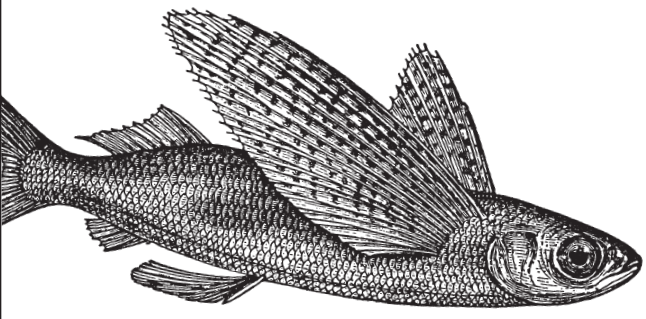
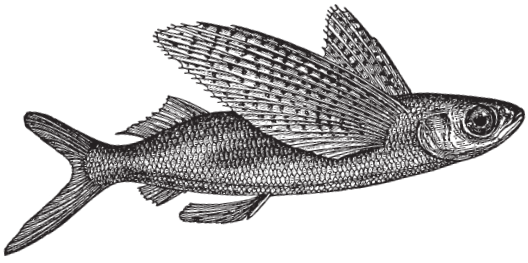
O'REILLY®

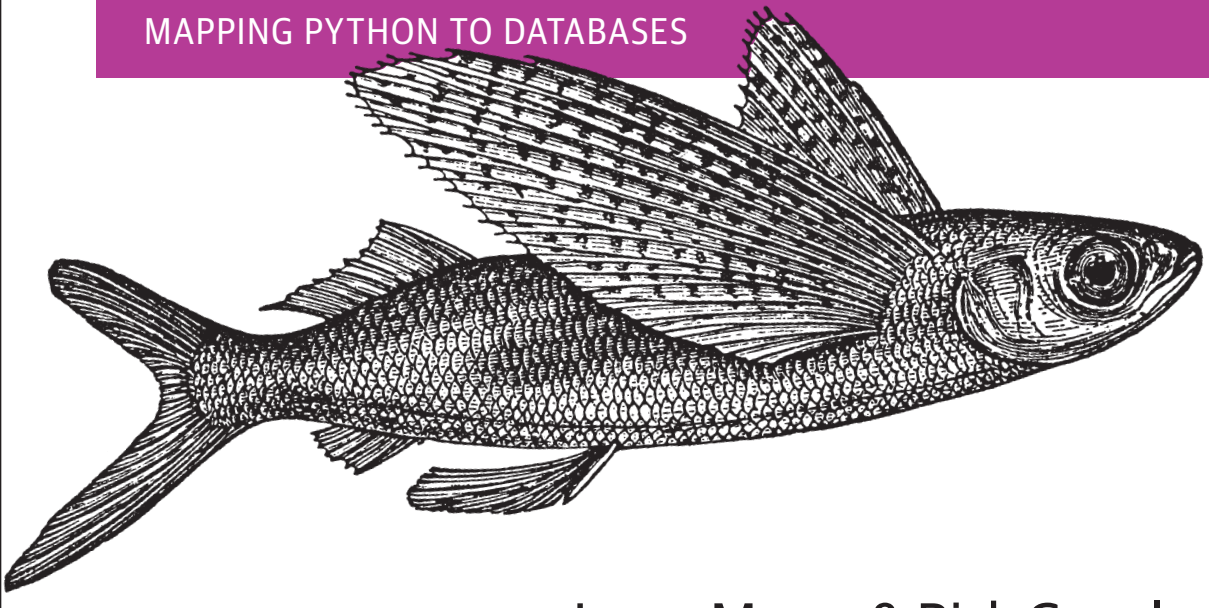
2nd Edition





Essential SQLAlchemy

MAPPING PYTHON TO DATABASES



Jason Myers & Rick Copeland

O'REILLY®

Essential SQLAlchemy

Dive into SQLAlchemy, the popular, open source code library that helps Python programmers work with relational databases such as Oracle, MySQL, PostgresSQL, 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 SQLAcodegen

"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)

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

Twitter: @oreillymedia
facebook.com/oreilly

SECOND EDITION

Essential SQLAlchemy

Jason Myers and Rick Copeland

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Essential SQLAlchemy
by Jason Myers and Rick Copeland

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Dawn Schanafelt and Meghan Blanchette
Production Editor: Shiny Kalapurakkel
Copyeditor: Charles Roumeliotis
Proofreader: Jasmine Kwityn
Indexer: Angela Howard
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2008: First Edition
December 2015: Second Edition

Revision History for the Second Edition
2015-11-20: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491916469* for release details.

978-1-491-91646-9

[LSI]

Table of Contents

Preface

We are surrounded by data everywhere, and your ability to store, update, and report on that data is critical to every application you build. Whether you are developing for the Web, the desktop, or other applications, you need fast and secure access to data. Relational databases are still one of the most common places to put that data.

SQL is a powerful language for querying and manipulating data in a database, but sometimes it's tough to integrate it with the rest of your application. You may have used string manipulation to generate queries to run over an ODBC interface, or used a DB API as a Python programmer. While those can be effective ways to handle data, they can make security and database changes very difficult.

This book is about a very powerful and flexible Python library named SQLAlchemy that bridges the gap between relational databases and traditional programming. While SQLAlchemy allows you to "drop down" into raw SQL to execute your queries, it encourages higher-level thinking through a more "Pythonic" and friendly approach to database queries and updates. It supplies the tools that let you map your application's classes and objects to database tables once and then to "forget about it," or to return to your model again and again to fine-tune performance.

SQLAlchemy is powerful and flexible, but it can also be a little daunting. SQLAlchemy tutorials expose only a fraction of what's available in this excellent library, and though the online documentation is extensive, it is often better as a reference than as a way to learn the library initially. This book is meant as a learning tool and a handy reference for when you're in "implementation mode" and need an answer *fast*.

This book focuses on the 1.0 release of SQLAlchemy; however, much of what we will cover has been available for many of the previous versions. It certainly works from 0.8 forward with minor tweaking, and most of it from 0.5.

This book has been written in three major parts: SQLAlchemy Core, SQLAlchemy ORM, and an Alembic section. The first two parts are meant to mirror each other as

closely as possible. We have taken care to perform the same examples in each part so that you can compare and contrast the two main ways of using SQLAlchemy. The book is also written so that you can read both the SQLAlchemy Core and ORM parts or just the one suits your needs at the moment.

Who This Book Is For

This book is intended for those who want to learn more about how to use relational databases in their Python programs, or have heard about SQLAlchemy and want more information on it. To get the most out of this book, the reader should have intermediate Python skills and at least moderate exposure to SQL databases. While we have worked hard to make the material accessible, if you are just getting started with Python, we recommend reading *Introducing Python* by Bill Lubanovic or watching the "Introduction to Python" videos by Jessica McKellar as they are both fantastic resources. If you are new to SQL and databases, check out *Learning SQL* by Alan Beaulieu. These will fill in any missing gaps as you work through this book.

How to Use the Examples

Most of the examples in this book are built to be run in a read-eval-print loop (REPL). You can use the built-in Python REPL by typing `python` at the command prompt. The examples also work well in an ipython notebook. There are a few parts of the book, such as Chapter 4, that will direct you to create and use files instead of a REPL. The supplied example code is provided in IPython notebooks for most examples, and Python files for the chapters that specify to use them. You can learn more about IPython at its website.

Assumptions This Book Makes

This book assumes basic knowledge about Python syntax and semantics, particularly versions 2.7 and later. In particular, the reader should be familiar with iteration and working with objects in Python, as these are used frequently throughout the book. The second part of the book deals extensively with object-oriented programming and the SQLAlchemy ORM. The reader should also know basic SQL syntax and relational theory, as this book assumes familiarity with the SQL concepts of defining schema and tables along with creating `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at link: *https://github.com/oreillymedia/essential-sqlalchemy-2e*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this

book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Essential SQLAlchemy, Second Edition*, by Jason Myers and Rick Copeland (O'Reilly). Copyright 2016 Jason Myers and Rick Copeland, 978-1-4919-1646-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like 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, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at the following link: *http://oreil.ly/1lwEdiw*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Many thanks go to Patrick Altman, Eric Floehr, and Alex Grönholm for their critical prepublication feedback. Without them, this book would have undoubtedly had many technical issues and been much harder to read.

My appreciation goes out to Mike Bayer, whose recommendation led to this book being written in the first place. I'm grateful to Meghan Blanchette and Dawn Schanafelt for pushing me to complete the book, making me a better writer, and putting up with me. I also would like to thank Brian Dailey for reading some of the roughest cuts of the book, providing great feedback, and laughing with me about it.

I want to thank the Nashville development community for supporting me, especially Cal Evans, Jacques Woodcock, Luke Stokes, and William Golden.

Thanks to my employer, Cisco Systems, for allowing me the time and providing support to finish the book. Thanks also to Justin at Mechanical Keyboards for keeping me supplied with everything I needed to keep my fingers typing.

Most importantly I want to thank my wife for putting up with me reading aloud to myself, disappearing to go write, and being my constant source of support and hope. I love you, Denise.

Introduction to SQLAlchemy

SQLAlchemy is a library used to interact with a wide variety of databases. It enables you to create data models and queries in a manner that feels like normal Python classes and statements. Created by Mike Bayer in 2005, SQLAlchemy is used by many companies great and small, and is considered by many to be the de facto way of working with relational databases in Python.

It can be used to connect to most common databases such as Postgres, MySQL, SQLite, Oracle, and many others. It also provides a way to add support for other relational databases as well. Amazon Redshift, which uses a custom dialect of PostgreSQL, is a great example of database support added by the community.

In this chapter, we'll explore why we need SQLAlchemy, learn about its two major modes, and get connected to a database.

Why Use SQLAlchemy?

The top reason to use SQLAlchemy is to abstract your code away from the underlying database and its associated SQL peculiarities. SQLAlchemy leverages powerful common statements and types to ensure its SQL statements are crafted efficiently and properly for each database type and vendor without you having to think about it. This makes it easy to migrate logic from Oracle to PostgreSQL or from an application database to a data warehouse. It also helps ensure that database input is sanitized and properly escaped prior to being submitted to the database. This prevents common issues like SQL injection attacks.

SQLAlchemy also provides a lot of flexibility by supplying two major modes of usage: SQL Expression Language (commonly referred to as Core) and ORM. These modes can be used separately or together depending on your preference and the needs of your application.

SQLAlchemy Core and the SQL Expression Language

The SQL Expression Language is a Pythonic way of representing common SQL statements and expressions, and is only a mild abstraction from the typical SQL language. It is focused on the actual database schema; however, it is standardized in such a way that it provides a consistent language across a large number of backend databases. The SQL Expression Language also acts as the foundation for the SQLAlchemy ORM.

ORM

The SQLAlchemy ORM is similar to many other object relational mappers (ORMs) you may have encountered in other languages. It is focused around the domain model of the application and leverages the Unit of Work pattern to maintain object state. It also provides a high-level abstraction on top of the SQL Expression Language that enables the user to work in a more idiomatic way. You can mix and match use of the ORM with the SQL Expression Language to create very powerful applications. The ORM leverages a declarative system that is similar to the active-record systems used by many other ORMs such as the one found in Ruby on Rails.

While the ORM is extremely useful, you must keep in mind that there is a difference between the way classes can be related, and how the underlying database relationships work. We'll more fully explore the ways in which this can affect your implementation in Chapter 6.

Choosing Between SQLAlchemy Core and ORM

Before you begin building applications with SQLAlchemy, you will need to decide if you are going to primarly use the ORM or Core. The choice of using SQLAlchemy Core or ORM as the dominant data access layer for an application often comes down to a few factors and personal preference.

The two modes use slightly different syntax, but the biggest difference between Core and ORM is the view of data as schema or business objects. SQLAlchemy Core has a schema-centric view, which like traditional SQL is focused around tables, keys, and index structures. SQLAlchemy Core really shines in data warehouse, reporting, analysis, and other scenarios where being able to tightly control the query or operating on unmodeled data is useful. The strong database connection pool and result-set optimizations are perfectly suited to dealing with large amounts of data, even in multiple databases.

However, if you intend to focus more on a domain-driven design, the ORM will encapsulate much of the underlying schema and structure in metadata and business objects. This encapsulation can make it easy to make database interactions feel more like normal Python code. Most common applications lend themselves to being modeled in this way. It can also be a highly effective way to inject domain-driven design

into a legacy application or one with raw SQL statements sprinkled throughout. Microservices also benefit from the abstraction of the underlying database, allowing the developer to focus on just the process being implemented.

However, because the ORM is built on top of SQLAlchemy Core, you can use its ability to work with services like Oracle Data Warehousing and Amazon Redshift in the same manner that it interoperates with MySQL. This makes it a wonderful compliment to the ORM when you need to combine business objects and warehoused data.

Here's a quick checklist to help you decide which option is best for you:

- If you are working with a framework that already has an ORM built in, but want to add more powerful reporting, use Core.
- If you want to view your data in a more schema-centric view (as used in SQL), use Core.
- If you have data for which business objects are not needed, use Core.
- If you view your data as business objects, use ORM.
- If you are building a quick prototype, use ORM.
- If you have a combination of needs that really could leverage both business objects and other data unrelated to the problem domain, use both!

Now that you know how SQLAlchemy is structured and the difference between Core and ORM, we are ready to install and start using SQLAlchemy to connect to a database.

Installing SQLAlchemy and Connecting to a Database

SQLAlchemy can be used with Python 2.6, Python 3.3, and Pypy 2.1 or greater. I recommend using pip to perform the install with the command `pip install sqlalchemy`. It's worth noting that it can also be installed with easy_install and distutils; however, pip is the more straightforward method. During the install, SQLAlchemy will attempt to build some C extensions, which are leveraged to make working with result sets fast and more memory efficient. If you need to disable these extensions due to the lack of a compiler on the system you are installing on, you can use `--global-option=--without-cextensions`. Note that using SQLAlchemy without C extensions will adversely affect performance, and you should test your code on a system with the C extensions prior to optimizing it.

Installing Database Drivers

By default, SQLAlchemy will support SQLite3 with no additional drivers; however, an additional database driver that uses the standard Python DBAPI (PEP-249) specification is needed to connect to other databases. These DBAPIs provide the basis for the dialect each database server speaks, and often enable the unique features seen in different database servers and versions. While there are multiple DBAPIs available for many of the databases, the following instructions focus on the most common:

PostgreSQL
: Psycopg2 provides wide support for PostgreSQL versions and features and can be installed with `pip install psycopg2`.

MySQL
: PyMySQL is my preferred Python library for connecting to a MySQL database server. It can be installed with a `pip install pymysql`. MySQL support in SQLAlchemy requires MySQL version 4.1 and higher due to the way passwords worked prior to that version. Also, if a particular statement type is only available in a certain version of MySQL, SQLAlchemy does not provide a method to use those statements on versions of MySQL where the statement isn't available. It's important to review the MySQL documentation if a particular component or function in SQLAlchemy does not seem to work in your environment.

Others
: SQLAlchemy can also be used in conjunction with Drizzle, Firebird, Oracle, Sybase, and Microsoft SQL Server. The community has also supplied external dialects for many other databases like IBM DB2, Informix, Amazon Redshift, EXASolution, SAP SQL Anywhere, Monet, and many others. Creating an additional dialect is well supported by SQLAlchemy, and Chapter 7 will examine the process of doing just that.

Now that we have SQLAlchemy and a DBAPI installed, let's actually build an engine to connect to a database.

Connecting to a Database

To connect to a database, we need to create a SQLAlchemy engine. The SQLAlchemy engine creates a common interface to the database to execute SQL statements. It does this by wrapping a pool of database connections and a dialect in such a way that they can work together to provide uniform access to the backend database. This enables our Python code not to worry about the differences between databases or DBAPIs.

SQLAlchemy provides a function to create an engine for us given a *connection string* and optionally some additional keyword arguments. A connection string is a specially formatted string that provides:

- Database type (Postgres, MySQL, etc.)
- Dialect unless the default for the database type (Psycopg2, PyMySQL, etc.)
- Optional authentication details (username and password)
- Location of the database (file or hostname of the database server)
- Optional database server port
- Optional database name

SQLite database connections strings have us represent a specific file or a storage location. Example P-1 defines a SQLite database file named *cookies.db* stored in the current directory via a relative path in the second line, an in-memory database on the third line, and a full path to the file on the fourth (Unix) and fifth (Windows) lines. On Windows, the connection string would look like `engine4`; the `\\` are required for proper string escaping unless you use a raw string (`r''`).

Example P-1. Creating an engine for a SQLite database

```
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')
```

The `create_engine` function returns an instance of an engine; however, it does not actually open a connection until an action is called that would require a connection, such as a query.

Let's create an engine for a local PostgreSQL database named `mydb`. We'll start by importing the `create_engine` function from the base `sqlalchemy` package. Next, we'll use that function to construct an engine instance. In Example P-2, you'll notice that I use `postgresql+psycopg2` as the engine and dialect components of the connection string, even though using only `postgres` will work. This is because I prefer to be explicit instead of implicit, as recommended in the Zen of Python.

Example P-2. Creating an engine for a local PostgreSQL database

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

Now let's look at a MySQL database on a remote server. You'll notice, in Example P-3, that after the connection string we have a keyword parameter, `pool_recycle`, to define how often to recycle the connections.

Example P-3. Creating an engine for a remote MySQL database

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

By default, MySQL closes connections idle for more than eight hours. To work around this issue, use `pool_recycle=3600` when creating an engine, as shown in Example P-3.

Some optional keywords for the `create_engine` function are:

`echo`
: This will log the actions processed by the engine, such as SQL statements and their parameters. It defaults to false.

`encoding`
: This defines the string encoding used by SQLAlchemy. It defaults to `utf-8`, and most DBAPIs support this encoding by default. This does not define the encoding type used by the backend database itself.

`isolation_level`
: This instructs SQLAlchemy to use a specific isolation level. For example, PostgreSQL with Psycopg2 has `READ COMMITTED`, `READ UNCOMMITTED`, `REPEATABLE READ`, `SERIALIZABLE`, and `AUTOCOMMIT` available with a default of `READ COMMITTED`. PyMySQL has the same options with a default of `REPEATABLE READ` for InnoDB databases.

Using the `isolation_level` keyword argument will set the isolation level for any given DBAPI. This functions the same as doing it via a key-value pair in the connection string in dialects such as Psycopg2 that support that method.

`pool_recycle`
: This recycles or times out the database connections at regular intervals. This is important for MySQL due to the connection timeouts we mentioned earlier. It defaults to `-1`, which means there is no timeout.

Once we have an engine initialized, we are ready to actually open a connection to the database. That is done by calling the `connect()` method on the engine as shown here:

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

Now that we have a database connection, we can start using either SQLAlchemy Core or the ORM. In Part I, we will begin exploring SQLAlchemy Core and learning how to define and query your database.

PART I

SQLAlchemy Core

Now that we can connect to databases, let's begin looking at how to use SQLAlchemy Core to provide database services to our applications. SQLAlchemy Core is a Pythonic way of representing elements of both SQL commands and data structures called SQL Expression Language. SQLAlchemy Core can be used with either the Django or SQLAlchemy ORM, or can be used as a standalone solution.

CHAPTER 1

Schema and Types

The first thing we must do is define what data our tables hold, how that data is interrelated, and any constraints on that data.

In order to provide access to the underlying database, SQLAlchemy needs a representation of the tables that should be present in the database. We can do this in one of three ways:

- Using user-defined `Table` objects
- Using declarative classes that represent your tables
- Inferring them from the database

This chapter focuses on the first of these, as that is the approach used with SQLAlchemy Core; we'll cover the other two options in later chapters after we have a grasp of the fundamentals. The `Table` objects contain a list of typed columns and their attributes, which are associated with a common metadata container. We'll begin our exploration of schema definitions by taking a look at the types that are available to build tables in SQLAlchemy.

Types

There are four categories of types we can use inside of SQLAlchemy:

- Generic
- SQL standard
- Vendor specific
- User defined

SQLAlchemy defines a large number of generic types that are abstracted away from the actual SQL types supported by each backend database. These types are all available in the `sqlalchemy.types` module, and for convenience they are also available in the `sqlalchemy` module. So let's think about how these generic types are useful.

The Boolean generic type typically uses the `BOOLEAN` SQL type, and on the Python side deals in true or false; however, it also uses `SMALLINT` on backend databases that don't support a `BOOLEAN` type. Thanks to SQLAlchemy, this minor detail is hidden from you, and you can trust that any queries or statements you build will operate properly against fields of that type regardless of the database type being used. You will only have to deal with true or false in your Python code. This kind of behavior makes the generic types very powerful, and useful during database transitions or split backend systems where the data warehouse is one database type and the transactional is another. The generic types and their associated type representations in both Python and SQL can be seen in Table 1-1.

Table 1-1. Generic type representations

SQLAlchemy	Python	SQL
`BigInteger`	`int`	`BIGINT`
`Boolean`	`bool`	`BOOLEAN` or `SMALLINT`
`Date`	`datetime.date`	`DATE` (SQLite: `STRING`)
`DateTime`	`datetime.datetime`	`DATETIME` (SQLite: `STRING`)
`Enum`	`str`	`ENUM` or `VARCHAR`
`Float`	`float` or `Decimal`	`FLOAT` or `REAL`
`Integer`	`int`	`INTEGER`
`Interval`	`datetime.timedelta`	`INTERVAL` or `DATE from epoch`
`LargeBinary`	`byte`	`BLOB` or `BYTEA`
`Numeric`	`decimal.Decimal`	`NUMERIC` or `DECIMAL`
`Unicode`	`unicode`	`UNICODE` or `VARCHAR`
`Text`	`str`	`CLOB` or `TEXT`
`Time`	`datetime.time`	`DATETIME`

It is important to learn these generic types, as you will need to use and define them regularly.

In addition to the generic types listed in Table 1-1, both SQL standard and vendor-specific types are available and are often used when a generic type will not operate as needed within the database schema due to its type or the specific type specified in an existing schema. A few good illustrations of this are the `CHAR` and `NVARCHAR` types, which benefit from using the proper SQL type instead of just the generic type. If we are working with a database schema that was defined prior to using SQLAlchemy, we would want to match types exactly. It's important to keep in mind that SQL standard type behavior and availability can vary from database to database. The SQL standard types are available within the `sqlalchemy.types` module. To help make a distinction between them and the generic types, the standard types are in all capital letters.

Vendor-specific types are useful in the same ways as SQL standard types; however, they are only available in specific backend databases. You can determine what is available via the chosen dialect's documentation or SQLALchemy's website. They are available in the `sqlalchemy.dialects` module and there are submodules for each database dialect. Again, the types are in all capital letters for distinction from the generic types. We might want to take advantage of the powerful JSON field from PostgreSQL, which we can do with the following statement:

```
from sqlalchemy.dialects.postgresql import JSON
```

Now we can define JSON fields that we can later use with the many PostgreSQL-specific JSON functions, such as `array_to_json`, within our application.

You can also define custom types that cause the data to be stored in a manner of your choosing. An example of this might be prepending characters onto text stored in a `VARCHAR` column when put into the database record, and stripping them off when retrieving that field from the record. This can be useful when working with legacy data still used by existing systems that perform this type of prefixing that isn't useful or important in your new application.

Now that we've seen the four variations of types we can use to construct tables, let's take a look at how the database structure is held together by metadata.

Metadata

Metadata is used to tie together the database structure so it can be quickly accessed inside SQLAlchemy. It's often useful to think of metadata as a kind of catalog of `Table` objects with optional information about the engine and the connection. Those tables can be accessed via a dictionary, `MetaData.tables`. Read operations are thread-safe;

however, table construction is not completely thread-safe. Metadata needs to be imported and initialized before objects can be tied to it. Let's initialize an instance of the `MetaData` objects that we can use throughout the rest of the examples in this chapter to hold our information catalog:

```
from sqlalchemy import MetaData
metadata = MetaData()
```

Once we have a way to hold the database structure, we're ready to start defining tables.

Tables

`Table` objects are initialized in SQLAlchemy Core in a supplied `MetaData` object by calling the `Table` constructor with the table name and metadata; any additional arguments are assumed to be column objects. There are also some additional keyword arguments that enable features that we will discuss later. Column objects represent each field in the table. The columns are constructed by calling `Column` with a name, type, and then arguments that represent any additional SQL constructs and constraints. For the remainder of this chapter, we are going to build up a set of tables that we'll use throughout Part I. In Example 1-1, we create a table that could be used to store the cookie inventory for our online cookie delivery service.

Example 1-1. Instantiating Table objects and columns

```
from sqlalchemy import Table, Column, Integer, Numeric, String, ForeignKey

cookies = Table('cookies', metadata,
    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('quantity', Integer()),
    Column('unit_cost', Numeric(12, 2)) ❸
)
```

❶ Notice the way we marked this column as the table's primary key. More on this in a second.

❷ We're making an index of cookie names to speed up queries on this column.

❸ This is a column which takes multiple arguments, length and precision, such as 11.2, which would give us numbers up to 11 digits long with two decimal places.

Before we get too far into tables, we need to understand their fundamental building blocks: columns.

Columns

Columns define the fields that exists in our tables, and they provide the primary means by which we define other constraints through their keyword arguments. Different types of columns have different primary arguments. For example, `String` type columns have length as their primary argument, while numbers with a fractional component will have precision and length. Most other types have no primary arguments.

Sometimes you will see examples that just show `String` columns without a length, which is the primary argument. This behavior is not universally supported—for example, MySQL and several other database backends do not allow for it.

Columns can also have some additional keyword arguments that help shape their behavior even further. We can mark columns as required and/or force them to be unique. We can also set default initial values and change values when the record is updated. A common use case for this is fields that indicate when a record was created or updated for logging or auditing purposes. Let's take a look at these keyword arguments in action in Example 1-2.

Example 1-2. Another Table with more Column options

```
from datetime import datetime
from sqlalchemy import DateTime

users = Table('users', metadata,
    Column('user_id', Integer(), primary_key=True),
    Column('username', String(15), nullable=False, unique=True), ❶
    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), ❷
    Column('updated_on', DateTime(), default=datetime.now, onupdate=datetime.now) ❸
)
```

❶ Here we are making this column required (`nullable=False`) and also requiring a unique value.

❷ The default sets this column to the current time if a date isn't specified.

❸ Using `onupdate` here will reset this column to the current time every time any part of the record is updated.

You'll notice that we set `default` and `onupdate` to the callable `datetime.now` instead of the function call itself, `datetime.now()`. If we had used the function call itself, it would have set the default to the time when the table was first instantiated. By using the callable, we get the time that each individual record is instantiated and updated.

We've been using column keyword arguments to define table constructs and constraints; however, it is also possible to declare them outside of a `Column` object. This is critical when you are working with an existing database, as you must tell SQLAlchemy the schema, constructs, and constraints present inside the database. For example, if you have an existing index in the database that doesn't match the default index naming schema that SQLAlchemy uses, then you must manually define this index. The following two sections show you how to do just that.

All of the commands in "Keys and Constraints" on page 6 and "Indexes" on page 7 are included as part of the `Table` constructor or added to the table via special methods. They will be persisted or attached to the metadata as standalone statements.

