(cc_cookie)
cc_cookie.cookie_name = 'Alterar pepitas de chocolate'
```

Agora vamos usar a propriedade modificada do inspetor para ver se ela mudou:

```
insp.modificado
```

Isso retornará True, e podemos usar a coleção attrs do inspetor para encontrar o que mudou. O [Exemplo 8-2](#) demonstra uma maneira de fazer isso.

Exemplo 8-2. Imprimindo o histórico de atributos alterados

```
para attr, attr_state em insp.attrs.items(): if
    attr_state.history.has_changes(): print('{}: {}')
        '{}'.format(attr, attr_state.value)) print('History: {}\\n'
        '{}'.format(attr_state.history))
```

②

- ➊ Verifica o estado do atributo para ver se a sessão pode encontrar alguma alteração
- ➋ Imprime o objeto de histórico do atributo alterado

Quando executarmos o [Exemplo 8-2](#), retornaremos:

```
cookie_name: Muda o chocolate
Histórico: Histórico(adicionado=['Alterar lascas de chocolate'], inalterado=(), excluído=())
```

A saída do [Exemplo 8-2](#) nos mostra que o cookie_name mudou. Quando olhamos para o registro de histórico desse atributo, ele nos mostra o que foi adicionado ou alterado nesse atributo. Isso nos dá uma boa visão de como os objetos interagem com a sessão.

Agora, vamos investigar como lidar com exceções que surgem ao usar o SQLAlchemy ORM.

Exceções

Existem inúmeras exceções que podem ocorrer no SQLAlchemy, mas vamos nos concentrar em duas em particular: `MultipleResultsFound` e `DetachedInstanceStateError`. Essas exceções são bastante comuns e também fazem parte de um grupo de exceções semelhantes. Ao aprender a lidar com essas exceções, você estará mais bem preparado para lidar com outras exceções que possa encontrar.

Exceção `MultipleResultsFound`

ocorre quando usamos o método de consulta `.one()`, mas obtemos mais de um resultado de volta. Antes de podermos demonstrar essa exceção, precisamos criar outro cookie e salvá-lo no banco de dados:

```
dcc = Cookie('chocolate amargo',
              'http://some.aweso.me/cookie/recipe_dark.html', 'CC02',
              1, 0,75)
session.add(dcc)
session.commit()
```

Agora que temos mais de um cookie no banco de dados, vamos acionar a exceção `MultipleResultsFound` ([Exemplo 8-3](#)).

Exemplo 8-3. Causando uma exceção `MultipleResultsFound`

```
resultados = session.query(Cookie).one()
```

O [Exemplo 8-3](#) resulta em uma exceção porque há dois cookies em nossa fonte de dados que correspondem à consulta. A exceção interrompe a execução do nosso programa. O [Exemplo 8-4](#) mostra como é essa exceção.

Exemplo 8-4. Saída de exceção do Exemplo 8-3

```
MultipleResultsFound                                     Traceback (última chamada mais recente) ①
<ipython-input-20-d88068ecde4b> em <module>()
---> 1 resultados = session.query(Cookie).one() ②

...b/python2.7/site-packages/sqlalchemy/orm/query.pyc em um (auto)
2480      senão:
2481          raise orm_exc.MultipleResultsFound(
-> 2482              "Foram encontradas várias linhas para uma()")
2483
2484      def escalar(auto):

MultipleResultsFound: várias linhas foram encontradas para one() ③
```

- ① Isso nos mostra o tipo de exceção e que um traceback está presente.
- ② Esta é a linha real onde ocorreu a exceção.
- ③ Esta é a parte interessante que precisamos focar.

No [Exemplo 8-4](#), temos o formato típico para uma exceção `MultipleResultsFound` do SQLAlchemy. Ele começa com uma linha que indica o tipo de exceção. Próximo, há um traceback mostrando onde ocorreu a exceção. O bloco final de linhas é onde os detalhes importantes podem ser encontrados: eles especificam o tipo de exceção e explicar por que ocorreu. Nesse caso, é porque nossa consulta retornou duas linhas e dissemos-lhe para devolver um e apenas um.



Outra exceção relacionada a isso é a exceção `NoResultFound` , o que ocorreria se usássemos o método `.one()` e a consulta não retornou nenhum resultado.

Se quiséssemos tratar esta exceção para que nosso programa não pare de executar e imprime uma mensagem de exceção mais útil, podemos usar o Python `try/except` bloco, como mostrado no [Exemplo 8-5](#).

Exemplo 8-5. Manipulando uma exceção `MultipleResultsFound`

```
de sqlalchemy.orm.exc import MultipleResultsFound ①
experimenter:
    resultados = session.query(Cookie).one()
except MultipleResultsFound como erro: ②
    print('Encontramos muitos cookies... isso é possível?')
```

- ➊ Todas as exceções do SQLAlchemy ORM estão disponíveis no módulo `sqlalchemy.orm.exc`.

- ➋ Capturando a exceção `MultipleResultsFound` como erro.

A execução do [Exemplo 8-5](#) resultará em “Encontramos muitos cookies... isso é possível?” sendo impresso e nosso aplicativo continuará funcionando normalmente. (Acho que não é possível encontrar muitos cookies.) Agora você sabe o que a exceção `MultipleResultsFound` está lhe dizendo e conhece pelo menos um método para lidar com ela de uma maneira que permita que seu programa continue executando.

Erro de Instância Desanexada

Essa exceção ocorre quando tentamos acessar um atributo em uma instância que precisa ser carregada do banco de dados, mas a instância que estamos usando não está atualmente anexada ao banco de dados. Antes de podermos explorar essa exceção, precisamos configurar os registros nos quais vamos operar. O [Exemplo 8-6](#) mostra como fazer isso.

Exemplo 8-6. Criando um usuário e um pedido para esse usuário

```
cookiemon = User('cookiemon', 'mon@cookie.com', '111-111-1111', 'password') session.add(cookiemon)
o1 = Order() o1.user = cookiemon
```

```
sessão.adicionar(o1)

cc = session.query(Cookie).filter(Cookie.cookie_name == "Alterar pepitas de
                                    chocolate ").one() line1 =
LineItem(pedido=o1, cookie=cc, quantidade=2, extended_cost=1.00)

session.add(line1)
session.commit()
```

Agora que criamos um usuário com um pedido e alguns itens de linha, temos o que precisamos para causar essa exceção. O [Exemplo 8-7](#) acionará a exceção para nós.

Exemplo 8-7. Causando um `DetachedInstanceError`

```
pedido = session.query(Order).first()
session.expunge(order) order.line_items
```

No [Exemplo 8-7](#), estamos consultando para obter uma instância de um pedido. Uma vez que temos nossa instância, nós a desanexamos da sessão com `expunge`. Em seguida, tentamos carregar o atributo `line_items`. Como `line_items` é um relacionamento, por padrão ele não carrega todos esses dados até que você os solicite. No nosso caso, separamos a instância da sessão

sion e o relacionamento não tem uma sessão para executar uma consulta para carregar o `line_items` e gera o `DetachedInstanceError`:

```
DetachedInstanceError <ipython-
input-35-233bbca5c715> em <module>()
    1 pedido = session.query(Order).first()
    2 sessão.expurgar(ordem)
--> 3 pedidos.line_items ①

site-packages/sqlalchemy/orm/attributes.py em __get__(self, instance, owner)
    235             return dict_[self.key]
    236         senão:
--> 237             return self.impl.get(instance_state(instance), dict_)
    238
    239

site-packages/sqlalchemy/orm/attributes.py em get(self, state, dict_, passiva)
    576             valor = callable_(estado, passivo)
    577         elif self.callable_:
--> 578             valor = self.callable_(estado, passivo)
    579         senão:
    580             valor = ATTR_EMPTY

site-packages/sqlalchemy/orm:strategies.py em _load_for_state(self, state,
passiva)
    499             "A instância pai %s não está vinculada a uma sessão; "a operação
    500             de carregamento lento do atributo '%s' não pode continuar" %
--> 501             (orm_util.state_str(estado), self.key)
    502         )
    503

DetachedInstanceError: instância pai <Order at 0x10dc31350> não está vinculada a
uma sessão; operação de carregamento lento do atributo 'line_items' não pode continuar ②
```

① Esta é a linha real onde ocorreu o erro.

② Esta é a parte interessante que precisamos focar.

Lemos essa saída de exceção exatamente como fizemos no [Exemplo 8-5](#); a parte interessante de a mensagem indica que tentamos carregar o atributo `line_items` em um pedido instância, e não poderia fazer isso porque essa instância não estava vinculada a uma sessão.

Embora este seja um exemplo forçado do `DetachedInstanceError`, ele age de maneira muito semelhante ao outras exceções, como `ObjectDeletedError`, `StaleDataError` e `ConcurrentModificationError`. Tudo isso está relacionado a informações que diferem entre os instâncias, a sessão e o banco de dados. O mesmo método usado no [Exemplo 8-6](#) com o bloco `try/except` também funcionará para esta exceção e permitirá que o código continue encontro. Você pode verificar uma instância detectada e, no bloco `except`, adicioná-la de volta

à sessão; no entanto, o `DetachedInstanceError` normalmente é um indicador de uma exceção ocorrendo antes desse ponto no código.



Para obter mais detalhes sobre exceções adicionais que podem ocorrer com o ORM, confira a documentação do SQLAlchemy em <http://docs.sqlalchemy.org/en/latest/orm/exceptions.html>.

Todas as exceções mencionadas até agora, como `MultipleResultsFound` e `DetachedInstanceError`, estão relacionadas à falha de uma única instrução. Se tivermos várias instruções, como várias adições ou exclusões que falham durante um commit ou um flush, precisamos tratá-las de maneira diferente. Especificamente, nós os tratamos controlando manualmente a transação. A seção a seguir aborda as transações em detalhes e ensina como usá-las para ajudar a lidar com exceções e recuperar uma sessão.

Transações

As transações são um grupo de instruções que precisamos para ter sucesso ou falhar como um grupo. Quando criamos uma sessão pela primeira vez, ela não está conectada ao banco de dados. Quando realizamos nossa primeira ação com a sessão, como uma consulta, ela inicia uma conexão e uma transação. Isso significa que, por padrão, não precisamos criar transações manualmente. No entanto, se precisarmos lidar com exceções em que parte da transação é bem-sucedida e outra parte falha ou onde o resultado de uma transação cria uma exceção, devemos saber como controlar a transação manualmente.

Já existe um bom exemplo de quando podemos querer fazer isso em nosso banco de dados existente. Depois que um cliente nos encomendar cookies, precisamos enviar esses cookies para o cliente e removê-los de nosso inventário. No entanto, e se não tivermos cookies suficientes para atender um pedido? Precisaremos detectar isso e não enviar esse pedido. Podemos resolver isso com transações.

Precisaremos de um shell Python novo com as tabelas do Capítulo 6; no entanto, precisamos adicionar um `CheckConstraint` à coluna de quantidade para garantir que ela não fique abaixo de 0, pois não podemos ter cookies negativos no inventário. Em seguida, precisamos recriar o usuário cookiemon, bem como os registros de biscoito de chocolate e chocolate amargo e definir a quantidade de biscoitos de chocolate para 12 e os biscoitos de chocolate amargo para 1. O Exemplo 8-8 mostra como para realizar todas essas etapas.

Exemplo 8-8. Configurando o ambiente de transações

```
do sqlalchemy import create_engine do
sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///memory:')
```

```

Sessão = criador de sessões(bind=motor)

sessão = Sessão()

de datetime importação datetime

da importação sqlalchemy (Tabela, Coluna, Inteiro, Numérico, String,
    DateTime, ForeignKey, Boolean, CheckConstraint)
da importação sqlalchemy.ext.declarative declarative_base da
relação de importação sqlalchemy.orm , backref

Base = declarative_base()

class Cookie(Base):
    __tablename__ = 'cookies'
    __table_args__ = (CheckConstraint('quantity >= 0', name='quantity_positive'),) ①

    cookie_id = Column(Integer, primary_key=True)
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255)) cookie_sku =
        Column(String(55)) quantidade = Column(Integer())
    unit_cost = Coluna(Numérico(12, 2))

    def __init__(self, nome, receita_url=Nenhum, sku=Nenhum, quantidade=0, unit_cost=0.00):
        self.cookie_name = nome
        self.cookie_recipe_url = receita_url
        self.cookie_sku = sku self.quantity =
            quantidade self.unit_cost = unit_cost

    def __repr__(self):
        return "Cookie(cookie_name='{self.cookie_name}', " \
            "cookie_recipe_url='{self.cookie_recipe_url}', " \
            "cookie_sku='{self.cookie_sku}', " \
            "quantidade ={self.quantity}, " \
            "unit_cost={self.unit_cost})".format(self=self)

class User(Base):
    __tablename__ = 'users'

    user_id = Column(Integer(), primary_key=True)
    username = Column(String(15), nullable=False, unique=True)
    email_address = Column(String(255), nullable=False) phone =
        Column(String(20) , nullable=False) senha = Column(String(25),
    nullable=False) created_on = Column(DateTime(), default=datetime.now)
    updated_on = Column(DateTime(), default=datetime.now,
    onupdate=datetime. agora)

```

```

def __init__(self, username, email_address, phone, password): self.username
    = username
    self.email_address = email_address
    self.phone = telefone self.password =
    senha

def __repr__(self):
    return "User(username='{self.username}', "
           "email_address='{self.email_address}', "
           "phone='{self.phone}', \"\n"
           "senha\n"
           "={self.password})".format(self=self)

class Order(Base):
    __tablename__ = 'orders'
    order_id = Column(Integer(), primary_key=True) user_id
    = Column(Integer(), ForeignKey('users.user_id')) enviado =
    Column(Boolean(), default= Falso)

    usuário = relacionamento("Usuário", backref=backref('pedidos', order_by=order_id))

    def __repr__(self):
        return "Order(user_id={self.user_id}, "
               "enviado={self.shipped})".format(self=self)

class Linelitem(Base):
    __tablename__ = 'line_items'
    line_item_id = Column(Integer(), primary_key=True) order_id
    = Column(Integer(), ForeignKey('orders.order_id')) cookie_id =
    Column(Integer(), ForeignKey( 'cookies.cookie_id')) quantidade =
    Column(Integer()) extended_cost = Column(Numeric(12, 2))

    pedido = relacionamento("Pedido", backref=backref('line_items',
                                                       order_by=line_item_id))
    cookie = relacionamento("Cookie", uselist=False)

    def __repr__(self):
        return "Linelitem(order_id={self.order_id}, "
               "cookie_id={self.cookie_id}, "
               "quantity={self.quantity}, "
               "extended_cost={self.extended_cost})".format( self=self)

Base.metadata.create_all(motor)

cookiemon = User('cookiemon', 'mon@cookie.com', '111-111-1111', 'password') cc =
Cookie('chocolate chip', 'http://some.aweso.me/cookie/ receita.html',
       'CC01', 12, 0,50)

```

②
③

```

dcc = Cookie(' chocolate amargo', 'http://      ④
              some.aweso.me/cookie/recipe_dark.html', 'CC02', 1, 0,75)

session.add(cookiemon)
session.add(cc)
session.add(dcc)

```

- ① Adiciona a quantidade positiva CheckConstraint.
- ② Adiciona o usuário cookiemon.
- ③ Adiciona o biscoito de chocolate.
- ④ Adiciona o biscoito de chocolate amargo.

Agora vamos definir dois pedidos para o usuário cookiemon. O primeiro pedido será de dois biscoitos com gotas de chocolate e nove biscoitos com gotas de chocolate, e o segundo pedido será de nove biscoitos com gotas de chocolate amargo. O [Exemplo 8-9](#) mostra os detalhes.

Exemplo 8-9. Adicionando os pedidos

```

o1 = Order()
o1.user = cookiemon
session.add(o1)

line1 = LineItem(pedido=o1, cookie=cc, quantidade=9, extended_cost=4,50)

```

```

session.add(line1)
session.commit() o2 ①
= Order() o2.user =
cookiemon
session.add(o2)

line1 = LineItem(pedido=o2, cookie=cc, quantidade=2, extended_cost=1,50) line2 =
LineItem(pedido=o2, cookie=dcc, quantidade=9, extended_cost=6,75)

```

```

session.add(line1)
session.add(line2)
session.commit() ②

```

- ① Adicionando o primeiro pedido.
- ② Adicionando a segunda ordem.

Isso nos dará todos os dados de pedidos que precisamos para explorar como as transações funcionam. Agora precisamos definir uma função chamada `ship_it`. Nossa função `ship_it` aceitará um `order_id`, removerá os cookies do inventário e marcará o pedido como enviado.

O Exemplo 8-10 mostra como isso funciona.

Exemplo 8-10. Definindo a função `ship_it`

