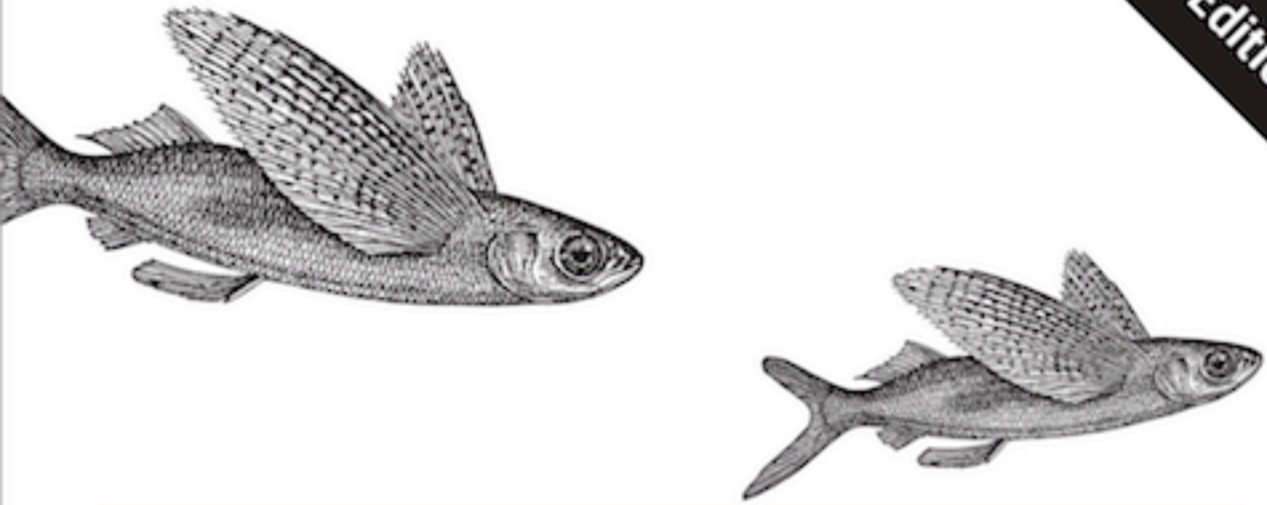
O'REILLY®

2nd Edition



Essential SQLAlchemy

MAPPING PYTHON TO DATABASES

Rick Copeland

SECOND EDITION

Essential SQLAlchemy

Jason Myers and Rick Copeland

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Essential SQLAlchemy, Second Edition

by Jason Myers and Rick Copeland

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: FIX ME!
Copyeditor: FIX ME!
Proofreader: FIX ME!
Indexer: FIX ME!
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

December 2015: Second Edition

Revision History for the Second Edition:

2015-07-07: Early release revision 1

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

ISBN: 978-1-491-91646-9

[?]

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. It doesn't matter if you are developing for the web, the desktop or other applications, they all 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 it 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 onto 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 focus the 1.0 release of SQLAlchemy; however, much of what 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 majors sections: SQLAlchemy Core, SQLAlchemy ORM, and a Cookbook section. The first two sections are meant to mirror each other as closely as possible. We have taken care to perform the same examples in each section 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 sections 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, I would 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 REPL (read, eval, print loop). You can use the builtin 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 to Come], 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 through out the book. The second section 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, file names, 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 *https://github.com/oreillymedia/title_title*.

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: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 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

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 *http://www.oreilly.com/catalog/0636920035800*.

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 Gronholm for their critical pre-publication feedback, without whom 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 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 wanna 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.

Most importantly I wanna 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.

CHAPTER 1

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 seperately or together depends 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 is 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 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 different between the way classes can be related, and how the underlying database relationships work. We'll explore more of how this can affect you in [Chapter to Come].

Choosing between SQLAlchemy Core and ORM

Before you begin building applications with SQLAlchemy, you will need to decide of you are going to primarly use the ORM or Core. The decision between choosing to using SQLAlchemy Core vs ORM as the dominate 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 points where being able to tightly control the query or operating on unmodeled data is useful. Having the strong database connection pool and ResultSet optimizations are perfectly suited to dealing with large amounts of data even in multiple databases.

However, if you wanted 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. Micro services also benefit from it's abstraction from the underlying database allowing the developer to focus on just the process being implemented.

However, since 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.

- If you are working with a framework that already has an ORM built in, but want to add more powerful reporting, use Core.
- If not do you want to take a view of your data in a more schema-centric view like used in SQL, use Core.
- Do you have data for which business objects are not needed, use Core.
- Do you view your data as business objects, use ORM.
- Are you building a quick prototype, use ORM.
- Do 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, and it can be done with a `pip install sqlalchemy`. It's worth noting that it can also be installed with easy_install and distutils as well; however, pip is the more straight forward 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 lack of a compiler on the system you are installing you, 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 driver; however, an additional database driver that uses the standard python DBAPI (PEP-249) specification is needed to connect to others 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 instructions below focus on the most common for PostgreSQL and MySQL.