Keys and Constraints

Keys and constraints are used as a way to ensure that our data meets certain requirements prior to being stored in the database. The objects that represent keys and constraints can be found inside the base SQLAlchemy module, and three of the more common ones can be imported as shown here:

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

The most common key type is a primary key, which is used as the unique identifier for each record in a database table and is used used to ensure a proper relationship between two pieces of related data in different tables. As you saw earlier in Example 1-1 and Example 1-2, a column can be made a primary key simply by using the `primary_key` keyword argument. You can also define composite primary keys by assigning the setting `primary_key` to `True` on multiple columns. The key will then essentially be treated like a tuple in which the columns marked as a key will be present in the order they were defined in the table. Primary keys can also be defined after the columns in the table constructor, as shown in the following snippet. You can add multiple columns separated by commas to create a composite key. If we wanted to explicitly define the key as shown in Example 1-2, it would look like this:

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

Another common constraint is the unique constraint, which is used to ensure that no two values are duplicated in a given field. For our online cookie delivery service, for example, we would want to ensure that each customer had a unique username to log

into our systems. We can also assign unique constraints on columns, as shown before in the username column, or we can define them manually as shown here:

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

Not shown in Example 1-2 is the check constraint type. This type of constraint is used to ensure that the data supplied for a column matches a set of user-defined criteria. In the following example, we are ensuring that `unit_cost` is never allowed to be less than 0.00 because every cookie costs something to make (remember from Economics 101: TINSTAAFC—that is, there is no such thing as a free cookie!):

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

In addition to keys and constraints, we might also want to make lookups on certain fields more efficient. This is where indexes come in.

Indexes

Indexes are used to accelerate lookups for field values, and in Example 1-1, we created an index on the `cookie_name` column because we know we will be searching by that often. When indexes are created as shown in that example, you will have an index called `ix_cookies_cookie_name`. We can also define an index using an explicit construction type. Multiple columns can be designated by separating them by a comma. You can also add a keyword argument of `unique=True` to require the index to be unique as well. When creating indexes explicitly, they are passed to the `Table` constructor after the columns. To mimic the index created in Example 1-1, we could do it explicitly as shown here:

```
from sqlalchemy import Index
Index('ix_cookies_cookie_name', 'cookie_name')
```

We can also create functional indexes that vary a bit by the backend database being used. This lets you create an index for situations where you often need to query based on some unusual context. For example, what if we want to select by cookie SKU and name as a joined item, such as *SKU0001 Chocolate Chip*? We could define an index like this to optimize that lookup:

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

Now it is time to dive into the most important part of relational databases: table relationships and how to define them.

Relationships and ForeignKeyConstraints

Now that we have a table with columns with all the right constraints and indexes, let's look at how we create relationships between tables. We need a way to track orders, including line items that represent each cookie and quantity ordered. To help visualize how these tables should be related, take a look at Figure 1-1.

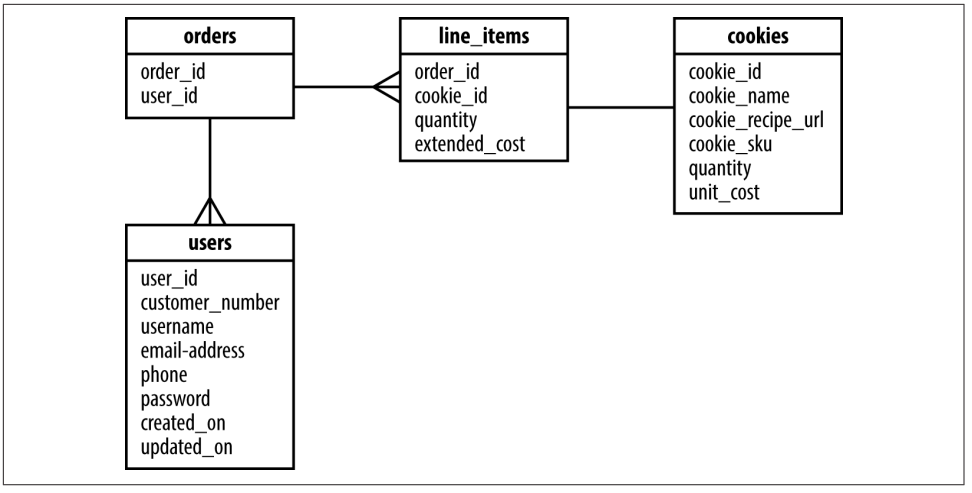


Figure 1-1. Relationship visualization

One way to implement a relationship is shown in Example 1-3 in the `line_items` table on the `order_id` column; this will result in a `ForeignKeyConstraint` to define the relationship between the two tables. In this case, many line items can be present for a single order. However, if you dig deeper into the `line_items` table, you'll see that we also have a relationship with the `cookies` table via the `cookie_id` `ForeignKey`. This is because `line_items` is actually an association table with some additional data on it between orders and cookies. Association tables are used to enable many-to-many relationships between two other tables. A single `ForeignKey` on a table is typically a sign of a one-to-many relationship; however, if there are multiple `ForeignKey` relationships on a table, there is a strong possibility that it is an association table.

Example 1-3. More tables with relationships

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

)

line_items = Table('line_items', metadata,
    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))
)
```

❶ Notice that we used a string instead of an actual reference to the column.

Using strings instead of an actual column allows us to separate the table definitions across multiple modules and/or not have to worry about the order in which our tables are loaded. This is because SQLAlchemy will only perform the resolution of that string to a table name and column the first time it is accessed. If we use hard references, such as `cookies.c.cookie_id`, in our `ForeignKey` definitions it will perform that resolution during module initialization and could fail depending on the order in which the tables are loaded.

You can also define a `ForeignKeyConstraint` explicitly, which can be useful if trying to match an existing database schema so it can be used with SQLAlchemy. This works in the same way as before when we created keys, constraints, and indexes to match name schemes and so on. You will need to import the `ForeignKeyConstraint` from the `sqlalchemy` module prior to defining one in your table definition. The following code shows how to create the `ForeignKeyConstraint` for the `order_id` field between the `line_items` and `orders` table:

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

Up until this point, we've been defining tables in such a way that SQLAlchemy can understand them. If your database already exists and has the schema already built, you are ready to begin writing queries. However, if you need to create the full schema or add a table, you'll want to know how to persist these in the database for permanent storage.

Persisting the Tables

All of our tables and additional schema definitions are associated with an instance of `metadata`. Persisting the schema to the database is simply a matter of calling the `create_all()` method on our `metadata` instance with the engine where it should create those tables:

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

By default, `create_all` will not attempt to re-create tables that already exist in the database, and it is safe to run multiple times. It's wiser to use a database migration tool like Alembic to handle any changes to existing tables or additional schema than to try to handcode changes directly in your application code (we'll explore this more fully in Chapter 11). Now that we have persisted the tables in the database, let's take a look at Example 1-4, which shows the complete code for the tables we've been working on in this chapter.

Example 1-4. Full in-memory SQLite code sample

```
from datetime import datetime

from sqlalchemy import (MetaData, Table, Column, Integer, Numeric, String,
                        DateTime, ForeignKey, create_engine)
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('quantity', Integer()),
    Column('unit_cost', Numeric(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)
)

orders = Table('orders', metadata,
    Column('order_id', Integer(), primary_key=True),
    Column('user_id', ForeignKey('users.user_id'))
)

line_items = Table('line_items', metadata,
    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)
```

In this chapter, we took a look at how metadata is used as a catalog by SQLAlchemy to store table schemas along with other miscellaneous data. We also can define a table with multiple columns and constraints. We explored the types of constraints and how to explicitly construct them outside of a column object to match an existing schema or naming scheme. Then we covered how to set default values and `onupdate` values for auditing. Finally, we now know how to persist or save our schema into the data-

base for reuse. The next step is to learn how to work with data within our schema via the SQL Expression Language.

CHAPTER 2

Working with Data via SQLAlchemy Core

Now that we have tables in our database, let's start working with data inside of those tables. We'll look at how to insert, retrieve, and delete data, and follow that with learning how to sort, group, and use relationships in our data. We'll be using the SQL Expression Language (SEL) provided by SQLAlchemy Core. We're going to continue using the tables we created in Chapter 1 for our examples in this chapter. Let's start by learning how to insert data.

Inserting Data

First, we'll build an `insert` statement to put my favorite kind of cookie (chocolate chip) into the `cookies` table. To do this, we can call the `insert()` method on the `cookies` table, and then use the `values()` statement with keyword arguments for each column that we are filling with data. Example 2-1 does just that.

Example 2-1. Single insert as a method

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

In Example 2-1, `print(str(ins))` shows us the actual SQL statement that will be executed:

```
    INSERT INTO cookies
        (cookie_name, cookie_recipe_url, cookie_sku, quantity, unit_cost)
```

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

Our supplied values have been replaced with `:column_name` in this SQL statement, which is how SQLAlchemy represents parameters displayed via the `str()` function. Parameters are used to help ensure that our data has been properly escaped, which mitigates security issues such as SQL injection attacks. It is still possible to view the parameters by looking at the compiled version of our insert statement, because each database backend can handle the parameters in a slightly different manner (this is controlled by the dialect). The `compile()` method on the `ins` object returns a `SQLCompiler` object that gives us access to the actual parameters that will be sent with the query via the `params` attribute:

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

This compiles the statement via our dialect but does not execute it, and we access the `params` attribute of that statement.

This results in:

```
{
    'cookie_name': 'chocolate chip',
    'cookie_recipe_url': 'http://some.aweso.me/cookie/recipe.html',
    'cookie_sku': 'CC01',
    'quantity': '12',
    'unit_cost': '0.50'
}
```

Now that we have a complete picture of the insert statement and understand what is going to be inserted into the table, we can use the `execute()` method on our connection to send the statement to the database, which will insert the record into the table (Example 2-2).

Example 2-2. Executing the insert statement

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

We can also get the ID of the record we just inserted by accessing the `inserted_primary_key` attribute:

```
result.inserted_primary_key
```

```
[1]
```

Let's take a quick detour here to discuss what happens when we call the `execute()` method. When we are building a SQL Expression Language statement like the insert statement we've been using so far, it is actually creating a tree-like structure that can be quickly traversed in a descending manner. When we call the `execute` method, it uses the statement and any other parameters passed to compile the statement with the

proper database dialect's compiler. That compiler builds a normal parameterized SQL statement by walking down that tree. That statement is returned to the `execute` method, which sends the SQL statement to the database via the connection on which the method was called. The database server then executes the statement and returns the results of the operation.

In addition to having `insert` as an instance method off a `Table` object, it is also available as a top-level function for those times that you want to build a statement "generatively" (a step at a time) or when the table may not be initially known. For example, our company might run two separate divisions, each with its own separate inventory tables. Using the `insert` function shown in Example 2-3 would allow us to use one statement and just swap the tables.

Example 2-3. Insert function

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

❶ Notice the table is now the argument to the `insert` function.

While `insert` works equally well as a method of a `Table` object and as a more generative standalone function, I prefer the generative approach because it more closely mirrors the SQL statements people are accustomed to seeing.

The `execute` method of the connection object can take more than just statements. It is also possible to provide the values as keyword arguments to the `execute` method after our statement. When the statement is compiled, it will add each one of the keyword argument keys to the columns list, and it adds each one of their values to the `VALUES` part of the SQL statement (Example 2-4).

Example 2-4. Values in execute statement

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

```
    quantity='1',
    unit_cost='0.75'
)
result.inserted_primary_key
```

❶ Our insert statement is the first argument to the `execute` function just like before.

❷ We add our values as keyword arguments to the `execute` function.

This results in:

```
[2]
```

While this isn't used often in practice for single inserts, it does provide a good illustration of how a statement is compiled and assembled prior to being sent to the database server. We can insert multiple records at once by using a list of dictionaries with data we are going to submit. Let's use this knowledge to insert two types of cookies into the `cookies` table (Example 2-5).

Example 2-5. Multiple inserts

```
inventory_list = [ ❶
    {
        'cookie_name': 'peanut butter',
        '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'
    }
]
result = connection.execute(ins, inventory_list) ❷
```

❶ Build our list of cookies.

❷ Use the list as the second parameter to execute.

The dictionaries in the list must have the exact same keys. SQLAlchemy will compile the statement against the first dictionary in the list, and the statement will fail if subsequent dictionaries are different, as the statement was already built with the prior columns.

Now that we have some data in our `cookies` table, let's learn how to query the tables and retrieve that data.

Querying Data

To begin building a query, we start by using the `select` function, which is analogous to the standard SQL `SELECT` statement. Initially, let's select all the records in our `cookies` table (Example 2-6).

Example 2-6. Simple select function

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

❶ Remember we can use `str(s)` to look at the SQL statement the database will see, which in this case is `SELECT cookies.cookie_id, cookies.cookie_name, cookies.cookie_recipe_url, cookies.cookie_sku, cookies.quantity, cookies.unit_cost FROM cookies`.

❷ This tells `rp`, the `ResultProxy`, to return all the rows.

The `results` variable now contains a list representing all the records in our `cookies` table:

```
[(1, u'chocolate chip', 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'peanut butter', 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'EWW01',
  100, Decimal('1.00'))]
```

In the preceding example, I passed a list containing the `cookies` table. The `select` method expects a list of columns to select; however, for convenience, it also accepts `Table` objects and selects all the columns on the table. It is also possible to use the `select` method on the `Table` object to do this, as shown in Example 2-7. Again, I prefer seeing it written more like Example 2-6.

Example 2-7. Simple select method

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

Before we go digging any further into queries, we need to know a bit more about `ResultProxy` objects.

ResultProxy

A `ResultProxy` is a wrapper around a DBAPI cursor object, and its main goal is to make it easier to use and manipulate the results of a statement. For example, it makes handling query results easier by allowing access using an index, name, or Column object. Example 2-8 demonstrates all three of these methods. It's very important to become comfortable using each of these methods to get to the desired column data.

Example 2-8. Handling rows with a ResultProxy

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

❶ Get the first row of the `ResultProxy` from Example 2-7.

❷ Access column by index.

❸ Access column by name.

❹ Access column by `Column` object.

These all result in `u'chocolate chip'` and they each reference the exact same data element in the first record of our `results` variable. This flexibility in access is only part of the power of the `ResultProxy`. We can also leverage the `ResultProxy` as an iterable, and perform an action on each record returned without creating another variable to hold the results. For example, we might want to print the name of each cookie in our database (Example 2-9).

Example 2-9. Iterating over a ResultProxy

```
rp = connection.execute(s) ❶
for record in rp:
    print(record.cookie_name)
```

❶ We are reusing the same select statement from earlier.

This returns:

```
    chocolate chip
    dark chocolate chip
```

```
peanut butter
oatmeal raisin
```

In addition to using the `ResultProxy` as an iterable or calling the `fetchall()` method, many other ways of accessing data via the `ResultProxy` are available. In fact, all the `result` variables in "Inserting Data" on page 13 were actually `ResultProxys`. Both the `rowcount()` and `inserted_primary_key()` methods we used in that section are just a couple of the other ways to get information from a `ResultProxy`. You can use the following methods as well to fetch results:

`first()`
: Returns the first record if there is one and closes the connection.

`fetchone()`
: Returns one row, and leaves the cursor open for you to make additional fetch calls.

`scalar()`
: Returns a single value if a query results in a single record with one column.

If you want to see the columns that are available in a result set you can use the `keys()` method to get a list of the column names. We'll be using the `first`, `scalar`, `fetchone`, and `fetchall` methods as well as the `ResultProxy` as an iterable throughout the remainder of this chapter.

Tips for Good Production Code

When writing production code, you should follow these guidelines:

- Use the `first` method for getting a single record over both the `fetchone` and `scalar` methods, because it is clearer to our fellow coders.
- Use the iterable version of the `ResultProxy` over the `fetchall` and `fetchone` methods. It is more memory efficient and we tend to operate on the data one record at a time.
- Avoid the `fetchone` method, as it leaves connections open if you are not careful.
- Use the `scalar` method sparingly, as it raises errors if a query ever returns more than one row with one column, which often gets missed during testing.

Every time we queried the database in the preceding examples, all the columns were returned for every record. Often we only need a portion of those columns to perform our work. If the data in these extra columns is large, it can cause our applications to slow down and consume far more memory than it should. SQLAlchemy does not add a bunch of overhead to the queries or `ResultProxys`; however, accounting for the

data you get back from a query is often the first place to look if a query is consuming too much memory. Let's look at how to limit the columns returned in a query.

Controlling the Columns in the Query

To limit the fields that are returned from a query, we need to pass the columns we want into the `select()` method constructor as a list. For example, you might want to run a query, that returns only the name and quantity of cookies, as shown in Example 2-10.

Example 2-10. Select only cookie_name and quantity

```
s = select([cookies.c.cookie_name, cookies.c.quantity])
rp = connection.execute(s)
print(rp.keys()) ❶
result = rp.first() ❷
```

❶ Returns the list of columns, which is `[u'cookie_name', u'quantity']` in this example (used only for demonstration, not needed for results).

❷ Notice this only returns the first result.

Result:

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

Now that we can build a simple select statement, let's look at other things we can do to alter how the results are returned in a select statement. We'll start with changing the order in which the results are returned.

Ordering

If you were to look at all the results from Example 2-10 instead of just the first record, you would see that the data is not really in any particular order. However, if we want the list to be returned in a particular order, we can chain an `order_by()` statement to our select, as shown in Example 2-11.In this case, we want the results to be ordered by the quantity of cookies we have on hand.

Example 2-11. Order by quantity ascending

```
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))
```

This results in:

```
1 - dark chocolate chip
12 - chocolate chip
24 - peanut butter
100 - oatmeal raisin
```

We saved the select statement into the `s` variable, used that `s` variable and added the `order_by` statement to it, and then reassigned that to the `s` variable. This is an example of how to compose statements in a generative or step-by-step fashion. This is the same as combining the `select` and the `order_by` statements into one line as shown here:

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

However, when we have the full list of columns in the select and the order columns in the `order_by` statement, it exceeds Python's 79 character per line limit (which was established in PEP8). By using the generative type statement, we can remain under that limit. Throughout the book, we'll see a few examples where this generative style can introduce additional benefits, such as conditionally adding things to the statement. For now, try to break your statements along those 79 character limits, and it will help make the code more readable.

If you want to sort in reverse or descending order, use the `desc()` statement. The `desc()` function wraps the specific column you want to sort in a descending manner, as shown in Example 2-12.

Example 2-12. Order by quantity descending

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

❶ Notice we are wrapping the `cookies.c.quantity` column in the `desc()` function.

The `desc()` function can also be used as a method on a `Column` object, such as `cookies.c.quantity.desc()`. However, that can be a bit more confusing to read in long statements, so I always use `desc()` as a function.

It's also possible to limit the number of results returned if we only need a certain number of them for our application.

Limiting

In prior examples, we used the `first()` or `fetchone()` methods to get just a single row back. While our `ResultProxy` gave us the one row we asked for, the actual query ran over and accessed all the results, not just the single record. If we want to limit the

query, we can use the `limit()` function to actually issue a limit statement as part of our query. For example, suppose you only have time to make two batches of cookies, and you want to know which two cookie types you should make. You can use our ordered query from earlier and add a limit statement to return the two cookie types that are most in need of being replenished (Example 2-13).

Example 2-13. Two smallest cookie inventories

```
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([result.cookie_name for result in rp]) ❶
```

❶ Here we are using the iterable capabilities of the `ResultsProxy` in a list comprehension.

This results in:

```
[u'dark chocolate chip', u'chocolate chip']
```

Now that you know what kind of cookies you need to bake, you're probably starting to get curious about how many cookies are now left in your inventory. Many databases include SQL functions designed to make certain operations available directly on the database server, such as SUM; let's explore how to use these functions next.

Built-In SQL Functions and Labels

SQLAlchemy can also leverage SQL functions found in the backend database. Two very commonly used database functions are `SUM()` and `COUNT()`. To use these functions, we need to import the `sqlalchemy.sql.func` module where they are found. These functions are wrapped around the column(s) on which they are operating. Thus, to get a total count of cookies, you would use something like Example 2-14.

Example 2-14. Summing our cookies

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

❶ Notice the use of `scalar`, which will return only the leftmost column in the first record.

This results in:

```
137
```

I tend to always import the `func` module, as importing `sum` directly can cause problems and confusion with Python's built-in `sum` function.

Now let's use the `count` function to see how many cookie inventory records we have in our `cookies` table (Example 2-15).

Example 2-15. Counting our inventory records

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

❶ This will show us the columns in the `ResultProxy`.

❷ The column name is autogenerated and is commonly `<func_name>_<position>`.

This results in:

```
[u'count_1']
4
```

This column name is annoying and cumbersome. Also, if we have several counts in a query, we'd have to know the occurrence number in the statement, and incorporate that into the column name, so the fourth `count()` function would be `count_4`. This simply is not as explicit and clear as we should be in our naming, especially when surrounded with other Python code. Thankfully, SQLAlchemy provides a way to fix this via the `label()` function. Example 2-16 performs the same query as Example 2-15; however, it uses `label` to give us a more useful name to access that column.

Example 2-16. Renaming our count column

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

❶ Notice that we just use the `label()` function on the column object we want to change.

This results in:

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

We've seen examples of how to restrict the columns or the number of rows returned from the database, so now it's time to learn about queries that filter data based on criteria we specify.

Filtering

Filtering queries is done by adding `where()` statements just like in SQL. A typical `where()` clause has a column, an operator, and a value or column. It is possible to chain multiple `where()` clauses together, and they will act like ANDs in traditional SQL statements. In Example 2-17, we'll find a cookie named "chocolate chip".

Example 2-17. Filtering by cookie name

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

❶ Here I'm calling the `items()` method on the row object, which will give me a list of columns and values.

This results in:

```
[
    (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'))
]
```

We can also use a `where()` statement to find all the cookie names that contain the word "chocolate" (Example 2-18).

Example 2-18. Finding names with chocolate in them

```
s = select([cookies]).where(cookies.c.cookie_name.like('%chocolate%'))
rp = connection.execute(s)
for record in rp.fetchall():
    print(record.cookie_name)
```

This results in:

```
chocolate chip
dark chocolate chip
```

In the `where()` statement of Example 2-18, we are using the `cookies.c.cookie_name` column inside of a `where()` statement as a type of `ClauseElement` to filter our results. We should take a brief moment and talk more about `ClauseElements` and the additional capabilities they provide.

ClauseElements

`ClauseElements` are just an entity we use in a clause, and they are typically columns in a table; however, unlike columns, `ClauseElements` come with many additional capabilities. In Example 2-18, we are taking advantage of the `like()` method that is available on `ClauseElements`. There are many other methods available, which are listed in Table 2-1. Each of these methods are analogous to a standard SQL statement construct. You'll find various examples of these used throughout the book.

Table 2-1. ClauseElement methods

Method	Purpose
`between(cleft, cright)`	Find where the column is between `cleft` and `cright`
`concat(column_two)`	Concatenate column with `column_two`
`distinct()`	Find only unique values for the column
`in_([list])`	Find where the column is in the list
`is_(None)`	Find where the column is `None` (commonly used for Null checks with None)
`contains(`*`string`*`)`	Find where the column has *`string`* in it (case-sensitive)
`endswith(`*`string`*`)`	Find where the column ends with *`string`* (case-sensitive)
`like(`*`string`*`)`	Find where the column is like *`string`* (case-sensitive)
`startswith(`*`string`*`)`	Find where the column begins with *`string`* (case-sensitive)
`ilike(`*`string`*`)`	Find where the column is like *`string`* (this is *not* case-sensitive)

There are also negative versions of these methods, such as `notlike` and `notin_()`. The only exception to the `not`*`<method>`* naming convention is the `isnot()` method, which drops the underscore.

If we don't use one of the methods listed in Table 2-1, then we will have an operator in our `where` clauses. Most of the operators work as you might expect; however, we should discuss operators in a bit more detail, as there are a few differences.

Operators

So far, we have only explored where a column was equal to a value or used one of the `ClauseElement` methods such as `like()`; however, we can also use many other common operators to filter data. SQLAlchemy provides overloading for most of the standard Python operators. This includes all the standard comparison operators (`==`, `!=`, `<`, `>`, `<=`, `>=`), which act exactly like you would expect in a Python statement. The `==` operator also gets an additional overload when compared to `None`, which converts it to an `IS NULL` statement. Arithmetic operators (`\+`, `-`, `*`, `/`, and `%`) are also supported with additional capabilities for database-independent string concatenation, as shown in Example 2-19.

Example 2-19. String concatenation with `\+`

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

This results in:

```
    (u'chocolate chip', u'SKU-CC01')
    (u'dark chocolate chip', u'SKU-CC02')
    (u'peanut butter', u'SKU-PB01')
    (u'oatmeal raisin', u'SKU-EWW01')
```

Another common usage of operators is to compute values from multiple columns. You'll often do this in applications and reports dealing with financial data or statistics. Example 2-20 shows a common inventory value calculation.

Example 2-20. Inventory value by cookie

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

❶ `Cast()` is another function that allows us to convert types. In this case, we will be getting back results such as 6.0000000000, so by casting it, we can make it look like currency. It is also possible to accomplish the same task in Python with `print('{} - {:.2f}'.format(row.cookie_name, row.inv_cost))`.

❷ Notice we are again using the `label()` function to rename the column. Without this renaming, the column would be named `anon_1`, as the operation doesn't result in a name.

This results in:

```
chocolate chip - 6.00
dark chocolate chip - 0.75
peanut butter - 6.00
oatmeal raisin - 100.00
```

Boolean Operators

SQLAlchemy also allows for the SQL Boolean operators AND, OR, and NOT via the bitwise logical operators (`&`, `|`, and `~`). Special care must be taken when using the AND, OR, and NOT overloads because of the Python operator precedence rules. For instance, `&` binds more closely than `<`, so when you write `A < B & C < D`, what you are actually writing is `A < (B&C) < D`, when you probably intended to get `(A < B) & (C < D)`. Please use conjunctions instead of these overloads, as they will make your code more expressive.

Often we want to chain multiple `where()` clauses together in inclusionary and exclusionary manners; this should be done via conjunctions.

Conjunctions

While it is possible to chain multiple `where()` clauses together, it's often more readable and functional to use conjunctions to accomplish the desired effect. The conjunctions in SQLAlchemy are `and_()`, `or_()`, and `not_()`. So if we wanted to get a list of cookies with a cost of less than an amount and above a certain quantity we could use the code shown in Example 2-21.

Example 2-21. Using the and() conjunction

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

The `or_()` function works as the opposite of `and_()` and includes results that match either one of the supplied clauses. If we wanted to search our inventory for cookie

types that we have between 10 and 50 of in stock or where the name contains *chip*, we could use the code shown in Example 2-22.

Example 2-22. Using the or() conjunction

```
from sqlalchemy import and_, or_, not_
s = select([cookies]).where(
    or_(
        cookies.c.quantity.between(10, 50),
        cookies.c.cookie_name.contains('chip')
    )
)
for row in connection.execute(s):
    print(row.cookie_name)
```

This results in:

```
    chocolate chip
    dark chocolate chip
    peanut butter
```

The `not_()` function works in a similar fashion to other conjunctions, and it simply is used to select records where a record does not match the supplied clause. Now that we can comfortably query data, we are ready to move on to updating existing data.

Updating Data

Much like the insert method we used earlier, there is also an update method with syntax almost identical to inserts, except that it can specify a `where` clause that indicates which rows to update. Like insert statements, update statements can be created by either the `update()` function or the `update()` method on the table being updated. You can update all rows in a table by leaving off the `where` clause.

For example, suppose you've finished baking those chocolate chip cookies that we needed for our inventory. In Example 2-23, we'll add them to our existing inventory with an update query, and then check to see how many we currently have in stock.

Example 2-23. Updating data

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

❶ Using the generative method of building our statement.

❷ Printing how many rows were updated.

This returns:

```
1
           cookie_id: 1
         cookie_name: chocolate chip
   cookie_recipe_url: http://some.aweso.me/cookie/recipe.html
          cookie_sku: CC01
            quantity: 132
           unit_cost: 0.50
```

In addition to updating data, at some point we will want to remove data from our tables. The following section explains how to do that.

Deleting Data

To create a delete statement, you can use either the `delete()` function or the `delete()` method on the table from which you are deleting data. Unlike `insert()` and `update()`, `delete()` takes no values parameter, only an optional `where` clause (omitting the `where` clause will delete all rows from the table). See Example 2-24.

Example 2-24. Deleting data

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

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

This returns:

```
1
0
```

OK, at this point, let's load up some data using what we already learned for the `users`, `orders`, and `line_items` tables. You can copy the code shown here, but you should also take a moment to play with different ways of inserting the data:

```
customer_list = [
    {
        '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()
result = connection.execute(ins, customer_list)
```

Now that we have customers, we can start to enter their orders and line items into the system as well:

```
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
    }
]
result = 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
```

```
        }
    ]
    result = connection.execute(ins, order_items)
```

In Chapter 1, you learned how to define `ForeignKeys` and relationships, but we haven't used them to perform any queries up to this point. Let's take a look at relationships next.

Joins

Now let's use the `join()` and `outerjoin()` methods to take a look at how to query related data. For example, to fulfill the order placed by the cookiemon user, we need to determine how many of each cookie type were ordered. This requires you to use a total of three joins to get all the way down to the name of the cookies. It's also worth noting that depending on how the joins are used in a relationship, you might want to rearrange the `from` part of a statement; one way to accomplish that in SQLAlchemy is via the `select_from()` clause. With `select_from()`, we can replace the entire `from` clause that SQLAlchemy would generate with one we specify (Example 2-25).

Example 2-25. Using join to select from multiple tables

```
columns = [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( ❶
                        line_items).join(cookies)).where(users.c.username ==
                            'cookiemon')
result = connection.execute(cookiemon_orders).fetchall()
for row in result:
    print(row)
```

❶ Notice we are telling SQLAlchemy to use the relationship joins as the `from` clause.

This results in:

```
    (u'1', u'cookiemon', u'111-111-1111', u'chocolate chip', 2, Decimal('1.00'))
    (u'1', u'cookiemon', u'111-111-1111', u'peanut butter', 12, Decimal('3.00'))
```

The SQL looks like this:

```
    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
```

It is also useful to get a count of orders by all users, including those who do not have any present orders. To do this, we have to use the `outerjoin()` method, and it

requires a bit more care in the ordering of the join, as the table we use the `outer join()` method on will be the one from which all results are returned (Example 2-26).

Example 2-26. Using outerjoin to select from multiple tables

```
columns = [users.c.username, func.count(orders.c.order_id)]
all_orders = select(columns)
all_orders = all_orders.select_from(users.outerjoin(orders)) ❶
all_orders = all_orders.group_by(users.c.username)
result = connection.execute(all_orders).fetchall()
for row in result:
    print(row)
```

❶ SQLAlchemy knows how to join the `users` and `orders` tables because of the foreign key defined in the `orders` table.

This results in:

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

So far, we have been using and joining different tables in our queries. However, what if we have a self-referential table like a table of employees and their bosses? To make this easy to read and understand, SQLAlchemy uses aliases.

Aliases

When using joins, it is often necessary to refer to a table more than once. In SQL, this is accomplished by using *aliases* in the query. For instance, suppose we have the following (partial) schema that tracks the reporting structure within an organization:

```
employee_table = Table(
    'employee', metadata,
    Column('id', Integer, primary_key=True),
    Column('manager', None, ForeignKey('employee.id')),
    Column('name', String(255)))
```

Now suppose we want to select all the employees managed by an employee named Fred. In SQL, we might write the following:

```
SELECT employee.name
FROM employee, employee AS manager
WHERE employee.manager_id = manager.id
    AND manager.name = 'Fred'
```

SQLAlchemy also allows the use of aliasing selectables in this type of situation via the `alias()` function or method:

```
>>> manager = employee_table.alias('mgr')
>>> stmt = select([employee_table.c.name],
...               and_(employee_table.c.manager_id==manager.c.id,
...                    manager.c.name=='Fred'))
>>> print(stmt)
SELECT employee.name
FROM employee, employee AS mgr
WHERE employee.manager_id = mgr.id AND mgr.name = ?
```

SQLAlchemy can also choose the alias name automatically, which is useful for guaranteeing that there are no name collisions:

```
>>> manager = employee_table.alias()
>>> stmt = select([employee_table.c.name],
...               and_(employee_table.c.manager_id==manager.c.id,
...                    manager.c.name=='Fred'))
>>> print(stmt)
SELECT employee.name
FROM employee, employee AS employee_1
WHERE employee.manager_id = employee_1.id AND employee_1.name = ?
```

It's also useful to be able to group data when we are looking to report on data, so let's look into that next.

Grouping

When using grouping, you need one or more columns to group on and one or more columns that it makes sense to aggregate with counts, sums, etc., as you would in normal SQL. Let's get an order count by customer (Example 2-27).

Example 2-27. Grouping data

```
columns = [users.c.username, func.count(orders.c.order_id)] ❶
all_orders = select(columns)
all_orders = all_orders.select_from(users.outerjoin(orders))
all_orders = all_orders.group_by(users.c.username) ❷
result = connection.execute(all_orders).fetchall()
for row in result:
    print(row)
```

❶ Aggregation via `count`

❷ Grouping by the nonaggregated included column

This results in:

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

We've shown the generative building of statements throughout the previous examples, but I want to focus on it specifically for a moment.

Chaining

We've used chaining several times throughout this chapter, and just didn't acknowledge it directly. Where query chaining is particularly useful is when you are applying logic when building up a query. So if we wanted to have a function that got a list of orders for us it might look like Example 2-28.

Example 2-28. Chaining

```
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()
    return result

get_orders_by_customer('cakeeater')
```

This results in:

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

However, what if we wanted to get only the orders that have shipped or haven't shipped yet? We'd have to write additional functions to support those additional desired filter options, or we can use conditionals to build up query chains. Another option we might want is whether or not to include details. This ability to chain queries and clauses together enables quite powerful reporting and complex query building (Example 2-29).

Example 2-29. Conditional chaining

```
def get_orders_by_customer(cust_name, shipped=None, 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)
    if shipped is not None:
        cust_orders = cust_orders.where(orders.c.shipped == shipped)
    result = connection.execute(cust_orders).fetchall()
    return result

get_orders_by_customer('cakeeater') ❶

get_orders_by_customer('cakeeater', details=True) ❷

get_orders_by_customer('cakeeater', shipped=True) ❸

get_orders_by_customer('cakeeater', shipped=False) ❹

get_orders_by_customer('cakeeater', shipped=False, details=True) ❺
```

❶ Gets all orders.

❷ Gets all orders with details.

❸ Gets only orders that have shipped.

❹ Gets orders that haven't shipped yet.

❺ Gets orders that haven't shipped yet with details.

This results in:

```
[(u'2', u'cakeeater', u'222-222-2222')]

[(u'2', u'cakeeater', u'222-222-2222', u'chocolate chip', 24, Decimal('12.00')),
 (u'2', u'cakeeater', u'222-222-2222', u'oatmeal raisin', 6, Decimal('6.00'))]

[]

[(u'2', u'cakeeater', u'222-222-2222')]

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

So far in this chapter, we've used the SQL Expression Language for all the examples; however, you might be wondering if you can execute standard SQL statements as well. Let's take a look.

Raw Queries

It is also possible to execute raw SQL statements or use raw SQL in part of a SQLAlchemy Core query. It still returns a `ResultProxy`, and you can continue to interact with it just as you would a query built using the SQL Expression syntax of

SQLAlchemy Core. I encourage you to only use raw queries and text when you must, as it can lead to unforeseen results and security vulnerabilities. First, we'll want to execute a simple select statement (Example 2-30).

Example 2-30. Full raw queries

```
result = connection.execute("select * from orders").fetchall()
print(result)
```

This results in:

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

While I rarely use a full raw SQL statement, I will often use small text snippets to help make a query clearer. Example 2-31 is of a raw SQL `where` clause using the `text()` function.

Example 2-31. Partial text query

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

This results in:

```
    [(1, None, u'cookiemon', 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))
    ]
```

Now you should have an understanding of how to use the SQL Expression Language to work with data in SQLAlchemy. We explored how to create, read, update, and delete operations. This is a good point to stop and explore a bit on your own. Try to create more cookies, orders, and line items, and use query chains to group them by order and user. Now that you've explored a bit more and hopefully broken something, let's investigate how to react to exceptions raised in SQLAlchemy, and how to use transactions to group statements that must succeed or fail as a group.

CHAPTER 3

Exceptions and Transactions

In the previous chapter, we did a lot of work with data in single statements, and we avoided doing anything that could result in an error. In this chapter, we will purposely perform some actions incorrectly so that we can see the types of errors that occur and how we should respond to them. We'll conclude the chapter by learning how to group statements that need to succeed together into transactions so that we can ensure that either the group executes properly or is cleaned up correctly. Let's start by blowing things up!

Exceptions

There are numerous exceptions that can occur in SQLAlchemy, but we'll focus on the most common ones: `AttributeErrors` and `IntegrityErrors`. By learning how to handle these common exceptions, you'll be better prepared to deal with the ones that occur less frequently.

To follow along with this chapter, make sure you start a new Python shell and load the tables that we built in Chapter 1 into your shell. Example 3-1 contains those tables and the connection again for reference.

Example 3-1. Setting up our shell environment

```
from datetime import datetime

from sqlalchemy import (MetaData, Table, Column, Integer, Numeric, String,
                        DateTime, ForeignKey, Boolean, create_engine,
                        CheckConstraint)
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('quantity', Integer()),
    Column('unit_cost', Numeric(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)
)

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

line_items = Table('line_items', metadata,
    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()
```

The first error we are going to learn about is the `AttributeError`; this is the most commonly encountered error I see in code I'm debugging.

AttributeError

We will start with an `AttributeError` that occurs when you attempt to access an attribute that doesn't exist. This often occurs when you are attempting to access a column on a `ResultProxy` that isn't present. `AttributeErrors` occur when you try to access an attribute of an object that isn't present on that object. You've probably run into this in normal Python code. I'm singling it out because this is a common error in SQLAlchemy and it's very easy to miss the reason why it is occuring. To demonstrate this error, let's insert a record into our `users` table and run a query against it. Then

we'll try to access a column on that table that we didn't select in the query (Example 3-2).

Example 3-2. Causing an AttributeError

```
from sqlalchemy import select, insert
ins = insert(users).values(
    username="cookiemon",
    email_address="mon@cookie.com",
    phone="111-111-1111",
    password="password"
)
result = 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 Inserting a test record.

❷ Password doesn't exist, as we only queried the username column.

The code in Example 3-2 causes Python to throw an `AttributeError` and stops the execution of our program. Let's look at the error output, and learn how to interpret what happened (Example 3-3).

Example 3-3. Error output from Example 3-2

```
cookiemon

AttributeError                            Traceback (most recent call last) ❶
<ipython-input-37-c4520631a10a> in <module>()
      3 for result in results:
      4     print(result.username)
----> 5     print(result.password) ❷

AttributeError: Could not locate column in row for column 'password' ❸
```

❶ This shows us the type of error and that a traceback is present.

❷ This is the actual line where the error occurred.

❸ This is the interesting part we need to focus on.

In Example 3-3, we have the typical format for an `AttributeError` in Python. It starts with a line that indicates the type of error. Next, there is a traceback showing us

where the error occurred. Because we tried to access the column in our code, it shows us the actual line that failed. The final block of lines is where the important details can be found. It again specifies the type of error, and right after it shows you why this occurred. In this case, it is because our row from the `ResultProxy` does not have a password column. We only queried for the username. While this is a common Python error that we can cause through a bug in our use of SQLAlchemy objects, there are also SQLAlchemy-specific errors that reference bugs we cause with SQLAlchemy statements themselves. Let's look at one example: the `IntegrityError`.

IntegrityError

Another common SQLAlchemy error is the `IntegrityError`, which occurs when we try to do something that would violate the constraints configured on a `Column` or `Table`. This type of error is commonly encountered in cases where you require something to be unique—for example, if you attempt to create two users with the same username, an `IntegrityError` will be thrown because usernames in our `users` table must be unique. Example 3-4 shows some code that will cause such an error.

Example 3-4. Causing an IntegrityError

```
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"
)
result = connection.execute(ins) ❷
```

❶ View the current records in the `users` table.

❷ Attempt to insert the second record, which will result in the error.

The code in Example 3-4 causes SQLAlchemy to create an `IntegrityError`. Let's look at the error output, and learn how to interpret what happened (Example 3-5).

Example 3-5. IntegrityError output

```
IntegrityError                            Traceback (most recent call last)
<ipython-input-7-6ecafb68a8ab> in <module>()
      5     password="password"
      6 )
```

```
----> 7 result = connection.execute(ins) ❶

... ❷

IntegrityError: (sqlite3.IntegrityError) UNIQUE constraint failed:
users.username [SQL: u'INSERT INTO users (username, email_address, phone,
password, created_on, updated_on) VALUES (?, ?, ?, ?, ?, ?)'] [parameters:
('cookiemon', 'damon@cookie.com', '111-111-1111', 'password',
'2015-04-26 10:52:24.275082', '2015-04-26 10:52:24.275099')] ❸
```

❶ This is the line that triggered the error.

❷ There is a long traceback here that I omitted.

❸ This is the interesting part we need to focus on.

Example 3-5 shows the typical format for an `IntegrityError` output in SQLAlchemy. It starts with the line that indicates the type of error. Next, it includes the traceback details. However, this is normally only our execute statement and internal SQLAlchemy code. Typically, the traceback can be ignored for the `IntegrityError` type. The final block of lines is where the important details are found. It again specifies the type of error, and tells you what caused it. In this case, it shows:

```
UNIQUE constraint failed: users.username
```

This points us to the fact that there is a unique constraint on the `username` column in the `users` table that we tried to violate. It then provides us with the details of the SQL statement and its compiled parameters similar to what we looked at in Chapter 2. The new data we tried to insert into the table was not inserted due to the error. This error also stops our program from executing.

While there are many other kinds of errors, the two we covered are the most common. The output for all the errors in SQLAlchemy will follow the same format as the two we just looked at. The SQLAlchemy documentation contains information on the other types of errors.

Because we don't want our programs to crash whenever they encounter an error, we need to learn how to handle errors properly.

Handling Errors

To prevent an error from crashing or stopping our program, errors need to be handled cleanly. We can do this just as we would for any Python error, with a `try/except` block. For example, we can use a `try/except` block to catch the error and print an error message, then carry on with the rest of our program; Example 3-6 shows the details.

Example 3-6. Catching an exception

```
from sqlalchemy.exc import IntegrityError ❶
ins = insert(users).values(
    username="cookiemon",
    email_address="damon@cookie.com",
    phone="111-111-1111",
    password="password"
)
try:
    result = connection.execute(ins)
except IntegrityError as error: ❷
    print(error.orig.message, error.params)
```

❶ All the SQLAlchemy exceptions are available in the `sqlalchemy.exc` module.

❷ We catch the `IntegrityError` exception as `error` so we can access the properties of the exception.

In Example 3-6, we are running the same statement as Example 3-4, but wrapping the statement execution in a `try/except` block that catches an `IntegrityError` and prints a message with the error message and statement parameters. While this example demonstrated how to print an error message, we can write any Python code we like in the exception clause. This can be useful for returning an error message to application users to inform them that the operation failed. By handling the error with a `try/except` block, our application continues to execute and run.

While Example 3-6 shows an `IntegrityError`, this method of handling errors will work for any type of error generated by SQLAlchemy. For more information on other SQLAlchemy exceptions, see the SQLAlchemy documentation at *http://docs.sqlalchemy.org/en/latest/core/exceptions.html*.

Remember it is best practice to wrap as little code as possible in a `try/except` block and only catch specific errors. This prevents catching unexpected errors that really should have a different behavior than the catch for the specific error you're watching for.

While we were able to handle the exceptions from a single statement using traditional Python tools, that method alone won't work if we have multiple database statements that are dependent on one another to be completely successful. In such cases, we need to wrap those statements in a database transaction, and SQLAlchemy provides a simple-to-use wrapper for that purpose built into the connection object: transactions.

Transactions

Rather than learning the deep database theory behind transactions, just think of transactions as a way to ensure that multiple database statements succeed or fail as a group. When we start a transaction, we record the current state of our database; then we can execute multiple SQL statements. If all the SQL statements in the transaction succeed, the database continues on normally and we discard the prior database state. Figure 3-1 shows the normal transaction workflow.

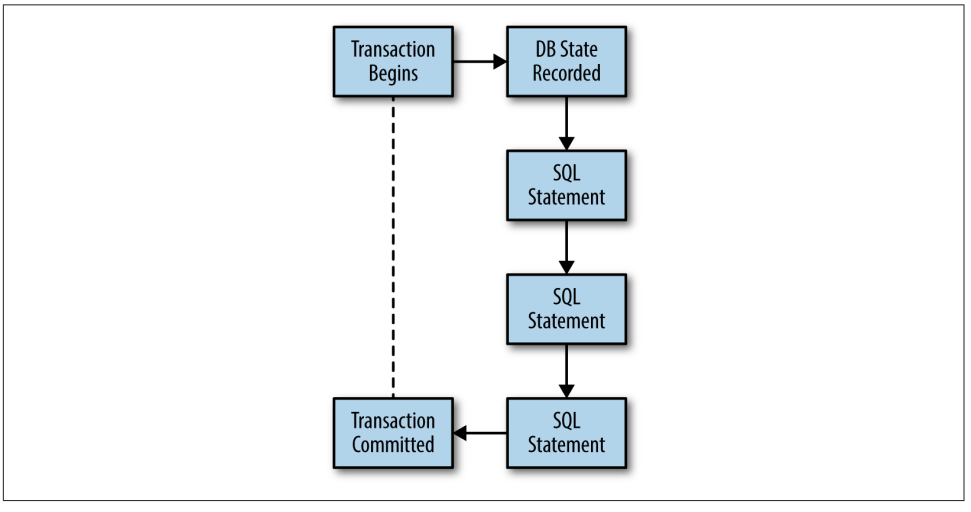


Figure 3-1. Successful transaction flow

However, if one or more of those statements fail, we can catch that error and use the prior state to roll back back any statements that succeeded. Figure 3-2 shows a transaction workflow in error.

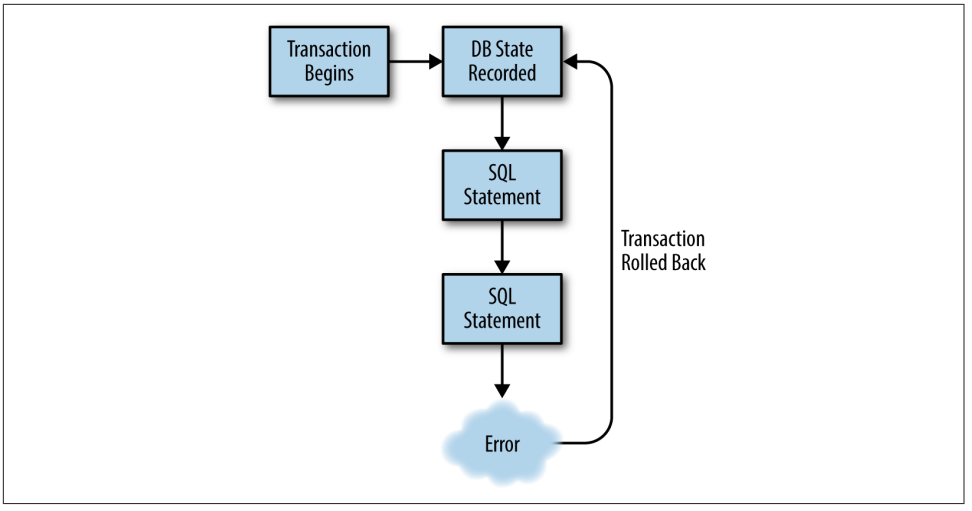


Figure 3-2. Failed transaction flow

There is already a good example of when we might want to do this in our existing database. After a customer has ordered cookies from us, we need to ship those cookies to the customer and remove them from our inventory. However, what if we do not have enough of the right cookies to fulfill an order? We will need to detect that and not ship that order. We can solve this with transactions.

We'll need a fresh Python shell with the tables from Chapter 2; however, we need to add a `CheckConstraint` to the quantity column to ensure it cannot go below 0, because we can't have negative cookies in inventory. Next, re-create the cookiemon user as well as the chocolate chip and dark chocolate chip cookie records. Set the quantity of chocolate chip cookies to 12 and the dark chocolate chip cookies to 1. Example 3-7 shows the full code for setting up the tables with the `CheckConstraint`, adding the cookiemon user, and adding the cookies.

Example 3-7. Setting up the transactions environment

```
from datetime import datetime

from sqlalchemy import (MetaData, Table, Column, Integer, Numeric, String,
                        DateTime, ForeignKey, Boolean, create_engine,
                        CheckConstraint)
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('quantity', Integer()),
    Column('unit_cost', Numeric(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)
)

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

line_items = Table('line_items', metadata,
    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="password"
)
result = connection.execute(ins)
ins = cookies.insert()
inventory_list = [
    {
        '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'
    }
]
result = connection.execute(ins, inventory_list)
```

We're now going to define two orders for the cookiemon user. The first order will be for nine chocolate chip cookies, and the second order will be for four chocolate chip cookies and one dark chocolate chip cookie. We'll do this using the insert statements discussed in the previous chapter. Example 3-8 shows the details.

Example 3-8. Adding the orders

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

❶ Adding the first order.

❷ Adding the second order.

That will give us all the order data we need to explore how transactions work; now we need to define a function called `ship_it`. Our `ship_it` function will accept an `order_id`, remove the cookies from inventory, and mark the order as shipped. Example 3-9 shows how this works.

Example 3-9. Defining the ship_it function

```
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)
    for cookie in cookies_to_ship: ❶
        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) ❷
    print("Shipped order ID: {}".format(order_id))
```

❶ For each cookie type we find in the order, we remove the quantity ordered for it from the `cookies` table quantity so we know how many cookies we have left.

❷ We update the order to mark it as shipped.

The `ship_it` function will perform all the actions required when we ship an order. Let's run it on our first order and then query the `cookies` table to make sure it reduced the cookie count correctly. Example 3-10 shows how to do that.

Example 3-10. Running ship_it on the first order

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

❶ Run `ship_it` on the first `order_id`.

❷ Look at our cookie inventory.

Running the code in Example 3-10 results in:

```
[(u'chocolate chip', 3), (u'dark chocolate chip', 1)]
```

Excellent! It worked. We can see that we don't have enough cookies in our inventory to fulfill the second order; however, in our fast-paced warehouse, these orders might be processed at the same time. Now try shipping our second order with the `ship_it` function, and watch what happens (as shown in Example 3-11).

Example 3-11. Running ship_it on the second order

```
ship_it(2)
```

That command gives us this result:

```
IntegrityError                            Traceback (most recent call last)
<ipython-input-9-47771be6653b> in <module>()
----> 1 ship_it(2)

<ipython-input-6-301c0ed7c4a1> in ship_it(order_id)
      7         u = update(cookies).where(cookies.c.cookie_id ==
          cookie.cookie_id)
      8         u = u.values(quantity = cookies.c.quantity-cookie.quantity)
----> 9         result = connection.execute(u)
     10     u = update(orders).where(orders.c.order_id == order_id)
     11     u = u.values(shipped=True)

...

IntegrityError: (sqlite3.IntegrityError) CHECK constraint failed:
quantity_positive
[SQL: u'UPDATE cookies SET quantity=(cookies.quantity - ?) WHERE
cookies.cookie_id = ?'] [parameters: (4, 1)]
```

We got an `IntegrityError` because we didn't have enough chocolate chip cookies to ship the order. However, let's see what happened to our `cookies` table using the last two lines of Example 3-10:

```
[(u'chocolate chip', 3), (u'dark chocolate chip', 0)]
```

It didn't remove the chocolate chip cookies because of the `IntegrityError`, but it did remove the dark chocolate chip cookies. This isn't good! We only want to ship whole orders to our customers. Using what you learned about `try/except` in "Handling Errors" on page 41 earlier, you could devise a complicated except method that would revert the partial shipment. However, transactions provide us a better way to handle just this type of event.

Transactions are initiated by calling the `begin()` method on the connection object. The result of this call is a transaction object that we can use to control the result of all our statements. If all our statements are successful, we commit the transaction by calling the `commit()` method on the transaction object. If not, we call the `rollback()` method on that same object. Let's rewrite the `ship_it` function to use a transaction to safely execute our statements; Example 3-12 shows what to do.

Example 3-12. Transactional ship_it

```
from sqlalchemy.exc import IntegrityError ❶
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)
transaction = connection.begin() ❷
cookies_to_ship = connection.execute(s).fetchall() ❸

try:
    for cookie in 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("Shipped order ID: {}".format(order_id))
    transaction.commit() ❹
except IntegrityError as error:
    transaction.rollback() ❺
    print(error)
```

❶ Importing the `IntegrityError` so we can handle its exception.

❷ Starting the transaction.

❸ Fetching all the results just to make it easier to follow what is happening.

❹ Commiting the transaction if no errors occur.

❺ Rolling back the transaction if an error occurs.

Now let's reset the dark chocolate chip cookies quantity back to 1:

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

We need to rerun our transaction-based `ship_it` on the second order. The program doesn't get stopped by the error, and prints us the error message without the traceback. Let's check the inventory like we did in Example 3-10 to make sure that it didn't mess up our inventory with a partial shipment:

```
[(u'chocolate chip', 3), (u'dark chocolate chip', 1)]
```

Great! Our transactional function didn't mess up our inventory or crash our application. We also didn't have to do a lot of coding to manually roll back the statements that did succeed. As you can see, transactions can be really useful in situations like this, and can save you a lot of manual coding.

In this chapter, we saw how to handle exceptions in both single statements and groups of statements. By using a normal `try/except` block on a single statement, we can prevent our application from crashing in case a database statement error occurs. We also looked at how transactions can help us avoid inconsistent databases, and application crashes in groups of statements. In the next chapter, we'll learn how to test our code to ensure it behaves the way we expect.

CHAPTER 4
Testing

Most testing inside of applications consists of both unit and functional tests; however, with SQLAlchemy, it can be a lot of work to correctly mock out a query statement or a model for unit testing. That work often does not truly lead to much gain over testing against a database during the functional test. This leads people to make wrapper functions for their queries that they can easily mock out during unit tests, or to just test against a database in both their unit and function tests. I personally like to use small wrapper functions when possible or—if that doesn't make sense for some reason or I'm in legacy code—mock it out.

This chapter covers how to perform functional tests against a database, and how to mock out SQLAlchemy queries and connections.

Testing with a Test Database

For our example application, we are going to have an *app.py* file that contains our application logic, and a *db.py* file that contains our database tables and connections. These files can be found in the *CH05/* folder of the example code.

How an application is structured is an implementation detail that can have quite an effect on how you need to do your testing. In *db.py*, you can see that our database is set up via the `DataAccessLayer` class. We're using this data access class to enable us to initialize a database schema, and connect it to an engine whenever we like. You'll see this pattern commonly used in web frameworks when coupled with SQLAlchemy. The `DataAccessLayer` class is initialized without an engine and a connection in the `dal` variable. Example 4-1 shows a snippet of our *db.py* file.

Example 4-1. DataAccessLayer class

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


class DataAccessLayer:
    connection = None
    engine = None
    conn_string = None
    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('quantity', Integer()),
                    Column('unit_cost', Numeric(12, 2))
                ) ❶

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

dal = DataAccessLayer() ❸
```

❶ In the complete file, we create all the tables that we have been using since Chapter 1, not just cookies.

❷ This provides a way to initialize a connection with a specific connection string like a factory.

❸ This provides an instance of the `DataAccessLayer` class that can be imported throughout our application.

We are going to write tests for the `get_orders_by_customer` function we built in Chapter 2, which is found the *app.py* file, shown in Example 4-2.