```
def ship_it(order_id): order
    = session.query(Order).get(order_id) for li em
    order.line_items:
        li.cookie.quantity = li.cookie.quantity - li.quantity
        session.add(li.cookie) order.shipped = True session.add(order)
    session.commit() print("ID ❷ do pedido enviado: {}".format(order_id))
```

❶

❶ Para cada item de linha encontrado no pedido, isso remove a quantidade desse item de linha da quantidade de cookies para sabermos quantos cookies ainda temos.

❷ Atualizamos o pedido para marcá-lo como enviado.

A função `ship_it` realizará todas as ações necessárias quando enviarmos um pedido.

Vamos executá-lo em nosso primeiro pedido e, em seguida, consultar a tabela de cookies para ter certeza de que reduziu a contagem de cookies corretamente. O Exemplo 8-11 mostra como fazer isso.

Exemplo 8-11. Executando `ship_it` no primeiro pedido

```
ship_it(1) ❶
print(session.query(Cookie.cookie_name, Cookie.quantity).all()) ❷
```

❶ Executa `ship_it` no primeiro `order_id`.

❷ Imprime nosso inventário de cookies.

A execução do código no Exemplo 8-11 resulta em:

```
ID do pedido enviado:
1 [(u'chocolate chip', 3), (u'dark chocolate chip', 1)]
```

Excelente! Funcionou. Podemos ver que não temos cookies suficientes em nosso estoque para atender o segundo pedido; no entanto, em nosso armazém de ritmo acelerado, esses pedidos podem ser processados ao mesmo tempo. Agora tente enviar nosso segundo pedido com a função `ship_it` executando a seguinte linha de código:

```
ship_it(2)
```

Esse comando nos dá este resultado:

```

IntegrityError                                                 Traceback (última chamada mais recente)
<ipython-input-7-8a7f7805a7f6> em <module>() --> 1
    ship_it(2) <ipython-input-5-c442ae46326c> em
ship_it(order_id) order.shipped = True session.add(order)
    6     session.commit() print("ID do pedido enviado:
    7     {}".format(order_id))
--> 8
    9
...

```

IntegrityError: (sqlite3.IntegrityError) Falha na restrição CHECK: quantidade_positiva
[SQL: u'UPDATE cookies SET quantidade=? WHERE cookies.cookie_id = ?']
[parâmetros: (-8, 2)]

Recebemos um IntegrityError porque não tínhamos biscoitos de chocolate amargo suficientes para enviar o pedido. Isso realmente interrompe nossa sessão atual. Se tentarmos emitir mais instruções por meio da sessão, como uma consulta para obter a lista de cookies, obteremos a saída mostrada no [Exemplo 8-12](#).

Exemplo 8-12. Consulta em uma sessão com erro

```
print(session.query(Cookie.cookie_name, Cookie.quantity).all())
```

```

InvalidRequestError                                         Traceback (última chamada mais recente)
<ipython-input-8-90b93364fb2d> em <module>()
--> 1 print(session.query(Cookie.cookie_name, Cookie.quantity).all())
...

```

InvalidRequestError: A transação desta sessão foi revertida devido a uma exceção anterior durante a liberação. Para iniciar uma nova transação com esta Sessão, primeiro emita Session.rollback(). A exceção original era: (sqlite3.IntegrityError)
Falha na restrição CHECK: quantidade_positiva [SQL: u'UPDATE cookies SET quantidade=? WHERE cookies.cookie_id = ?'] [parâmetros: ((1, 1), (-8, 2))]

Eliminando todos os detalhes do traceback, uma exceção InvalidRequestError é gerada devido ao IntegrityError anterior. Para recuperar desse estado de sessão, precisamos reverter manualmente a transação. A mensagem mostrada no [Exemplo 8-12](#) pode ser um pouco confusa porque diz “A transação desta sessão foi revertida”, mas indica que você deve executar a reversão manualmente. O método rollback() na sessão irá restaurar nossa sessão para uma estação de trabalho. Sempre que encontramos uma exceção, queremos emitir um rollback():

```

session.rollback()
print(session.query(Cookie.cookie_name, Cookie.quantity).all())

```

Este código será gerado normalmente com os dados que esperamos:

```
[(u'chocolate', 3), (u'chocolate amargo', 1)]
```

Com nossa sessão funcionando normalmente, agora podemos usar um bloco try/except para garantir que, se ocorrer a exceção IntegrityError, uma mensagem seja impressa e a transação seja revertida para que nosso aplicativo continue funcionando normalmente, conforme mostrado no [Exemplo 8-13](#).

Exemplo 8-13. Transacional ship_it

```
de sqlalchemy.exc import IntegrityError def      ①
ship_it(order_id):
    pedido = session.query(Order).get(order_id) para li
    em order.line_items:
        li.cookie.quantity = li.cookie.quantity - li.quantity session.add(li.cookie)
        order.shipped = True session.add(order) try: session.commit()
    print("ID do pedido enviado: {}".format(order_id)) exceto IntegrityError
    como erro: print('ERROR: {}'.format(error.orig)) session.rollback()
```

②

③

- ① Importando o IntegrityError para que possamos tratar sua exceção.
- ② Confirmando a transação se não ocorrer nenhuma exceção.
- ③ Revertendo a transação se ocorrer uma exceção.

Vamos executar novamente nosso ship_it baseado em transações no segundo pedido, conforme mostrado aqui:

```
ship_it(2)
```

```
ERRO: falha na restrição CHECK: quantidade_positiva
```

O programa não é interrompido pela exceção e nos imprime a mensagem de exceção sem o traceback. Vamos verificar o estoque como fizemos no [Exemplo 8-12](#) para ter certeza de que ele não atrapalhou nosso estoque com uma remessa parcial:

```
[(u'chocolate', 3), (u'chocolate amargo', 1)]
```

Excelente! Nossa função transacional não atrapalhou nosso inventário ou travou nosso aplicativo. Também não tivemos que fazer muita codificação para reverter manualmente as instruções que tiveram sucesso. Como você pode ver, as transações podem ser realmente úteis em situações como essa e podem economizar muita codificação manual.

Neste capítulo, vimos como lidar com exceções tanto em instruções simples quanto em grupos de instruções. Usando um bloco try/except normal em uma única instrução, podemos evitar que nosso aplicativo falhe caso ocorra um erro na instrução do banco de dados.

Também analisamos a transação de sessão para evitar bancos de dados inconsistentes e aplicamos

cation falha em grupos de instruções. No próximo capítulo, aprenderemos como testar nosso código para garantir que ele se comporte da maneira que esperamos.

CAPÍTULO 9

Testando com SQLAlchemy ORM

A maioria dos testes dentro de aplicativos consiste em testes unitários e funcionais; no entanto, com SQLAlchemy, pode ser muito trabalhoso simular corretamente uma instrução de consulta ou um modelo para teste de unidade. Esse trabalho geralmente não leva a muito ganho em relação ao teste em um banco de dados durante o teste funcional. Isso leva as pessoas a criar funções de wrapper para suas consultas que podem ser facilmente simuladas durante os testes de unidade ou apenas testar em um banco de dados em seus testes de unidade e função. Eu pessoalmente gosto de usar pequenas funções de wrapper quando possível ou – se isso não fizer sentido por algum motivo ou se eu estiver em código legado – zombar disso.

Este capítulo aborda como realizar testes funcionais em um banco de dados e como simular consultas e conexões SQLAlchemy.

Testando com um banco de dados de teste

Para nosso aplicativo de exemplo, teremos um arquivo app.py que contém nossa lógica de aplicativo e um arquivo db.py que contém nossos modelos de dados e sessão. Esses arquivos podem ser encontrados na pasta CH10/ do código de exemplo deste livro.

Como um aplicativo é estruturado é um detalhe de implementação que pode ter um grande efeito sobre como você precisa fazer seus testes. Em db.py, você pode ver que nosso banco de dados é configurado por meio da classe DataAccessLayer . Estamos usando essa classe de acesso a dados para nos permitir inicializar um mecanismo de banco de dados e uma sessão sempre que quisermos. Você verá esse padrão comumente usado em estruturas da Web quando acoplado ao SQLAlchemy. A classe DataAccessLayer é inicializada sem um mecanismo e uma sessão na variável dal .



Embora testaremos com um banco de dados SQLite em nossos exemplos, é altamente recomendável que você teste com o mesmo mecanismo de banco de dados usado em seu ambiente de produção. Se você usar um banco de dados diferente em testes, algumas de suas restrições e instruções que podem funcionar com SQLite podem não funcionar com, digamos, PostgreSQL.

O [Exemplo 9-1](#) mostra um trecho do nosso arquivo db.py que contém o DataAccessLayer.

Exemplo 9-1. Classe DataAccessLayer

```
de sqlalchemy.ext.declarative importação declarative_base
```

```
Base = declarative_base()
```

classe DataAccessLayer:

```
def __init__(self):          ❶
    self.engine = Nenhum
    self.conn_string = 'alguma string de conexão'

def connect(self):           ❷
    self.engine = create_engine(self.conn_string)
    Base.metadata.create_all(self.engine)
    self.Session = sessionmaker(bind=self.engine)

dal = DataAccessLayer()      ❸
```

- ❶ `__init__` fornece uma maneira de inicializar uma conexão com uma string de conexão específica como uma fábrica.
- ❷ O método `connect` cria todas as tabelas em nossa classe `Base` e usa o `sessionmaker` para criar uma maneira fácil de fazer sessões para uso em nosso aplicativo.
- ❸ O método `connect` fornece uma instância da classe `DataAccessLayer` que pode ser importada em todo o nosso aplicativo.

Além do nosso `DataAccessLayer`, também temos todos os nossos modelos de dados definidos no arquivo `db.py`. Esses modelos são os mesmos que usamos no [Capítulo 8](#), todos reunidos para usarmos em nosso aplicativo. Aqui estão todos os modelos no arquivo `db.py`:

```
de datetime importação datetime
```

```
da importação sqlalchemy (Coluna, Inteiro, Numérico, String, DateTime, ForeignKey,
                           Boolean, create_engine) do
relacionamento de importação sqlalchemy.orm , backref, sessionmaker
```

```

class Cookie(Base):
    __tablename__ = 'cookies'

    cookie_id = Column(Integer, primary_key=True)
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255))
    cookie_sku =
    Column(String(55))
    quantidade = Column(Integer())
    unit_cost = Coluna(Numérico(12, 2))

    def __repr__(self):
        return "Cookie(cookie_name='{self.cookie_name}', " \
            "cookie_recipe_url='{self.cookie_recipe_url}', " \
            "cookie_sku='{self.cookie_sku}', " \
            "quantity={self.quantity}, " \
            "unit_cost={self.unit_cost})".format( eu = eu)

class User(Base):
    __tablename__ = 'users'

    user_id = Column(Integer(), primary_key=True)
    username = Column(String(15), nullable=False, unique=True)
    email_address = Column(String(255), nullable=False)
    phone =
    Column(String(20))
    senha = Column(String(25),
    nullable=False)
    created_on = Column(DateTime(), default=datetime.now)
    updated_on = Column(DateTime(), default=datetime.now,
    onupdate=datetime.agora)

    def __repr__(self):
        return "Usuário(username='{self.username}', " \
            "email_address='{self.email_address}', " \
            "phone='{self.phone}', " \
            "password='{self.password}')".format(self=self)

class Order(Base):
    __tablename__ = 'orders'
    order_id = Column(Integer(), primary_key=True)
    user_id =
    Column(Integer(), ForeignKey('users.user_id'))
    enviado =
    Column(Boolean(), default= Falso)

    usuário = relacionamento("Usuário", backref=backref('pedidos', order_by=order_id))

    def __repr__(self):
        return "Order(user_id={self.user_id}, " \
            "enviado={self.shipped})".format(self=self)

class LineItem(Base):
    __tablename__ = 'line_items'

```

```

line_item_id = Column(Integer(), primary_key=True) order_id =
= Column(Integer(), ForeignKey('orders.order_id')) cookie_id =
Column(Integer(), ForeignKey('cookies.cookie_id')) quantidade =
Column(Integer()) extended_cost = Column(Numeric(12, 2))

pedido = relacionamento("Pedido", backref=backref('line_items',
                                                 order_by=line_item_id))
cookie = relacionamento("Cookie", uselist=False)

def __repr__(self):
    return "LineItem(order_id={self.order_id}, "
           "cookie_id={self.cookie_id}, " \
           "quantity={self.quantity}, "
           "extended_cost={self.extended_cost})".format(self=self)

```

Com nossos modelos de dados e classe de acesso em vigor, estamos prontos para examinar o código que vamos testar. Vamos escrever testes para a função `get_orders_by_customer` que construímos no [Capítulo 7](#), que é o arquivo `app.py`, mostrado no [Exemplo 9-2](#).

Exemplo 9-2. `app.py` para testar

```

from db import dal, Cookie, LineItem, Order, User, DataAccessLayer ①

def get_orders_by_customer(cust_name, enviado=Nenhum, detalhes=False):
    query = dal.session.query(Order.order_id, User.username, User.phone) query = ②
    query.join(User) se detalhes:

        query = query.add_columns(Cookie.cookie_name, LineItem.quantity,
                                  LineItem.extended_cost) query =
        query.join(LineItem).join(Cookie)
    se enviado não for Nenhum:
        query = query.filter(Order.shipped == enviado) resultados
        = query.filter(User.username == cust_name).all() retornar resultados

```

- ➊ Esta é nossa instância `DataAccessLayer` do arquivo `db.py` e os modelos de dados que estamos usando no código do nosso aplicativo.
- ➋ Como nossa sessão é um atributo do objeto `dal`, precisamos acessá-la de lá.

Vejamos todas as maneiras pelas quais a função `get_orders_by_customer` pode ser usada. Vamos supor para este exercício que já validamos que as entradas para a função são do tipo correto. No entanto, seria muito sensato certificar-se de testar com dados que funcionarão corretamente e dados que possam causar erros. Aqui está uma lista das variáveis que nossa função pode aceitar e seus possíveis valores:

- `cust_name` pode estar em branco, uma string contendo o nome de um cliente válido ou um string que não contém o nome de um cliente válido.
- `enviado` pode ser Nenhum, Verdadeiro ou Falso.
- os detalhes podem ser Verdadeiros ou Falso.

Se quisermos testar todas as combinações possíveis, precisaremos de 12 (são 3 testes para $^* 3 * 2$) testar completamente esta função.



É importante não testar coisas que são apenas parte da funcionalidade básica do SQLAlchemy, pois o SQLAlchemy já vem com uma grande coleção de testes bem escritos. Por exemplo, não gostaríamos de testar uma instrução simples de inserção, seleção, exclusão ou atualização, pois elas são testadas no próprio projeto SQLAlchemy. Em vez disso, procure testar coisas que seu código manipula que podem afetar como a instrução SQLAlchemy é executada ou os resultados retornados por ela.

Para este exemplo de teste, usaremos o módulo unittest integrado. Não se preocupe se você não estiver familiarizado com este módulo; Vou explicar os pontos-chave. Primeiro, precisamos configurar a classe de teste e inicializar a conexão do dal criando um novo arquivo chamado `test_app.py`, conforme mostrado no [Exemplo 9-3](#).

Exemplo 9-3. Configurando os testes

```
importar teste unitário

de db import dal

class TestApp(unittest.TestCase): ①

    @classmethod
    def setUpClass(cls): ②
        dal.conn_string = 'sqlite:///memory:' dal.connect()③
```

- ① unittest requer classes de teste herdadas de `unittest.TestCase`.
- ② O método `setUpClass` é executado uma vez antes de todos os testes.
- ③ Define a string de conexão para nosso banco de dados na memória para teste.

Com nosso banco de dados conectado, estamos prontos para escrever alguns testes. Vamos adicionar esses testes como funções dentro da classe `TestApp` conforme mostrado no [Exemplo 9-4](#).

Exemplo 9-4. Os primeiros seis testes para nomes de usuário em branco

```
def test_orders_by_customer_blank(self): results ①
    = get_orders_by_customer('')
    self.assertEqual(results, []) ②

def test_orders_by_customer_blank_shipped(self): results
    = get_orders_by_customer("", True)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_notshipped(self): results
    = get_orders_by_customer("", False) self.assertEqual(results,
    [])

def test_orders_by_customer_blank_details(self):
    resultados = get_orders_by_customer("", details=True)
    self.assertEqual(results, [])

def test_orders_by_customer_blank_shipped_details(self): results =
    get_orders_by_customer("", True, True) self.assertEqual(results,
    [])

def test_orders_by_customer_blank_notshipped_details(self): results =
    get_orders_by_customer("", False, True) self.assertEqual(results, [])
```

- ➊ unittest espera que cada teste comece com o teste de letras.
- ➋ unittest usa o método assertEquals para verificar se o resultado corresponde ao esperado. Como um usuário não foi encontrado, você deve obter uma lista vazia de volta.

Agora salve o arquivo de teste como `test_app.py` e execute os testes de unidade com o seguinte comando:

```
# python -m unittest test_app
.....
```

Executou 6 testes em 0,018s



Você pode receber um aviso sobre os tipos SQLite e decimal ao executar os testes de unidade; apenas ignore isso, pois é normal para nossos exemplos. Isso ocorre porque o SQLite não tem um tipo decimal verdadeiro, e o SQLAlchemy quer que você saiba que pode haver algumas esquisitices devido à conversão do tipo float do SQLite. É sempre aconselhável investigar essas mensagens, porque no código de produção elas normalmente indicarão a maneira correta de fazer algo. Estamos acionando este aviso propositalmente aqui para que você possa ver como é.

Agora precisamos carregar alguns dados e garantir que nossos testes ainda funcionem. Vamos reutilizar o trabalho que fizemos no [Capítulo 7](#) e inserir os mesmos usuários, pedidos e itens de linha.

No entanto, desta vez vamos envolver as inserções de dados em uma função chamada `db_prep`.

Isso nos permitirá inserir esses dados antes de um teste com uma simples chamada de função. Para simplificar, coloquei esta função dentro do arquivo `db.py` (veja o [Exemplo 9-5](#)); no entanto, em situações do mundo real, muitas vezes ele estará localizado em um arquivo de acessórios ou utilitários de teste.

Exemplo 9-5. Inserindo alguns dados de teste

```
def prep_db(session):
    c1 = Cookie(cookie_name='dark chocolate chip',
                cookie_recipe_url='http://some.aweso.me/cookie/dark_cc.html',
                cookie_sku='CC02', quantidade=1, unit_cost=0.75)
    c2 = Cookie(cookie_name='manteiga de amendoim',
                cookie_recipe_url='http://some.aweso.me/cookie/peanut.html',
                cookie_sku='PB01', quantidade=24, unit_cost=0.25)
    c3 = Cookie(cookie_name='oatmeal raisin', cookie_recipe_url='http://
                  some.okay.me/cookie/raisin.html', cookie_sku='EWW01', quantidade=100,
                unit_cost=1.00)
    session.bulk_save_objects([c1, c2, c3])
    session.commit()

    cookiemon = User(username='cookiemon',
                      email_address='mon@cookie.com',
                      phone='111-111-1111',
                      password='password')
    cakeeater = User(username='cakeeater',
                      email_address='cakeeater@cake.com',
                      phone='222-222-2222',
                      password='password')
    pieperson = User(username='pieperson',
                      email_address='person@pie.com',
                      phone='333-333-3333',
                      password='password')

    session.add(cookiemon)
    session.add(cakeeater)
    session.add(pieperson)
    session.commit()

    o1 = Order()
    o1.user = cookiemon
    session.add(o1)
```

```

line1 = Linelitem(cookie=c1, quantidade=2, extended_cost=1,00)

line2 = Linelitem(cookie=c3, quantidade=12, extended_cost=3,00)

o1.line_items.append(line1)
o1.line_items.append(line2)
session.commit()

o2 = Order()
o2.user = cakeeater

line1 = Linelitem(cookie=c1, quantidade=24, extended_cost=12,00) line2 =
Linelitem(cookie=c3, quantidade=6, extended_cost=6,00)

o2.line_items.append(line1)
o2.line_items.append(line2)

session.add(o2)
session.commit()

```

Agora que temos uma função `prep_db`, podemos usá-la em nosso método `setUpClass` para carregar dados no banco de dados antes de executar nossos testes. Então agora nosso método `setUpClass` se parece com isso:

```

@classmethod
def setUpClass(cls):
    dal.conn_string = 'sqlite:///memory:' dal.connect()
    dal.session = dal.Session() prep_db(dal.session)
    dal.session.close()

```

Também precisamos criar uma nova sessão antes de cada teste e reverter todas as alterações feitas durante essa sessão após cada teste. Podemos fazer isso adicionando um método `setUp` que é executado automaticamente antes de cada teste individual e um método `tearDown` que é executado automaticamente após cada teste, como segue:

```

def setUp(self):      ❶
    dal.session = dal.Session()

def tearDown(self):   ❷
    dal.session.rollback()
    dal.session.close()

```

- ❶ Estabelece uma nova sessão antes de cada teste.
- ❷ Reverte todas as alterações durante o teste e limpa a sessão após cada teste.

Também adicionaremos nossos resultados esperados como propriedades da classe `TestApp`, conforme mostrado aqui:

```

cookie_orders = [(1, u'cookiemon', u'111-111-1111')]
cookie_details = [ (1, u'cookiemon', u'111-111-1111', u'dark
    chocolate chip', 2, Decimal('1.00)), (1, u'cookiemon',
    u'111-111-1111', u'oatmeal raisin', 12, Decimal('3.00))]
```

Podemos usar os dados de teste para garantir que nossa função faça a coisa certa quando recebe um nome de usuário válido. Esses testes vão para dentro da nossa classe TestApp como novas funções, como mostra o [Exemplo 9-6](#).

[Exemplo 9-6](#). Testes para um usuário válido

```

def test_orders_by_customer(self): results
    = get_orders_by_customer('cookiemon')
    self.assertEqual(results, self.cookie_orders)

def test_orders_by_customer_shipped_only(self):
    resultados = get_orders_by_customer('cookiemon', True)
    self.assertEqual(resultados, [])

def test_orders_by_customer_unshipped_only(self):
    resultados = get_orders_by_customer('cookiemon', False)
    self.assertEqual(resultados, self.cookie_orders)

def test_orders_by_customer_with_details(self):
    resultados = get_orders_by_customer('cookiemon', details=True)
    self.assertEqual(results, self.cookie_details)

def test_orders_by_customer_shipped_only_with_details(self):
    resultados = get_orders_by_customer('cookiemon', True, True)
    self.assertEqual(results, [])

def test_orders_by_customer_unshipped_only_details(self):
    resultados = get_orders_by_customer('cookiemon', False, True)
    self.assertEqual(results, self.cookie_details)
```

Usando os testes do [Exemplo 9-6](#) como orientação, você pode concluir os testes para ver o que acontece com um usuário diferente, como cakeeater? Que tal os testes para um nome de usuário que ainda não existe no sistema? Ou se obtivermos um inteiro em vez de uma string para o nome de usuário, qual será o resultado? Quando terminar, compare seus testes com os do código de exemplo fornecido para este capítulo para ver se seus testes são semelhantes aos usados neste livro.

Agora aprendemos como podemos usar SQLAlchemy em testes funcionais para determinar se uma função se comporta conforme o esperado em um determinado conjunto de dados. Também vimos como configurar um arquivo unittest e como preparar o banco de dados para uso em nossos testes.

Em seguida, estamos prontos para examinar os testes sem atingir o banco de dados.

Usando simulações

Essa técnica pode ser uma ferramenta poderosa quando você tem um ambiente de teste onde a criação de um banco de dados de teste não faz sentido ou simplesmente não é viável. Se você tiver uma grande quantidade de lógica que opera no resultado da consulta, pode ser útil simular o código SQLAlchemy para retornar os valores desejados para que você possa testar apenas a lógica circundante. Normalmente, quando vou zombar de alguma parte da consulta, ainda crio o banco de dados na memória, mas não carrego nenhum dado nele e zombou da própria conexão do banco de dados. Isso me permite controlar o que é retornado pelos métodos execute e fetch. Vamos explorar como fazer isso nesta seção.

Para aprender a usar mocks em nossos testes, vamos fazer um único teste para um usuário válido. Usaremos a poderosa biblioteca simulada do Python para controlar o que é retornado pela conexão. O Mock faz parte do módulo unittest no Python 3. No entanto, se você estiver usando o Python 2, precisará instalar a biblioteca mock usando o pip para obter os recursos simulados mais recentes. Para fazer isso, execute este comando:

```
simulação de instalação do pip
```

Agora que temos o mock instalado, podemos usá-lo em nossos testes. O Mock tem uma função de patch que nos permite substituir um determinado objeto em um arquivo Python por um MagicMock que podemos controlar em nosso teste. Um MagicMock é um tipo especial de objeto Python que rastreia como ele é usado e nos permite definir como ele se comporta com base em como está sendo usado.

Primeiro, precisamos importar a biblioteca simulada. No Python 2, precisamos fazer o seguinte:

```
importação simulada
```

No Python 3, precisamos fazer o seguinte:

```
da simulação de importação unittest
```

Com o mock importado, vamos usar o método patch como um decorador para substituir a parte da sessão do objeto dal . Um decorador é uma função que encapsula outra função e altera o comportamento da função encapsulada. Como o objeto dal é importado por nome no arquivo app.py, precisaremos corrigi-lo dentro do módulo app . Isso será passado para a função de teste como um argumento. Agora que temos um objeto simulado, podemos definir um valor de retorno para o método execute , que neste caso não deve ser nada além de um método fetchall encadeado cujo valor de retorno são os dados com os quais queremos testar.

O [Exemplo 9-7](#) mostra o código necessário para usar o mock no lugar do objeto dal .

Exemplo 9-7. Teste de conexão simulado

```
importar teste unitário do
decimal importar decimal
```

```
importação simulada
```

da importação do aplicativo get_orders_by_customer

```
class TestApp(unittest.TestCase):
    cookie_orders = [(1, u'cookiemon', u'111-111-1111')]
    cookie_details = [ (1, u'cookiemon', u'111-111-1111', u'dark
                      chocolate chip', 2, Decimal('1.00)), (1, u'cookiemon',
                      u'111-111-1111', u'oatmeal raisin', 12, Decimal('3.00))]

    @mock.patch('app.dal.session') def ❶
        test_orders_by_customer(self, mock_dal): ❷
            mock_dal.query.return_value.join.return_value.filter.return_value. \
                all.return_value = self.cookie_orders resultados ❸
            = get_orders_by_customer('cookiemon') ❹
            self.assertEqual(results, self.cookie_orders)
```

- ❶ Corrigindo dal.session no módulo do aplicativo com uma simulação.
- ❷ Esse mock é passado para a função de teste como mock_dal.
- ❸ Definimos o valor de retorno do método execute para o valor de retorno encadeado do todo o método que definimos como self.cookie_order.
- ❹ Agora chamamos a função de teste onde o dal.connection será simulado e retornamos o valor que definimos na etapa anterior.

Você pode ver que uma consulta complicada como a do [Exemplo 9-7](#) pode se tornar tediosa rapidamente ao tentar simular a consulta ou conexão completa. Não fuja do trabalho; pode ser muito útil para encontrar bugs obscuros. Como exercício, você deve trabalhar na construção do restante dos testes que construímos com o banco de dados na memória com os tipos de teste simulados. Recomendo que você zombe da consulta e da conexão para se familiarizar com o processo de simulação.

Agora você deve se sentir confortável em testar uma função que contém funções e modelos SQLAlchemy ORM dentro dela. Você também deve entender como pré-preencher dados no banco de dados de teste para uso em seu teste. Por fim, você deve entender como zombar dos objetos de consulta e de conexão. Embora este capítulo tenha usado um exemplo simples, vamos mergulhar mais fundo nos testes no [Capítulo 14](#), que trata do Flask e da Pyramid.

A seguir, veremos como lidar com um banco de dados existente com SQLAlchemy sem a necessidade de reciar todo o esquema em Python via reflexão com Automap.

CAPÍTULO 10

Reflexão com SQLAlchemy

ORM e Automap

Como você aprendeu no [Capítulo 5](#), a reflexão permite preencher um objeto SQLAlchemy de um banco de dados existente; A reflexão funciona em tabelas, exibições, índices e chaves estrangeiras. Mas se você quiser refletir um esquema de banco de dados em classes de estilo ORM? Felizmente, o prático mapa automático da extensão SQLAlchemy permite que você faça exatamente isso.

A reflexão via automap é uma ferramenta muito útil; entretanto, a partir da versão 1.0 do SQLAlchemy não podemos refletir CheckConstraints, comentários ou gatilhos. Você também não pode refletir os padrões do lado do cliente ou uma associação entre uma sequência e uma coluna. No entanto, é possível adicioná-los manualmente usando os métodos que aprendemos no [Capítulo 6](#).

Assim como no [Capítulo 5](#), vamos usar o banco de dados Chinook para teste. Usaremos a versão SQLite, que está disponível na pasta CH11/ do código de exemplo deste livro. Essa pasta também contém uma imagem do esquema do banco de dados para que você possa visualizar o esquema com o qual trabalharemos ao longo deste capítulo.

Refletindo um banco de dados com o Automap

Para refletir um banco de dados, em vez de usar o declarative_base que usamos com o ORM até agora, vamos usar o automap_base. Vamos começar criando um objeto Base para trabalhar, como mostrado no [Exemplo 10-1](#)

Exemplo 10-1. Criando um objeto Base com automap_base

```
de sqlalchemy.ext.automap importar automap_base    ①  
Base = automap_base()    ②
```

- ➊ Importa o automap_base da extensão automap.
- ➋ Inicializa um objeto Base .

Em seguida, precisamos de um mecanismo conectado ao banco de dados que queremos refletir.

O [Exemplo 10-2](#) demonstra como se conectar ao banco de dados Chinook.

Exemplo 10-2. Inicializando um mecanismo para o banco de dados Chinook

```
de sqlalchemy import create_engine

engine = create_engine('sqlite:///Chinook_Sqlite.sqlite')
```

- ➊ Essa cadeia de conexão pressupõe que você esteja no mesmo diretório que o banco de dados de exemplo.

Com a configuração da Base e do mecanismo, temos tudo o que precisamos para refletir o banco de dados.

Usar o método `prepare` no objeto Base que criamos no [Exemplo 10-1](#) examinará tudo disponível no mecanismo que acabamos de criar e refletirá tudo o que puder.

Veja como você reflete o banco de dados usando o objeto Automap Base :

```
Base.prepare(engine, reflect=True)
```

Essa linha de código é tudo que você precisa para refletir todo o banco de dados! Essa reflexão criou objetos ORM para cada tabela que é acessível sob a propriedade `class` do automap Base. Para imprimir uma lista desses objetos, basta executar esta linha de código:

```
Base.classes.keys()
```

Aqui está a saída que você obtém quando faz isso:

```
['Álbum',
 'Cliente',
 'Lista de reprodução',
 'Artista',
 'Acompanhar',
 'Empregado',
 'Tipo de mídia',
 'Linha da fatura',
 'Fatura',
 'Gênero']
```

Agora vamos criar alguns objetos para referenciar as tabelas Artista e Álbum :

```
Artista = Base.classes.Artist
Album = Base.classes.Album
```

A primeira linha de código cria uma referência ao objeto Artist ORM refletido e a segunda linha cria uma referência ao objeto Album ORM refletido. Podemos usar o objeto Artist exatamente como fizemos no [Capítulo 7](#) com nosso ORM definido manualmente

objetos. O [Exemplo 10-3](#) demonstra como realizar uma consulta simples com o objeto para obter os 10 primeiros registros da tabela.

Exemplo 10-3. Usando a tabela Artista

da sessão de importação do sqlalchemy.orm

```
session = Session(engine)
para o artista em session.query(Artist).limit(10):
    print (artist.ArtistId, artist.Name)
```

O [Exemplo 10-3](#) produzirá o seguinte:

```
(1, u'AC/DC')
(2, u'Accept') (3,
u'Aerosmith') (4,
u'Alanis Morissette') (5, u'Alice
In Chains') (6, u'Anthrax')
Carlos Jobim') (7, u'Apocalyptica') (8,
u'Audioslave') (9, u'BackBeat') (10,
u'Billy Cobham')
```

Agora que sabemos como refletir um banco de dados e mapeá-lo para objetos, vamos ver como os relacionamentos são refletidos por meio do automap.

Relacionamentos refletidos

O Automap pode refletir e estabelecer automaticamente relacionamentos muitos para um, um para muitos e muitos para muitos. Vejamos como foi estabelecida a relação entre nossas tabelas Álbum e Artista . Quando o automap cria um relacionamento, ele cria uma propriedade <related_object>_collection no objeto, conforme mostrado no objeto Artist no [Exemplo 10-4](#).

Exemplo 10-4. Usando a relação entre Artista e Álbum para imprimir dados relacionados

```
artist = session.query(Artist).first() for album in
artist.album_collection: print('{} -'
    '{}'.format(artist.Name, album.Title))
```

Isso irá produzir:

```
AC/DC - Para quem está prestes a arrasar, nós te saudamos
AC/DC - Que haja rock
```

Você também pode configurar o automap e substituir certos aspectos de seu comportamento para adequar as classes que ele cria a especificações precisas; no entanto, isso está muito além do escopo deste livro. Você pode aprender muito mais sobre isso na documentação do [SQLAlchemy](#).

Este capítulo encerra as partes essenciais do SQLAlchemy ORM. Espero que você tenha percebido o quão poderosa é essa parte do SQLAlchemy. Este livro oferece uma introdução sólida ao ORM, mas há muito mais para aprender; para mais informações, consulte a [documentação](#).

Na próxima seção, vamos aprender como usar o Alembic para gerenciar migrações de banco de dados para que possamos alterar nosso esquema de banco de dados sem precisar destruir e recriar o banco de dados.

PARTE III

Alambique

Alembic é uma ferramenta para lidar com alterações de banco de dados que aproveita o SQLAlchemy para realizar as migrações. Como o SQLAlchemy só cria tabelas ausentes quando usamos o método `create_all` dos metadados , ele não atualiza as tabelas do banco de dados para corresponder a quaisquer alterações que possamos fazer nas colunas. Nem excluiria as tabelas que removemos do código. O Alembic fornece uma maneira de fazer coisas como adicionar/excluir tabelas, alterar nomes de colunas e adicionar novas restrições. Como o Alembic usa SQLAlchemy para realizar as migrações, eles podem ser usados em uma ampla variedade de bancos de dados back-end.

CAPÍTULO 11

Introdução ao Alambique

O Alembic fornece uma maneira de criar e executar migrações programaticamente para lidar com as alterações no banco de dados que precisaremos fazer à medida que nosso aplicativo evolui. Por exemplo, podemos adicionar colunas às nossas tabelas ou remover atributos de nossos modelos. Também podemos adicionar modelos inteiramente novos ou dividir um modelo existente em vários modelos. O Alembic fornece uma maneira de pré-formar esses tipos de alterações, aproveitando o poder do SQLAlchemy.

Para começar, precisamos instalar o Alembic, o que podemos fazer com o seguinte:

```
pip install alembic
```

Depois de instalar o Alembic, precisamos criar o ambiente de migração.

Criando o ambiente de migração

Para criar o ambiente de migração, vamos criar uma pasta chamada CH12 e mudar para esse diretório. Em seguida, execute o comando alembic init alembic para criar nosso ambiente de migração no diretório alembic/. As pessoas geralmente criam o ambiente de migração em um diretório migrations/, o que você pode fazer com as migrações init do alambique. Você pode escolher qualquer nome de diretório que desejar, mas eu o encorajo a nomeá-lo com algo distinto que não será usado como um nome de módulo em seu código em nenhum lugar. Este processo de inicialização cria o ambiente de migração e também cria um arquivo alembic.ini com as opções de configuração. Se olharmos para o nosso diretório agora, veremos a seguinte estrutura:

```
ÿÿÿ alambique
ÿ ÿÿÿ README ÿ
ÿÿÿ escript.py.mako
ÿ ÿÿÿ alembic.ini
```

Dentro do nosso ambiente de migração recém-criado, encontraremos env.py e o arquivo de modelo script.py.mako junto com um diretório versões/. O diretório versões/ conterá nossos scripts de migração. O arquivo env.py é usado pelo Alembic para definir e instanciar um mecanismo SQLAlchemy, conectar-se a esse mecanismo, iniciar uma transação e chamar o mecanismo de migração corretamente ao executar um comando do Alembic. O modelo script.py.mako é usado ao criar uma migração e define a estrutura básica de uma migração.

Com o ambiente criado, é hora de configurá-lo para funcionar com nossa aplicação.

Configurando o ambiente de migração

As configurações nos arquivos alembic.ini e env.py precisam ser ajustadas para que o Alembic possa trabalhar com nosso banco de dados e aplicativo. Vamos começar com o arquivo alembic.ini, onde precisamos alterar a opção sqlalchemy.url para corresponder à nossa string de conexão do banco de dados. Queremos configurá-lo para se conectar a um arquivo SQLite chamado alembictest.db no diretório atual; para fazer isso, edite a linha sqlalchemy.url para se parecer com o seguinte:

```
sqlalchemy.url = sqlite:///alembictest.db
```

Em todos os nossos exemplos do Alembic, estaremos criando todo o nosso código em um arquivo app/db.py usando o estilo declarativo do ORM. Primeiro, crie um diretório app/ e um __init__.py vazio nele para tornar app um módulo.

Em seguida, adicionaremos o seguinte código em app/db.py para configurar o SQLAlchemy para usar o mesmo banco de dados que o Alembic está configurado para usar no arquivo alembic.ini e definir uma base declarativa:

```
de sqlalchemy import create_engine de
sqlalchemy.ext.declarative import declarative_base de
sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///alembictest.db')

Base = declarative_base()
```

Agora precisamos alterar o arquivo env.py para apontar para nossos metadados, que é um atributo da instância Base que criamos em app/db.py. Ele usa os metadados para comparar o que encontra no banco de dados com os modelos definidos no SQLAlchemy. Começaremos adicionando o diretório atual ao caminho que o Python usa para localizar módulos para que ele possa ver nosso módulo de aplicativo :

```
import os  
import sys  
  
sys.path.append(os.getcwd()) ①
```

- ① Adiciona o diretório de trabalho atual (CH12) ao sys.path que o Python usa ao procurar por módulos

Por fim, alteraremos a linha de metadados de destino em env.py para corresponder ao nosso objeto de metadados no arquivo app/db.py:

```
from app.db import Base ①  
target_metadata = Base.metadata ②
```

- ① Importe nossa instância de Base.
- ② Instrua o Alembic a usar o Base.metadata como seu destino.

Com isso concluído, temos nosso ambiente Alembic configurado corretamente para usar o banco de dados e metadados do aplicativo, e construímos um esqueleto de um aplicativo onde definiremos nossos modelos de dados no [Capítulo 12](#).

Neste capítulo, aprendemos como criar um ambiente de migração e configurá-lo para compartilhar o mesmo banco de dados e metadados do nosso aplicativo. No próximo capítulo, começaremos a criar migrações manualmente e usando os recursos de geração automática.

CAPÍTULO 12

Como criar migrações

No Capítulo 11, inicializamos e configuramos o ambiente de migração Alembic para preparar a adição de classes de dados ao nosso aplicativo e criar migrações para adicioná-las ao nosso banco de dados. Vamos explorar como usar a geração automática para adicionar tabelas e como criar migrações manuais para realizar coisas que a geração automática não pode fazer. É sempre uma boa ideia começar com uma migração vazia, então vamos começar por aí, pois isso nos dá um ponto de partida limpo para nossas migrações.

Gerando uma migração de base vazia

Para criar a migração de base vazia, certifique-se de estar na pasta CH12/ do código de exemplo deste livro. Criaremos uma migração vazia usando este comando:

```
# revisão do alambique -m "Empty Init" ①  
Gerando ch12/alembic/versions/8a8a9d067_empty_init.py ... concluído
```

- ① Execute o comando de revisão do alambique e adicione a mensagem (-m) "Empty Init" à migração

Isso criará um arquivo de migração na subpasta alambique/versions/. Os nomes dos arquivos sempre começarão com um hash que representa o ID da revisão e, em seguida, qualquer mensagem que você fornecer. Vamos olhar dentro deste arquivo:

```
"""Init vazio ①  
  
ID da revisão: 8a8a9d067  
Revisões:  
Data de criação: 2015-09-13 20:10:05.486995  
  
"""  
  
# identificadores de revisão, usados pelo Alembic.
```

```

revisão = '8a8a9d067'          ②
down_revision = Nenhum          ③
branch_labels = Nenhum          ④
Depend_on = Nenhum             ⑤

```

```

do alambique import op
import sqlalchemy as sa

```

```

def atualização():
    passar

```

```

def downgrade():
    passar

```

- ① A mensagem de migração que especificamos
- ② O ID da revisão do Alambique
- ③ A revisão anterior usada para determinar como fazer o downgrade
- ④ A filial associada a esta migração
- ⑤ Quaisquer migrações das quais este depende

O arquivo começa com um cabeçalho que contém a mensagem que fornecemos (se houver), o ID da revisão e a data e hora em que foi criado. A seguir está a seção de identificadores, que explica o que é essa migração e quaisquer detalhes sobre o que ela faz downgrade ou depende, ou qualquer ramificação associada a essa migração. Normalmente, se você estiver fazendo alterações de classe de dados em uma ramificação do seu código, também desejará ramificar suas migrações do Alembic.

Em seguida, há um método de atualização que conteria qualquer código Python necessário para realizar as alterações no banco de dados ao aplicar essa migração. Por fim, há um método de downgrade que contém o código necessário para desfazer essa migração e restaurar o banco de dados para a etapa de migração anterior.

Como não temos nenhuma classe de dados e não fizemos alterações, nossos métodos de upgrade e downgrade estão vazios. Portanto, executar essa migração não terá efeito, mas fornecerá uma ótima base para nossa cadeia de migração. Para executar todas as migrações de qualquer que seja o estado atual do banco de dados para a migração mais alta do Alembic, executamos o seguinte comando:

```

# alambique upgrade head      ①
INFO [alembic.runtime.migration] Contexto impl SQLiteImpl.
INFO [alembic.runtime.migration] Asumirá DDL não transacional.
INFO [alembic.runtime.migration] Executando atualização -> 8a8a9d067, Empty Init

```

- Atualiza o banco de dados para a revisão principal (mais recente).

Na saída anterior, ele listará todas as revisões pelas quais passou para chegar à revisão mais recente. No nosso caso, temos apenas uma, a migração Empty Init, que foi executada por último.

Com nossa base instalada, podemos começar a adicionar nossa classe de dados de usuário ao nosso aplicativo. Uma vez que adicionamos a classe de dados, podemos construir uma migração gerada automaticamente e usá-la para criar a tabela associada no banco de dados.

Gerando automaticamente uma migração

Agora vamos adicionar nossa classe de dados de usuário a app/db.py. Será a mesma classe Cookie que usamos na seção ORM. Vamos adicionar as importações do sqlalchemy, incluindo os tipos Column e column que são usados na classe Cookie e, em seguida, adicionar a própria classe Cookie , conforme mostrado no [Exemplo 12-1](#).

Exemplo 12-1. Aplicativo/db.py atualizado

```
de sqlalchemy import create_engine, Column, Integer, Numeric, String

de sqlalchemy.ext.declarative importe declarative_base de
sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///alembictest.db')

Base = declarative_base()

class Cookie(Base):
    __tablename__ = 'cookies'

    cookie_id = Column(Integer, primary_key=True)
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255))
    cookie_sku =
    Column(String(55))
    quantidade = Column(Integer())
    unit_cost = Coluna(Numérico(12, 2))
```

Com nossa classe adicionada, estamos prontos para criar uma migração para adicionar esta tabela ao nosso banco de dados. Esta é uma migração muito direta, e podemos aproveitar a capacidade do Alembic de gerar automaticamente a migração ([Exemplo 12-2](#)).

Exemplo 12-2. Gerando automaticamente a migração

```
# revisão do alambique --autogenerate -m "Adicionado modelo de cookie" ❶
INFO [alembic.runtime.migration] Contexto impl SQLiteImpl.
INFO [alembic.runtime.migration] Assumirá DDL não transacional.
INFO [alembic.autogenerate.compare] Detectada tabela adicionada 'cookies'
```

```
INFO [alembic.autogenerate.compare] Detectado índice adicionado 'ix_cookies_cookie_name' em
  '['cookie_name']'
Gerando ch12/alembic/versions/34044511331_added_cookie_model.py ... concluído
```

- ➊ Gera automaticamente uma migração com a mensagem "Modelo de cookie adicionado".

Então, quando executamos o comando de autogeração, o Alembic inspecciona os metadados de nossa base SQLAlchemy e compara com o estado atual do banco de dados. No [Exemplo 12-2](#), ele detectou que adicionamos uma tabela chamada cookies (com base em `__tablename__`) e um índice no campo `cookie_name` dessa tabela. Ele marca as diferenças como alterações e adiciona lógica para criá-las no arquivo de migração. Vamos investigar o arquivo de migração que ele criou no [Exemplo 12-3](#).

Exemplo 12-3. O arquivo de migração de cookies

```
"""Adicionado modelo de cookie
```

```
ID da revisão: 34044511331
```

```
Revisões: 8a8a9d067
```

```
Data de criação: 2015-09-14 20:37:25.924031
```

```
----
```

```
# identificadores de revisão, usados pelo Alembic.
revision = '34044511331' down_revision =
'8a8a9d067' branch_labels = Nenhum Depend_on
= Nenhum
```

```
do alambique import op
import sqlalchemy as sa
```

```
def upgrade():
    ### comandos gerados automaticamente pelo Alembic - por favor ajuste!
    ### op.create_table('cookies', sa.Column('cookie_id', sa.Integer(),
    nullable=False), sa.Column('cookie_name', sa.String(length=50),
    nullable=True), sa.Column('cookie_recipe_url', sa.String(length=255),
    nullable=True), sa.Column('cookie_sku', sa.String(length=55), nullable=True),
    sa.Column('quantity', sa.Integer(), nullable=True), sa.Column('unit_cost',
    sa.Numeric(precision=12, scale=2, nullable=True), sa.PrimaryKeyConstraint('cookie_id'))
    op.create_index(op.f('ix_cookies_cookie_name'), 'cookies', ['cookie_name'], unique=False)
    ### end Comandos do Alembic ###
    ❷
```

```
❸
```

```
def downgrade():
```

```
#### comandos gerados automaticamente pelo Alembic - ajuste por favor!
#### op.drop_index(op.f('ix_cookies_cookie_name'), table_name='cookies')
op.drop_table('cookies') #### endComandos do Alembic ####
```

④

- ① Usa o método `create_table` do Alembic para adicionar nossa tabela de cookies .
- ② Adiciona a chave primária na coluna `cookie_id` .
- ③ Usa o método `create_index` do Alembic para adicionar um índice na coluna `cookie_name` hum.
- ④ Elimina o índice `cookie_name` com o método `drop_index` do Alembic.
- ⑤ Descarta a tabela de biscoitos .

No método de atualização do [Exemplo 12-3](#), a sintaxe do método `create_table` é a mesma do construtor `Table` que usamos com SQLAlchemy Core no [Capítulo 1](#): é o nome da tabela seguido por todas as colunas da tabela e quaisquer restrições. O método `create_index` é o mesmo que o construtor `Index` do [Capítulo 1](#) .

Finalmente, o método de downgrade do [Exemplo 12-3](#) usa os métodos `drop_index` e `drop_table` do Alembic na ordem correta para garantir que tanto o índice quanto a tabela sejam removidos.

Vamos executar esta migração usando o mesmo comando que fizemos para executar a migração vazia:

```
# alambique upgrade head
INFO [alembic.runtime.migration] Contexto impl SQLiteImpl.
INFO [alembic.runtime.migration] Assumirá DDL não transacional.
INFO [alembic.runtime.migration] Executando atualização 8a8a9d067 -> 34044511331, modelo
de cookie adicionado
```

Depois de executar essa migração, podemos dar uma olhada no banco de dados para garantir que as alterações ocorreram:

```
# sqlite3 alembictest.db SQLite ①
versão 3.8.5 15/08/2014 22:37:57 Digite ".help"
para dicas de uso .sqlite> .tables alembic_version
cookies sqlite> .schema cookies CREATE TABLE
cookies ( cookie_id INTEGER NOT NULL,
cookie_name VARCHAR(50), cookie_recipe_url
VARCHAR(255), cookie_sku VARCHAR(55),
quantidade INTEGER,
```

```

    unit_cost NUMERIC(12, 2),
CHAVE PRIMÁRIA (cookie_id)
);
CRIAR ÍNDICE ix_cookies_cookie_name ON cookies (cookie_name);

```

- ➊ Acessando o banco de dados através do comando sqlite3 .
- ➋ Listando as tabelas no banco de dados.
- ➌ Imprimindo o esquema para a tabela de cookies .

Excelente! Nossa migração criou nossa tabela e nosso índice perfeitamente. No entanto, há são algumas limitações do que o Autogerar Alambique pode fazer. A [Tabela 12-1](#) mostra uma lista de alterações de esquema comuns que a geração automática pode detectar e, depois disso, a [Tabela 12-2](#) mostra algumas alterações de esquema que a geração automática não pode detectar ou detecta incorretamente.

Tabela 12-1. Alterações de esquema que são geradas automaticamente podem detectar

Ações do elemento de esquema	
Tabelas	Adições e remoções
Coluna	Adições, remoções, mudança de status anulável em colunas
Índice	Alterações básicas em índices e restrições exclusivas explicitamente nomeadas, suporte para geração automática de índices e restrições únicas
Chaves	Renomeações básicas

Tabela 12-2. Alterações de esquema que são geradas automaticamente não podem ser detectadas

Ações do elemento de esquema	
Tabelas	Alterações de nome
Coluna	Alterações de nome
Restrições	Restrições sem um nome explícito
Tipos	Tipos como ENUM que não são suportados diretamente em um back-end de banco de dados

Além das ações que a geração automática pode e não pode suportar, existem algumas recursos opcionalmente suportados que exigem configuração especial ou código personalizado para implemento. Alguns exemplos disso são alterações em um tipo de coluna ou alterações em um padrão do servidor. Se você quiser saber mais sobre as capacidades e limites da autogeração,ções, você pode ler mais na [documentação de geração automática do Alembic](#).

Criando uma migração manualmente

O Alembic não consegue detectar uma mudança de nome de tabela, então vamos aprender como fazer isso sem autogeração e mudar a tabela de cookies para a tabela new_cookies . Precisamos começar alterando o __tablename__ em nossa classe Cookie em app/db.py. Vamos alterá-lo para ficar assim:

```
class Cookie(Base):
    __tablename__ = 'new_cookies'
```

Agora precisamos criar uma nova migração que podemos editar com a mensagem “Renomeando cookies para new_cookies”:

```
# revisão do alambique -m "Renomeando cookies para new_cookies"
Gerando ch12/alembic/versions/2e6a6cc63e9_rename_cookies_to_new_cookies.py ... feito
```

Com nossa migração criada, vamos editar o arquivo de migração e adicionar a operação de renomeação aos métodos de upgrade e downgrade conforme mostrado aqui:

```
def upgrade():
    op.rename_table('cookies', 'new_cookies')

def downgrade():
    op.rename_table('new_cookies', 'cookies')
```

Agora, estamos prontos para executar nossa migração com o comando alambique upgrade :

```
# alambique upgrade head
INFO [alembic.runtime.migration] Contexto impl SQLiteImpl.
INFO [alembic.runtime.migration] Assumirá DDL não transacional.
INFO [alembic.runtime.migration] Executando atualização 34044511331 -> 2e6a6cc63e9,
Renomeando cookies para new_cookies
```

Isso renomeará nossa tabela no banco de dados para new_cookies . Vamos confirmar que aconteceu com o comando sqlite3 :

```
± sqlite3 alembictest.db SQLite
versão 3.8.5 15/08/2014 22:37:57 Digite ".help"
para dicas de uso .sqlite> .tables alembic_version
new_cookies
```

Como esperávamos, não há mais uma tabela de cookies , pois ela foi substituída por new_cookies . Além de rename_table, o Alembic possui muitas operações que você pode usar para ajudar nas alterações do banco de dados, conforme mostrado na [Tabela 12-3](#).

Tabela 12-3. Operações de alambique

Operação	Usado para
add_column	Adiciona uma nova coluna
alterar_coluna	Altera um tipo de coluna, padrão de servidor ou nome
create_check_constraint	Adiciona um novo CheckConstraint
create_foreign_key	Adiciona uma nova chave estrangeira
índice_criação	Adiciona um novo índice
create_primary_key	Adiciona uma nova chave primária
criar a tabela	Adiciona uma nova tabela
create_unique_constraint	Adiciona uma nova UniqueConstraint
drop_column	Remove uma coluna
drop_constraint	Remove uma restrição
drop_index	Elimina um índice
drop_table	Derruba uma mesa
executar	Executar uma instrução SQL bruta
renomear_tabela	Renomeia uma tabela



Embora o Alambique suporte todas as operações da [Tabela 12-3](#), elas são não suportadas em todos os bancos de dados de back-end. Por exemplo, alter_column não funciona em bancos de dados SQLite porque SQLite não suporta alterar uma coluna de forma alguma. SQLite também não suporta a queda de uma coluna.

Neste capítulo, vimos como gerar automaticamente e criar migrações manuais para fazer alterações em nosso banco de dados de maneira repetível. Também aprendemos como aplicar a migrações com o comando de atualização do alambique . Lembre-se, além da operação do Alambique , podemos usar qualquer código Python válido na migração para nos ajudar a realizar nossos metas. No próximo capítulo, exploraremos como realizar downgrades e controle Alambique .

CAPÍTULO 13

Alambique de controle

No capítulo anterior, aprendemos como criar e aplicar migrações, e neste capítulo vamos discutir como controlar ainda mais o Alembic. Exploraremos como aprender o nível de migração atual do banco de dados, como fazer downgrade de uma migração e como marcar o banco de dados em um determinado nível de migração.

Determinando o nível de migração de um banco de dados

Antes de realizar as migrações, você deve verificar novamente quais migrações foram aplicadas ao banco de dados. Você pode determinar qual é a última migração aplicada ao banco de dados usando o comando alambique atual . Ele retornará o ID de revisão da migração atual e informará se é a migração mais recente (também conhecida como head). Vamos executar o comando alambique current na pasta CH12/ do código de exemplo deste livro:

```
# alambique atual
INFO [alembic.runtime.migration] Contexto impl SQLiteImpl.
INFO [alembic.runtime.migration] Assumirá DDL não transacional. 2e6a6cc63e9
(cabeça) ①
```

- ① A última migração aplicada ao banco de dados

Embora esta saída mostre que estamos na migração 2e6a6cc63e9, ela também nos diz que estamos na cabeça ou na migração mais recente. Essa foi a migração que alterou a tabela de cookies para new_cookies. Podemos confirmar isso com o comando histórico do alambique , que nos mostrará uma lista de nossas migrações. O [Exemplo 13-1](#) mostra como é a história do Alambique.

Exemplo 13-1. Nossa história de migração

```
# história do alambique
34044511331 -> 2e6a6cc63e9 (head), Renomeando cookies para new_cookies
8a8a9d067 -> 34044511331, Adicionado Cookie model <base> -> 8a8a9d067, Empty
Init
```

A saída do [Exemplo 13-1](#) nos mostra todas as etapas desde nossa migração inicial vazia até nossa migração atual que nomeou a tabela de cookies . Acho que todos podemos concordar, new_cookies era um nome terrível para uma tabela, especialmente quando mudamos os cookies novamente e temos new_new_cookies. Para corrigir isso e reverter para o nome anterior, vamos aprender como fazer downgrade de uma migração.

Fazendo downgrade de migrações

Para fazer downgrade de migrações, precisamos escolher o ID de revisão para a migração para a qual queremos voltar. Por exemplo, se você quisesse retornar à migração inicial vazia, você escolheria o ID de revisão 8a8a9d067. Na maioria das vezes, queremos desfazer a renomeação da tabela, então para o [Exemplo 13-2](#), vamos reverter para o ID de revisão 34044511331 usando o comando downgrade do alambique .

Exemplo 13-2. Fazendo downgrade da migração de renomear cookies

```
# downgrade do alambique 34044511331
INFO [alembic.runtime.migration] Contexto impl SQLiteImpl.
INFO [alembic.runtime.migration] Assumirá DDL não transacional.
INFO [alembic.runtime.migration] Executando downgrade 2e6a6cc63e9 -> 34044511331,
Renomeando cookies para new_cookies ①
```

① Linha de downgrade

Como podemos ver na linha de downgrade na saída do [Exemplo 13-2](#), ele reverteu a renomeação. Podemos verificar as tabelas com o mesmo comando SQLite .tables que usamos no [Capítulo 12](#). Podemos ver a saída desse comando aqui:

```
# sqlite3 alembictest.db SQLite
versão 3.8.5 15/08/2014 22:37:57 Digite ".help" para
dicas de uso. cookies sqlite> .tables alambic_version
```

①

① Nossa mesa voltou a ser chamada de cookies.

Podemos ver que a tabela foi devolvida ao seu nome original de cookies. Agora vamos usar a corrente do alambique para ver em qual nível de migração o banco de dados está:

```
± alambique INFO
atual [alembic.runtime.migration] Contexto impl SQLiteImpl.
```

INFO [alembic.runtime.migration] Assumirá DDL não transacional. 34044511331

1

- O ID de revisão para a migração do modelo de adição de cookies

Podemos ver que voltamos à migração que especificamos no comando downgrade .

Também precisamos atualizar o código do nosso aplicativo para usar o nome da tabela de cookies , assim como fizemos no [Capítulo 12](#), se quisermos restaurar o aplicativo para um estado de funcionamento.

Você também pode ver que não estamos na última migração, pois o head não está nessa última linha.

Isso cria uma questão interessante que precisamos lidar. Se não quisermos mais usar os cookies renomear para migração new_cookies novamente, podemos apenas excluí-lo do nosso diretório alambique/versions/.

Se você deixá-lo para trás, ele será executado na próxima vez que a cabeça de atualização do alambique for executada, e isso pode causar resultados desastrosos. Vamos deixá-lo no lugar para que possamos explorar como marcar o banco de dados como estando em um determinado nível de migração.

Marcando o nível de migração do banco de dados

Quando queremos fazer algo como pular uma migração ou restaurar um banco de dados, é possível que o banco de dados acredice que estamos em uma migração diferente de onde realmente estamos.

Vamos querer marcar explicitamente o banco de dados como sendo um nível de migração específico para corrigir o problema. No [Exemplo 13-3](#), vamos marcar o banco de dados com o ID de revisão 2e6a6cc63e9 com o comando alambique stamp .

Exemplo 13-3. Marcando o nível de migração do banco de dados

```
# carimbo do alambique
2e6a6cc63e9 INFO [alembic.runtime.migration] Contexto impl SQLiteImpl.
INFO [alembic.runtime.migration] Assumirá DDL não transacional.
INFO [alembic.runtime.migration] Executando stamp_revision 34044511331 ->
2e6a6cc63e9
```

Podemos ver no [Exemplo 13-3](#) que ele marcou a revisão 34044511331 como o nível de migração do banco de dados atual. No entanto, se observarmos as tabelas do banco de dados, ainda veremos que a tabela de cookies está presente. Marcar o nível de migração do banco de dados não executa as migrações, apenas atualiza a tabela Alembic para refletir o nível de migração que fornecemos no comando. Isso efetivamente ignora a aplicação da migração 34044511331 .



Se você pular uma migração como essa, ela será aplicada apenas ao banco de dados atual. Se você apontar seu ambiente de migração do Alembic para um banco de dados diferente alterando o sqlalchemy.url, e esse novo banco de dados estiver abaixo do nível da migração ignorada ou estiver em branco, a execução do cabeçalho de atualização do alambique aplicará essa migração.

Gerando SQL

Se você deseja alterar o esquema do seu banco de dados de produção com SQL, o Alembic também suporta isso. Isso é comum em ambientes com controles de gerenciamento de mudanças mais rígidos ou para aqueles que têm ambientes massivamente distribuídos e precisam executar muitos servidores de banco de dados diferentes. O processo é o mesmo que realizar uma atualização “online” do Alembic como fizemos no [Capítulo 12](#). Podemos especificar as versões inicial e final do SQL gerado. Se você não fornecer uma migração inicial, o Alembic construirá os scripts SQL para atualização de um banco de dados vazio.

O [Exemplo 13-4](#) mostra como gerar o SQL para nossa migração de renomeação.

Exemplo 13-4. Gerando SQL de migração de renomeação

```
# atualização do alambique 34044511331:2e6a6cc63e9 --sql ①
INFO [alembic.runtime.migration] Context impl SQLiteImpl.
INFO [alembic.runtime.migration] Gerando SQL INFO estático
[alembic.runtime.migration] Assumirá DDL não transacional.
INFO [alembic.runtime.migration] Executando atualização 34044511331 -> 2e6a6cc63e9,
    Renomeando cookies para new_cookies
-- Executando atualização 34044511331 -> 2e6a6cc63e9

cookies ALTER TABLE RENAME TO new_cookies;

UPDATE versão_alambique SET version_num='2e6a6cc63e9'
WHERE alambic_version.version_num = '34044511331';
```

① Atualizando de 34044511331 para 2e6a6cc63e9.

A saída do [Exemplo 13-4](#) mostra as duas instruções SQL necessárias para renomear a tabela de cookies e atualizar o nível de migração do Alembic para o novo nível designado pela ID de revisão 2e6a6cc63e9. Podemos escrever isso em um arquivo simplesmente redirecionando a saída padrão para o nome do arquivo de nossa escolha, como mostrado aqui:

```
# atualização do alambique 34044511331:2e6a6cc63e9 --sql > migration.sql | INFO
[alembic.runtime.migration] Context impl SQLiteImpl.
INFO [alembic.runtime.migration] Gerando SQL INFO estático
[alembic.runtime.migration] Assumirá DDL não transacional.
INFO [alembic.runtime.migration] Executando atualização 34044511331 -> 2e6a6cc63e9,
    Renomeando cookies para new_cookies
```

Depois que o comando for executado, podemos cat o arquivo migration.sql para ver nossas instruções SQL assim:

```
# cat migration.sql --
Executando atualização 34044511331 -> 2e6a6cc63e9

cookies ALTER TABLE RENAME TO new_cookies;
```

```
UPDATE versão_alambique SET version_num='2e6a6cc63e9'  
WHERE alambic_version.version_num = '34044511331';
```

Com nossas instruções SQL preparadas, agora podemos pegar esse arquivo e executá-lo por meio de qualquer ferramenta que desejarmos em nossos servidores de banco de dados.



Se você desenvolver em um backend de banco de dados (como SQLite) e implantar em um banco de dados diferente (como PostgreSQL), certifique-se de alterar a configuração `sqlalchemy.url` que definimos no [Capítulo 11](#) para usar a string de conexão para um banco de dados PostgreSQL.

Caso contrário, você pode obter uma saída SQL incorreta para seu banco de dados de produção! Para evitar problemas como esse, sempre desenvolvo no banco de dados que uso em produção.

Isso encerra o básico do uso do Alembic. Neste capítulo, aprendemos como ver o nível de migração atual, fazer downgrade de uma migração e criar scripts SQL que podemos aplicar sem usar o Alembic diretamente em nossos bancos de dados de produção. Se você quiser aprender mais sobre o Alembic, por exemplo, como lidar com ramificações de código, você pode obter detalhes adicionais na [documentação](#).

O próximo capítulo cobre uma coleção de coisas comuns que as pessoas precisam fazer com SQLAlchemy, incluindo como usá-lo com o framework web Flask.

CAPÍTULO 14

Livro de receitas

Este capítulo é diferente dos anteriores, pois cada seção se concentra em um aspecto diferente do uso do SQLAlchemy. A cobertura aqui não é tão detalhada quanto nos capítulos anteriores; considere isso um resumo rápido de ferramentas úteis. No final de cada seção, há informações sobre onde você pode aprender mais se estiver interessado. Estas seções não pretendem ser tutoriais completos, mas sim receitas curtas sobre como realizar uma tarefa específica. A primeira parte deste capítulo se concentra em alguns usos avançados do SQLAlchemy, a segunda parte no uso do SQLAlchemy com estruturas da Web como o Flask e a seção final em bibliotecas adicionais que podem ser usadas com o SQLAlchemy.

Atributos híbridos

Atributos híbridos são aqueles que exibem um comportamento quando acessados como um método de classe e outro comportamento quando acessados em uma instância. Outra maneira de pensar nisso é que o atributo gerará SQL válido quando usado em uma instrução SQLAlchemy e, quando acessado em uma instância, o atributo híbrido executará o código Python diretamente na instância. Acho mais fácil entender isso ao olhar para o código. Vamos usar nossa classe declarativa Cookie para ilustrar isso no [Exemplo 14-1](#).

Exemplo 14-1. Nosso modelo de dados de usuário de Cookie

```
de datetime import datetime
```

```
de sqlalchemy import Column, Integer, Numeric, String, create_engine de
sqlalchemy.ext.declarative import declarative_base de sqlalchemy.ext.hybrid
import hybrid_property, hybrid_method de sqlalchemy.orm import sessionmaker
```

```

engine = create_engine('sqlite:///memory:')

Base = declarative_base()

class Cookie(Base):
    __tablename__ = 'cookies'

    cookie_id = Column(Integer, primary_key=True)
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255))
    cookie_sku = Column(String(55))
    quantidade = Column(Integer())
    unit_cost = Column(Numerico(12, 2))

    @hybrid_property ❶
    def inventory_value(self): return
        self.unit_cost * self.quantity

    @hybrid_method ❷
    def bake_more(self, min_quantity):
        return self.quantity < min_quantity

    def __repr__(self):
        return "Cookie(cookie_name='{self.cookie_name}', " \
            "cookie_recipe_url='{self.cookie_recipe_url}', " \
            "cookie_sku='{self.cookie_sku}', " \
            "quantity={self.quantity}, " \
            "unit_cost={self.unit_cost})".format( eu = eu)

Base.metadata.create_all(motor)
Sessão = criador de sessões(bind=motor)

```

- ❶ Cria uma propriedade híbrida.
- ❷ Cria um método híbrido porque precisamos de uma entrada adicional para realizar a comparação.

No Exemplo 14-1, `valor_inventário` é uma propriedade híbrida que realiza um cálculo com dois atributos de nossa classe de dados `Cookie` e `bake_more` é um método híbrido que recebe uma entrada adicional para determinar se devemos assar mais cookies. (Isso deve retornar `false`? Sempre há espaço para mais cookies!) Com nossa classe de dados definida, podemos começar a examinar o que híbrido realmente significa. Vejamos como o `valor_inventário` funciona quando usado em uma consulta (Exemplo 14-2).

Exemplo 14-2. Propriedade híbrida: Classe

```
print(Cookie.inventory_value < 10,00)
```

O Exemplo 14-2 produzirá o seguinte:

```
cookies.unit_cost * cookies.quantity < :param_1
```

Na saída do Exemplo 14-2, podemos ver que a propriedade híbrida foi expandida em uma cláusula SQL válida. Isso significa que podemos filtrar, ordenar, agrupar e usar funções de banco de dados nessas propriedades. Vamos fazer a mesma coisa com o método híbrido bake_more :