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 8: Custom Dialects 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 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 if 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 1-1 contains a line that 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 that connection string would look like `engine4` and that the `\\` are required for proper string escaping unless you use a raw string (`r''`).

Example 1-1. 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')
engine3 = 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.

So lets 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 for us. In Example 1-2, you'll notice that I tend to use postgresql+psycopg2 as the engine and dialect components of the connection string when just postgres will do. This is because I prefer to be explicit instead of implicit as mentioned in the (Zen of Python).

Example 1-2. Creating an engine for a local PostgreSQL Database

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

Now lets look at a MySQL database on a remote server. You'll notice, in Example 1-3, after the connection string we have a keyword parameter, `pool_recycle` to define how often to recycle the connections.

Example 1-3. 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 8 hours. To work around this issue, use `pool_recycle=3600` when creating an engine as shown above.

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 is not define the encoding type used by the backend database itself.

`isolation_level`: This instructs SQLAlchemy to use a specific isolation level. For example with 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 and is the same as doing it via a key-value pair in the connection string for those that support that such as psycopg2.

`pool_recycle`: This recycles or times out the database connections every so many seconds. This is important for MySQL due to the connection timeouts we mentioned in the MySQL section above. 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 the next section, 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. Due to the nature of the way SQLAlchemy Core works it can be used with the Django or the SQLAlchemy ORM in addition to its usage as a standalone solution.

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

CHAPTER 2

Schema and Types

In order to provide access to the underlying database, SQLAlchemy has to have 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 will focus on the first of these as that one and the third one are used with SQLAlchemy Core, and we'll cover the other two 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 with 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 as well. 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 details is hidden from you, and you can trust that the 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 the table below.

Table 2-1. 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 above, 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 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 that SQL standard types are; however they are only available in specific backend databases. You can determine was 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 it off when retrieving that field from the record. This can be useful when working with legacy data still used by existing systems that preform 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 through out the rest of the examples in this chapter to hold our information catalog.

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

Once we have a way 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, metadata, and 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. Through out the rest of this chapter, we are going to be building up a set of tables that we'll use through out the entire SQLAlchemy Core section. In Example 2-1 below, we create a table that could be used as a way to store the cookie inventory for our online cookie delivery service.

Example 2-1. 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 who takes multiple arguments length and precision, such as 11.20

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 type 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. Not all database backends, include MySQL, support this behavior.

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 indicated when a record was created or updated for logging or auditing purposes. Let's take a look at these keyword arguments in action in the example below.

Example 2-2. 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 to have 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 the 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 the *Keys and Constraints* and *Index* Sections are included as part of the Table constructor or added to the table via special methods. They will 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 to pieces of related data in different tables. As you can see in examples 1 and 2 above, you can make a column a primary key just 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 below. 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 2 above, it would look like the following.

```
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. In our cookie shop, it would be confusing to have two users that used the same username to log into our systems. We can also assign unique constraints on columns as shown above on the username column or we can define them manually as shown below.

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

Not shown in example 2 above 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 example below, we are ensuring that unit_cost is never allowed to be less than 0.00 because every cookie cost something to make. Remember from high school economics, TINSTAFC. (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

Index are used to accelerate lookups for field values, and in example 1 above, 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, 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 index for situations where you need to 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 the one below to optimize that lookup.

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

Now it is time to dive the most important part of relational databases, which is 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 relationships between tables are made. 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 the diagram below.

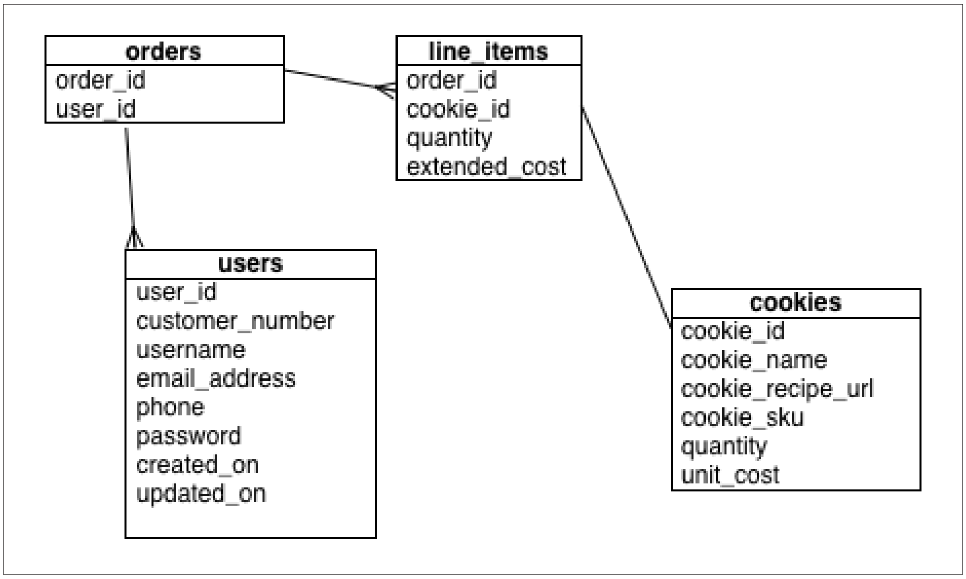


Figure 2-1. Chapter 3 Relationship Visualization

One way to implement a relationship is shown below in Example 3 in the line_items table on the order_id column, this will result in ForeignKeyConstraint to define the relationship between the two tables. In this case, we can have many line_items 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. So 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 this is an association table.

Example 2-3. 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 order in which our tables are loaded. This is because it will only perform the resolution of that string to a table name and column the first time it is accessed. If we used hard references, such as `cookies.c.cookie_id`, in our ForeignKey definitions it would perform that resolution during module initialization and could fail depending on the order in which the tables where loaded.

You can define a ForeignKeyConstraint explicitly as well, which can be useful if trying to match an existing database schema so it can be used with SQLAlchemy. This is just like the way we talked about create keys, constraints, and indexes above to matching name schemes etc. You will need to import the `ForeignKeyConstraint` from the `sqlalchemy` module prior to defining one in your table definition. The code below shows how to create the ForeignKeyConstraint for the order_id field between the line_items and orders table.

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

Everything prior to now has shown you how to define 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 being writing queries. However, if you need to create the full schema or added a table, we'll want to know how to persist them in the database for permanent storage.

Persisting the Tables

All of our tables and additional schema definitions are associated with a 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 the create_all will not attempt to recreate tables that already exist in the database, and 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 hand code changes directly in your application code. We'll explore this more in a later chapter. Well now that we have persisted the tables in the database, let's see the full example of the work we've done in chapter 2.

Example 2-4. Full In Memory SQLite Example

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

CHAPTER 3

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 to use the tables we created in Chapter 3 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. In Example 1 below are going to do just that.

Example 3-1. 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)) ❶
```