Example 4-2. app.py to test

```
from db import dal ❶
from sqlalchemy.sql import select
```

```
def get_orders_by_customer(cust_name, shipped=None, details=False):
    columns = [dal.orders.c.order_id, dal.users.c.username, dal.users.c.phone]
    joins = dal.users.join(dal.orders) ❷

    if details:
        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)

    if shipped is not None:
        cust_orders = cust_orders.where(dal.orders.c.shipped == shipped)

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

❶ This is our `DataAccessLayer` instance from the *db.py* file.

❷ Because our tables are inside of the `dal` object, we access them from there.

Let's look at all the ways the `get_orders_by_customer` function can be used. We're going to assume for this exercise that we have already validated that the inputs to the function are of the correct type. However, in your testing, it would be very wise to make sure you test with data that will work correctly and data that could cause errors. Here's a list of the variables our function can accept and their possible values:

- `cust_name` can be blank, a string containing the name of a valid customer, or a string that does not contain the name of a valid customer.
- `shipped` can be `None`, `True`, or `False`.
- `details` can be `True` or `False`.

If we want to test all of the possible combinations, we will need 12 (that 3 * 3 * 2) tests to fully test this function.

It is important not to test things that are just part of the basic functionality of SQLAlchemy, as SQLAlchemy already comes with a large collection of well-written tests. For example, we wouldn't want to test a simple insert, select, delete, or update statement, as those are tested within the SQLAlchemy project itself. Instead, look to test things that your code manipulates that could affect how the SQLAlchemy statement is run or the results returned by it.

For this testing example, we're going to use the built-in `unittest` module. Don't worry if you're not familiar with this module; we'll explain the key points. First, we need to set up the test class, and initialize the `dal`'s connection, which is shown in Example 4-3 by creating a new file named *test_app.py*.

Example 4-3. Setting up the tests

```
import unittest

class TestApp(unittest.TestCase): ❶

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

❶ `unittest` requires test classes inherited from `unittest.TestCase`.

❷ The `setUpClass` method is run once for the entire test class.

❸ This line initializes a connection to an in-memory database for testing.

Now we need to load some data to use during our tests. I'm not going to include the full code here, as it is the same inserts we worked with in Chapter 2, modified to use the `DataAccessLayer`; it is available in the *db.py* example file. With our data loaded, we are ready to write some tests. We're going to add these tests as functions within the `TestApp` class, as shown in Example 4-4.

Example 4-4. The first six tests for blank usernames

```
    def test_orders_by_customer_blank(self): ❶
        results = get_orders_by_customer('')
        self.assertEqual(results, []) ❷

    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):
        results = 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, [])
```

❶ `unittest` expects each test to begin with the letters `test`.

❷ `unittest` uses `assertEqual` to verify that the result matches what you expect. Because a user is not found, you should get an empty list back.

Now save the test file as *test_app.py*, and run the unit tests with the following command:

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

Ran 6 tests in 0.018s
```

You might get a warning about SQLite and decimal types; just ignore this as it's normal for our examples. It occurs because SQLite doesn't have a true decimal type, and SQLAlchemy wants you to know that there could be some oddities due to the conversion from SQLite's float type. It is always wise to investigate these messages, because in production code they will normally point you to the proper way you should be doing something. We are purposely triggering this warning here so that you will see what it looks like.

Now we need to load up some data, and make sure our tests still work. Again, we're going to reuse the work we did in Chapter 2, and insert the same users, orders, and line items. However, this time we are going to wrap the data inserts in a function called `db_prep`. This will allow us to insert this data prior to a test with a simple function call. For simplicity's sake, I have put this function inside the *db.py* file (see Example 4-5); however, in real-world situations, it will often be located in a test fixtures or utilities file.

Example 4-5. Inserting some test data

```
def prep_db():
    ins = dal.cookies.insert()
    dal.connection.execute(ins, 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')
    inventory_list = [
        {
            'cookie_name': 'peanut butter',
```

```
            '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, inventory_list)

    customer_list = [
        {
            '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)
```

Now that we have a `prep_db` function, we can use that in our *test_app.py* `setUpClass` method to load data into the database prior to running our tests. So now our `setUpClass` method looks like this:

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

We can use this test data to ensure that our function does the right thing when given a valid username. These tests go inside of our *TestApp* class as new functions, as Example 4-6 shows.

Example 4-6. Tests for a valid user

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

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

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

```
    def test_orders_by_customer_with_details(self):
        expected_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'))
        ]
        results = get_orders_by_customer('cookiemon', details=True)
        self.assertEqual(results, expected_results)

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

    def test_orders_by_customer_unshipped_only_details(self):
        expected_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'))
        ]
        results = get_orders_by_customer('cookiemon', False, True)
        self.assertEqual(results, expected_results)
```

Using the tests in Example 4-6 as guidance, can you complete the tests for what happens for a different user, such as `cakeeater`? How about the tests for a username that doesn't exist in the system yet? Or if we get an integer instead of a string for the username, what will be the result? Compare your tests to those in the supplied example code when you are done to see if your tests are similar to the ones used in this book.

We've learned how we can use SQLAlchemy in functional tests to determine whether a function behaves as expected on a given data set. We have also looked at how to set up a `unittest` file and how to prepare the database for use in our tests. Next, we are ready to look at testing without hitting the database.

Using Mocks

This technique can be a powerful tool when you have a test environment where creating a test database doesn't make sense or simply isn't feasible. If you have a large amount of logic that operates on the result of the query, it can be useful to mock out the SQLAlchemy code to return the values you want so you can test only the surrounding logic. Normally when I am going to mock out some part of the query, I still create the in-memory database, but I don't load any data into it, and I mock out the database connection itself. This allows me to control what is returned by the execute and fetch methods. We are going to explore how to do that in this section.

To learn how to use mocks in our tests, we are going to make a single test for a valid user. This time we will use the powerful Python mock library to control what is

returned by the connection. Mock is part of the `unittest` module in Python 3. However, if you are using Python 2, you will need to install the mock library using pip to get the latest mock features. To do that, run this command:

```
pip install mock
```

Now that we have mock installed, we can use it in our tests. Mock has a patch function that will let us replace a given object in a Python file with a `MagicMock` that we can control from our test. A `MagicMock` is a special type of Python object that tracks how it is used and allows us to define how it behaves based on how it is being used.

First, we need to import the mock library. In Python 2, we need to do the following:

```
import mock
```

In Python 3, we need to do the following:

```
from unittest import mock
```

With mock imported, we are going to use the `patch` method as a decorator to replace the connection part of the `dal` object. A decorator is a function that wraps another function, and alters the wrapped function's behavior. Because the `dal` object is imported by name into the *app.py* file, we will need to patch it inside the `app` module. This will get passed into the test function as an argument. Now that we have a mock object, we can set a return value for the `execute` method, which in this case should be nothing but a chained `fetchall` method whose return value is the data we want to test with. Example 4-7 shows the code needed to use the mock in place of the `dal` object.

Example 4-7. Mocked connection test

```
import unittest
from decimal import Decimal

import mock

from db import dal, prep_db
from 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') ❶
```

```
    def test_orders_by_customer(self, mock_conn): ❷
        mock_conn.execute.return_value.fetchall.return_value = self.cookie_orders ❸
        results = get_orders_by_customer('cookiemon') ❹
        self.assertEqual(results, self.cookie_orders)
```

❶ Patching `dal.connection` in the `app` module with a mock.

❷ That mock is passed into the test function as `mock_conn`.

❸ We set the return value of the `execute` method to the chained returned value of the `fetchall` method, which we set to `self.cookie_order`.

❹ Now we call the test function where the `dal.connection` will be mocked and return the value we set in the prior step.

You can see that a complicated query or `ResultProxy` like the one in Example 4-7 might get tedious quickly when trying to mock out the full query or connection. Don't shy away from the work though; it can be very useful for finding obscure bugs.

If you did want to mock out the query, you would follow the same pattern of using the `mock.patch` decorator and mocking out the `select` object in the `app` module. Let's try that with one of the empty test queries. We have to mock out all the chained query element return values, which in this query are the `select`, `select_from`, and `where` clauses. Example 4-8 demonstrates how to do this.

Example 4-8. Mocking out the query as well

```
@mock.patch('app.select') ❶
@mock.patch('app.dal.connection')
def test_orders_by_customer_blank(self, mock_conn, mock_select): ❷
    mock_select.return_value.select_from.return_value.\
        where.return_value = '' ❸
    mock_conn.execute.return_value.fetchall.return_value = [] ❹
    results = get_orders_by_customer('')
    self.assertEqual(results, [])
```

❶ Mocking out the `select` method, as it starts the query chain.

❷ The decorators are passed into the function in order. As we progress up the decorators stack from the function, the arguments get added to the left.

❸ We have to mock the return value for all parts of the chained query.

❹ We still need to mock the connection or the `app` module SQLAlchemy code would try to make the query.

As an exercise, you should work on building the remainder of the tests that we built with the in-memory database with the mocked test types. I would encourage you to mock both the query and the connection to get familiar with the mocking process.

You should now feel comfortable with how to test a function that contains SQLAlchemy functions within it. You should also understand how to prepopulate data into the test database for use in your test. Finally, you should understand how to mock both the query and connection objects. While this chapter used a simple example, we will dive further into testing in Chapter 14, which looks at Flask, Pyramid, and pytest.

Up next, we are going to look at how to handle an existing database with SQLAlchemy without the need to re-create all the schema in Python via reflection.

CHAPTER 5

Reflection

Reflection is a technique that allows us to populate a SQLAlchemy object from an existing database. You can reflect tables, views, indexes, and foreign keys. This chapter will explore how to use reflection on an example database.

For testing, I recommend using Chinook database. You can learn more about it at *http://chinookdatabase.codeplex.com/*. We'll be using the SQLite version, which is available in the *CH06/* folder of this book's sample code. That folder also contains an image of the database schema so you can visualize the schema that we'll be working with throughout this chapter. We'll begin by reflecting a single table.

Reflecting Individual Tables

For our first reflection, we are going to generate the `Artist` table. We'll need a metadata object to hold the reflected table schema information, and an engine attached to the Chinook database. Example 5-1 demonstrates how to set up both of these things; the process should be very familiar to you now.

Example 5-1. Setting up our initial objects

```
from sqlalchemy import MetaData, create_engine
metadata = MetaData()
engine = create_engine('sqlite:///Chinook_Sqlite.sqlite') ❶
```

❶ This connection string assumes you are in the same directory as the example database.

With the metadata and engine set up, we have everything we need to reflect a table. We are going to create a table object using code similar to the table creation code in Chapter 1; however, instead of defining the columns by hand, we are going to use the

autoload and autoload_with keyword arguments. This will reflect the schema information into the metadata object and store a reference to the table in the artist variable. Example 5-2 demonstrates how to perform the reflection.

Example 5-2. Reflecting the Artist table

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

That last line really is all we need to reflect the Artist table! We can use this table just as we did in Chapter 2 with our manually defined tables. Example 5-3 shows how to perform a simple query with the table to see what kind of data is in it.

Example 5-3. Using the Artist table

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

❶ Listing the columns.

❷ Selecting the first 10 records.

This results in:

```
['ArtistId', 'Name']

[(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')]
```

Looking at the database schema, we can see that there is an Album table that is related to the Artist table. Let's reflect it the same way as we did for the Artist table:

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

Now let's check the Album table's metadata to see what got reflected. Example 5-4 shows how to do that.

Example 5-4. Viewing the metadata

```
metadata.tables['album']
```

```
Table('album',
      MetaData(bind=None),
      Column('AlbumId', INTEGER(), table=<album>, primary_key=True, nullable=False),
      Column('Title', NVARCHAR(length=160), table=<album>, nullable=False),
      Column('ArtistId', INTEGER(), table=<album>, nullable=False),
             schema=None)
)
```

Interestingly, the foreign key to the `Artist` table does not appear to have been reflected. Let's check the `foreign_keys` attribute of the `Album` table to make sure that it is not present:

```
album.foreign_keys
```

```
set()
```

So it really didn't get reflected. This occurred because the two tables weren't reflected at the same time, and the target of the foreign key was not present during the reflection. In an effort to not leave you in a semi-broken state, SQLAlchemy discarded the one-sided relationship. We can use what we learned in Chapter 1 to add the missing `ForeignKey`, and restore the relationship:

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

Now if we rerun Example 5-4, we can see the `ArtistId` column is a `ForeignKey`:

```
Table('album',
      MetaData(bind=None),
      Column('AlbumId', INTEGER(), table=<album>, primary_key=True,
             nullable=False),
      Column('Title', NVARCHAR(length=160), table=<album>, nullable=False),
      Column('ArtistId', INTEGER(), ForeignKey('artist.ArtistId'), table=<album>,
             nullable=False),
      schema=None)
```

Now let's see if we can use the relationship to join the tables properly. We can run the following code to test the relationship:

```
str(artist.join(album))
```

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

Excellent! Now we can perform queries that use this relationship. It works just like the queries discussed in "Joins" on page 31.

It would be quite a bit of work to repeat the reflection process for each individual table in our database. Fortunately, SQLAlchemy lets you reflect an entire database at once.

Reflecting a Whole Database

In order to reflect a whole database, we can use the `reflect` method on the `metadata` object. The `reflect` method will scan everything available on the engine supplied, and reflect everything it can. Let's use our existing `metadata` and `engine` objects to reflect the entire Chinook database:

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

This statement doesn't return anything if it succeeds; however, we can run the following code to retrieve a list of table names to see what was reflected into our `metadata`:

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

```
dict_keys(['InvoiceLine', 'Employee', 'Invoice', 'album', 'Genre',
           'PlaylistTrack', 'Album', 'Customer', 'MediaType', 'Artist',
           'Track', 'artist', 'Playlist'])
```

The tables we manually reflected are listed twice but with different case letters. This is due to that fact that SQLAlchemy reflects the tables as they are named, and in the Chinook database they are uppercase. Due to SQLite's handling of case sensitivity, both the lower- and uppercase names point to the same tables in the database.

Be careful, as case sensitivity can trip you up on other databases, such as Oracle.

Now that we have our database reflected, we are ready to discuss using reflected tables in queries.

Query Building with Reflected Objects

As you saw in Example 5-3, querying tables that we reflected and stored in a variable works just like it did in Chapter 2. However, for the rest of the tables that were reflected when we reflected the entire database, we'll need a way to refer to them in our query. We can do that by assigning them to a variable from the `tables` attribute of the `metadata`, as shown in Example 5-5.

Example 5-5. Using a reflected table in query

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

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

❶ Establish a variable to be a reference to the table.

❷ Use that variable in the query.

Running the code in Example 5-5 gives us this result:

```
engine.execute(s).fetchall()
[(1, 'Music'),
 (2, 'Movies'),
 (3, 'TV Shows'),
 (4, 'Audiobooks'),
 (5, '90's Music'),
 (6, 'Audiobooks'),
 (7, 'Movies'),
 (8, 'Music'),
 (9, 'Music Videos'),
 (10, 'TV Shows')]
```

By assigning the reflected tables we want to use into a variable, they can be used in the same way as previous chapters.

Reflection is a very useful tool; however, as of version 1.0 of SQLAlchemy, we cannot reflect `CheckConstraints`, comments, or triggers. You also can't reflect client-side defaults or an association between a sequence and a column. However, it is possible to add them manually using the methods described in Chapter 1.

You now understand how to reflect individual tables and repair issues like missing relationships in those tables. Additionally, you learned how to reflect an entire database, and use individual tables in queries from the reflected tables.

This chapter wraps up the essential parts of SQLAlchemy Core and the SQL Expression Language. Hopefully, you've gotten a sense for how powerful these parts of SQLAlchemy are. They are often overlooked due to the SQLAlchemy ORM, but using them can add even more capabilities to your applications. Let's move on to learning about the SQLAlchemy ORM.

PART II

SQLAlchemy ORM

The SQLAlchemy ORM is what most people think of when you mention SQLAlchemy. It provides a very effective way to bind database schema and operations to the same data objects used in your application. It offers a way to rapidly build applications and get them into customers' hands. Using the ORM is so simple that most people don't consider the possible side effects of their code. It is still important to think about how the database will be used by your code. However, the wonderful thing about the ORM is that you can think about that when the need arises and not for every query. Let's get started using the ORM.

CHAPTER 6

Defining Schema with SQLAlchemy ORM

You define schema slightly different when using the SQLAlchemy ORM because it is focused around user-defined data objects instead of the schema of the underlying database. In SQLAlchemy Core, we created a metadata container and then declared a `Table` object associated with that metadata. In SQLAlchemy ORM, we are going to define a class that inherits from a special base class called the `declarative_base`. The `declarative_base` combines a metadata container and a mapper that maps our class to a database table. It also maps instances of the class to records in that table if they have been saved. Let's dig into defining tables in this manner.

Defining Tables via ORM Classes

A proper class for use with the ORM must do four things:

- Inherit from the `declarative_base` object.
- Contain `__tablename__`, which is the table name to be used in the database.
- Contain one or more attributes that are `Column` objects.
- Ensure one or more attributes make up a primary key.

We need to examine the last two attribute-related requirements a bit closer. First, defining columns in an ORM class is very similar to defining columns in a `Table` object, which we discussed in Chapter 2; however, there is one very important difference. When defining columns in an ORM class, we don't have to supply the column name as the first argument to the `Column` constructor. Instead, the column name will be set to the name of the class attribute to which it is assigned. Everything else we covered in "Types" on page 1 and "Columns" on page 5 applies here too, and works as expected. Second, the requirement for a primary key might seem odd at first; how-

ever, the ORM has to have a way to uniquely identify and associate an instance of the class with a specific record in the underlying database table. Let's look at our `cookies` table defined as an ORM class (Example 6-1).

Example 6-1. Cookies table defined as an ORM class

```
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))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))
```

❶ Create an instance of the `declarative_base`.

❷ Inherit from the `Base`.

❸ Define the table name.

❹ Define an attribute and set it to be a primary key.

This will result in the same table as shown in Example 2-1, and that can be verified by looking at `Cookie._table_` property:

```
>>> 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('quantity', Integer(), table=<cookies>),
    Column('unit_cost', Numeric(precision=12, scale=2),
    table=<cookies>), schema=None)
```

Let's look at another example. This time I want to re-create our `users` table from Example 2-2. Example 6-2 demonstrates how additional keywords work the same in both the ORM and Core schemas.

Example 6-2. Another table with more column options

```
from datetime import datetime
from sqlalchemy import DateTime


class User(Base):
    __tablename__ = 'users'

    user_id = Column(Integer(), primary_key=True)
    username = Column(String(15), nullable=False, unique=True) ❶
    email_address = Column(String(255), nullable=False)
    phone = Column(String(20), nullable=False)
    password = Column(String(25), nullable=False)
    created_on = Column(DateTime(), default=datetime.now) ❷
    updated_on = Column(DateTime(), default=datetime.now, onupdate=datetime.now) ❸
```

❶ Here we are making this column required (`nullable=False`) and requiring the values to be unique.

❷ The default sets this column to the current time if a date isn't specified.

❸ Using `onupdate` here will reset this column to the current time every time any part of the record is updated.

Keys, Constraints, and Indexes

In "Keys and Constraints" on page 6, we discussed that it is possible to define keys and constraints in the table constructor in addition to being able to do it in the columns themselves, as shown earlier. However, when using the ORM, we are building classes and not using the table constructor. In the ORM, these can be added by using the `__table_args__` attribute on our class. `__table_args__` expects to get a tuple of additional table arguments, as shown here:

```
    class SomeDataClass(Base):
        __tablename__ = 'somedatatable'
        __table_args__ = (ForeignKeyConstraint(['id'], ['other_table.id']),
                          CheckConstraint(unit_cost >= 0.00',
                                          name='unit_cost_positive'))
```

The syntax is the same as when we use them in a `Table()` constructor. This also applies to what you learned in "Indexes" on page 7.

In this section, we covered how to define a couple of tables and their schemas with the ORM. Another important part of defining data models is establishing relationships between multiple tables and objects.

Relationships

Related tables and objects are another place where there are differences between SQLAlchemy Core and ORM. The ORM uses a similar `ForeignKey` column to constrain and link the objects; however, it also uses a `relationship` directive to provide a property that can be used to access the related object. This does add some extra database usage and overhead when using the ORM; however, the pluses of having this capability far outweigh the drawbacks. I'd love to give you a rough estimate of the additional overhead, but it varies based on your data models and how you use them. In most cases, it's not even worth considering. Example 6-3 shows how to define a relationship using the `relationship` and `backref` methods.

Example 6-3. Table with a relationship

```
from sqlalchemy import ForeignKey, Boolean
from sqlalchemy.orm import relationship, backref ❶


class Order(Base):
    __tablename__ = 'orders'

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

    user =  relationship("User", backref=backref('orders', order_by=order_id)) ❸
```

❶ Notice how we import the `relationship` and `backref` methods from `sqlalchemy.orm`.

❷ We are defining a `ForeignKey` just as we did with SQLAlchemy Core.

❸ This establishes a one-to-many relationship.

Looking at the user relationship defined in the `Order` class, it establishes a one-to-many relationship with the `User` class. We can get the `User` related to this `Order` by accessing the `user` property. This relationship also establishes an `orders` property on the `User` class via the `backref` keyword argument, which is ordered by the `order_id`. The `relationship` directive needs a target class for the relationship, and can optionally include a back reference to be added to target class. SQLAlchemy knows to use the `ForeignKey` we defined that matches the class we defined in the relationship. In the preceding example, the `ForeignKey(`*`users.user_id`*`)`, which has the `users` table's `user_id` column, maps to the `User` class via the `__tablename__` attribute of `users` and forms the relationship.

It is also possible to establish a one-to-one relationship: in Example 6-4, the `LineItem` class has a one-to-one relationship with the `Cookie` class. The `uselist=False` keyword argument defines it as a one-to-one relationship. We also use a simpler back reference, as we do not care to control the order.

Example 6-4. More tables with relationships

```
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'))
    quantity = Column(Integer())
    extended_cost = Column(Numeric(12, 2))

    order = relationship("Order", backref=backref('line_items',
                                                  order_by=line_item_id))

    cookie = relationship("Cookie", uselist=False) ❶
```

❶ This establishes a one-to-one relationship.

These two examples are good places to start with relationships; however, relationships are an extremely powerful part of the ORM. We've barely scrathed the surface on what can be done with relationships, and we will explore them more in the Cookbook in Chapter 14. With our classes all defined and our relationships established, we are ready to create our tables in the database.

Persisting the Schema

To create our database tables, we are going to use the `create_all` method on the metadata within our `Base` instance. It requires an instance of an engine, just as it did in SQLAlchemy Core:

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

Base.metadata.create_all(engine)
```

Before we discuss how to work with our classes and tables via the ORM, we need to learn about how sessions are used by the ORM. That's what the next chapter is all about.

CHAPTER 7

Working with Data via SQLAlchemy ORM

Now that we have defined classes that represent the tables in our database and persisted them, let's start working with data via those classes. In this chapter, we'll look at how to insert, retrieve, update, and delete data. Then we'll learn how to sort and group that data and take a look at how relationships work. We'll begin by learning about the SQLAlchemy session, one of the most important parts of the SQLAlchemy ORM.

The Session

The session is the way SQLAlchemy ORM interacts with the database. It wraps a database connection via an engine, and provides an identity map for objects that you load via the session or associate with the session. The identity map is a cache-like data structure that contains a unique list of objects determined by the object's table and primary key. A session also wraps a transaction, and that transaction will be open until the session is committed or rolled back, very similar to the process described in "Transactions" on page 43.

To create a new session, SQLAlchemy provides the `sessionmaker` class to ensure that sessions can be created with the same parameters throughout an application. It does this by creating a `Session` class that has been configured according to the arguments passed to the `sessionmaker` factory. The `sessionmaker` factory should be used just once in your application global scope, and treated like a configuration setting. Let's create a new session associated with an in-memory SQLite database:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker ❶

engine = create_engine('sqlite:///:memory:')
```

```
Session = sessionmaker(bind=engine) ❷

session = Session() ❸
```

❶ Imports the `sessionmaker` class.

❷ Defines a `Session` class with the `bind` configuration supplied by `sessionmaker`.

❸ Creates a `session` for our use from our generated `Session` class.

Now we have a `session` that we can use to interact with the database. While `session` has everything it needs to connect to the database, it won't connect until we give it some instructions that require it to do so. We're going to continue to use the classes we created in Chapter 6 for our examples in the chapter. We're going to add some `__repr__` methods to make it easy to see and re-create object instances; however, these methods are not required:

```
from datetime import datetime

from sqlalchemy import (Table, Column, Integer, Numeric, String, DateTime,
                        ForeignKey)
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, 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))
    quantity = 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}', " \
                       "quantity={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)
    password = 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}', " \
                    "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'))

    user =  relationship("User", backref=backref('orders', order_by=order_id))

    def __repr__(self):
        return "Order(user_id={self.user_id}, " \
                     "shipped={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'))
    quantity = Column(Integer())
    extended_cost = Column(Numeric(12, 2))
    order = relationship("Order", backref=backref('line_items',
                         order_by=line_item_id))
    cookie = relationship("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(engine) ❷
```

❶ A `__repr__` method defines how the object should be represented. It typically is the constructor call required to re-create the instance. This will show up later in our print output.

❷ Creates the tables in the database defined by the engine.

With our classes re-created, we are ready to begin learning how to work with data in our database, and we are going to start with inserting data.

Inserting Data

To create a new cookie record in our database, we initialize a new instance of the `Cookie` class that has the desired data in it. We then add that new instance of the `Cookie` object to the session and commit the session. This is even easier to do because inheriting from the `declarative_base` provides a default constructor we can use (Example 7-1).

Example 7-1. Inserting a single object

```
cc_cookie = Cookie(cookie_name='chocolate chip', ❶
                   cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
                   cookie_sku='CC01',
                   quantity=12,
                   unit_cost=0.50)
session.add(cc_cookie) ❷
session.commit() ❸
```

❶ Creating an instance of the `Cookie` class.

❷ Adding the instance to the session.

❸ Committing the session.

When `commit()` is called on the session, the cookie is actually inserted into the database. It also updates `cc_cookie` with the primary key of the record in the database. We can see that by doing the following:

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

```
1
```

Let's take a moment to discuss what happens to the database when we run the code in Example 7-1. When we create the instance of the `Cookie` class and then add it to the session, nothing is sent to the database. It's not until we call `commit()` on the session that anything is sent to the database. When `commit()` is called, the following happens:

```
INFO:sqlalchemy.engine.base.Engine:BEGIN (implicit) ❶

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

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

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

❶ Start a transaction.

❷ Insert the record into the database.

❸ The values for the insert.

❹ Commit the transaction.

If you want to see the details of what is happening here, you can add `echo=True` to your `create_engine` statement as a keyword argument after the connection string. Make sure to only do this for testing, and don't use `echo=True` in production!

First, a fresh transaction is started, and the record is inserted into the database. Next, the engine sends the values of our insert statement. Finally, the transaction is committed to the database, and the transaction is closed. This method of processing is often called the Unit of Work pattern.

Next, let's look at a couple of ways to insert multiple records. If you are going to do additional work with the objects after inserting them, you'll want to use the method shown in Example 7-2. We simple create multiple instances, then add them both to the session prior to committing the session.

Example 7-2. Multiple inserts

```
dcc = Cookie(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)
mol = Cookie(cookie_name='molasses',
             cookie_recipe_url='http://some.aweso.me/cookie/recipe_molasses.html',
             cookie_sku='MOL01',
             quantity=1,
             unit_cost=0.80)
session.add(dcc) ❶
session.add(mol) ❷
session.flush() ❸
print(dcc.cookie_id)
print(mol.cookie_id)
```

❶ Adds the dark chocolate chip cookie.

❷ Adds the molasses cookie.

❸ Flushes the session.

Notice that we used the `flush()` method on the session instead of `commit()` in Example 7-2. A flush is like a commit; however, it doesn't perform a database commit and end the transaction. Because of this, the `dcc` and `mol` instances are still connected to the session, and can be used to perform additional database tasks without triggering additional database queries. We also issue the `session.flush()` statement one time, even though we added multiple records into the database. This actually results in two insert statements being sent to the database inside a single transaction. Example 7-2 will result in the following:

```
2
3
```

The second method of inserting multiple records into the database is great when you want to insert data into the table and you don't need to perform additional work on that data. Unlike the method we used in Example 7-2, the method in Example 7-3 does not associate the records with the session.