```
print(Cookie.bake_more(12))
```

Isso irá produzir:

```
cookies.quantity < :quantity_1
```

Novamente, ele cria uma cláusula SQL válida a partir do código Python em nosso método híbrido. Para ver o que acontece quando o acessamos como um método de instância, precisaremos adicionar alguns dados ao banco de dados, o que faremos no Exemplo 14-3.

Exemplo 14-3. Adicionando alguns registros ao banco de dados

```
session = Session()
cc_cookie = Cookie(cookie_name='chocolate',
                   cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
                   cookie_sku='CC01', quantidade=12, unit_cost=0.50) dcc =
Cookie(cookie_name='dark chocolate chip',
       cookie_recipe_url='http://some.aweso.me/cookie/recipe_dark.html',
       cookie_sku='CC02', quantidade=1, unit_cost=0.75) mol =
Cookie(cookie_name='molasses',
       cookie_recipe_url='http://some.aweso.me/cookie/recipe_molasses.html',
       cookie_sku='MOL01', quantidade=1, custo_unidade=0.80)
session.add(cc_cookie)
session.add(dcc)
session.add(mol)
session.flush()
```

Com esses dados adicionados, estamos prontos para ver o que acontece quando usamos nossa propriedade e método híbridos em uma instância. Podemos acessar a propriedade inventory_value da instância dcc conforme mostrado aqui:

```
dcc.inventory_value
0,75
```

Você pode ver que quando usado em uma instância, Inventory_value executa o código Python especificado na propriedade. Esta é exatamente a mágica de uma propriedade híbrida ou

método. Também podemos executar o método híbrido `bake_more` na instância, conforme mostrado aqui:

```
dcc.bake_more(12)
```

Verdadeiro

Novamente, devido ao comportamento do método híbrido quando acessado em uma instância, `bake_more` executa o código Python conforme o esperado. Agora que entendemos como as propriedades e métodos híbridos funcionam, vamos usá-los em algumas consultas. Começaremos usando o valor_inventário no [Exemplo 14-4](#).

Exemplo 14-4. Usando uma propriedade híbrida em uma consulta

```
de sqlalchemy import desc
```

```
para cookie em session.query(Cookie).order_by(desc(Cookie.inventory_value)):
    print('{:>20} - {:.2f}'.format(cookie.cookie_name, cookie.inventory_value))
```

Quando executamos o [Exemplo 14-4](#), obtemos a seguinte saída:

```
pepitas de chocolate -
6,00 melado -
0,80 pepitas de chocolate negro - 0,75
```

Isso se comporta exatamente como esperaríamos que qualquer atributo de classe ORM funcionasse. No [Exemplo 14-5](#), usaremos o método `bake_more` para decidir quais cookies precisamos assar.

Exemplo 14-5. Usando um método híbrido em uma consulta

```
para cookie em session.query(Cookie).filter(Cookie.bake_more(12)):
    print('{:>20} - {}'.format(cookie.cookie_name, cookie.quantity))
```

[Exemplo 14-5](#) saídas:

```
pepitas de chocolate preto -
1 melado - 1
```

Novamente, isso se comporta exatamente como esperávamos. Embora esse seja um conjunto incrível de comportamentos do nosso código, os atributos híbridos podem fazer muito mais, e você pode ler sobre isso na documentação do SQLAlchemy sobre [atributos híbridos](#).

Procuração de Associação

Um proxy de associação é um ponteiro em um relacionamento com um atributo específico; ele pode ser usado para facilitar o acesso a um atributo em um relacionamento no código. Por exemplo, isso seria útil se quiséssemos uma lista de nomes de ingredientes que são usados para fazer nossos biscoitos. Vamos ver como isso funciona com um relacionamento típico. o

a relação entre biscoitos e ingredientes será uma relação de muitos para muitos.
O Exemplo 14-6 configura nossos modelos de dados e seus relacionamentos.

Exemplo 14-6. Configurando nossos modelos e relacionamentos

```

da importação de data e hora da
importação de sqlalchemy (Coluna, Inteiro, Numérico, String, Tabela,
    ForeignKey, create_engine)
do relacionamento de importação do
sqlalchemy.orm do sqlalchemy.ext.declarative import
declarative_base do sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///memory:')

Base = declarative_base()
Sessão = criador de sessões(bind=engine)

cookieingredients_table = Table('cookieingredients', Base.metadata,           ①
    Column('cookie_id', Integer, ForeignKey("cookies.cookie_id"), primary_key=True),
    Column('ingredient_id', Integer, ForeignKey("ingredients.ingredient_id"),
           primary_key=True)
)

class Ingredient(Base):
    __tablename__ = 'ingredients'

    ingrediente_id = Column(Integer, primary_key=True) nome
    = Column(String(255), index=True)

    def __repr__(self):
        return "Ingredient(name='{self.name}')".format(self=self)

class Cookie(Base):
    __tablename__ = 'cookies'

    cookie_id = Column(Integer, primary_key=True)
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255)) cookie_sku =
    Column(String(55)) quantidade = Column(Integer())
    unit_cost = Coluna(Numérico(12, 2))

    ingredientes = relacionamento("Ingrediente",
        secundário=cookieingredients_table)           ②

    def __repr__(auto):

```

```

return "Cookie(cookie_name='{self.cookie_name}', " \
        "cookie_recipe_url='{self.cookie_recipe_url}', " \
        "cookie_sku='{self.cookie_sku}', " \ "quantity={self.quantity} , " \
        "\ "unit_cost={self.unit_cost})".format(self=self)

Base.metadata.create_all(engine)

```

- ➊ Criando a tabela de passagem usada para nosso relacionamento muitos-para-muitos.
- ➋ Criando a relação com os ingredientes através da tabela de passagem.

Como esse é um relacionamento muitos-para-muitos, precisamos de uma tabela de passagem para mapear os vários relacionamentos. Usamos o cookieingredients_table para fazer isso. Em seguida, precisamos adicionar um biscoito e alguns ingredientes, como mostrado aqui:

```

session = Session()
cc_cookie = Cookie(cookie_name='chocolate', ➊
                    cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
                    cookie_sku='CC01', quantidade=12, unit_cost=0,50)

```

```

farinha = Ingrediente(nome='Farinha') ➋
açúcar = Ingrediente(nome='Açúcar')
ovo = Ingrediente(nome='Ovo') cc =
Ingrediente(nome=' Raspas de chocolate')
cc_cookie.ingredients.extend([farinha, açúcar, ovo, cc]) ➌
session.add(cc_cookie) session.flush()

```

- ➊ Criando o biscoito.
- ➋ Criando os ingredientes.
- ➌ Adicionando os ingredientes à relação com o biscoito.

Então, para adicionar os ingredientes ao biscoito, temos que criá-los e depois adicioná-los à relação entre biscoitos, ingredientes. Agora, se quisermos listar os nomes de todos os ingredientes, que era nosso objetivo original, temos que percorrer todos os ingredientes e obter o atributo name . Podemos fazer isso da seguinte forma:

```
[ingredient.name para ingrediente em cc_cookie.ingredients]
```

Isso retornará:

```
['Farinha', 'Açúcar', 'Ovo', 'Raspas de Chocolate']
```

Isso pode ser complicado se tudo o que realmente queremos dos ingredientes é o atributo name . Usar um relacionamento tradicional também exige que criemos manualmente cada ingrediente

e adicione-o ao biscoito. Torna-se mais trabalhoso se precisarmos determinar se o ingrediente já existe. É aqui que o proxy de associação pode ser útil para simplificar esse tipo de uso.

Para estabelecer um proxy de associação que podemos usar para acesso a atributos e criação de ingredientes, precisamos fazer três coisas:

- Importe o proxy de associação.

Adicione um método `__init__` ao objeto de destino que facilita a criação de novas instâncias apenas com os valores necessários.

- Crie um proxy de associação que segmente o nome da tabela e o nome da coluna que você deseja proxy.

Vamos iniciar um novo shell Python e configurar nossas classes novamente. No entanto, desta vez vamos adicionar um proxy de associação para facilitar o acesso aos nomes dos ingredientes no Exemplo 14-7.

Exemplo 14-7. Configuração do proxy de associação

```
do sqlalchemy import create_engine do
sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///memory:')

Sessão = criador de sessões(bind=engine)

de datetime importação datetime

from sqlalchemy import Column, Integer, Numeric, String, Table, ForeignKey from
sqlalchemy.orm import relacionamento from sqlalchemy.ext.declarative import
declarative_base de sqlalchemy.ext.associationproxy import associação_proxy
①

Base = declarative_base()

cookieingredients_table = Table('cookieingredients', Base.metadata,
    Column('cookie_id', Integer, ForeignKey("cookies.cookie_id"), primary_key=True),
    Column('ingredient_id', Integer, ForeignKey("ingredients.ingredient_id"),
           primary_key=True)
)

class Ingredient(Base):
    __tablename__ = 'ingredients'
```

```

ingrediente_id = Column(Integer, primary_key=True) nome
= Column(String(255), index=True)

def __init__(self, name): ❷
    self.name = name

def __repr__(self):
    return "Ingredient(name='{self.name}')".format(self=self)

class Cookie(Base):
    __tablename__ = 'cookies'

    cookie_id = Column(Integer, primary_key=True)
    cookie_name = Column(String(50), index=True)
    cookie_recipe_url = Column(String(255)) cookie_sku =
    Column(String(55)) quantidade = Column(Integer())
    unit_cost = Coluna(Numérico(12, 2))

    ingredientes = relacionamento("Ingrediente",
                                    secundário=cookieingredients_table)

    nomes_ingredientes = associação_proxy('ingredientes', 'nome') ❸

    def __repr__(self):
        return "Cookie(cookie_name='{self.cookie_name}', \" \
            cookie_recipe_url='{self.cookie_recipe_url}', \" \
            cookie_sku='{self.cookie_sku}', \" \
            quantidade ={self.quantity}, \
            \" \
            unit_cost={self.unit_cost})".format(self=self)

Base.metadata.create_all(motor)

```

- ❶ Importando o proxy de associação.
- ❷ Definindo um método `__init__` que requer apenas um nome.
- ❸ Estabelecendo um proxy de associação para o atributo de nome dos ingredientes que podemos referenciar como `ingredientes_nomes`.

Com essas três coisas feitas, agora podemos usar nossas classes como fizemos anteriormente:

```

session = Session()
cc_cookie = Cookie(cookie_name='chocolate',
                   cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
                   cookie_sku='CC01', quantidade=12,

```

```

unit_cost=0.50) dcc =
Cookie(cookie_name='dark chocolate chip', cookie_recipe_url='http://
some.aweso.me/cookie/recipe_dark.html', cookie_sku='CC02', quantidade=1, unit_cost=0.75)
farinha = Ingrediente(nome='Farinha') açúcar = Ingrediente(nome='Açúcar') ovo =
Ingrediente(nome='Ovo') cc = Ingrediente(nome=' Raspas de chocolate')
cc_cookie.ingredients.extend([farinha, açúcar, ovo, cc]) session.add(cc_cookie) session.add
(dcc) session.flush()

```

Assim, o relacionamento ainda funciona como esperamos; no entanto, para obter os nomes dos ingredientes, podemos usar o proxy de associação para evitar a compreensão da lista que usamos antes. Aqui está um exemplo de uso de nomes_ingredientes:

```
cc_cookie.ingredient_names
```

Isso irá produzir:

```
['Farinha', 'Açúcar', 'Ovo', 'Raspas de Chocolate']
```

Isso é muito mais fácil do que fazer um loop por todos os ingredientes para criar a lista de nomes de ingredientes. No entanto, esquecemos de adicionar um ingrediente-chave, "Óleo". Podemos adicionar este novo ingrediente usando o proxy de associação, conforme mostrado aqui:

```
cc_cookie.ingredient_names.append('Oil') session.flush()
```

Quando fazemos isso, o proxy de associação cria um novo ingrediente usando o método Ingrediente.__init__ para nós automaticamente. É importante observar que, se já tivéssemos um ingrediente chamado Óleo, o proxy de associação ainda tentaria criá-lo para nós. Isso pode levar a registros duplicados ou causar uma exceção se a adição violar uma restrição.

Para contornar isso, podemos consultar os ingredientes existentes antes de usar o proxy de associação.

O Exemplo 14-8 mostra como podemos fazer isso.

Exemplo 14-8. Filtrando ingredientes

```

dcc_ingredient_list = ['Farinha', 'Açúcar', 'Ovo', 'Raspas de Chocolate Escuro',
'Óleo'] ①
existing_ingredients =
    session.query(Ingredient).filter(Ingredient.name.in_(dcc_ingredient_list)).all() ②
missing = set(dcc_ingredient_list) - set([x.name for x in existing_ingredients]) ③

```

- Definindo a lista de ingredientes para nossos biscoitos de chocolate amargo.

- ② Consultando para encontrar os ingredientes que já existem em nosso banco de dados.
- ③ Encontrar os pacotes ausentes usando a diferença dos dois conjuntos de ingredientes entes.

Depois de encontrar os ingredientes que já tínhamos, comparamos com a nossa lista de ingredientes necessária para determinar o que estava faltando, que são “lascas de chocolate escuro”. Agora podemos adicionar todos os ingredientes existentes por meio do relacionamento e, em seguida, os novos ingredientes por meio do proxy de associação, conforme mostrado no [Exemplo 14-9](#).

[Exemplo 14-9.](#) Adicionando ingredientes ao biscoito de chocolate escuro

```
dcc.ingredients.extend(existing_ingredients)           ①
dcc.ingredient_names.extend(missing)                 ②
```

- ① Adicionando os ingredientes existentes através do relacionamento.
- ② Adicionando os novos ingredientes por meio do proxy de associação.

Agora podemos imprimir uma lista dos nomes dos ingredientes, conforme mostrado aqui:

```
dcc.ingredient_names
```

E teremos a seguinte saída:

```
['Ovo', 'Farinha', 'Azeite', 'Açúcar', 'Raspas de Chocolate Escuro']
```

Isso nos permitiu lidar rapidamente com ingredientes novos e existentes quando os adicionamos ao nosso cookie, e a saída resultante foi exatamente o que desejávamos. Os proxies de associação também têm muitos outros usos; você pode saber mais na [documentação de procuração de associação](#) [ção](#).

Integrando SQLAlchemy com Flask

É comum ver o SQLAlchemy usado com um aplicativo da Web do Flask. O criador do Flask também criou um pacote Flask-SQLAlchemy para facilitar essa integração.

O uso do Flask-SQLAlchemy fornecerá sessões com escopo pré-configuradas que estão vinculadas ao ciclo de vida da página do seu aplicativo Flask. Você pode instalar o Flask-SQLAlchemy com pip como mostrado aqui:

```
# pip install flask-sqlalchemy
```

Ao usar o Flask-SQLAlchemy, eu recomendo que você use o padrão de fábrica de aplicativos, que não é o que é mostrado na seção de início rápido da documentação do Flask-SQLAlchemy. O padrão de fábrica de aplicativos usa uma função que monta um aplicativo com todos os complementos e configurações apropriados. Isso normalmente é colocado em seu aplicativo

arquivo app/__init__.py do cation. O Exemplo 14-10 contém um exemplo de como estruturar seu método create_app .

Exemplo 14-10. Função de fábrica de aplicativos

```
da importação do frasco Frasco
da importação do frasco.ext.sqlalchemy SQLAlchemy

da configuração de importação de configuração

db = SQLAlchemy()          ①

def create_app(config_name): app =      ②
    Flask(__name__)
    app.config.from_object(config[config_name])

    db.init_app(app)          ③
    aplicativo de retorno
```

- ① Cria uma instância não configurada do Flask-SQLAlchemy.
- ② Define a fábrica de aplicativos create_app .
- ③ Inicializa a instância com o contexto do aplicativo .

A fábrica de aplicativos precisa de uma configuração que defina as configurações do Flask e a cadeia de conexão SQLAlchemy. No Exemplo 14-11, definimos um arquivo de configuração, config.py, na raiz do aplicativo.

Exemplo 14-11. Configuração do aplicativo Flask

```
import os

basedir = os.path.abspath(os.path.dirname(__ arquivo__))

classe Config: ①
    SECRET_KEY = 'chave de desenvolvimento'
    ADMINS = frozenset(['jason@jasonamyers.com', ])

class DevelopmentConfig(Config): ②
    DEBUG = Verdadeiro
    SQLALCHEMY_DATABASE_URI = "sqlite:///tmp/dev.db"

class ProductionConfig(Config): ③
```

```
SECRET_KEY = 'Chave de produção'
SQLALCHEMY_DATABASE_URI = "sqlite:///tmp/prod.db"
```