Example 7-3. Bulk inserting multiple records

```
c1 = Cookie(cookie_name='peanut butter',
            cookie_recipe_url='http://some.aweso.me/cookie/peanut.html',
            cookie_sku='PB01',
            quantity=24,
            unit_cost=0.25)
c2 = Cookie(cookie_name='oatmeal raisin',
            cookie_recipe_url='http://some.okay.me/cookie/raisin.html',
            cookie_sku='EWW01',
            quantity=100,
            unit_cost=1.00)
session.bulk_save_objects([c1,c2]) ❶
session.commit()
print(c1.cookie_id)
```

❶ Adds the cookies to a list and uses the `bulk_save_objects` method.

Example 7-3 will not result in anything being printed to the screen because the `c1` object isn't associated with the session, and can't refresh its `cookie_id` for printing. If we look at what was sent to the database, we can see only a single insert statement was in the transaction:

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

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

❶ A single insert

The method demonstrated in Example 7-3 is substantially faster than performing multiple individual adds and inserts as we did in Example 7-2. This speed does come at the expense of some features we get in the normal add and commit, such as:

- Relationship settings and actions are not respected or triggered.
- The objects are not connected to the session.
- Fetching primary keys is not done by default.
- No events will be triggered.

In addition to `bulk_save_objects`, there are additional methods to create and update objects via a dictionary, and you can learn more about bulk operations and their performance in the SQLAlchemy documentation.

If you are inserting multiple records and don't need access to relationships or the inserted primary key, use `bulk_save_objects` or its related methods. This is especially true if you are ingesting data from an external data source such as a CSV or a large JSON document with nested arrays.

Now that we have some data in our `cookies` table, let's learn how to query the tables and retrieve that data.

Querying Data

To begin building a query, we start by using the `query()` method on the session instance. Initially, let's select all the records in our `cookies` table by passing the `Cookie` class to the `query()` method, as shown in Example 7-4.

Example 7-4. Get all the cookies

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

❶ Returns a list of `Cookie` instances that represent all the records in the `cookies` table.

Example 7-4 will output the following:

```
[
    Cookie(cookie_name='chocolate chip',
           cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
           cookie_sku='CC01', quantity=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', quantity=1, unit_cost=0.75),
    Cookie(cookie_name='molasses',
           cookie_recipe_url='http://some.aweso.me/cookie/recipe_molasses.html',
           cookie_sku='MOL01', quantity=1, unit_cost=0.80),
    Cookie(cookie_name='peanut butter',
           cookie_recipe_url='http://some.aweso.me/cookie/peanut.html',
           cookie_sku='PB01', quantity=24, unit_cost=0.25),
    Cookie(cookie_name='oatmeal raisin',
           cookie_recipe_url='http://some.okay.me/cookie/raisin.html',
           cookie_sku='EWW01', quantity=100, unit_cost=1.00)
]
```

Because the returned value is a list of objects, we can use those objects as we would normally. These objects are connected to the session, which means we can change them or delete them and persist that change to the database as shown later in this chapter.

In addition to being able to get back a list of objects all at once, we can use the query as an iterable, as shown in Example 7-5.

Example 7-5. Using the query as an iterable

```
for cookie in session.query(Cookie): ❶
    print(cookie)
```

❶ We don't append an `all()` when using an iterable.

Using the iterable approach allows us to interact with each record object individually, release it, and get the next object.

Example 7-5 would display the following output:

```
Cookie(cookie_name='chocolate chip',
       cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
       cookie_sku='CC01', quantity=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', quantity=1, unit_cost=0.75),
Cookie(cookie_name='molasses',
       cookie_recipe_url='http://some.aweso.me/cookie/recipe_molasses.html',
       cookie_sku='MOL01', quantity=1, unit_cost=0.80),
Cookie(cookie_name='peanut butter',
       cookie_recipe_url='http://some.aweso.me/cookie/peanut.html',
```

```
        cookie_sku='PB01', quantity=24, unit_cost=0.25),
    Cookie(cookie_name='oatmeal raisin',
        cookie_recipe_url='http://some.okay.me/cookie/raisin.html',
        cookie_sku='EWW01', quantity=100, unit_cost=1.00)
```

In addition to using the query as an iterable or calling the `all()` method, there are many other ways of accessing the data. You can use the following methods to fetch results:

`first()`
: Returns the first record object if there is one.

`one()`
: Queries all the rows, and raises an exception if anything other than a single result is returned.

`scalar()`
: Returns the first element of the first result, `None` if there is no result, or an error if there is more than one result.

Tips for Good Production Code

When writing production code, you should follow these guidelines:

- Use the iterable version of the query over the `all()` method. It is more memory efficient than handling a full list of objects and we tend to operate on the data one record at a time anyway.
- To get a single record, use the `first()` method (rather than `one()` or `scalar()`) because it is clearer to our fellow coders. The only exception to this is when you must ensure that there is one and only one result from a query; in that case, use `one()`.
- Use the `scalar()` method sparingly, as it raises errors if a query ever returns more than one row with one column. In a query that selects entire records, it will return the entire record object, which can be confusing and cause errors.

Every time we queried the database in the preceding examples, all the columns were returned for every record. Often we only need a portion of those columns to perform our work. If the data in these extra columns is large, it can cause our applications to slow down and consume far more memory than it should. SQLAlchemy does not add a bunch of overhead to the queries or objects; however, accounting for the data you get back from a query is often the first place to look if a query is consuming too much memory. Let's look at how to limit the columns returned in a query.

Controlling the Columns in the Query

To limit the fields that are returned from a query, we need to pass in the columns we want in the `query()` method constructor separated by columns. For example, you might want to run a query that returns only the name and quantity of cookies, as shown in Example 7-6.

Example 7-6. Select only cookie_name and quantity

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

❶ Selects the `cookie_name` and `quantity` from the `cookies` table and returns the first result.

When we run Example 7-6, it outputs the following:

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

The output from a query where we supply the column names is a tuple of those column values.

Now that we can build a simple select statement, let's look at other things we can do to alter how the results are returned in a query. We'll start with changing the order in which the results are returned.

Ordering

If you were to look at all the results from Example 7-6 instead of just the first record, you would see that the data is not really in any particular order. However, if we want the list to be returned in a particular order, we can chain an `order_by()` statement to our select, as shown in Example 7-7. In this case, we want the results to be ordered by the quantity of cookies we have on hand.

Example 7-7. Order by quantity ascending

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

Example 7-7 will print the following output:

```
  1 - dark chocolate chip
  1 - molasses
 12 - chocolate chip
 24 - peanut butter
100 - oatmeal raisin
```

If you want to sort in reverse or descending order, use the `desc()` statement. The `desc()` function wraps the specific column you want to sort in a descending manner, as shown in Example 7-8.

Example 7-8. Order by quantity descending

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

❶ We wrap the column we want to sort descending in the `desc()` function.

The `desc()` function can also be used as a method on a column object, such as `Cookie.quantity.desc()`. However, that can be a bit more confusing to read in long statements, and so I always use `desc()` as a function.

It's also possible to limit the number of results returned if we only need a certain number of them for our application. The following section explains how.

Limiting

In prior examples, we used the `first()` method to get just a single row back. While our `query()` gave us the one row we asked for, the actual query ran over and accessed all the results, not just the single record. If we want to limit the query, we can use array slice notation to actually issue a limit statement as part of our query. For example, suppose you only have time to bake two batches of cookies, and you want to know which two cookie types you should make. You can use our ordered query from earlier and add a limit statement to return the two cookie types that are most in need of being replenished. Example 7-9 shows how this can be done.

Example 7-9. Two fewest cookie inventories

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

❶ This runs the query and slices the returned list. This can be very ineffecient with a large result set.

The output from Example 7-9 looks like this:

```
[u'dark chocolate chip', u'molasses']
```

In addition to using the array slice notation, it is also possible to use the `limit()` statement.

Example 7-10. Two fewest cookie inventories with limit

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

Now that you know what kind of cookies you need to bake, you're probably starting to get curious about how many cookies are now left in your inventory. Many databases include SQL functions designed to make certain operations available directly on the database server such as SUM; let's explore how to use these functions.

Built-In SQL Functions and Labels

SQLAlchemy can also leverage SQL functions found in the backend database. Two very commonly used database functions are `SUM()` and `COUNT()`. To use these functions, we need to import the `sqlalchemy.func` module generator that makes them available. These functions are wrapped around the column(s) on which they are operating. Thus, to get a total count of cookies, you would use something like Example 7-11.

Example 7-11. Summing our cookies

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

❶ Notice the use of `scalar`, which will return only the leftmost column in the first record.

I tend to always import the `func` module generator, as importing `sum` directly can cause problems and confusion with Python's built-in `sum` function

When we run Example 7-11, it results in:

```
138
```

Now let's use the `count` function to see how many cookie inventory records we have in our `cookies` table (Example 7-12).

Example 7-12. Counting our inventory records

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

Unlike Example 7-11 where we used `scalar` and got a single value, Example 7-12 results in a tuple for us to use, as we used the `first` method instead of `scalar`:

```
(5,)
```

Using functions such as `count()` and `sum()` will end up returning tuples or results with column names like `count_1`. These types of returns are often not what we want. Also, if we have several counts in a query we'd have to know the occurrence number in the statement, and incorporate that into the column name, so the fourth `count()` function would be `count_4`. This simply is not as explicit and clear as we should be in our naming, especially when surrounded with other Python code.

Thankfully, SQLAlchemy provides a way to fix this via the `label()` function. Example 7-13 performs the same query as Example 7-12; however, it uses `label()` to give us a more useful name to access that column.

Example 7-13. Renaming our count column

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

❶ I used the `label()` function on the column object I want to change.

Example 7-13 results in:

```
[u'inventory_count']
5
```

We've seen examples of how to restrict the columns or the number of rows returned from the database, so now it's time to learn about queries that filter data based on criteria we specify.

Filtering

Filtering queries is done by appending `filter()` statements to our query. A typical `filter()` clause has a column, an operator, and a value or column. It is possible to chain multiple `filters()` clauses together or comma separate multiple `ClauseElement` expressions in a single filter, and they will act like ANDs in traditional SQL statements. In Example 7-14, we'll find a cookie named "chocolate chip."

Example 7-14. Filtering by cookie name with filter

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

Example 7-14 prints out the chocolate chip cookie record:

```
Cookie(cookie_name='chocolate chip',
       cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
       cookie_sku='CC01', quantity=12, unit_cost=0.50)
```

There is also a `filter_by()` method that works similarly to the `filter()` method except instead of explicity providing the class as part of the filter expression it uses attribute keyword expressions from the primary entity of the query or the last entity that was joined to the statement. It also uses a keyword assignment instead of a Boolean. Example 7-15 performs the exact same query as Example 7-14.

Example 7-15. Filtering by cookie name with filter_by

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

We can also use a where statement to find all the cookie names that contain the word "chocolate," as shown in Example 7-16.

Example 7-16. Finding names with "chocolate" in them

```
query = session.query(Cookie).filter(Cookie.cookie_name.like('%chocolate%'))
for record in query:
    print(record.cookie_name)
```

The code in Example 7-16 will return:

```
chocolate chip
dark chocolate chip
```

In Example 7-16, we are using the `Cookie.cookie_name` column inside of a filter statement as a type of `ClauseElement` to filter our results, and we are taking advantage of the `like()` method that is available on `ClauseElements`. There are many other methods available, which are listed in Table 2-1.

If we don't use one of the `ClauseElement` methods, then we will have an operator in our filter clauses. Most of the operators work as you might expect, but as the following section explains, there are a few differences.

Operators

So far, we have only explored situations where a column was equal to a value or used one of the `ClauseElement` methods such as `like()`; however, we can also use many other common operators to filter data. SQLAlchemy provides overloading for most of the standard Python operators. This includes all the standard comparison operators (`==`, `!=`, `<`, `>`, `<=`, `>=`), which act exactly like you would expect in a Python statement. The `==` operator also gets an additional overload when compared to `None`,

which converts it to an `IS NULL` statement. Arithmetic operators (`\+`, `-`, `*`, `/`, and `%`) are also supported with additional capabilities for database-independent string concatenation, as shown in Example 7-17.

Example 7-17. String concatenation with +

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

Example 7-17 results in:

```
    ('chocolate chip', 'SKU-CC01')
    ('dark chocolate chip', 'SKU-CC02')
    ('molasses', 'SKU-MOL01')
    ('peanut butter', 'SKU-PB01')
    ('oatmeal raisin', 'SKU-EWW01')
```

Another common usage of operators is to compute values from multiple columns. You'll often do this in applications and reports dealing with data or statistics. Example 7-18 shows a common inventory value calculation.

Example 7-18. Inventory value by 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
for result in query:
    print('{} - {}'.format(result.cookie_name, result.inv_cost))
```

❶ `cast` is a function that allows us to convert types. In this case, we will be getting back results such as 6.0000000000, so by casting it, we can make it look like currency. It is also possible to accomplish the same task in Python with `print('{} - {:.2f}'.format(row.cookie_name, row.inv_cost))`.

❷ We are using the `label()` function to rename the column. Without this renaming, the column would not be listed in the keys of the `result` object, as the operation doesn't have a name.

Example 7-18 results in:

```
    chocolate chip - 6.00
    dark chocolate chip - 0.75
    molasses - 0.80
    peanut butter - 6.00
    oatmeal raisin - 100.00
```

If we need to combine where statements, we can use a couple of different methods. One of those methods is known as Boolean operators.

Boolean Operators

SQLAlchemy also allows for the SQL Boolean operators AND, OR, and NOT via the bitwise logical operators (`&`, `|`, and `~`). Special care must be taken when using the AND, OR, and NOT overloads because of the Python operator precedence rules. For instance, `&` binds more closely than `<`, so when you write `A < B & C < D`, what you are actually writing is `A < (B&C) < D`, when you probably intended to get `(A < B) & (C < D)`.

Often we want to chain multiple `where` clauses together in inclusive and exclusionary manners; this should be done via conjunctions.

Conjunctions

While it is possible to chain multiple `filter()` clauses together, it's often more readable and functional to use conjunctions to accomplish the desired effect. I also prefer to use conjunctions instead of Boolean operators, as conjunctions will make your code more expressive. The conjunctions in SQLAlchemy are `and_()`, `or_()`, and `not_()`. They have underscores to separate them from the built-in keywords. So if we wanted to get a list of cookies with a cost of less than an amount and above a certain quantity we could use the code shown in Example 7-19.

Example 7-19. Using filter with multiple ClauseElement expressions to perform an AND

```
query = session.query(Cookie).filter(
    Cookie.quantity > 23,
    Cookie.unit_cost < 0.40
)
for result in query:
    print(result.cookie_name)
```

The `or_()` function works as the opposite of `and_()` and includes results that match either one of the supplied clauses. If we wanted to search our inventory for cookie types that we have between 10 and 50 of in stock or where the name contains *chip*, we could use the code shown in Example 7-20.

Example 7-20. Using the or() conjunction

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

```
    )
)
for result in query:
    print(result.cookie_name)
```

Example 7-20 results in:

```
chocolate chip
dark chocolate chip
peanut butter
```

The `not_()` function works in a similar fashion to other conjunctions, and it is used to select records where a record does not match the supplied clause.

Now that we can comfortably query data, we are ready to move on to updating existing data.

Updating Data

Much like the insert method we used earlier, there is also an update method with syntax almost identical to inserts, except that they can specify a `where` clause that indicates which rows to update. Like insert statements, update statements can be created by using either the `update()` function or the `update()` method on the table being updated. You can update all rows in a table by leaving off the `where` clause.

For example, suppose you've finished baking those chocolate chip cookies that we needed for our inventory. In Example 7-21, we'll add them to our existing inventory with an update query, and then check to see how many we have currently have in stock.

Example 7-21. Updating data via object

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

❶ Lines 1 and 2 are using the generative method to build our statement.

❷ We're querying to get the object here; however, if you already have it, you can directly edit it without querying it again.

Example 7-21 returns:

```
132
```

It is also possible to update data in place without having the object originally (Example 7-22).

Example 7-22. Updating data in place

```
query = session.query(Cookie)
query = query.filter(Cookie.cookie_name == "chocolate chip")
query.update({Cookie.quantity: Cookie.quantity - 20}) ❶

cc_cookie = query.first() ❷
print(cc_cookie.quantity)
```

❶ The `update()` method causes the record to be updated outside of the session, and returns the number of rows updated.

❷ We are reusing `query` here because it has the same selection criteria we need.

Example 7-22 returns:

```
112
```

In addition to updating data, at some point we will want to remove data from our tables. The following section explains how to do that.

Deleting Data

To create a delete statement, you can use either the `delete()` function or the `delete()` method on the table from which you are deleting data. Unlike `insert()` and `update()`, `delete()` takes no values parameter, only an optional `where` clause (omitting the `where` clause will delete all *rows* from the table). See Example 7-23.

Example 7-23. Deleting data

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

Example 7-23 returns:

```
None
```

It is also possible to delete data in place without having the object (Example 7-24).

Example 7-24. Deleting data

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

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

Example 7-24 returns:

```
None
```

OK, at this point, let's load up some data using what we already learned for the `users`, `orders`, and `line_items` tables. You can copy the code shown here, but you should also take a moment to play with different ways of inserting the data:

```
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()
```

Now that we have customers, we can start to enter their orders and line items into the system as well. We're going to take advantage of the relationship between these two objects when inserting them into the table. If we have an object that we want to associate to another one, we can do that by assigning it to the relationship property just like we would any other property. Let's see this in action a few times in Example 7-25.

Example 7-25. Adding related objects

```
o1 = Order()
o1.user = cookiemon ❶
session.add(o1)

cc = session.query(Cookie).filter(Cookie.cookie_name ==
                                  "chocolate chip").one()
line1 = LineItem(cookie=cc, quantity=2, extended_cost=1.00) ❷

pb = session.query(Cookie).filter(Cookie.cookie_name ==
                                  "peanut butter").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()
```

❶ Sets `cookiemon` as the user who placed the order.

❷ Builds a line item for the order, and associates it with the cookie.

❸ Builds a line item a piece at a time.

❹ Adds the line item to the order via the relationship.

In Example 7-25, we create an empty `Order` instance, and set its `user` property to the `cookiemon` instance. Next, we add it to the session. Then we query for the chocolate chip cookie and create a `LineItem`, and set the cookie to be the chocolate chip one we just queried for. We repeat that process for the second line item on the order; however, we build it up piecemeal. Finally, we add the line items to the order and commit it.

Before we move on, let's add one more order for another user:

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

cc = session.query(Cookie).filter(Cookie.cookie_name ==
                                  "chocolate chip").one()
line1 = LineItem(cookie=cc, quantity=24, extended_cost=12.00)

oat = session.query(Cookie).filter(Cookie.cookie_name ==
                                   "oatmeal raisin").one()
line2 = LineItem(cookie=oat, quantity=6, extended_cost=6.00)

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

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

❶ I moved this line down with the other adds, and SQLAlchemy determined the proper order in which to create them to ensure it was successful.

In Chapter 6, you learned how to define `ForeignKeys` and relationships, but we haven't used them to perform any queries up to this point. Let's take a look at relationships.

Joins

Now let's use the `join()` and `outerjoin()` methods to take a look at how to query related data. For example, to fulfill the order placed by the cookiemon user, we need to determine how many of each cookie type were ordered. This requires you to use a total of three joins to get all the way down to the name of the cookies. The ORM makes this query much easier than it normally would be with raw SQL (Example 7-26).

Example 7-26. Using join to select from multiple tables

```
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) ❶
results = query.filter(User.username == 'cookiemon').all()
print(results)
```

❶ Tells SQLAlchemy to join the related objects.

Example 7-26 will output the following:

```
[
    (u'1', u'cookiemon', u'111-111-1111', u'chocolate chip', 2, Decimal('1.00'))
    (u'1', u'cookiemon', u'111-111-1111', u'peanut butter', 12, Decimal('3.00'))
]
```

It is also useful to get a count of orders by all users, including those who do not have any present orders. To do this, we have to use the `outerjoin()` method, and it requires a bit more care in the ordering of the join, as the table we use the `outerjoin()` method on will be the one from which all results are returned (Example 7-27).

Example 7-27. Using outerjoin to select from multiple tables

```
query = session.query(User.username, func.count(Order.order_id))
query = query.outerjoin(Order).group_by(User.username)
for row in query:
    print(row)
```

Example 7-27 results in:

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

So far, we have been using and joining different tables in our queries. However, what if we have a self-referential table like a table of managers and their reports? The ORM

allows us to establish a relationship that points to the same table; however, we need to specify an option called `remote_side` to make the relationship a many to one:

```
class Employee(Base):
    __tablename__ = 'employees'

    id = Column(Integer(), primary_key=True)
    manager_id = Column(Integer(), ForeignKey('employees.id'))
    name = Column(String(255), nullable=False)

    manager = relationship("Employee", backref=backref('reports'),
                           remote_side=[id]) ❶

Base.metadata.create_all(engine)
```

❶ Establishes a relationship back to the same table, specifies the `remote_side`, and makes the relationship a many to one.

Let's add an employee and another employee that reports to her:

```
marsha = Employee(name='Marsha')
fred = Employee(name='Fred')

marsha.reports.append(fred)

session.add(marsha)
session.commit()
```

Now if we want to print the employees that report to Marsha, we would do so by accessing the `reports` property as follows:

```
for report in marsha.reports:
    print(report.name)
```

It's also useful to be able to group data when we are looking to report on data, so let's look into that next.

Grouping

When using grouping, you need one or more columns to group on and one or more columns that it makes sense to aggregate with counts, sums, etc., as you would in normal SQL. Let's get an order count by customer (Example 7-28).

Example 7-28. Grouping data

```
query = session.query(User.username, func.count(Order.order_id)) ❶
query = query.outerjoin(Order).group_by(User.username) ❷
for row in query:
    print(row)
```

❶ Aggregation via `count`

❷ Grouping by the nonaggregated included column

Example 7-28 results in:

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

We've shown the generative building of statements throughout the previous examples, but I want to focus on it specifically for a moment.

Chaining

We've used chaining several times throughout this chapter, and just didn't acknowledge it directly. Where query chaining is particularly useful is when you are applying logic when building up a query. So if we wanted to have a function that got a list of orders for us it might look like Example 7-29.

Example 7-29. Chaining

```
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()
    return results

get_orders_by_customer('cakeeater')
```

Example 7-29 results in:

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

However, what if we wanted to get only the orders that have shipped or haven't shipped yet? We'd have to write additional functions to support those additional filter options, or we can use conditionals to build up query chains. Another option we might want is whether or not to include details. This ability to chain queries and clauses together enables quite powerful reporting and complex query building (Example 7-30).

Example 7-30. Conditional chaining

```
def get_orders_by_customer(cust_name, shipped=None, details=False):
    query = session.query(Order.order_id, User.username, User.phone)
```

```
    query = query.join(User)
    if details:
        query = query.add_columns(Cookie.cookie_name, LineItem.quantity,
                                  LineItem.extended_cost)
        query = query.join(LineItem).join(Cookie)
    if shipped is not None:
        query = query.where(Order.shipped == shipped)
    results = query.filter(User.username == cust_name).all()
    return results

get_orders_by_customer('cakeeater') ❶

get_orders_by_customer('cakeeater', details=True) ❷

get_orders_by_customer('cakeeater', shipped=True) ❸

get_orders_by_customer('cakeeater', shipped=False) ❹

get_orders_by_customer('cakeeater', shipped=False, details=True) ❺
```

❶ Gets all orders.

❷ Gets all orders with details.

❸ Gets only orders that have shipped.

❹ Gets orders that haven't shipped yet.

❺ Gets orders that haven't shipped yet with details.

Example 7-30 results in:

```
    [(u'2', u'cakeeater', u'222-222-2222')]

    [(u'2', u'cakeeater', u'222-222-2222', u'chocolate chip', 24, Decimal('12.00')),
     (u'2', u'cakeeater', u'222-222-2222', u'oatmeal raisin', 6, Decimal('6.00'))]

    []

    [(u'2', u'cakeeater', u'222-222-2222')]

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

So far in this chapter, we've used the ORM for all the examples; however, it is also possible to use literal SQL at times as well. Let's take a look.

Raw Queries

While I rarely use a full raw SQL statement, I will often use small text snippets to help make a query clearer. Example 7-31 shows a raw SQL `where` clause using the `text()` function.

Example 7-31. Partial text query

```
from sqlalchemy import text
query = session.query(User).filter(text("username='cookiemon'"))
print(query.all())
```

Example 7-31 results in:

```
    [User(username='cookiemon', email_address='mon@cookie.com',
          phone='111-111-1111', password='password')]
```

Now you should have an understanding of how to use the ORM to work with data in SQLAlchemy. We explored how to create, read, update, and delete operations. This is a good point to stop and explore a bit on your own. Try to create more cookies, orders, and line items, and use query chains to group them by order and user.

Now that you've explored a bit more and hopefully broken something, let's investigate how to react to exceptions raised in SQLAlchemy, and how to use transactions to group statements that must succeed or fail as a group.

CHAPTER 8

Understanding the Session and Exceptions

In the previous chapter, we did a lot of work with the session, and we avoided doing anything that could result in an exception. In this chapter, we will learn a bit more about how our objects and the SQLAlchemy session interact. We'll conclude this chapter by purposely performing some actions incorrectly so that we can see the types of exceptions that occur and how we should respond to them. Let's start by learning more about the session. First, we'll set up an in-memory SQLite database using the tables from Chapter 6:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///:memory:')

Session = sessionmaker(bind=engine)

session = Session()

from datetime import datetime

from sqlalchemy import (Table, Column, Integer, Numeric, String, DateTime,
                        ForeignKey, Boolean)
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, 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))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))

    def __init__(self, name, recipe_url=None, sku=None, quantity=0,
                 unit_cost=0.00):
        self.cookie_name = name
        self.cookie_recipe_url = recipe_url
        self.cookie_sku = sku
        self.quantity = quantity
        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}', " \
                       "quantity={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)
    password = Column(String(25), nullable=False)
    created_on = Column(DateTime(), default=datetime.now)
    updated_on = Column(DateTime(), default=datetime.now, onupdate=datetime.now)

    def __init__(self, username, email_address, phone, password):
        self.username = username
        self.email_address = email_address
        self.phone = phone
        self.password = password

    def __repr__(self):
        return "User(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'))
    shipped = Column(Boolean(), default=False)

    user =  relationship("User", backref=backref('orders', order_by=order_id))
```

```
    def __repr__(self):
        return "Order(user_id={self.user_id}, " \
                    "shipped={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'))
    quantity = Column(Integer())
    extended_cost = Column(Numeric(12, 2))

    order = relationship("Order", backref=backref('line_items',
                                                  order_by=line_item_id))
    cookie = relationship("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(engine)
```

Now that we have our initial database and objects defined, we're ready to learn more about how the session works with objects.

The SQLAlchemy Session

We covered a bit about the SQLAlchemy session in Chapter 7; however, here I want to focus on how our data objects and the session interact.

When we use a query to get an object, we get back an object that is connected to a session. That object could move through several states in relationship to the session.

Session States

Understanding the session states can be useful for troubleshooting exceptions and handling unexpected behaviors. There are four possible states for data object instances:

Transient
: The instance is not in session, and is not in the database.

Pending
: The instance has been added to the session with `add()`, but hasn't been flushed or committed.

Persistent
: The object in session has a corresponding record in the database.

Detached
: The instance is no longer connected to the session, but has a record in the database.

We can watch an instance move through these states as we work with it. We'll start by creating an instance of a cookie:

```
cc_cookie = Cookie('chocolate chip',
                   'http://some.aweso.me/cookie/recipe.html',
                   'CC01', 12, 0.50)
```

To see the instance state, we can use the powerful `inspect()` method provided by SQLAlchemy. When we use `inspect()` on an instance, we gain access to several useful pieces of information. Let's get an inspection of our `cc_cookie` instance:

```
from sqlalchemy import inspect
insp = inspect(cc_cookie)
```

In this case, we are interested in the `transient`, `pending`, `persistent`, and `detached` properties that indicate the current state. Let's loop through those properties as shown in Example 8-1.

Example 8-1. Getting the session state of an instance

```
for state in ['transient', 'pending', 'persistent', 'detached']:
    print('{:>10}: {}'.format(state, getattr(insp, state)))
```

Example 8-1 results in a list of states and a Boolean indicating whether that is the current state:

```
 transient: True
   pending: False
persistent: False
  detached: False
```

As you can see in the output, the current state of our cookie instance is transient, which is the state newly created objects are in prior to being flushed or commited to the database. If we add `cc_cookie` to the current session and rerun Example 8-1, we will get the following output:

```
transient: False
   pending: True
persistent: False
  detached: False
```

Now that our cookie instance has been added to the current session, we can see it has been moved to pending. If we commit the session and rerun Example 8-1 yet again, we can see the state is updated to persistent:

```
 transient: False
   pending: False
persistent: True
  detached: False
```

Finally, to get `cc_cookie` into the detached state, we want to call the `expunge()` method on the session. You might do this if you are moving data from one session to another. One case in which you might want to move data from one session to another is when you are archiving or consolidating data from your primary database to your data warehouse:

```
session.expunge(cc_cookie)
```

If we rerun Example 8-1 one last time after expunging `cc_cookie`, we'll see that it is now in a detached state:

```
 transient: False
   pending: False
persistent: False
  detached: True
```

If you are using the `inspect()` method in normal code, you'll probably want to use `insp.transient`, `insp.pending`, `insp.persistent`, and `insp.detached`. We used `getattr` to access them so we could loop through the states instead of hardcoding each state individually.

Now that we've seen how an object moves through the session states, let's look at how we can use the inspector to see the history of an instance prior to committing it. First, we'll add our object back to the session and change the `cookie_name` attribute:

```
session.add(cc_cookie)
cc_cookie.cookie_name = 'Change chocolate chip'
```

Now let's use the inspector's `modified` property to see if it has changed:

```
insp.modified
```

This will return `True`, and we can use the inspector's `attrs` collection to find what has changed. Example 8-2 demonstrates one way we can accomplish this.

Example 8-2. Printing the changed attribute history

```
for attr, attr_state in insp.attrs.items():
    if attr_state.history.has_changes(): ❶
        print('{}: {}'.format(attr, attr_state.value))
        print('History: {}\n'.format(attr_state.history)) ❷
```

❶ Checks the attribute state to see if the session can find any changes

❷ Prints the `history` object of the changed attribute

When we run Example 8-2, we will get back:

```
cookie_name: Change chocolate chip
History: History(added=['Change chocolate chip'], unchanged=(), deleted=())
```

The output from Example 8-2 shows us that the `cookie_name` changed. When we look at the history record for that attribute, it shows us what was added or changed on that attribute. This gives us a good view of how objects interact with the session. Now, let's investigate how to handle exceptions that arise while using SQLAlchemy ORM.

Exceptions

There are numerous exceptions that can occur in SQLAlchemy, but we'll focus on two in particular: `MultipleResultsFound` and `DetachedInstanceError`. These exceptions are fairly common, and they also are part of a group of similar exceptions. By learning how to handle these exceptions, you'll be better prepared to deal with other exceptions you might encounter.

MultipleResultsFound Exception

This exception occurs when we use the `.one()` query method, but get more than one result back. Before we can demonstrate this exception, we need to create another cookie and save it to the database:

```
dcc = Cookie('dark chocolate chip',
             'http://some.aweso.me/cookie/recipe_dark.html',
             'CC02', 1, 0.75)
session.add(dcc)
session.commit()
```

Now that we have more than one cookie in the database, let's trigger the `MultipleResultsFound` exception (Example 8-3).

Example 8-3. Causing a MultipleResultsFound exception

```
results = session.query(Cookie).one()
```

Example 8-3 results in an exception because there are two cookies in our data source that match the query. The exception stops the execution of our program. Example 8-4 shows what this exception looks like.

Example 8-4. Exception output from Example 8-3

```
MultipleResultsFound                    Traceback (most recent call last) ❶
<ipython-input-20-d88068ecde4b> in <module>()
----> 1 results = session.query(Cookie).one() ❷

...b/python2.7/site-packages/sqlalchemy/orm/query.pyc in one(self)
   2480         else:
   2481             raise orm_exc.MultipleResultsFound(
-> 2482                 "Multiple rows were found for one()")
   2483
   2484     def scalar(self):

MultipleResultsFound: Multiple rows were found for one() ❸
```

❶ This shows us the type of exception and that a traceback is present.

❷ This is the actual line where the exception occurred.

❸ This is the interesting part we need to focus on.

In Example 8-4, we have the typical format for a `MultipleResultsFound` exception from SQLAlchemy. It starts with a line that indicates the type of exception. Next, there is a traceback showing us where the exception occurred. The final block of lines is where the important details can be found: they specify the type of exception and explain why it occurred. In this case, it is because our query returned two rows and we told it to return one and only one.

Another exception related to this is the `NoResultFound` exception, which would occur if we used the `.one()` method and the query returned no results.

If we wanted to handle this exception so that our program doesn't stop executing and prints out a more helpful exception message, we can use the Python `try/except` block, as shown in Example 8-5.

Example 8-5. Handling a MultipleResultsFound exception

```
from sqlalchemy.orm.exc import MultipleResultsFound ❶
try:
    results = session.query(Cookie).one()
except MultipleResultsFound as error: ❷
    print('We found too many cookies... is that even possible?')
```

❶ All of the SQLAlchemy ORM exceptions are available in the `sqlalchemy.orm.exc` module.

❷ Catching the `MultipleResultsFound` exception as `error`.

Executing Example 8-5 will result in "We found too many cookies… is that even possible?" being printed and our application will keep on running normally. (I don't think is it possible to find too many cookies.) Now you know what the `MultipleResultsFound` exception is telling you, and you know at least one method for dealing with it in a way that allows your program to continue executing.

DetachedInstanceError

This exception occurs when we attempt to access an attribute on an instance that needs to be loaded from the database, but the instance we are using is not currently attached to the database. Before we can explore this exception, we need to set up the records we are going to operate on. Example 8-6 shows how to do that.

Example 8-6. Creating a user and an order for that user

```
cookiemon = User('cookiemon', 'mon@cookie.com', '111-111-1111', 'password')
session.add(cookiemon)
o1 = Order()
o1.user = cookiemon
session.add(o1)

cc = session.query(Cookie).filter(Cookie.cookie_name ==
                                  "Change chocolate chip").one()
line1 = LineItem(order=o1, cookie=cc, quantity=2, extended_cost=1.00)

session.add(line1)
session.commit()
```

Now that we've created a user with an order and some line items, we have what we need to cause this exception. Example 8-7 will trigger the exception for us.

Example 8-7. Causing a DetachedInstanceError

```
order = session.query(Order).first()
session.expunge(order)
order.line_items
```

In Example 8-7, we are querying to get an instance of an order. Once we have our instance, we detach it from the session with `expunge`. Then we attempt to load the `line_items` attribute. Because `line_items` is a relationship, by default it doesn't load all that data until you ask for it. In our case, we detached the instance from the ses-

sion and the relationship doesn't have a session to execute a query to load the `line_items` attribute, and it raises the `DetachedInstanceError`:

```
DetachedInstanceError                    Traceback (most recent call last)
<ipython-input-35-233bbca5c715> in <module>()
      1 order = session.query(Order).first()
      2 session.expunge(order)
----> 3 order.line_items ❶

site-packages/sqlalchemy/orm/attributes.pyc in __get__(self, instance, owner)
    235             return dict_[self.key]
    236         else:
--> 237             return self.impl.get(instance_state(instance), dict_)
    238
    239

site-packages/sqlalchemy/orm/attributes.pyc in get(self, state, dict_, passive)
    576                     value = callable_(state, passive)
    577                 elif self.callable_:
--> 578                     value = self.callable_(state, passive)
    579                 else:
    580                     value = ATTR_EMPTY

site-packages/sqlalchemy/orm/strategies.pyc in _load_for_state(self, state,
passive)
    499                 "Parent instance %s is not bound to a Session; "
    500                 "lazy load operation of attribute '%s' cannot proceed" %
--> 501                 (orm_util.state_str(state), self.key)
    502             )
    503

DetachedInstanceError: Parent instance <Order at 0x10dc31350> is not bound to
a Session; lazy load operation of attribute 'line_items' cannot proceed ❷
```

❶ This is the actual line where the error occurred.

❷ This is the interesting part we need to focus on.

We read this exception output just like we did in Example 8-5; the interesting part of the message indicates that we tried to load the `line_items` attribute on an `Order` instance, and it couldn't do that because that instance wasn't bound to a session.

While this is a forced example of the `DetachedInstanceError`, it acts very similarly to other exceptions such as `ObjectDeletedError`, `StaleDataError`, and `ConcurrentModificationError`. All of these are related to information differing between the instance, the session, and the database. The same method used in Example 8-6 with the `try/except` block will also work for this exception and allow the code to continue running. You could check for a detacted instance, and in the `except` block add it back

to the session; however, the `DetachedInstanceError` is normally an indicator of an exception occuring prior to this point in the code.

To get more details on additional exceptions that can occur with the ORM, check out the SQLAlchemy documentation at *http://docs.sqlalchemy.org/en/latest/orm/exceptions.html*.

All the exceptions mentioned so far, such as `MultipleResultsFound` and `DetachedInstanceError`, are related to a single statement failing. If we have multiple statements, such as multiple adds or deletes that fail during a commit or a flush, we need to handle those differently. Specifically, we handle them by manually controlling the transaction. The following section covers transactions in detail, and teaches you how to use them to help handle exceptions and recover a session.

Transactions

Transactions are a group of statements that we need to succeed or fail as a group. When we first create a session, it is not connected to the database. When we undertake our first action with the session such as a query, it starts a connection and a transaction. This means that by default, we don't need to manually create transactions. However, if we need to handle any exceptions where part of the transaction succeeds and another part fails or where the result of a transaction creates an exception, then we must know how to control the transaction manually.

There is already a good example of when we might want to do this in our existing database. After a customer has ordered cookies from us, we need to ship those cookies to the customer and remove them from our inventory. However, what if we do not have enough of the right cookies to fulfill an order? We will need to detect that and not ship that order. We can solve this with transactions.

We'll need a fresh Python shell with the tables from Chapter 6; however, we need to add a `CheckConstraint` to the quantity column to ensure it cannot go below 0, because we can't have negative cookies in inventory. Next, we need to re-create the cookiemon user as well as the chocolate chip and dark chocolate chip cookie records and set the quantity of chocolate chip cookies to 12 and the dark chocolate chip cookies to 1. Example 8-8 shows how to perform all those steps.

Example 8-8. Setting up the transactions environment

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///:memory:')
```

```
Session = sessionmaker(bind=engine)

session = Session()

from datetime import datetime

from sqlalchemy import (Table, Column, Integer, Numeric, String,
                        DateTime, ForeignKey, Boolean, CheckConstraint)
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, backref


Base = declarative_base()


class Cookie(Base):
    __tablename__ = 'cookies'
    __table_args__ = (CheckConstraint('quantity >= 0', name='quantity_positive'),) ❶

    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))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))

    def __init__(self, name, recipe_url=None, sku=None, quantity=0, unit_cost=0.00):
        self.cookie_name = name
        self.cookie_recipe_url = recipe_url
        self.cookie_sku = sku
        self.quantity = quantity
        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}', " \
                       "quantity={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)
    password = Column(String(25), nullable=False)
    created_on = Column(DateTime(), default=datetime.now)
    updated_on = Column(DateTime(), default=datetime.now, onupdate=datetime.now)
```

```
    def __init__(self, username, email_address, phone, password):
        self.username = username
        self.email_address = email_address
        self.phone = phone
        self.password = password

    def __repr__(self):
        return "User(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'))
    shipped = Column(Boolean(), default=False)

    user =  relationship("User", backref=backref('orders', order_by=order_id))

    def __repr__(self):
        return "Order(user_id={self.user_id}, " \
                     "shipped={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'))
    quantity = Column(Integer())
    extended_cost = Column(Numeric(12, 2))

    order = relationship("Order", backref=backref('line_items',
                                                  order_by=line_item_id))
    cookie = relationship("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)

Base.metadata.create_all(engine)

cookiemon = User('cookiemon', 'mon@cookie.com', '111-111-1111', 'password') ❷
cc = Cookie('chocolate chip', 'http://some.aweso.me/cookie/recipe.html', ❸
            'CC01', 12, 0.50)
```

```
dcc = Cookie('dark chocolate chip', ❹
             'http://some.aweso.me/cookie/recipe_dark.html',
             'CC02', 1, 0.75)

session.add(cookiemon)
session.add(cc)
session.add(dcc)
```

❶ Adds the positive quantity `CheckConstraint`.

❷ Adds the cookiemon user.

❸ Adds the chocolate chip cookie.

❹ Adds the dark chocolate chip cookie.

We're now going to define two orders for the cookiemon user. The first order will be for two chocolate chip cookies and nine chocolate chip cookies, and the second order will be for nine dark chocolate chip cookies. Example 8-9 shows the details.

Example 8-9. Adding the orders

```
o1 = Order()
o1.user = cookiemon
session.add(o1)

line1 = LineItem(order=o1, cookie=cc, quantity=9, extended_cost=4.50)


session.add(line1)
session.commit() ❶
o2 = Order()
o2.user = cookiemon
session.add(o2)

line1 = LineItem(order=o2, cookie=cc, quantity=2, extended_cost=1.50)
line2 = LineItem(order=o2, cookie=dcc, quantity=9, extended_cost=6.75)


session.add(line1)
session.add(line2)
session.commit() ❷
```

❶ Adding the first order.

❷ Adding the second order.

That will give us all the order data we need to explore how transactions work. Now we need to define a function called `ship_it`. Our `ship_it` function will accept an `order_id`, remove the cookies from inventory, and mark the order as shipped. Example 8-10 shows how this works.

Example 8-10. Defining the ship_it function

```
def ship_it(order_id):
    order = session.query(Order).get(order_id)
    for li in order.line_items:
        li.cookie.quantity = li.cookie.quantity - li.quantity ❶
        session.add(li.cookie)
    order.shipped = True ❷
    session.add(order)
    session.commit()
    print("shipped order ID: {}".format(order_id))
```

❶ For each line item we find on the order, this removes the quantity on that line item from the cookie quantity so we know how many cookies we have left.

❷ We update the order to mark it as shipped.

The `ship_it` function will perform all the actions required when we ship an order. Let's run it on our first order and then query the `cookies` table to make sure it reduced the cookie count correctly. Example 8-11 shows how to do that.

Example 8-11. Running ship_it on the first order

```
ship_it(1) ❶
print(session.query(Cookie.cookie_name, Cookie.quantity).all()) ❷
```

❶ Runs `ship_it` on the first `order_id`.

❷ Prints our cookie inventory.

Running the code in Example 8-11 results in:

```
shipped order ID: 1
[(u'chocolate chip', 3), (u'dark chocolate chip', 1)]
```

Excellent! It worked. We can see that we don't have enough cookies in our inventory to fulfill the second order; however, in our fast-paced warehouse, these orders might be processed at the same time. Now try shipping our second order with the `ship_it` function by running the following line of code:

```
ship_it(2)
```

That command gives us this result:

```
IntegrityError                            Traceback (most recent call last)
<ipython-input-7-8a7f7805a7f6> in <module>()
 ---> 1 ship_it(2)
<ipython-input-5-c442ae46326c> in ship_it(order_id)
      6     order.shipped = True
      7     session.add(order)
 ---> 8     session.commit()
      9     print("shipped order ID: {}".format(order_id))
...

IntegrityError: (sqlite3.IntegrityError) CHECK constraint failed:
quantity_positive [SQL: u'UPDATE cookies SET quantity=? WHERE
cookies.cookie_id = ?'] [parameters: (-8, 2)]
```

We got an `IntegrityError` because we didn't have enough dark chocolate chip cookies to ship the order. This actually breaks our current session. If we attempt to issue any more statements via the session such as a query to get the list of cookies, we'll get the output shown in Example 8-12.

Example 8-12. Query on an errored session

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

InvalidRequestError                       Traceback (most recent call last)
<ipython-input-8-90b93364fb2d> in <module>()
 ---> 1 print(session.query(Cookie.cookie_name, Cookie.quantity).all())

...

InvalidRequestError: This Session's transaction has been rolled back due to a
previous exception during flush. To begin a new transaction with this Session,
first issue Session.rollback(). Original exception was: (sqlite3.IntegrityError)
CHECK constraint failed: quantity_positive [SQL: u'UPDATE cookies SET
quantity=? WHERE cookies.cookie_id = ?'] [parameters: ((1, 1), (-8, 2))]
```

Trimming out all the traceback details, an `InvalidRequestError` exception is raised due to the previous `IntegrityError`. To recover from this session state, we need to manually roll back the transaction. The message shown in Example 8-12 can be a bit confusing because it says "This Session's transaction has been rolled back," but it does point you toward performing the roll back manually. The `rollback()` method on the session will restore our session to a working station. Whenever we encounter an exception, we want to issue a `rollback()`:

```
session.rollback()
print(session.query(Cookie.cookie_name, Cookie.quantity).all())
```

This code will output normally with the data we expect:

```
[(u'chocolate chip', 3), (u'dark chocolate chip', 1)]
```

With our session working normally, now we can use a `try/except` block to ensure that if the `IntegrityError` exception occurs a message is printed and transaction is rolled back so our application will continue to function normally, as shown in Example 8-13.

Example 8-13. Transactional ship_it

```
from sqlalchemy.exc import IntegrityError ❶
def ship_it(order_id):
    order = session.query(Order).get(order_id)
    for li in 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("shipped order ID: {}".format(order_id))
    except IntegrityError as error: ❷
        print('ERROR: {!s}'.format(error.orig))
        session.rollback() ❸
```

❶ Importing the `IntegrityError` so we can handle its exception.

❷ Committing the transaction if no exceptions occur.

❸ Rolling back the transaction if an exception occurs.

Let's rerun our transaction-based `ship_it` on the second order as shown here:

```
ship_it(2)

ERROR: CHECK constraint failed: quantity_positive
```

The program doesn't get stopped by the exception, and prints us the exception message without the traceback. Let's check the inventory like we did in Example 8-12 to make sure that it didn't mess up our inventory with a partial shipment:

```
[(u'chocolate chip', 3), (u'dark chocolate chip', 1)]
```

Excellent! Our transactional function didn't mess up our inventory or crash our application. We also didn't have to do a lot of coding to manually roll back the statements that did succeed. As you can see, transactions can be really useful in situations like this, and can save you a lot of manual coding.

In this chapter, we saw how to handle exceptions in both single statements and groups of statements. By using a normal `try/except` block on a single statements, we can prevent our application from crashing in case a database statement error occurs. We also looked at the session transaction to avoid inconsistent databases, and appli-

cation crashes in groups of statements. In the next chapter, we'll learn how to test our code to ensure it behaves the way we expect.

CHAPTER 9

Testing with SQLAlchemy ORM

Most testing inside of applications consists of both unit and functional tests; however, with SQLAlchemy, it can be a lot of work to correctly mock out a query statement or a model for unit testing. That work often does not truly lead to much gain over testing against a database during the functional test. This leads people to make wrapper functions for their queries that they can easily mock out during unit tests, or to just test against a database in both their unit and function tests. I personally like to use small wrapper functions when possible or—if that doesn't make sense for some reason or I'm in legacy code—mock it out.

This chapter covers how to perform functional tests against a database, and how to mock out SQLAlchemy queries and connections.

Testing with a Test Database

For our example application, we are going to have an *app.py* file that contains our application logic, and a *db.py* file that contains our data models and session. These files can be found in the *CH10/* folder of this book's example code.

How an application is structured is an implementation detail that can have quite an effect on how you need to do your testing. In *db.py*, you can see that our database is set up via the `DataAccessLayer` class. We're using this data access class to enable us to initialize a database engine and session whenever we like. You'll see this pattern commonly used in web frameworks when coupled with SQLAlchemy. The `DataAccessLayer` class is initialized without an engine and a session in the `dal` variable.

While we will be testing with a SQLite database in our examples, it is highly recommended that you test against the same database engine you use in your production environment. If you use a different database in testing, some of your constraints and statements that may work with SQLite might not work with, say, PostgreSQL.

Example 9-1 shows a snippet of our *db.py* file that contains the `DataAccessLayer`.

Example 9-1. DataAccessLayer class

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()


class DataAccessLayer:

    def __init__(self): ❶
        self.engine = None
        self.conn_string = 'some conn string'

    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__` provides a way to initialize a connection with a specific connection string like a factory.

❷ The `connect` method creates all the tables in our `Base` class, and uses `sessionmaker` to create a easy way to make sessions for use in our application.

❸ The `connect` method provides an instance of the `DataAccessLayer` class that can be imported throughout our application.

In addition to our `DataAccessLayer`, we also have all our data models defined in the *db.py* file. These models are the same models we used in Chapter 8, all gathered together for us to use in our application. Here are all the models in the *db.py* file:

```
from datetime import datetime

from sqlalchemy import (Column, Integer, Numeric, String, DateTime, ForeignKey,
                        Boolean, create_engine)
from sqlalchemy.orm import relationship, 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))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(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(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)
    password = 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}', " \
            "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'))
    shipped = Column(Boolean(), default=False)

    user = relationship("User", backref=backref('orders', order_by=order_id))

    def __repr__(self):
        return "Order(user_id={self.user_id}, " \
            "shipped={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'))
    quantity = Column(Integer())
    extended_cost = Column(Numeric(12, 2))

    order = relationship("Order", backref=backref('line_items',
                                                  order_by=line_item_id))
    cookie = relationship("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)
```

With our data models and access class in place, we are ready to look at the code we are going to test. We are going to write tests for the `get_orders_by_customer` function we built in Chapter 7, which is found the *app.py file*, shown in Example 9-2.

Example 9-2. app.py to test

```
from db import dal, Cookie, LineItem, Order, User, DataAccessLayer ❶


def get_orders_by_customer(cust_name, shipped=None, details=False):
    query = dal.session.query(Order.order_id, User.username, User.phone) ❷
    query = query.join(User)
    if details:
        query = query.add_columns(Cookie.cookie_name, LineItem.quantity,
                                  LineItem.extended_cost)
        query = query.join(LineItem).join(Cookie)
    if shipped is not None:
        query = query.filter(Order.shipped == shipped)
    results = query.filter(User.username == cust_name).all()
    return results
```

❶ This is our `DataAccessLayer` instance from the *db.py* file, and the data models we are using in our application code.

❷ Because our session is an attribute of the `dal` object, we need to access it from there.

Let's look at all the ways the `get_orders_by_customer` function can be used. We're going to assume for this exercise that we have already validated that the inputs to the function are of the correct type. However, it would be very wise to make sure you test with data that will work correctly and data that could cause errors. Here's a list of the variables our function can accept and their possible values:

- `cust_name` can be blank, a string containing the name of a valid customer, or a string that does not contain the name of a valid customer.
- `shipped` can be `None`, `True`, or `False`.
- `details` can be `True` or `False`.

If we want to test all of the possible combinations, we will need 12 (that's 3 * 3 * 2) tests to fully test this function.

It is important not to test things that are just part of the basic functionality of SQLAlchemy, as SQLAlchemy already comes with a large collection of well-written tests. For example, we wouldn't want to test a simple insert, select, delete, or update statement, as those are tested within the SQLAlchemy project itself. Instead, look to test things that your code manipulates that could affect how the SQLAlchemy statement is run or the results returned by it.

For this testing example, we're going to use the built-in `unittest` module. Don't worry if you're not familiar with this module; I'll explain the key points. First, we need to set up the test class and initialize the `dal`'s connection by creating a new file named *test_app.py*, as shown in Example 9-3.

Example 9-3. Setting up the tests

```
import unittest

from db import dal


class TestApp(unittest.TestCase): ❶

    @classmethod
    def setUpClass(cls): ❷
        dal.conn_string = 'sqlite:///:memory:' ❸
        dal.connect()
```

❶ `unittest` requires test classes inherited from `unittest.TestCase`.

❷ The `setUpClass` method is run once prior to all the tests.

❸ Sets the connection string to our in-memory database for testing.

With our database connected, we are ready to write some tests. We're going to add these tests as functions within the `TestApp` class as shown in Example 9-4.

Example 9-4. The first six tests for blank usernames

```
    def test_orders_by_customer_blank(self): ❶
        results = get_orders_by_customer('')
        self.assertEqual(results, []) ❷

    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):
        results = 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, [])
```

❶ `unittest` expects each test to begin with the letters `test`.

❷ `unittest` uses the `assertEqual` method to verify that the result matches what you expect. Because a user is not found, you should get an empty list back.

Now save the test file as *test_app.py*, and run the unit tests with the following command:

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

Ran 6 tests in 0.018s
```

You might get a warning about SQLite and decimal types when running the unit tests; just ignore this as it's normal for our examples. It occurs because SQLite doesn't have a true decimal type, and SQLAlchemy wants you to know that there could be some oddities due to the conversion from SQLite's float type. It is always wise to investigate these messages, because in production code they will normally point you to the proper way you should be doing something. We are purposely triggering this warning here so that you can see what it looks like.

Now we need to load up some data, and make sure our tests still work. We're going to reuse the work we did in Chapter 7, and insert the same users, orders, and line items. However, this time we are going to wrap the data inserts in a function called `db_prep`. This will allow us to insert this data prior to a test with a simple function call. For simplicity's sake, I have put this function inside the *db.py* file (see Example 9-5); however, in real-world situations, it will often be located in a test fixtures or utilities file instead.