```
config =      ④
    { 'development': DevelopmentConfig,
      'production': ProductionConfig, 'default':
      DevelopmentConfig
    }
```

- ① Define a configuração básica do Flask.
- ② Define a configuração usada durante o desenvolvimento.
- ③ Define a configuração usada quando em produção.
- ④ Cria um dicionário que é usado para mapear de um nome simples para a classe de configuração.

Com a fábrica do aplicativo e a configuração prontas, temos um aplicativo Flask em funcionamento. Agora estamos prontos para definir nossas classes de dados. Definiremos nosso modelo de Cookie em apps/models.py, conforme demonstrado no [Exemplo 14-12](#).

Exemplo 14-12. Definindo o modelo de cookie

```
do banco de dados de importação do aplicativo

class Cookie(db.Model):
    __tablename__ = 'cookies'

    cookie_id = db.Column(db.Integer(), primary_key=True)
    cookie_name = db.Column(db.String(50), index=True)
    cookie_recipe_url = db.Column(db.String(255)) quantidade = db.
        Coluna(db.Integer())
```

No [Exemplo 14-12](#), usamos a instância db que criamos em app/__init__.py para acessar a maioria das partes da biblioteca SQLAlchemy. Agora podemos usar nossos modelos de cookies em consultas como fizemos na seção ORM do livro. A única mudança é que nossas sessões estão aninhadas no objeto db.session .

Uma outra coisa que o Flask-SQLAlchemy adiciona que não é encontrada no código SQLAlchemy normal é a adição de um método de consulta a cada classe de dados ORM. Eu recomendo que você não use essa sintaxe, porque pode ser confusa quando misturada e combinada com outras consultas SQLAlchemy. Este estilo de consulta se parece com o seguinte:

```
Cookie.query.all()
```

Isso deve lhe dar um gostinho de como integrar o SQLAlchemy com o Flask; você pode aprender mais na documentação do Flask-SQLAlchemy . Miguel Grinberg também escreveu um livro fantástico sobre como construir um aplicativo inteiro intitulado [Flask Web Development](#).

SQLACodegenGenericName

Aprendemos sobre reflexão com SQLAlchemy Core no [Capítulo 5](#) e automap para uso com ORM no [Capítulo 10](#). Embora ambas as soluções sejam ótimas, elas exigem que você execute a reflexão toda vez que o aplicativo for reiniciado, ou no caso de módulos dinâmicos , toda vez que o módulo for recarregado. SQLACodegen usa reflexão para construir uma coleção de classes de dados ORM que você pode usar em sua base de código de aplicativo para evitar refletir o banco de dados várias vezes. SQLACodegen tem a capacidade de detectar relacionamentos muitos para um, um para um e muitos para muitos. Pode ser instalado via pip:

```
pip instalar sqlacodegen
```

Para executar o SQLACodegen, precisamos especificar uma string de conexão do banco de dados para ele se conectar. Usaremos uma cópia do banco de dados Chinook com o qual trabalhamos anteriormente (disponível na pasta CH15/ do código de exemplo deste livro). Enquanto estiver dentro da pasta CH15/, execute o comando SQLACodegen com a string de conexão apropriada, conforme mostrado no [Exemplo 14-13](#).

Exemplo 14-13. Executando SQLACodegen no banco de dados Chinook

```
# sqlacodegen sqlite:///Chinook_Sqlite.sqlite
# codificação: utf-8 da
importação sqlalchemy (Tabela, Coluna, Inteiro, Numérico, Unicode, DateTime,
Foreignkey) do
relacionamento de importação sqlalchemy.orm de
sqlalchemy.ext.declarative import declarative_base

Base = metadados declarative_base()
= Base.metadata

class Álbum(Base):
    __tablename__ = 'Álbum'

    AlbumId = Column(Integer, primary_key=True, unique=True)
    Título = Coluna(Unicode(160), anulável=False)
    ArtistaId = Column (ForeignKey (u'Artist.ArtistId'), anulável = False, index = True)

    Artista = relacionamento(u'Artista')
```

```

class Artist(Base):
    __tablename__ = 'Artista'

    ArtistId = Column(Integer, primary_key=True, unique=True)
    Nome = Coluna(Unicode(120))

class Cliente(Base):
    __tablename__ = 'Cliente'

    CustomerId = Column(Integer, primary_key=True, unique=True)
    FirstName = Column(Unicode(40), nullable=False)
    Sobrenome = Coluna(Unicode(20), anulável=False)
    Empresa = Coluna(Unicode(80))
    Endereço = Coluna(Unicode(70))
    Cidade = Coluna(Unicode(40))
    Estado = Coluna(Unicode(40))
    País = Coluna(Unicode(40))
    Código Postal = Coluna(Unicode(10))
    Telefone = Coluna(Unicode(24))
    Fax = Coluna(Unicode(24))
    Email = Coluna(Unicode(60), anulável=False)
    SupportRepld = Column(ForeignKey(u'Employee.EmployeeId'), index=True)

    Funcionário = relacionamento(u'Funcionário')

class Funcionário(Base):
    __tablename__ = 'Funcionário'

    EmployeeId = Column(Integer, primary_key=True, unique=True)
    Sobrenome = Coluna(Unicode(20), anulável=False)
    FirstName = Column(Unicode(20), nullable=False)
    Título = Coluna(Unicode(30))
    ReportsTo = Column(ForeignKey(u'Employee.EmployeeId'), index=True)
    DataNascimento = Coluna(DataHora)
    DataData = Coluna(DataHora)
    Endereço = Coluna(Unicode(70))
    Cidade = Coluna(Unicode(40))
    Estado = Coluna(Unicode(40))
    País = Coluna(Unicode(40))
    Código Postal = Coluna(Unicode(10))
    Telefone = Coluna(Unicode(24))
    Fax = Coluna(Unicode(24))
    E- mail = Coluna(Unicode(60))

    pai = relacionamento(u'Employee', remote_side=[EmployeeId])

class Gênero(Base):
    __tablename__ = 'Gênero'

    GenreId = Column(Integer, primary_key=True, unique=True)

```

```

Nome = Coluna(Unicode(120))

class Invoice(Base):
    __tablename__ = 'Fatura'

    Invoiceld = Column(Integer, primary_key=True, unique=True)
    CustomerId = Column(ForeignKey(u'Customer.CustomerId'), nullable=False,
        índice=Verdadeiro)
    InvoiceDate = Column(DateTime, nullable=False)
    Endereço de cobrança = Column(Unicode(70))
    BillingCity = Column(Unicode(40))
    BillingState = Column(Unicode(40))
    BillingCountry = Column(Unicode(40))
    BillingPostalCode = Column(Unicode(10))
    Total = Coluna(Numérico(10, 2), anulável=False)

    Cliente = relacionamento(u'Cliente')

class InvoiceLine(Base):
    __tablename__ = 'InvoiceLine'

    InvoiceLineId = Column(Integer, primary_key=True, unique=True)
    Invoiceld = Column(ForeignKey(u'Invoice.Invoiceld'), nullable=False, index=True)
    TrackId = Column(ForeignKey(u'Track.TrackId'), nullable=False, index=True)
    Preço Unitário = Column(Numeric(10, 2), nullable=False)
    Quantidade = Coluna(Inteiro, anulável=False)

    Fatura = relacionamento(u'Fatura')
    Track = relacionamento(u'Track')

class MediaType(Base):
    __tablename__ = 'MediaType'

    MediaTypeId = Column(Integer, primary_key=True, unique=True)
    Nome = Coluna(Unicode(120))

class Playlist(Base):
    __tablename__ = 'Playlist'

    PlaylistId = Column(Integer, primary_key=True, unique=True)
    Nome = Coluna(Unicode(120))

    Faixa = relacionamento(u'Track', secundário='PlaylistTrack')

t_PlaylistTrack =
    Table( 'PlaylistTrack', metadados,
        Column('PlaylistId', ForeignKey(u'Playlist.PlaylistId'), primary_key=True,

```

```

        anulável=False),
Column('TrackId', ForeignKey(u'Track.TrackId'), primary_key=True, nullable=False, index=True), Index('PK_PlaylistTrack',
'PlaylistId', 'TrackId', unique=True)

)

class Track(Base):
    __tablename__ = 'Track'

    TrackId = Column(Integer, primary_key=True, unique=True)
    Nome = Coluna(Unicode(200), anulável=False)
    AlbumId = Column (ForeignKey ('Album.AlbumId '), index = True)
    MediaTypeId = Column(ForeignKey(u'MediaType.MediaTypeId'), nullable=False,
        indice=Verdadeiro)
    GenreId = Column(ForeignKey(u'Genre.GenreId'), index=True)
    Compositor = Coluna(Unicode(220))
    Milissegundos = Coluna(Inteiro, anulável=False)
    Bytes = Coluna(Inteiro)
    Preço Unitário = Column(Numeric(10, 2), nullable=False)

    Álbum = relacionamento(u'Álbum')
    Gênero = relacionamento(u'Gênero')
    MediaType = relacionamento(u'MediaType')

```

① Executa o SQLAcodegen no banco de dados local Chinook SQLite

Como você pode ver no [Exemplo 14-13](#), quando executamos o comando, ele cria um arquivo completo que contém todas as classes de dados ORM do banco de dados juntamente com as importações apropriadas. Este arquivo está pronto para uso em nosso aplicativo. Você pode precisar ajustar as configurações para o objeto Base se ele foi estabelecido em outro lugar. Também é possível gerar o código apenas para algumas tabelas usando o argumento --tables . No [Exemplo 14-14](#), vamos especificar as tabelas Artista e Faixa .

Exemplo 14-14. Executando SQLAcodegen nas tabelas Artista e Faixa

```
# sqlacodegen sqlite:///Chinook_Sqlite.sqlite --tables Artist,Track
# codificação: utf-8 de
sqlalchemy import Column, ForeignKey, Integer, Numeric, Unicode de sqlalchemy.orm import
relacionamento de sqlalchemy.ext.declarative import declarative_base
```

```
Base = metadados declarative_base()
= Base.metadata
```

```
class Álbum(Base):
    __tablename__ = 'Álbum'
```

```

AlbumId = Column(Integer, primary_key=True, unique=True)
Título = Coluna(Unicode(160), anulável=False)
ArtistId = Column (ForeignKey (u'Artist.ArtistId '), anulável = False, index = True)

Artista = relacionamento(u'Artista')

```

```

class Artist(Base):
    __tablename__ = 'Artista'

ArtistId = Column(Integer, primary_key=True, unique=True)
Nome = Coluna(Unicode(120))

```

```

class Gênero(Base):
    __tablename__ = 'Gênero'

GenreId = Column(Integer, primary_key=True, unique=True)
Nome = Coluna(Unicode(120))

```

```

class MediaType(Base):
    __tablename__ = 'MediaType'

MediaTypeId = Column(Integer, primary_key=True, unique=True)
Nome = Coluna(Unicode(120))

```

```

class Track(Base):
    __tablename__ = 'Track'

TrackId = Column(Integer, primary_key=True, unique=True)
Nome = Coluna(Unicode(200), anulável=False)
AlbumId = Column (ForeignKey (u'Album.AlbumId '), index = True)
MediaTypeId = Column(ForeignKey(u'MediaType.MediaTypeId'), nullable=False,
                     índice=Verdadeiro)
GenreId = Column(ForeignKey(u'Genre.GenreId'), index=True)
Compositor = Coluna(Unicode(220))
Milissegundos = Coluna(Inteiro, anulável=False)
Bytes = Coluna(Inteiro)
Preço Unitário = Column(Numeric(10, 2), nullable=False)

Álbum = relacionamento(u'Álbum')
Gênero = relacionamento(u'Gênero')
MediaType = relacionamento(u'MediaType')

```

- ① Executa o SQLAcodegen no banco de dados Chinook SQLite local, mas apenas para o Tabelas de artistas e faixas .

Quando especificamos as tabelas Artist e Track no [Exemplo 14-14](#), SQLAcodegen construiu classes para essas tabelas e todas as tabelas que tinham um relacionamento com essas tabelas. SQLAcodegen faz isso para garantir que o código que ele cria esteja pronto para ser usado. Se você deseja salvar as classes geradas diretamente em um arquivo, pode fazê-lo com o redirecionamento padrão, conforme mostrado aqui:

```
# sqlacodegen sqlite:///Chinook_Sqlite.sqlite --tables Artist,Track > db.py
```

Se você quiser mais informações sobre o que você pode fazer com SQLAcodegen, você pode encontrá-lo executando sqlacodegen --help para ver as opções adicionais. Não há nenhuma documentação oficial no momento da redação deste artigo.

Isso conclui a seção do livro de receitas do livro. Espero que você tenha gostado desses poucos exemplos de como fazer mais com o SQLAlchemy.

CAPÍTULO 15

Para onde ir a partir daqui

Espero que você tenha gostado de aprender a usar SQLAlchemy Core e ORM junto com a ferramenta de migração de banco de dados Alembic. Lembre-se que também existe uma extensa documentação no [site SQLAlchemy](#), junto com uma [coleção](#) de apresentações de várias conferências onde você pode aprender sobre tópicos específicos com mais detalhes. Há também várias palestras adicionais sobre SQLAlchemy no [site PyVideo](#), incluindo alguns por [seu verdadeiramente](#).

Para onde ir a seguir depende muito do que você está tentando realizar com SQLAlchemy:

- Se você quiser saber mais sobre Flask e SQLAlchemy, certifique-se de ler [Flask Web Development](#) por Miguel Grinberg.
- Se você estiver interessado em aprender mais sobre testes ou como usar o pytest, Alex Grönholm tem várias postagens de blog excelentes sobre estratégias de teste eficazes: [Parte 1](#) e [Parte 2](#).
- Se você quiser saber mais sobre plug-ins e extensões SQLAlchemy, confira [Awesome SQLAlchemy](#) por Hong Minhee. Este recurso possui uma ótima lista de tecnologias relacionadas ao SQLAlchemy.
- Se você estiver interessado em aprender mais sobre os componentes internos do SQLAlchemy, isso foi abordado por Mike Bayer em [The Architecture of Open Source Applications](#).

Esperamos que as informações que você aprendeu neste livro lhe dêem uma base sólida sobre a qual melhorar suas habilidades. Desejo-lhe sucesso em seus futuros empreendimentos!

Índice

- Um** método add() (Sessão), [80](#)
 método add_column() (Alambique), [149](#)
 Alembic, [139-141](#)
 - documentação para, [148, 155](#)
 - migração vazia, criando, [143-145](#) instalando, [139](#) migração ambiente, configurando, [140-141](#)
 - ambiente de migração, criação, [139-140](#) nível de migração, determinação, [151-152](#) nível de migração, configuração, [153](#) migração, geração automática, [145-148](#) migração, construção manual, [149-150](#) migração, downgrade, [152-153](#) operações executadas por, lista de, [149](#) SQL, gerando, [154-155](#) comando atual do alambique, [151](#) comando downgrade do alambique, [152](#) comando histórico do alambique, [151](#) comando init alambique do alambique, [139](#) comando de revisão do alambique, [143](#) comando carimbo do alambique, [153](#) arquivo alambique.ini, [139, 140](#) alias() função, [32](#) alias() método (Tabela), [32](#) aliases para tabelas, [32-33](#) método all() (Sessão), [83-93](#) método alter_column() (Alambique), [149](#) e_() função, [27-28, 92](#) padrão de fábrica de aplicativos, [166](#) The Architecture of Open Source Applications (Bayer), [175](#) operadores aritméticos, em consultas, [26, 91](#)
 - notação de fatia de matriz, [87](#) proxies de associação, [160-166](#)
 - método de associação_proxy(), [164](#)
 - Exceção AttributeError, [38-40](#) atributos híbridos, [157-160](#) inexistentes, erro para, [38-40](#) ponteiro para, entre relacionamentos, [160-166](#) coleção attrs (inspetor), [107](#) argumento autoload (Tabela), [63](#) autoload_with argumento (Tabela), [63](#) objetos automap_base, [133-136](#)

Método **B** backref(), [74](#)
 Objetos básicos, [133-136](#)
 Bayer, Mike (desenvolvedor, SQLAlchemy), [xiii](#)
 Beaulieu, Alan (Aprendendo SQL), [viii](#)
 método begin() (conexão), [48-50](#) método between() (ClauseElement), [25](#)
 tipo BIGINT, [2](#)
 Tipo BigInteger, [2](#)
 operadores lógicos bit a bit, em consultas, [27](#)
 Tipo BLOB, [2](#)
 tipo bool, [2](#)
 Operadores booleanos, em consultas, [27, 92](#)
 Tipo booleano, [2](#)
 Tipo BOOLEAN, [2](#)
 método bulk_save_objects() (Session), [82](#) objetos de negócios, tipo [xiv](#) byte, [2](#)

tipo BYTEA, [2](#)

- tipos personalizados, 3
- F**
- Função C cast(), 91
 consultas de encadeamento, 34-35, 99-100
 Verificar Restrição, 7
 banco de dados Chinook, 63
 Objetos ClauseElement, 25, 90
 Tipo CLOB, 2
 exemplos de código
 Banco de dados Chinook,
 63 downloads, ix em
 notebooks IPython, viii permissões
 de uso, ix
 Objetos de coluna, 18
 colunas
 controlando para uma consulta, 20, 86
 valor padrão para, 73 agrupamento, 33,
 98 rotulagem, nos resultados da
 consulta, 23, 89 necessários, 73
 redefinindo na atualização, 73 em objetos
 Tabela, 5-6 método commit() (Session),
 80-81 método commit() (transação), 48-50
 operadores de comparação, em consultas, 26,
 90 método compile(), 14 método concat()
 (ClauseElement), 25 encadeamento condicional, 34,
 99 conjunções, em consultas, 27-28, 92 método
 connect() (mecanismo), xix cadeia de conexão, xvii-
 xviii tempos limite de conexão, xix restrições em
 classes ORM, 73 em objetos de tabela, 6-7, 7-9
 informações de contato para este livro, xi contém()
 método (ClauseElement), 25 convenções usadas
 neste livro, ix
- Core (veja SQLAlchemy Core) função
 count(), 23 método create_all()
 (MetaData), 9-10, 75 método create_check_constraint()
 (Alembic),
 149
 função create_engine(), xvii-xix método
 create_foreign_key() (Alembic), 149 método create_index()
 (Alembic), 147, 149 método create_primary_key() (Alembic),
 149 método create_table() (Alembic), 147, 149
 create_unique_constraint() método (Alambique),
 149
- funcção desc(), 21, 87 estado
 de sessão desanexado, 106
 Exceção DetachedInstanceError, 110-112 método
 distinct() (ClauseElement), 25 design orientado a
 domínio, xiv método downgrade() (Alembic), 144, 147,
 149 método drop_column() (Alembic), 149 método
 drop_constraint() (Alambique), 149 método drop_index()
 (Alambique), 147, 149 método drop_table() (Alambique), 147,
 149
- D**
- data warehouse, xiv
 Classe DataAccessLayer, 51-53
 bancos de dados
 conectando a, xvi-xix
 excluindo dados, 29, 94
 drivers necessários para, xvi
 inserindo dados, 13-17, 29-31, 80-83, 95-96 migrações
 (consulte Alembic) consultando dados, 17-28, 31-36 ,
 83-85, 97-101 refletindo (veja reflexão) suportado, xiii,
 xvi tabelas em (veja tabelas) atualizando dados, 28, 93
 Tipo de data, 2 tipo DATE, 2 tipo DateTime, 2 tipo
 DATETIME, 2 tipo datetime.date , 2 datetime.datetime
 type, 2 datetime.time type, 2 datetime.timedelta type, 2
 db.py file, 140 Decimal type, 2 DECIMAL type, 2
 decimal.Decimal type, 2 declarative classes (consulte ORM
 classes) declarative_base objects, 71-72 decoradores, 59
 função delete(), 29, 94 método delete() (Tabela), 29, 94
 excluindo dados SQLAlchemy Core, 29 SQLAlchemy ORM,
 94
- E**
- argumento de eco (create_engine), xviii

argumento de codificação (create_engine),
método **xviii** endswith() (ClauseElement), **25**
engine, criando, xvi-xix tipo Enum, **2** tipo ENUM,
2 arquivo env.py, **140**, **140** tratamento de erro
exceção AttributeError, 38-40 exceção
DetachedInstanceError, 110-112 Exceção
IntegrityError, **40-41**, **117** Exceção
 InvalidRequestError, **117** Exceção
MultipleResultsFound, 108-110 SQLAlchemy Core,
37-42 bloco try/except, **41-42**, **109**, **111**

método execute() (Alambique), **149**
método execute() (conexão), **14**, **15** método
expunge() (Sessão), **107**

F método fetchall() (ResultProxy), **17**
método fetchone() (ResultProxy), **19** função
filter(), **89** resultados de consulta de filtragem,
24-28, **89-90** função filter_by(), **90** primeiro
método () (ResultProxy), **19** primeiro ()
método (Sessão), **85**

Frasco, **166-169**, **175**
Desenvolvimento Web Flask (Grinberg), **175**
Pacote Flask-SQLAlchemy, **166**
Tipo flutuante, **2**
tipo flutuante, **2**
Tipo FLOAT,
método **2** flush() (Session), **82**
fontes usadas neste livro, **ix**
definição de chaves estrangeiras,
8-9, **74** refletindo, **65**

ForeignKeyConstraint, **8-9**
testes funcionais, **51-53**, **121-125**

G construção de declaração generativa, **15**, **15**,
21 tipos genéricos, **2-3**
Grinberg, Miguel (Desenvolvimento Web Flask),
175
agrupamento em consultas, **33**,
98 método group_by() (Tabela), **33**, **98**

Atributos híbridos **H**, **157-160**

I ícones usados neste livro, **ix**
método ilike() (ClauseElement), **25** índices
em objetos Table, **7** método __init__(),
163 arquivo __init__.py, **140**, **166**
função insert(), **15**, **30** método insert()
(Tabela), **13-15**, **29** método
insert_primary_key() (Result-Proxy), **14**
inserindo dados SQLAlchemy Core, **13-17**,
29-31 SQLAlchemy ORM, **80-83**, **95-96** método
 inspect(), **106** Int type, **2** tipo int, **2** tipo Integer,
2 tipo INTEGER, **2** exceção IntegrityError, **40-41**,
117 Tipo de intervalo, **2** tipo INTERVAL, **2**
 Introdução ao Python (Lubanovic), **viii**
Introdução ao Python (vídeos, McKellar), **viii**
Exceção InvalidRequestError, **117** método in_()
(ClauseElement), **25** IPython, **viii** nível de
isolamento, configuração, argumento **xviii**
isolamento_level (create_engine), **xviii** método
is_() (ClauseElement), **25**

J join() método (Tabela), **31**, **97**
junções em consultas, **31-32**, **97-98**

K

keys (veja chaves estrangeiras; chaves
primárias) método keys() (ResultProxy), **19**

L label() função, **23**, **89**
Tipo binário grande, **2**
Aprendendo SQL (Beaulieu), **viii**
método like() (ClauseElement), **25**, **90** função
limit(), **21**, **87**

- limitando os resultados da consulta, 21,
87 ações do banco de dados de log, xviii
Lubanovic, Bill (Apresentando Python), viii
- M**
- McKellar, Jessica (Introdução ao Python, vídeos), viii
Objetos MetaData, 3 dicionário MetaData.tables, 3
microserviços, xv migrations (ver Alembic) biblioteca
simulada, 58, 130 mocks, para teste, 58-61, 130- 131
propriedade modificada (inspetor), 107 exceção
MultipleResultsFound, 108-110 MySQL se conectando,
xviii tempos limite de conexão, xix drivers para, xvi versões
suportadas, xvi
- Função N not_(), 28, 93
not..() métodos (ClauseElement), 25 Tipo
numérico, 2 tipo NUMERIC, 2
- O**
- método one() (Session), 85, 108
operadores, em consultas, 26-27, 90-92
ordenando resultados de consulta, 20-21,
86 função order_by(), 20-21, 86
ORM (consulte SQLAlchemy ORM)
Aulas de ORM
restrições, 73
definindo tabelas usando, 71-73
excluindo dados, 94 inserindo
dados, 80-83, 95-96 chaves, 73
persistentes, 75 consultando
dados, 83-84, 97-101
relacionamentos entre, 74-75, 95
-96, 135 atualizando dados, 93 ou função _(),
27-28, 92 método outerjoin() (Tabela), 31, 97
- P**
- patch() método (simulado), 59
- estado de sessão pendente, 105
estado de sessão persistente, 106
persistente
aulas ORM, 75
Objetos de tabela, 9-10
pip install command, xv, 139
argumento pool_recycle (create_engine), xix
PostgreSQL
conectando, drivers
xvii para, xvii chaves
primárias definindo, 6, 73
determinando, para
registro inserido, 14
Restrição de Chave Primária, 6
Driver Psycopg2, xvi
Driver PyMySQL, xvi
pytest, recursos para, 175
Pitão
REPL para, viii
recursos para, viii
versões suportadas, xv
Python DBAPI (PEP-249), xvi
Site PyVideo, 175
- Q**
- query() método (Sessão), 83-93
consultando objetos refletidos de dados,
66-67
SQLAlchemy Core, 17-28, 31-36
SQLAlchemy ORM, 83-93, 97-101
consultando objetos refletidos, 135
- R**
- consultas brutas, 35, 101
leitura, eval, loop de impressão (REPL), viii
Tipo REAL,
método 2 reflect() (Metadados), 66
reflexão
Automap, 133-136
SQLACodegen, 169-174
SQLAlchemy Core, 63-66
SQLAlchemy ORM, 133
método de relacionamento (), 74, 97
opção remote_side, 97 método
rename_table() (Alambique), 149, 149
REPL (read, eval, print loop), viii método
__repr__(), 78-80 recursos, viii, xi, 175
(veja também recursos do site)

- Desenvolvimento Web Flask (Grinberg), [175](#)
 Apresentando Python (Lubanovic), [viii](#)
 Introdução ao Python (vídeos, McKellar), [viii](#)
 IPython, [viii](#)
 Aprendendo SQL (Beaulieu), [viii](#)
 Objetos ResultProxy, [18-20](#)
 método rollback() (Session), [117-118](#) método
 rollback() (transação), [48-50](#)
- S** scalar() function, [88](#)
 scalar() method (ResultProxy), [19](#) scalar()
 method (Session), [85](#) schema, [xiv, 71](#)
 script.py.mako file, [140](#) select() function,
[17, 20](#) select() method (Tabela), [17](#)
 método select_from() (Tabela), [31](#) Classe
 de sessão, [77-80](#) classe de criador de
 sessão, [77-80](#) sessões, [77-80](#) estados
 de, [105-108](#) como transações, [77, 80-82,](#)
[112-118](#) SMALLINT type, [2](#) SQL gerando
 para migração, [154-155](#) consultas brutas
 usando, [35, 101](#) recursos para, [viii](#)
 SQL Expression Language, [xiv](#) (consulte
 também SQLAlchemy Core)
- Funções SQL, usando em consultas, [22, 88-89](#)
 SQLAcodegen, [169-174](#) comando sqlacodegen,
[169, 174](#) SQLAlchemy, [xiii-xv](#) C extensions for,
 disable, [xv](#) documentation for, [175](#) engine, criando,
 xvi-xix instalando, [xv- xvi](#) plug-ins e extensões,
[175](#) SQLAlchemy Core comparado ao ORM,
 xiv-xv excluindo dados, [29](#) tratamento de erros,
[37-42](#) inserindo dados, [13-17, 29-31](#)
 consultando dados, [17-28, 31-36](#) tabelas
 representadas em (consulte Objetos de tabela)
 teste, [51-61](#) transações, [43-50](#)
- Atributo T __table__, [72](#)
 Objetos de tabela, 1,
 4-9 restrições, [6-7, 7-9](#)
 construindo, [4](#) excluindo
 dados, [29](#) índices, [7](#)
 inserindo dados, [13-17,](#)
[29-31](#) chaves, [6](#) persistentes,
[9-10](#) consultando dados , [17-28,](#)
[31-36](#) relacionamentos entre, [7-9](#)
 atualizando dados, [28](#) atributo
 __tablename__, [71-72](#) tabelas
- colunas em, [5-6](#)
 classes declarativas representando (veja classes
 ORM) refletindo tudo em um banco de dados,
[66](#)

- refletindo individualmente, 63-66
 representações de, 1 definido pelo usuário (consulte Objetos de tabela)
 atributo `__table_args__`, 73 testes
- SQLAlchemy Core, 51-61
 SQLAlchemy ORM, 121-131
- Tipo de texto, 2
 Tipo de TEXTO, 2,
 2 função de texto(), 101
- Tipo de tempo,
 2 objetos de transação, 48-50
 transações
- SQLAlchemy Core, 43-50
 SQLAlchemy ORM, 77, 80-82, 112-118 estado de sessão transitória, 105
- bloco try/except, 41-42, 109, 111 tipos, 1-3 tipos personalizados, 3 genéricos, 2-3 tipos SQL padrão, 3 tipos específicos do fornecedor, 3
- convenções tipográficas usadas neste livro, ix
- Tipo Unicode, 2
 tipo Unicode, 2
 Tipo UNICODE, 2
 Restrição única, 6
 Padrão de unidade de trabalho, teste de 81 unidades, módulo 54-58, 125-129 unittest, 54-58, 125-129 função update(), 28, 93 método update() (tabela), 28, 93 atualização de dados
- SQLAlchemy Core, 28
 SQLAlchemy ORM, 93
- método upgrade() (Alambique), 144, 147, 149
- Função de valores V (), 13
 Tipo VARCHAR, 2
 tipos específicos do fornecedor, 3 diretórios de versões, 140
- Recursos do site W , xi, 175
 (consulte também recursos)
 Documentação do alambique, 148, 155
 A Arquitetura de Aplicativos de Código Aberto (Bayer), 175 proxies de associação, 166 operações em massa, 83
- Banco de dados Chinook, 63 exemplos de código, ix atributos híbridos, 160 IPython, viii plug-ins e extensões, 175 Driver Psycopg2, xvi pytest, 175 PyVideo, 175 relacionamentos refletidos, 135 SQLAlchemygen, 174 Documentação do SQLAlchemy, 175 Zen of Python, xvii função where(), 24-28
- Zen of Python (site), xvii

sobre os autores

Jason Myers trabalha na Cisco como engenheiro de software trabalhando no OpenStack. Antes de mudar para o desenvolvimento há alguns anos, ele passou vários anos como arquiteto de sistemas, construindo data centers e arquiteturas de nuvem para várias das maiores empresas de tecnologia, hospitalares, estádios e provedores de telecomunicações. Ele é um desenvolvedor apaixonado que fala regularmente em eventos locais e nacionais sobre tecnologia. Ele também é o presidente da conferência PyTennessee. Ele adora resolver problemas relacionados à saúde e tem um projeto paralelo, Sucratrend, dedicado a ajudar os diabéticos a gerenciar sua condição e melhorar sua qualidade de vida. Ele usou o SQLAlchemy em aplicativos da Web, data warehouse e análises.

Rick Copeland é cofundador e CEO da Synapp.io, uma empresa com sede em Atlanta que fornece uma solução SaaS para o espaço de conformidade e entregabilidade de e-mail. Ele também é um desenvolvedor Python experiente com foco em bancos de dados relacionais e NoSQL, e foi homenageado como MongoDB Master pela MongoDB Inc. por suas contribuições à comunidade. Ele é um palestrante frequente em vários grupos de usuários e conferências, e um membro ativo da comunidade de startups de Atlanta.

Colofão

O animal na capa do Essential SQLAlchemy é um peixe voador de grande porte (*Cypselurus oligolepis*). Peixe-voador é o nome mais comum para membros da família Exocoetidae, que compreende cerca de 40 espécies que habitam as águas quentes tropicais e subtropicais dos oceanos Atlântico, Pacífico e Índico. Os peixes voadores variam de 7 a 12 polegadas de comprimento e são caracterizados por suas barbatanas peitorais extraordinariamente grandes e semelhantes a asas. Algumas espécies também têm barbatanas pélvicas aumentadas e, portanto, são conhecidas como peixes voadores de quatro asas.

Como o próprio nome sugere, os peixes voadores têm a capacidade única de saltar da água e deslizar pelo ar por distâncias de até 400 metros. Seus corpos semelhantes a torpedos os ajudam a reunir a velocidade necessária para se impulsionar do oceano (cerca de 37 milhas por hora), e suas barbatanas peitorais distintas e barbatanas bifurcadas os mantêm no ar. Os biólogos acreditam que essa característica notável pode ter evoluído como uma maneira de os peixes voadores escaparem de seus muitos predadores, que incluem atum, cavala, espadarte, espadim e outros peixes maiores. No entanto, os peixes voadores às vezes têm mais dificuldade em fugir de seus predadores humanos. Atraídos por uma luz atraente que os pescadores prendem às suas canoas à noite, os peixes saltam e são incapazes de saltar de volta.

O peixe-voador seco é um alimento básico para o povo Tao da Ilha das Orquídeas, localizada na costa de Taiwan, e as ovas de peixe-voador são comuns na culinária japonesa. Eles também são uma iguaria cobiçada em Barbados, conhecida como “Terra dos Peixes Voadores” antes que a poluição do transporte e a pesca excessiva esgotassem seus números. O peixe-voador mantém um destaque

status cultural lá, no entanto; é o principal ingrediente do prato nacional (cou-cou e peixe voador) e está presente em moedas, obras de arte e até mesmo no logotipo da Autoridade de Turismo de Barbados.

Muitos dos animais nas capas da O'Reilly estão ameaçados de extinção; todos eles são importantes para o mundo. Para saber mais sobre como você pode ajudar, acesse animals.oreilly.com.

A imagem da capa é da Dover's Animals. As fontes da capa são URW Typewriter e Guardian Sans. A fonte do texto é Adobe Minion Pro; a fonte do título é Adobe Myriad Condensed; e a fonte do código é Ubuntu Mono da Dalton Maag.