Example 9-5. Inserting some test data

```
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',
                quantity=1,
                unit_cost=0.75)
    c2 = Cookie(cookie_name='peanut butter',
                cookie_recipe_url='http://some.aweso.me/cookie/peanut.html',
                cookie_sku='PB01',
                quantity=24,
                unit_cost=0.25)
    c3 = Cookie(cookie_name='oatmeal raisin',
                cookie_recipe_url='http://some.okay.me/cookie/raisin.html',
                cookie_sku='EWW01',
                quantity=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 = LineItem(cookie=c1, quantity=2, extended_cost=1.00)

line2 = LineItem(cookie=c3, quantity=12, extended_cost=3.00)

o1.line_items.append(line1)
o1.line_items.append(line2)
session.commit()

o2 = Order()
o2.user = cakeeater

line1 = LineItem(cookie=c1, quantity=24, extended_cost=12.00)
line2 = LineItem(cookie=c3, quantity=6, extended_cost=6.00)

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

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

Now that we have a `prep_db` function, we can use it in our *test_app.py* `setUpClass` method to load data into the database prior to running our tests. So now our `setUpClass` method looks like this:

```
@classmethod
def setUpClass(cls):
    dal.conn_string = 'sqlite:///:memory:'
    dal.connect()
    dal.session = dal.Session()
    prep_db(dal.session)
    dal.session.close()
```

We also need to create a new session before each test, and roll back any changes made during that session after each test. We can do that by adding a `setUp` method that is automatically run prior to each individual test, and a `tearDown` method that is automatically run after each test, as follows:

```
def setUp(self): ❶
    dal.session = dal.Session()

def tearDown(self): ❷
    dal.session.rollback()
    dal.session.close()
```

❶ Establishes a new session before each test.

❷ Rolls back any changes during the test and cleans up the session after each test.

We're also going to add our expected results as properties of the `TestApp` class as shown here:

```
    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'))]
```

We can use the test data to ensure that our function does the right thing when given a valid username. These tests go inside of our `TestApp` class as new functions, as Example 9-6 shows.

Example 9-6. Tests for a valid user

```
    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):
        results = get_orders_by_customer('cookiemon', True)
        self.assertEqual(results, [])

    def test_orders_by_customer_unshipped_only(self):
        results = get_orders_by_customer('cookiemon', False)
        self.assertEqual(results, self.cookie_orders)

    def test_orders_by_customer_with_details(self):
        results = get_orders_by_customer('cookiemon', details=True)
        self.assertEqual(results, self.cookie_details)

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

    def test_orders_by_customer_unshipped_only_details(self):
        results = get_orders_by_customer('cookiemon', False, True)
        self.assertEqual(results, self.cookie_details)
```

Using the tests in Example 9-6 as guidance, can you complete the tests for what happens for a different user, such as `cakeeater`? How about the tests for a username that doesn't exist in the system yet? Or if we get an integer instead of a string for the username, what will be the result? When you are done, compare your tests to those in the supplied example code for this chapter to see if your tests are similar to the ones used in this book.

We've now learned how we can use SQLAlchemy in functional tests to determine whether a function behaves as expected on a given data set. We have also looked at how to set up a `unittest` file and how to prepare the database for use in our tests. Next, we are ready to look at testing without hitting the database.

Using Mocks

This technique can be a powerful tool when you have a test environment where creating a test database doesn't make sense or simply isn't feasible. If you have a large amount of logic that operates on the result of the query, it can be useful to mock out the SQLAlchemy code to return the values you want so you can test only the surrounding logic. Normally when I am going to mock out some part of the query, I still create the in-memory database, but I don't load any data into it, and I mock out the database connection itself. This allows me to control what is returned by the execute and fetch methods. We are going to explore how to do that in this section.

To learn how to use mocks in our tests, we are going to make a single test for a valid user. We will use the powerful Python mock library to control what is returned by the connection. Mock is part of the `unittest` module in Python 3. However, if you are using Python 2, you will need to install the mock library using pip to get the latest mock features. To do that, run this command:

```
pip install mock
```

Now that we have mock installed, we can use it in our tests. Mock has a patch function that will let us replace a given object in a Python file with a `MagicMock` that we can control from our test. A `MagicMock` is a special type of Python object that tracks how it is used and allows us to define how it behaves based on how it is being used.

First, we need to import the mock library. In Python 2, we need to do the following:

```
import mock
```

In Python 3, we need to do the following:

```
from unittest import mock
```

With mock imported, we are going to use the `patch` method as a decorator to replace the session part of the `dal` object. A decorator is a function that wraps another function, and alters the wrapped function's behavior. Because the `dal` object is imported by name into the *app.py* file, we will need to patch it inside the `app` module. This will get passed into the test function as an argument. Now that we have a mock object, we can set a return value for the `execute` method, which in this case should be nothing but a chained `fetchall` method whose return value is the data we want to test with. Example 9-7 shows the code needed to use the mock in place of the `dal` object.

Example 9-7. Mocked connection test

```
import unittest
from decimal import Decimal

import mock
```

```
from app import 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') ❶
    def 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 ❸
        results = get_orders_by_customer('cookiemon') ❹
        self.assertEqual(results, self.cookie_orders)
```

❶ Patching `dal.session` in the `app` module with a mock.

❷ That mock is passed into the test function as `mock_dal`.

❸ We set the return value of the `execute` method to the chained return value of the `all` method which we set to `self.cookie_order`.

❹ Now we call the test function where the `dal.connection` will be mocked and return the value we set in the prior step.

You can see that a complicated query like the one in Example 9-7 might get tedious quickly when trying to mock out the full query or connection. Don't shy away from the work though; it can be very useful for finding obscure bugs

As an exercise, you should work on building the remainder of the tests that we built with the in-memory database with the mocked test types. I would encourage you to mock both the query and the connection to get familiar with the mocking process.

You should now feel comfortable with testing a function that contains SQLAlchemy ORM functions and models within it. You should also understand how to prepopulate data into the test database for use in your test. Finally, you should understand how to mock both the query and connection objects. While this chapter used a simple example, we will dive further into testing in Chapter 14, which looks at Flask and Pyramid.

Up next, we are going to look at how to handle an existing database with SQLAlchemy without the need to re-create all the schema in Python via reflection with Automap.

CHAPTER 10

Reflection with SQLAlchemy ORM and Automap

As you learned in Chapter 5, reflection lets you populate a SQLAlchemy object from an existing database; reflection works on tables, views, indexes, and foreign keys. But what if you want to reflect a database schema into ORM-style classes? Fortunately, the handy SQLAlchemy extension automap lets you do just that.

Reflection via automap is a very useful tool; however, as of version 1.0 of SQLAlchemy we cannot reflect `CheckConstraints`, comments, or triggers. You also can't reflect client-side defaults or an association between a sequence and a column. However, it is possible to add them manually using the methods we learned in Chapter 6.

Just like in Chapter 5, we are going to use the Chinook database for testing. We'll be using the SQLite version, which is available in the *CH11/* folder of this book's sample code. That folder also contains an image of the database schema so you can visualize the schema we'll be working with throughout this chapter.

Reflecting a Database with Automap

In order to reflect a database, instead of using the `declarative_base` we've been using with the ORM so far, we're going to use the `automap_base`. Let's start by creating a `Base` object to work with, as shown in Example 10-1

Example 10-1. Creating a Base object with automap_base

```
from sqlalchemy.ext.automap import automap_base ❶

Base = automap_base() ❷
```

❶ Imports the `automap_base` from the automap extension.

❷ Initializes a `Base` object.

Next, we need an engine connected to the database that we want to reflect. Example 10-2 demonstrates how to connect to the Chinook database.

Example 10-2. Initializaing an engine for the Chinook database

```
from sqlalchemy import create_engine

engine = create_engine('sqlite:///Chinook_Sqlite.sqlite') ❶
```

❶ This connection string assumes you are in the same directory as the example database.

With the `Base` and engine setup, we have everything we need to reflect the database. Using the `prepare` method on the `Base` object we created in Example 10-1 will scan everything available on the engine we just created, and reflect everything it can. Here's how you reflect the database using the automap `Base` object:

```
Base.prepare(engine, reflect=True)
```

That one line of code is all you need to reflect the entire database! This reflection has created ORM objects for each table that is accessible under the `class` property of the automap `Base`. To print a list of those objects, simply run this line of code:

```
Base.classes.keys()
```

Here's the output you get when you do that:

```
['Album',
 'Customer',
 'Playlist',
 'Artist',
 'Track',
 'Employee',
 'MediaType',
 'InvoiceLine',
 'Invoice',
 'Genre']
```

Now let's create some objects to reference the `Artist` and `Album` tables:

```
Artist = Base.classes.Artist
Album = Base.classes.Album
```

The first line of code creates a reference to the `Artist` ORM object we reflected, and the second line creates a reference to the `Album` ORM object we reflected. We can use the `Artist` object just as we did in Chapter 7 with our manually defined ORM

objects. Example 10-3 demonstrates how to perform a simple query with the object to get the first 10 records in the table.

Example 10-3. Using the Artist table

```
from sqlalchemy.orm import Session

session = Session(engine)
for artist in session.query(Artist).limit(10):
    print(artist.ArtistId, artist.Name)
```

Example 10-3 will output the following:

```
    (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')
```

Now that we know how to reflect a database and map it to objects, let's look at how relationships are reflected via automap.

Reflected Relationships

Automap can automatically reflect and establish many-to-one, one-to-many, and many-to-many relationships. Let's look at how the relationship between our `Album` and `Artist` tables was established. When automap creates a relationship, it creates a `<related_object>_collection` property on the object, as shown on the `Artist` object in Example 10-4.

Example 10-4. Using the relationship between Artist and Album to print related data

```
artist = session.query(Artist).first()
for album in artist.album_collection:
    print('{} - {}'.format(artist.Name, album.Title))
```

This will output:

```
    AC/DC - For Those About To Rock We Salute You
    AC/DC - Let There Be Rock
```

You can also configure automap and override certain aspects of its behavior to tailor the classes it creates to precise specifications; however, that is well beyond the scope of this book. You can learn much more about this in the SQLAlchemy documenation.

This chapter wraps up the essential parts of SQLAlchemy ORM. Hopefully, you've gotten a sense for how powerful this part of SQLAlchemy is. This book has given you a solid introduction to ORM, but there's plenty more to learn; for more information, check out the documentation.

In the next section, we're going to learn how to use Alembic to manage database migrations so that we can change our database schema without having to destroy and re-create the database.

PART III
Alembic

Alembic is a tool for handling database changes that leverages SQLAlchemy to perform the migrations. Since SQLAlchemy will only create missing tables when we use the metadata's `create_all` method, it doesn't update the database tables to match any changes we might make to the columns. Nor would it delete tables that we removed from the code. Alembic provides a way to do things like adding/deleting tables, changing column names, and adding new constraints. Because Alembic uses SQLAlchemy to perform the migrations, they can be used on a wide array of backend databases.

CHAPTER 11

Getting Started with Alembic

Alembic provides a way for us to programmically create and perform migrations to handle changes to the database that we'll need to make as our application evolves. For example, we might add columns to our tables or remove attributes from our models. We might also add entirely new models, or split an existing model into multiple models. Alembic provides a way for us to preform these types of changes by leveraging the power of SQLAlchemy.

To get started, we need to install Alembic, which we can do with the following:

```
pip install alembic
```

Once we have Alembic installed, we need to create the migration environment.

Creating the Migration Environment

To create the migration environment, we are going to create a folder labeled *CH12*, and change into that directory. Next, run the `alembic init alembic` command to create our migration environment in the *alembic/* directory. People commonly create the migration environment in a *migrations/* directory, which you can do with `alembic init migrations`. You can choose whatever directory name you like, but I encourage you to name it something distinctive that won't be used as a module name in your code anywhere. This initialization process creates the migration environment and also creates an *alembic.ini* file with the configuration options. If we look at our directory now, we'll see the following structure:

```
.
├── alembic
│   ├── README
│   ├── env.py
│   ├── script.py.mako
│   └── versions
└── alembic.ini
```

Inside our newly created migration environment, we'll find *env.py* and the *script.py.mako* template file along with a *versions/* directory. The *versions/* directory will hold our migration scripts. The *env.py* file is used by Alembic to define and instantiate a SQLAlchemy engine, connect to that engine, and start a transaction, and calls the migration engine properly when you run an Alembic command. The *script.py.mako* template is used when creating a migration, and it defines the basic structure of a migration.

With the environment created, it's time to configure it to work with our application.

Configuring the Migration Environment

The settings in the *alembic.ini* and *env.py* files need to be tweaked so that Alembic can work with our database and application. Let's start with the *alembic.ini* file where we need to change the `sqlalchemy.url` option to match our database connection string. We want to set it to connect to a SQLite file named *alembictest.db* in the current directory; to do that, edit the `sqlalchemy.url` line to look like the following:

```
sqlalchemy.url = sqlite:///alembictest.db
```

In all our Alembic examples, we will be creating all our code in an *app/db.py* file using the declarative style of the ORM. First, create an *app/* directory and an empty *__init__.py* in it to make `app` a module.

Then we'll add the following code into *app/db.py* to set up SQLAlchemy to use the same database that Alembic is configured to use in the *alembic.ini* file and define a declarative base:

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///alembictest.db')

Base = declarative_base()
```

Now we need to change the *env.py* file to point to our metadata, which is an attribute of the `Base` instance we created in *app/db.py*. It uses the metadata to compare what it finds in the database to the models defined in SQLAlchemy. We'll start by adding the current directory to the path that Python uses for locating modules so that it can see our `app` module:

```
import os
import sys

sys.path.append(os.getcwd()) ❶
```

❶ Adds the current working directory (*CH12*) to the *sys.path* that Python uses when searching for modules

Finally, we'll change the target metadata line in *env.py* to match our `metadata` object in the *app/db.py* file:

```
from app.db import Base ❶
target_metadata = Base.metadata ❷
```

❶ Import our instance of `Base`.

❷ Instruct Alembic to use the `Base.metadata` as its target.

With this completed, we have our Alembic environment set up properly to use the application's database and metadata, and we have built a skeleton of an application where we will define our data models in Chapter 12.

In this chapter, we learned how to create a migration environment and configure it to share the same database and metadata as our application. In the next chapter, we will begin building migrations both by hand and using the autogeneration capabilities.

CHAPTER 12

Building Migrations

In Chapter 11, we initialized and configured the Alembic migration environment to prepare for adding data classes to our application and to create migrations to add them to our database. We're going to explore how to use autogenerate for adding tables, and how to handcraft migrations to accomplish things that autogenerate cannot do. It's always a good idea to start with an empty migration, so let's begin there as it gives us a clean starting point for our migrations.

Generating a Base Empty Migration

To create the base empty migration, make sure you are in the *CH12/* folder of the example code for this book. We'll create an empty migration using this command:

```
# alembic revision -m "Empty Init" ❶
   Generating ch12/alembic/versions/8a8a9d067_empty_init.py ... done
```

❶ Run the `alembic revision` command and add the message (`-m`) `"Empty Init"` to the migration

This will create a migration file in the *alembic/versions/* subfolder. The filenames will always begin with a hash that represents the revision ID and then whatever message you supply. Let's look inside this file:

```
"""Empty Init ❶

Revision ID: 8a8a9d067
Revises:
Create Date: 2015-09-13 20:10:05.486995

"""

# revision identifiers, used by Alembic.
```

```
revision = '8a8a9d067' ❷
down_revision = None ❸
branch_labels = None ❹
depends_on = None ❺

from alembic import op
import sqlalchemy as sa


def upgrade():
    pass


def downgrade():
    pass
```

❶ The migration message we specified

❷ The Alembic revision ID

❸ The previous revision used to determine how to downgrade

❹ The branch associated with this migration

❺ Any migrations that this one depends on

The file starts with a header that contains the message we supplied (if any), the revision ID, and the date and time at which it was created. Following that is the identifiers section, which explains what migration this is and any details about what it downgrades to or depends on, or any branch associated with this migration. Typically, if you are making data class changes in a branch of your code, you'll also want to branch your Alembic migrations as well.

Next, there is an `upgrade` method that would contain any Python code required to perform the changes to the database when applying this migration. Finally, there is a `downgrade` method that contains the code required to undo this migration and restore the database to the prior migration step.

Because we don't have any data classes and have made no changes, both our `upgrade` and `downgrade` methods are empty. So running this migration will have no effect, but it will provide a great foundation for our migration chain. To run all the migrations from whatever the current database state is to the highest Alembic migration, we execute the following command:

```
# alembic upgrade head ❶
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> 8a8a9d067, Empty Init
```

❶ Upgrades the database to the head (newest) revision.

In the preceding output, it will list every revision it went through to get to the newest revision. In our case, we only have one, the Empty Init migration, which it ran last.

With our base in place, we can begin to add our user data class to our application. Once we add the data class, we can build an autogenerated migration and use it to create the associated table in the database.

Autogenerating a Migration

Now let's add our user data class to *app/db.py*. It's going to be the same `Cookie` class we've been using in the ORM section. We're going to add to the imports from `sqlalchemy`, including the `Column` and column types that are used in the `Cookie` class, and then add the `Cookie` class itself, as shown in Example 12-1.

Example 12-1. Updated app/db.py

```
from sqlalchemy import create_engine, Column, Integer, Numeric, String

from sqlalchemy.ext.declarative import declarative_base
from 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))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))
```

With our class added, we are ready to create a migration to add this table to our database. This is a very straightforward migration, and we can take advantage of Alembic's ability to autogenerate the migration (Example 12-2).

Example 12-2. Autogenerating the migration

```
# alembic revision --autogenerate -m "Added Cookie model" ❶
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'cookies'
```

```
INFO  [alembic.autogenerate.compare] Detected added index 'ix_cookies_cookie_name'
      on '['cookie_name']'
  Generating ch12/alembic/versions/34044511331_added_cookie_model.py ... done
```

❶ Autogenerates a migration with the message "Added Cookie model."

So when we run the autogeneration command, Alembic inspects the metadata of our SQLAlchemy `Base` and then compares that to the current database state. In Example 12-2, it detected that we added a table named `cookies` (based on the `__tablename__`) and an index on the `cookie_name` field of that table. It marks the differences as changes, and adds logic to create them in the migration file. Let's investigate the migration file it created in Example 12-3.

Example 12-3. The cookies migration file

```
"""Added Cookie model

Revision ID: 34044511331
Revises: 8a8a9d067
Create Date: 2015-09-14 20:37:25.924031

"""

# revision identifiers, used by Alembic.
revision = '34044511331'
down_revision = '8a8a9d067'
branch_labels = None
depends_on = None

from alembic import op
import sqlalchemy as sa


def upgrade():
    ### commands auto generated by Alembic - please adjust! ###
    op.create_table('cookies', ❶
    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') ❷
    )
    op.create_index(op.f('ix_cookies_cookie_name'), 'cookies', ['cookie_name'],
                    unique=False) ❸
    ### end Alembic commands ###


def downgrade():
```

```
    ### commands auto generated by Alembic - please adjust! ###
    op.drop_index(op.f('ix_cookies_cookie_name'), table_name='cookies') ❹
    op.drop_table('cookies') ❺
    ### end Alembic commands ###
```

❶ Uses Alembic's `create_table` method to add our `cookies` table.

❷ Adds the primary key on the `cookie_id` column.

❸ Uses Alembic's `create_index` method to add an index on the `cookie_name` column.

❹ Drops the `cookie_name` index with Alembic's `drop_index` method.

❺ Drops the `cookies` table.

In the `upgrade` method of Example 12-3, the syntax of the `create_table` method is the same as the `Table` constructor we used with SQLAlchemy Core in Chapter 1: it's the table name followed by all the table's columns, and any constraints. The `create_index` method is the same as the `Index` constructor from Chapter 1 as well.

Finally, the `downgrade` method of Example 12-3 uses Alembic's `drop_index` and `drop_table` methods in the proper order to ensure that both the index and the table are removed.

Let's run this migration using the same command we did for running the empty migration:

```
    # alembic upgrade head
    INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
    INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
    INFO  [alembic.runtime.migration] Running upgrade 8a8a9d067 -> 34044511331,
    Added Cookie model
```

After running this migration, we can take a peek in the database to make sure the changes happened:

```
    # sqlite3 alembictest.db ❶
    SQLite version 3.8.5 2014-08-15 22:37:57
    Enter ".help" for usage hints.
    sqlite> .tables ❷
    alembic_version  cookies
    sqlite> .schema cookies ❸
    CREATE TABLE cookies (
            cookie_id INTEGER NOT NULL,
            cookie_name VARCHAR(50),
            cookie_recipe_url VARCHAR(255),
            cookie_sku VARCHAR(55),
            quantity INTEGER,
```

```
        unit_cost NUMERIC(12, 2),
        PRIMARY KEY (cookie_id)
);
CREATE INDEX ix_cookies_cookie_name ON cookies (cookie_name);
```

❶ Accessing the database via the `sqlite3` command.

❷ Listing the tables in the database.

❸ Printing the schema for the `cookies` table.

Excellent! Our migration created our table and our index perfectly. However, there are some limitations to what Alembic autogenerate can do. Table 12-1 shows a list of common schema changes that autogenerate can detect, and after that Table 12-2 shows some schema changes that autogenerate cannot detect or detects improperly.

Table 12-1. Schema changes that autogenerate can detect

Schema element	Changes
Table	Additions and removals
Column	Additions, removals, change of nullable status on columns
Index	Basic changes in indexes and explicitly named unique constraints, support for autogenerate of indexes and unique constraints
Keys	Basic renames

Table 12-2. Schema changes that autogenerate cannot detect

Schema element	Actions
Tables	Name changes
Column	Name changes
Constraints	Constraints without an explicit name
Types	Types like ENUM that aren't directly supported on a database backend

In addition to the actions that autogenerate can and cannot support, there are some optionally supported capabilities that require special configuration or custom code to implement. Some examples of these are changes to a column type or changes in a server default. If you want to know more about autogeneration's capabilities and limitations, you can read more at the Alembic autogenerate documentation.

Building a Migration Manually

Alembic can't detect a table name change, so let's learn how to do that without autogeneration and change the `cookies` table to be the `new_cookies` table. We need to start by changing the `__tablename__` in our `Cookie` class in *app/db.py*. We'll change it to look like the following:

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

Now we need to create a new migration that we can edit with the message "Renaming cookies to new_cookies":

```
# alembic revision -m "Renaming cookies to new_cookies"
  Generating ch12/alembic/versions/2e6a6cc63e9_renaming_cookies_to_new_cookies.py
... done
```

With our migration created, let's edit the migration file and add the rename operation to the `upgrade` and `downgrade` methods as shown here:

```
def upgrade():
    op.rename_table('cookies', 'new_cookies')


def downgrade():
    op.rename_table('new_cookies', 'cookies')
```

Now, we're ready to run our migration with the `alembic upgrade` command:

```
# alembic upgrade head
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade 34044511331 -> 2e6a6cc63e9,
      Renaming cookies to new_cookies
```

This will rename our table in the database to `new_cookies`. Let's confirm that has happened with the `sqlite3` command:

```
± sqlite3 alembictest.db
SQLite version 3.8.5 2014-08-15 22:37:57
Enter ".help" for usage hints.
sqlite> .tables
alembic_version  new_cookies
```

As we expected, there is no longer a `cookies` table, as it has been replaced by `new_cookies`. In addition to `rename_table`, Alembic has many operations that you can use to help with your database changes, as shown in Table 12-3.

Table 12-3. Alembic operations

Operation	Used for
`add_column`	Adds a new column
`alter_column`	Changes a column type, server default, or name
`create_check_constraint`	Adds a new CheckConstraint
`create_foreign_key`	Adds a new ForeignKey
`create_index`	Adds a new Index
`create_primary_key`	Adds a new PrimaryKey
`create_table`	Adds a new table
`create_unique_constraint`	Adds a new UniqueConstraint
`drop_column`	Removes a column
`drop_constraint`	Removes a constraint
`drop_index`	Drops an index
`drop_table`	Drops a table
`execute`	Run a raw SQL statement
`rename_table`	Renames a table

While Alembic supports all the operations in Table 12-3, they are not supported on every backend database. For example, `alter_column` does not work on SQLite databases because SQLite doesn't support altering a column in any way. SQLite also doesn't support dropping a column.

In this chapter, we've seen how to autogenerate and handcraft migrations to make changes to our database in a repeatable manner. We also learned how to apply migrations with the `alembic upgrade` command. Remember, in addition to Alembic operations, we can use any valid Python code in the migration to help us accomplish our goals. In the next chapter, we'll explore how to perform downgrades and further control Alembic.

CHAPTER 13

Controlling Alembic

In the previous chapter, we learned how to create and apply migrations, and in this chapter we're going to discuss how to further control Alembic. We'll explore how to learn the current migration level of the database, how to downgrade from a migration, and how to mark the database at a certain migration level.

Determining a Database's Migration Level

Before performing migrations, you should double-check to make sure what migrations have been applied to the database. You can determine what the last migration applied to the database is by using the `alembic current` command. It will return the revision ID of the current migration, and tell you whether it is the latest migration (also known as the head). Let's run the `alembic current` command in the *CH12/* folder of the sample code for this book:

```
# alembic current
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
2e6a6cc63e9 (head) ❶
```

❶ The last migration applied to the database

While this output shows that we are on migration `2e6a6cc63e9`, it also tells us we are on the head or latest migration. This was the migration that changed the `cookies` table to `new_cookies`. We can confirm that with the `alembic history` command, which will show us a list of our migrations. Example 13-1 shows what the Alembic history looks like.

Example 13-1. Our migration history

```
# alembic history
34044511331 -> 2e6a6cc63e9 (head), Renaming cookies to new_cookies
8a8a9d067 -> 34044511331, Added Cookie model
<base> -> 8a8a9d067, Empty Init
```

The output from Example 13-1 shows us every step from our initial empty migration to our current migration that named the `cookies` table. I think we can all agree, `new_cookies` was a terrible name for a table, especially when we change cookies again and have `new_new_cookies`. To fix that and revert to the previous name, let's learn how to downgrade a migration.

Downgrading Migrations

To downgrade migrations, we need to choose the revision ID for the migration that we want to go back to. For example, if you wanted to return all the way back to the initial empty migration, you would choose revision ID `8a8a9d067`. We mostly want to undo the table rename, so for Example 13-2, let's revert to revision ID `34044511331` using the `alembic downgrade` command.

Example 13-2. Downgrading the rename cookies migration

```
# alembic downgrade 34044511331
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running downgrade 2e6a6cc63e9 -> 34044511331,
      Renaming cookies to new_cookies ❶
```

❶ Downgrade line

As we can see from the downgrade line in the output of Example 13-2, it has reverted the rename. We can check the tables with the same SQLite `.tables` command we used in Chapter 12. We can see the output of that command here:

```
# sqlite3 alembictest.db
SQLite version 3.8.5 2014-08-15 22:37:57
Enter ".help" for usage hints.
sqlite> .tables
alembic_version  cookies ❶
```

❶ Our table is back to being named `cookies`.

We can see the table has been returned to it's original name of `cookies`. Now let's use `alembic current` to see what migration level the database is at:

```
± alembic current
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
```

```
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
34044511331 ❶
```

❶ The revision ID for the add cookies model migration

We can see that we're back to the migration we specified in the `downgrade` command. We also need to update our application code to use the `cookies` table name as well, just as we did in Chapter 12, if we want to restore the application to a working state. You can also see that we are not at the latest migration, as `head` is not on that last line. This creates an interesting issue that we need to handle. If we no longer want to use the rename cookies to new_cookies migration again, we can just delete it from our *alembic/versions/* directory. If you leave it behind it will get run the next time `alembic upgrade head` is run, and this can cause disastrous results. We're going to leave it in place so that we can explore how to mark the database as being at a certain migration level.

Marking the Database Migration Level

When we want to do something like skip a migration or restore a database, it is possible that the database believes we are at a different migration than where we really are. We will want to explicitly mark the database as being a specific migration level to correct the issue. In Example 13-3, we are going to mark the database as being at revision ID `2e6a6cc63e9` with the `alembic stamp` command.

Example 13-3. Marking the database migration level

```
# alembic stamp 2e6a6cc63e9
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running stamp_revision 34044511331
       -> 2e6a6cc63e9
```

We can see from Example 13-3 that it stamped revision `34044511331` as the current database migration level. However, if we look at the database tables, we'll still see the `cookies` table is present. Stamping the database migration level does not actually run the migrations, it merely updates the Alembic table to reflect the migration level we supplied in the command. This effectively skips applying the `34044511331` migration.

If you skip a migration like this, it will only apply to the current database. If you point your Alembic migration environment to a different database by changing the `sqlalchemy.url`, and that new database is below the skipped migration's level or is blank, running `alembic upgrade head` will apply this migration.

Generating SQL

If you would like to change your production database's schema with SQL, Alembic supports that as well. This is common in environments with tighter change management controls, or for those who have massively distributed environments and need to run many different database servers. The process is the same as performing an "online" Alembic upgrade like we did in Chapter 12. We can specify both the starting and ending versions of the SQL generated. If you don't supply a starting migration, Alembic will build the SQL scripts for upgrading from an empty database. Example 13-4 shows how to generate the SQL for our rename migration.

Example 13-4. Generating rename migration SQL

```
# alembic upgrade 34044511331:2e6a6cc63e9 --sql ❶
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Generating static SQL
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade 34044511331 -> 2e6a6cc63e9,
      Renaming cookies to new_cookies
-- Running upgrade 34044511331 -> 2e6a6cc63e9

ALTER TABLE cookies RENAME TO new_cookies;

UPDATE alembic_version SET version_num='2e6a6cc63e9'
WHERE alembic_version.version_num = '34044511331';
```

❶ Upgrading from `34044511331` to `2e6a6cc63e9`.

The output from Example 13-4 shows the two SQL statements required to rename the `cookies` table, and update the Alembic migration level to the new level designated by revision ID `2e6a6cc63e9`. We can write this to a file by simple redirecting the standard output to the filename of our choice, as shown here:

```
# alembic upgrade 34044511331:2e6a6cc63e9 --sql > migration.sql
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Generating static SQL
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade 34044511331 -> 2e6a6cc63e9,
      Renaming cookies to new_cookies
```

After the command executes, we can `cat` the *migration.sql* file to see our SQL statements like so:

```
# cat migration.sql
-- Running upgrade 34044511331 -> 2e6a6cc63e9

ALTER TABLE cookies RENAME TO new_cookies;
```

```
UPDATE alembic_version SET version_num='2e6a6cc63e9'
WHERE alembic_version.version_num = '34044511331';
```

With our SQL statements prepared, we can now take this file and run it via whatever tool we like on our database servers.

If you develop on one database backend (such as SQLite) and deploy to a different database (such as PostgreSQL), make sure to change the `sqlalchemy.url` configuration setting that we set in Chapter 11 to use the connection string for a PostgreSQL database. If not, you could get SQL output that is not correct for your production database! To avoid issues such as this, I always develop on the database I use in production.

This wraps ups the basics of using Alembic. In this chapter, we learned how to see the current migration level, downgrade a migration, and create SQL scripts that we can apply without using Alembic directly on our production databases. If you want to learn more about Alembic, such as how to handle code branches, you can get additional details in the documentation.

The next chapter covers a collection of common things people need to do with SQLAlchemy, including how to use it with the Flask web framework.

CHAPTER 14

Cookbook

This chapter is different than previous ones in that each section focuses on a different aspect of using SQLAlchemy. The coverage here isn't as detailed as in previous chapters; consider this a quick rundown of useful tools. At the end of each section, there's information about where you can learn more if you're interested. These sections are not meant to be full tutorials, but rather short recipes on how to accomplish a particular task. The first part of this chapter focuses on a few advanced usages of SQLAlchemy, the second part on using SQLAlchemy with web frameworks like Flask, and the final section on additional libraries that can be used with SQLAlchemy.

Hybrid Attributes

Hybrid attributes are those that exhibit one behavior when accessed as a class method, and another behavior when accessed on an instance. Another way to think of this is that the attribute will generate valid SQL when it is used in a SQLAlchemy statement, and when accessed on an instance the hybrid attribute will execute the Python code directly against the instance. I find it easiest to understand this when looking at code. We're going to use our `Cookie` declarative class to illustrate this in Example 14-1.

Example 14-1. Our Cookie user data model

```
from datetime import datetime

from sqlalchemy import Column, Integer, Numeric, String, create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.ext.hybrid import hybrid_property, hybrid_method
from 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))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(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(self=self)


Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)
```

❶ Creates a hybrid property.

❷ Creates a hybrid method because we need an additional input to perform the comparison.

In Example 14-1, `inventory_value` is a hybrid property that performs a calculation with two attributes of our `Cookie` data class, and `bake_more` is a hybrid method that takes an additional input to determine if we should bake more cookies. (Should this ever return false? There's always room for more cookies!) With our data class defined, we can begin to examine what hybrid really means. Let's look at how the `inventory_value` works when used in a query (Example 14-2).

Example 14-2. Hybrid property: Class

```
print(Cookie.inventory_value < 10.00)
```

Example 14-2 will output the following:

```
cookies.unit_cost * cookies.quantity < :param_1
```

In the output from Example 14-2, we can see that the hybrid property was expanded into a valid SQL clause. This means that we can filter, order by, group by, and use database functions on these properties. Let's do the same thing with the `bake_more` hybrid method:

```
print(Cookie.bake_more(12))
```

This will output:

```
cookies.quantity < :quantity_1
```

Again, it builds a valid SQL clause out of the Python code in our hybrid method. In order to see what happens when we access it as an instance method, we'll need to add some data to database, which we'll do in Example 14-3.

Example 14-3. Adding some records to the database

```
session = Session()
cc_cookie = Cookie(cookie_name='chocolate chip',
                   cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
                   cookie_sku='CC01',
                   quantity=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',
             quantity=1,
             unit_cost=0.75)
mol = Cookie(cookie_name='molasses',
             cookie_recipe_url='http://some.aweso.me/cookie/recipe_molasses.html',
             cookie_sku='MOL01',
             quantity=1,
             unit_cost=0.80)
session.add(cc_cookie)
session.add(dcc)
session.add(mol)
session.flush()
```

With this data added, we're ready to look at what happens when we use our hybrid property and method on an instance. We can access the `inventory_value` property of the `dcc` instance as shown here:

```
dcc.inventory_value
0.75
```

You can see that when used on an instance, `inventory_value` executes the Python code specified in the property. This is exactly the magic of a hybrid property or

method. We can also run the `bake_more` hybrid method on the instance, as shown here:

```
dcc.bake_more(12)
True
```

Again, because of the hybrid method's behavior when accessed on an instance, `bake_more` executes the Python code as expected. Now that we understand how the hybrid properties and methods work, let's use them in some queries. We'll start by using `inventory_value` in Example 14-4.

Example 14-4. Using a hybrid property in a query

```
from sqlalchemy import desc

for cookie in session.query(Cookie).order_by(desc(Cookie.inventory_value)):
    print('{:>20} - {:.2f}'.format(cookie.cookie_name, cookie.inventory_value))
```

When we run Example 14-4, we get the following output:

```
        chocolate chip - 6.00
             molasses - 0.80
dark chocolate chip - 0.75
```

This behaves just as we would expect any ORM class attribute to work. In Example 14-5, we will use the `bake_more` method to decide what cookies we need to bake.

Example 14-5. Using a hybrid method in a query

```
for cookie in session.query(Cookie).filter(Cookie.bake_more(12)):
    print('{:>20} - {}'.format(cookie.cookie_name, cookie.quantity))
```

Example 14-5 outputs:

```
dark chocolate chip - 1
           molasses - 1
```

Again, this behaves just like we hoped it would. While this is an amazing set of behaviors from our code, hybrid attributes can do so much more, and you can read about it in the SQLAlchemy documentation on hybrid attributes.

Association Proxy

An association proxy is a pointer across a relationship to a specific attribute; it can be used to make it easier to access an attribute across a relationship in code. For example, this would come in handy if we wanted a list of ingredient names that are used to make our cookies. Let's walk through how that works with a typical relationship. The

relationship between cookies and ingredients will be a many-to-many relationship. Example 14-6 sets up our data models and their relationships.

Example 14-6. Setting up our models and relationships

```
from datetime import datetime
from sqlalchemy import (Column, Integer, Numeric, String, Table,
                        ForeignKey, create_engine)
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///:memory:')

Base = declarative_base()
Session = sessionmaker(bind=engine)



cookieingredients_table = Table('cookieingredients', Base.metadata, ❶
    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'

    ingredient_id = Column(Integer, primary_key=True)
    name = 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))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))

    ingredients = relationship("Ingredient",
                               secondary=cookieingredients_table) ❷

    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(self=self)


Base.metadata.create_all(engine)
```

❶ Creating the through table used for our many-to-many relationship.

❷ Creating the relationship to ingredients via the through table.

Because this is a many-to-many relationship, we need a through table to map the multiple relationships. We use the `cookieingredients_table` to do that. Next, we need to add a cookie and some ingredients, as shown here:

```
session = Session()
cc_cookie = Cookie(cookie_name='chocolate chip', ❶
                   cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
                   cookie_sku='CC01',
                   quantity=12,
                   unit_cost=0.50)

flour = Ingredient(name='Flour') ❷
sugar = Ingredient(name='Sugar')
egg = Ingredient(name='Egg')
cc = Ingredient(name='Chocolate Chips')
cc_cookie.ingredients.extend([flour, sugar, egg, cc]) ❸
session.add(cc_cookie)
session.flush()
```

❶ Creating the cookie.

❷ Creating the ingredients.

❸ Adding the ingredients to the relationship with the cookie.

So to add the ingredients to the cookie, we have to create them and then add them to the relationship on cookies, `ingredients`. Now, if we want to list the names of all the ingredients, which was our original goal, we have to iterate through all the ingredients and get the `name` attribute. We can accomplish this as follows:

```
[ingredient.name for ingredient in cc_cookie.ingredients]
```

This will return:

```
['Flour', 'Sugar', 'Egg', 'Chocolate Chips']
```

This can be cumbersome if all we really want from ingredients is the `name` attribute. Using a traditional relationship also requires us to manually create each ingredient

and add it to the cookie. It becomes more work if we need to determine if the ingredient already exists. This is where the association proxy can be useful in simplifying this type of usage.

To establish an association proxy that we can use for attribute access and ingredient creation, we need to do three things:

- Import the association proxy.
- Add an `__init__` method to the targeted object that makes it easy to create new instances with just the required values.
- Create an association proxy that targets the table name and column name you want to proxy.

We're going to start a fresh Python shell, and set up our classes again. However, this time we are going to add an association proxy to give us easier access to the ingredient names in Example 14-7.

Example 14-7. Association proxy setup

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///:memory:')

Session = sessionmaker(bind=engine)

from datetime import datetime

from sqlalchemy import Column, Integer, Numeric, String, Table, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.ext.associationproxy import association_proxy ❶


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'
```

```
    ingredient_id = Column(Integer, primary_key=True)
    name = 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))
    quantity = Column(Integer())
    unit_cost = Column(Numeric(12, 2))

    ingredients = relationship("Ingredient",
                               secondary=cookieingredients_table)

    ingredient_names = association_proxy('ingredients', 'name') ❸


    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(self=self)


Base.metadata.create_all(engine)
```

❶ Importing the association proxy.

❷ Defining an `__init__` method that only requires a name.

❸ Establishing an association proxy to the ingredients' `name` attribute that we can reference as `ingredient_names`.

With these three things done, we can now use our classes as we did previously:

```
    session = Session()
    cc_cookie = Cookie(cookie_name='chocolate chip',
                       cookie_recipe_url='http://some.aweso.me/cookie/recipe.html',
                       cookie_sku='CC01',
                       quantity=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',
                 quantity=1,
                 unit_cost=0.75)
    flour = Ingredient(name='Flour')
    sugar = Ingredient(name='Sugar')
    egg = Ingredient(name='Egg')
    cc = Ingredient(name='Chocolate Chips')
    cc_cookie.ingredients.extend([flour, sugar, egg, cc])
    session.add(cc_cookie)
    session.add(dcc)
    session.flush()
```

So the relationship still works like we expect; however, to get the ingredient names we can use the association proxy to avoid the list comprehension we used before. Here is an example of using `ingrendient_names`:

```
    cc_cookie.ingredient_names
```

This will output:

```
    ['Flour', 'Sugar', 'Egg', 'Chocolate Chips']
```

This is much easier than looping though all the ingredients to create the list of ingredient names. However, we forgot to add one key ingredient, "Oil". We can add this new ingredient using the association proxy as shown here:

```
    cc_cookie.ingredient_names.append('Oil')
    session.flush()
```

When we do this, the association proxy creates a new ingredient using the `Ingredient.__init__` method for us automatically. It's important to note that if we already had an ingredient named Oil, the association proxy would still attempt to create it for us. That could lead to duplicate records or cause an exception if the addition violates a constraint.

To work around this, we can query for existing ingredients prior to using the association proxy. Example 14-8 shows how we can accomplish that.

Example 14-8. Filtering ingredients

```
dcc_ingredient_list = ['Flour', 'Sugar', 'Egg', 'Dark Chocolate Chips',
                       'Oil'] ❶
existing_ingredients = session.query(Ingredient).filter(
    Ingredient.name.in_(dcc_ingredient_list)).all() ❷
missing = set(dcc_ingredient_list) - set([x.name for x in
                                          existing_ingredients]) ❸
```

❶ Defining the list of ingredients for our dark chocolate chip cookies.

❷ Querying to find the ingredients that already exist in our database.

❸ Finding the missing packages by using the difference of the two sets of ingredients.

After finding the ingredients we already had, we then compared it with our required ingredient list to determine what was missing, which is "Dark Chocolate Chips." Now we can add all the existing ingredients via the relationship, and then the new ingredients via the association proxy as shown in Example 14-9.

Example 14-9. Adding ingredients to dark chocolate chip cookie

```
dcc.ingredients.extend(existing_ingredients) ❶
dcc.ingredient_names.extend(missing) ❷
```

❶ Adding the existing ingredients via the relationship.

❷ Adding the new ingredients via the association proxy.

Now we can print a list of the ingredient names, as shown here:

```
dcc.ingredient_names
```

And we will get the following output:

```
['Egg', 'Flour', 'Oil', 'Sugar', 'Dark Chocolate Chips']
```

This enabled us to quickly handle existing and new ingredients when we added them to our cookie, and the resulting output was just what we desired. Association proxies have lots of other uses, too; you can learn more in the association proxy documentation.

Integrating SQLAlchemy with Flask

It's common to see SQLAlchemy used with a Flask web application. The creator of Flask has also created a Flask-SQLalchemy package to make this integration easy. Using Flask-SQLalchemy will provide preconfigured scoped sessions that are tied to the page life cycle of your Flask application. You can install Flask-SQLalchemy with pip as shown here:

```
# pip install flask-sqlalchemy
```

When using Flask-SQLalchemy, I highly recommend you use the app factory pattern, which is not what is shown in the quick start section of the Flask-SQLalchemy documentation. The app factory pattern uses a function that assembles an application with all the appropriate add-ons and configuration. This is typically placed in your appli-

cation's *app/__init__.py* file. Example 14-10 contains an example of how to structure your `create_app` method.

Example 14-10. App factory function

```
from flask import Flask
from flask.ext.sqlalchemy import SQLAlchemy

from config import config

db = SQLAlchemy() ❶


def create_app(config_name): ❷
    app = Flask(__name__)
    app.config.from_object(config[config_name])

    db.init_app(app) ❸
    return app
```

❶ Creates an unconfigured Flask-SQLalchemy instance.

❷ Defines the `create_app` app factory.

❸ Initializes the instance with the `app` context.

The app factory needs a configuration that defines the Flask settings and the SQLAlchemy connection string. In Example 14-11, we define a configuration file, *config.py*, in the root of the application.

Example 14-11. Flask app configuration

```
import os


basedir = os.path.abspath(os.path.dirname(__file__))


class Config: ❶
    SECRET_KEY = 'development key'
    ADMINS = frozenset(['jason@jasonamyers.com', ])


class DevelopmentConfig(Config): ❷
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = "sqlite:////tmp/dev.db"


class ProductionConfig(Config): ❸
```

```
    SECRET_KEY = 'Prod key'
    SQLALCHEMY_DATABASE_URI = "sqlite:////tmp/prod.db"


config = { ❹
    'development': DevelopmentConfig,
    'production': ProductionConfig,
    'default': DevelopmentConfig
}
```

❶ Defines the base Flask config.

❷ Defines the configuration used when developing.

❸ Defines the configuration used when in production.

❹ Creates a dictionary that is used to map from a simple name to the configuration class.

With the app factory and configuration ready, we have a functioning Flask application. We're now ready to define our data classes. We'll define our `Cookie` model in *apps/models.py*, as demonstrated in Example 14-12.

Example 14-12. Defining the Cookie model

```
from app import db


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))
    quantity = db.Column(db.Integer())
```

In Example 14-12, we used the `db` instance we created in *app/__init__.py* to access most parts of the SQLAlchemy library. Now we can use our cookie models in queries just like we did in the ORM section of the book. The only change is that our sessions are nested in the `db.session` object.

One other thing that Flask-SQLalchemy adds that is not found in normal SQLAlchemy code is the addition of a `query` method to every ORM data class. I highly encourage you to *not* use this syntax, because it can be confusing when mixed and matched with other SQLAlchemy queries. This style of query looks like the following:

```
    Cookie.query.all()
```

This should give you a taste of how to integrate SQLAlchemy with Flask; you can learn more in the Flask-SQLalchemy documentation. Miguel Grinberg also wrote a fantastic book on building out a whole application titled Flask Web Development.

SQLAcodegen

We learned about reflection with SQLAlchemy Core in Chapter 5 and automap for use with the ORM in Chapter 10. While both of those solutions are great, they require you to perform the reflection every time the application is restarted, or in the case of dynamic modules, every time the module is reloaded. SQLAcodegen uses reflection to build a collection of ORM data classes that you can use in your application code base to avoid reflecting the database multiple times. SQLAcodegen has the ability to detect many-to-one, one-to-one, and many-to-many relationships. It can be installed via pip:

```
pip install sqlacodegen
```

To run SQLAcodegen, we need to specify a database connection string for it to connect to. We'll use a copy of the Chinook database that we have worked with previously (available in the *CH15/* folder of the example code from this book). While inside the *CH15/* folder, Run the SQLAcodegen command with the appropriate connection string, as shown in Example 14-13.

Example 14-13. Running SQLAcodegen on the Chinook database

```
# sqlacodegen sqlite:///Chinook_Sqlite.sqlite ❶

# coding: utf-8
from sqlalchemy import (Table, Column, Integer, Numeric, Unicode, DateTime,
                        ForeignKey)
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base


Base = declarative_base()
metadata = Base.metadata


class Album(Base):
    __tablename__ = 'Album'

    AlbumId = Column(Integer, primary_key=True, unique=True)
    Title = Column(Unicode(160), nullable=False)
    ArtistId = Column(ForeignKey(u'Artist.ArtistId'), nullable=False, index=True)

    Artist = relationship(u'Artist')
```

```
class Artist(Base):
    __tablename__ = 'Artist'

    ArtistId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(120))


class Customer(Base):
    __tablename__ = 'Customer'

    CustomerId = Column(Integer, primary_key=True, unique=True)
    FirstName = Column(Unicode(40), nullable=False)
    LastName = Column(Unicode(20), nullable=False)
    Company = Column(Unicode(80))
    Address = Column(Unicode(70))
    City = Column(Unicode(40))
    State = Column(Unicode(40))
    Country = Column(Unicode(40))
    PostalCode = Column(Unicode(10))
    Phone = Column(Unicode(24))
    Fax = Column(Unicode(24))
    Email = Column(Unicode(60), nullable=False)
    SupportRepId = Column(ForeignKey(u'Employee.EmployeeId'), index=True)

    Employee = relationship(u'Employee')

class Employee(Base):
    __tablename__ = 'Employee'

    EmployeeId = Column(Integer, primary_key=True, unique=True)
    LastName = Column(Unicode(20), nullable=False)
    FirstName = Column(Unicode(20), nullable=False)
    Title = Column(Unicode(30))
    ReportsTo = Column(ForeignKey(u'Employee.EmployeeId'), index=True)
    BirthDate = Column(DateTime)
    HireDate = Column(DateTime)
    Address = Column(Unicode(70))
    City = Column(Unicode(40))
    State = Column(Unicode(40))
    Country = Column(Unicode(40))
    PostalCode = Column(Unicode(10))
    Phone = Column(Unicode(24))
    Fax = Column(Unicode(24))
    Email = Column(Unicode(60))

    parent = relationship(u'Employee', remote_side=[EmployeeId])


class Genre(Base):
    __tablename__ = 'Genre'

    GenreId = Column(Integer, primary_key=True, unique=True)
```

```
    Name = Column(Unicode(120))


class Invoice(Base):
    __tablename__ = 'Invoice'

    InvoiceId = Column(Integer, primary_key=True, unique=True)
    CustomerId = Column(ForeignKey(u'Customer.CustomerId'), nullable=False,
                        index=True)
    InvoiceDate = Column(DateTime, nullable=False)
    BillingAddress = Column(Unicode(70))
    BillingCity = Column(Unicode(40))
    BillingState = Column(Unicode(40))
    BillingCountry = Column(Unicode(40))
    BillingPostalCode = Column(Unicode(10))
    Total = Column(Numeric(10, 2), nullable=False)

    Customer = relationship(u'Customer')


class InvoiceLine(Base):
    __tablename__ = 'InvoiceLine'

    InvoiceLineId = Column(Integer, primary_key=True, unique=True)
    InvoiceId = Column(ForeignKey(u'Invoice.InvoiceId'), nullable=False, index=True)
    TrackId = Column(ForeignKey(u'Track.TrackId'), nullable=False, index=True)
    UnitPrice = Column(Numeric(10, 2), nullable=False)
    Quantity = Column(Integer, nullable=False)

    Invoice = relationship(u'Invoice')
    Track = relationship(u'Track')


class MediaType(Base):
    __tablename__ = 'MediaType'

    MediaTypeId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(120))


class Playlist(Base):
    __tablename__ = 'Playlist'

    PlaylistId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(120))

    Track = relationship(u'Track', secondary='PlaylistTrack')


t_PlaylistTrack = Table(
    'PlaylistTrack', metadata,
    Column('PlaylistId', ForeignKey(u'Playlist.PlaylistId'), primary_key=True,
```

```
            nullable=False),
    Column('TrackId', ForeignKey(u'Track.TrackId'), primary_key=True, nullable=False,
            index=True),
    Index('IPK_PlaylistTrack', 'PlaylistId', 'TrackId', unique=True)
)


class Track(Base):
    __tablename__ = 'Track'

    TrackId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(200), nullable=False)
    AlbumId = Column(ForeignKey(u'Album.AlbumId'), index=True)
    MediaTypeId = Column(ForeignKey(u'MediaType.MediaTypeId'), nullable=False,
                         index=True)
    GenreId = Column(ForeignKey(u'Genre.GenreId'), index=True)
    Composer = Column(Unicode(220))
    Milliseconds = Column(Integer, nullable=False)
    Bytes = Column(Integer)
    UnitPrice = Column(Numeric(10, 2), nullable=False)

    Album = relationship(u'Album')
    Genre = relationship(u'Genre')
    MediaType = relationship(u'MediaType')
```

❶ Runs SQLAcodegen against the local Chinook SQLite database

As you can see in Example 14-13, when we run the command, it builds up a complete file that contains all the ORM data classes of the database along with the proper imports. This file is ready for use in our application. You might need to tweak the settings for the `Base` object if it was established elsewhere. It is also possible to only generate the code for a few tables by using the `--tables` argument. In Example 14-14, we are going to specify the `Artist` and `Track` tables.

Example 14-14. Running SQLAcodegen on the Artist and Track tables

```
# sqlacodegen sqlite:///Chinook_Sqlite.sqlite --tables Artist,Track ❶

# coding: utf-8
from sqlalchemy import Column, ForeignKey, Integer, Numeric, Unicode
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base


Base = declarative_base()
metadata = Base.metadata


class Album(Base):
    __tablename__ = 'Album'
```

```
    AlbumId = Column(Integer, primary_key=True, unique=True)
    Title = Column(Unicode(160), nullable=False)
    ArtistId = Column(ForeignKey(u'Artist.ArtistId'), nullable=False, index=True)

    Artist = relationship(u'Artist')


class Artist(Base):
    __tablename__ = 'Artist'

    ArtistId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(120))


class Genre(Base):
    __tablename__ = 'Genre'

    GenreId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(120))


class MediaType(Base):
    __tablename__ = 'MediaType'

    MediaTypeId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(120))


class Track(Base):
    __tablename__ = 'Track'

    TrackId = Column(Integer, primary_key=True, unique=True)
    Name = Column(Unicode(200), nullable=False)
    AlbumId = Column(ForeignKey(u'Album.AlbumId'), index=True)
    MediaTypeId = Column(ForeignKey(u'MediaType.MediaTypeId'), nullable=False,
                         index=True)
    GenreId = Column(ForeignKey(u'Genre.GenreId'), index=True)
    Composer = Column(Unicode(220))
    Milliseconds = Column(Integer, nullable=False)
    Bytes = Column(Integer)
    UnitPrice = Column(Numeric(10, 2), nullable=False)

    Album = relationship(u'Album')
    Genre = relationship(u'Genre')
    MediaType = relationship(u'MediaType')
```

❶ Runs SQLAcodegen against the local Chinook SQLite database, but only for the `Artist` and `Track` tables.

When we specified the `Artist` and `Track` tables in Example 14-14, SQLAcodegen built classes for those tables, and all the tables that had a relationship with those tables. SQLAcodegen does this to ensure that the code it builds is ready to be used. If you want to save the generated classes directly to a file, you can do so with standard redirection, as shown here:

```
# sqlacodegen sqlite:///Chinook_Sqlite.sqlite --tables Artist,Track > db.py
```

If you want more information on what you can do with SQLAcodegen, you can find it by running `sqlacodegen --help` to see the additional options. There is not any official documentation at the time of this writing.

That concludes the cookbook section of the book. I hope you enjoyed these few samples of how to do more with SQLAlchemy.

CHAPTER 15

Where to Go from Here

I hope you've enjoyed learning how to use SQLAlchemy Core and ORM along with the Alembic database migration tool. Remember that there is also extensive documentation on the SQLAlchemy website, along with a collection of presentations from various conferences where you can learn about specific topics in more detail. There are also several additional talks on SQLAlchemy on the PyVideo website, including a few by yours truly.

Where to go next greatly depends on what you are trying to accomplish with SQLAlchemy:

- If you want to know more about Flask and SQLAlchemy, make sure to read *Flask Web Development* by Miguel Grinberg.
- If you are interested in learning more about testing or how to use pytest, Alex Grönholm has several excellent blog posts on effective testing strategies: Part 1 and Part 2.
- If you want to know more about SQLAlchemy plug-ins and extensions, check out Awesome SQLAlchemy by Hong Minhee. This resource has a great list of SQLAlchemy-related technologies.
- If you are interested in learning more about the internals of SQLAlchemy, this has been covered by Mike Bayer in *The Architecture of Open Source Applications*.

Hopefully the information you've learned in this book will give you a solid foundation upon which to improve your skills. I wish you well in your future endeavors!

Index

F

G

H

I

J

K

L

S

T

U

V

W

Z

About the Authors

Jason Myers works at Cisco as a Software Engineer working on OpenStack. Prior to switching to development a few years ago, he spent several years as a systems architect, building data centers and cloud architectures for several of the largest tech companies, hospitals, stadiums, and telecom providers. He's a passionate developer who regularly speaks at local and national events about technology. He's also the chair of the PyTennessee conference. He loves solving health-related problems, and has a side project, Sucratrend, devoted to helping diabetics manage their condition and improve their quality of life. He has used SQLAlchemy in web, data warehouse, and analytics applications.

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, and has been honored as a MongoDB Master by MongoDB Inc. for his contributions to the community. He is a frequent speaker at various user groups and conferences, and an active member of the Atlanta startup community.

Colophon

The animal on the cover of *Essential SQLAlchemy* is a largescale flying fish (*Cypselurus oligolepis*). Flying fish is the more common name for members of the *Exocoetidae* family, which comprises roughly 40 species that inhabit the warm tropical and subtropical waters of the Atlantic, Pacific, and Indian oceans. Flying fish range from 7 to 12 inches in length and are characterized by their unusually large, winglike pectoral fins. Some species also have enlarged pelvic fins and are thus known as four-winged flying fish.

As their name suggests, flying fish have the unique ability to leap from the water and glide through the air for distances of up to a quarter of a mile. Their torpedo-like bodies help them gather the speed necessary to propel themselves from the ocean (about 37 miles per hour), and their distinctive pectoral fins and forked tailfins keep them airborne. Biologists believe this remarkable trait may have evolved as a way for flying fish to escape their many predators, which include tuna, mackerel, swordfish, marlin, and other larger fish. However, flying fish sometimes have a more difficult time evading their human predators. Attracted by a luring light that fishermen attach to their canoes at night, the fish leap in and are unable to vault themselves back out.

Dried flying fish are a dietary staple for the Tao people of Orchid Island, located off the coast of Taiwan, and flying fish roe is common in Japanese cuisine. They are also a coveted delicacy in Barbados, known as "Land of the Flying Fish" before shipping pollution and overfishing depleted their numbers. The flying fish retains a prominent

cultural status there, however; it's the main ingredient in the national dish (cou-cou and flying fish) and it is featured on coins, artwork, and even in the Barbados Tourism Authority's logo.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *Dover's Animals*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.