❶ This 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 the SQL statement above, 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 mitigate 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 which 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 ❶
```

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

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 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 3-2. Executing the Insert Statement

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

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

```
result.inserted_primary_key
```

```
[1]
```

Let's take a quick aside here and talk about what happens when we call the `execute()` method at a high level. 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 the table may not be initially known. For example, maybe we run two divisions of the company that have separate inventory tables. Using the insert function as shown below in Example 3-3 would allow us to use one statement and just swap the tables.

Example 3-3. 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 both methods of table object and the more generative standalone function, I prefer the generative approach because it more closely mirrors the SQL statements most people are accustom 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 3-4. 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 the knowledge to insert two types of cookies into the cookies table.

Example 3-5. 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. As it will compile the statement against the first dictionary in the list and the statement will fail if subsequent dictionaries are different since 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 that is very analogous to the standard SQL SELECT statement. Initially, let's select all the records in our cookies table.

Example 3-6. Simple Select function

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

❶ Remember we can 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 above 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 select all the columns on the table. It is also possible to use the `select()` method on the table object to do this as shown below in Example 3-7. Again, I prefer seeing it written more like Example 3-6 above.

Example 3-7. 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 DB-API cursor object, and it's main goal is to make dealing with the results of a statement easier to use manipulate. For example, it makes handling query results easier by allowing access using an index, name, or Column object. Example 3-8 will demonstrate all three of these methods. It's very important to become comfortable using each of these ways to get to the desired columns data.

Example 3-8. Handling Rows with a result proxy

```
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 3-7

❷ Access column by index

❸ Access column by column name

❹ Access column by Column object

These all result in `u'chocolate chip'` and are all 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 3-9. 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

Returns:

```
chocolate chip
dark chocolate chip
peanut butter
oatmeal raisin
```

In addition to using the ResultProxy as an interable or calling the `fetchall()` method, many other ways of accessing data via the ResultProxy are available. In fact, all the `result` variables in that section on inserting data where actually ResultProxys. Both the `rowcount()` and `inserted_primary_key()` methods we used in that section are just a few 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 close 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 through-out the remainder of this chapter.

In Production Code

When I am writing production code, I use the following 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 query ever returns more than one row with one column, which often gets missed during testing.

Every time we queried the database in the examples above, 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 onto the queries or ResultProxys; however, accounting the data you get back from a query is often the first place to look if a query is consuming to 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 `select()` method constructor as a list. For example, I might only want to get the name and quantity of cookies in a query as shown in Example 3-10.

Example 3-10. 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, we're going to look at other things we can do to alter how the results are returned in a select statement. We're going to start with changing the order in which the results are returned.

Ordering

If you were to look at all the results from Example 10 instead of the just first record, you would see that the data is not really in any particular order. However, if we want get the list back in order by the quantity we have on hand, we can chain an `order_by()` statement onto our select as shown in Example 3-11.

Example 3-11. Order by Quanity Asscending

```
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))
```

Results in:

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

I saved the select statement into the `s` variable, and then used that `s` variable and added the `order_by` statement onto it 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 all 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 would exceed Python's 79 character per line limit expressed in PEP8. By using the generative type statement, it gets me in under that limit. We'll see a few examples through out the book were this generative style can introduce a more useful 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 I wanted to get these sorted in reverse or descending order, I would need to the use the `desc()` statement. The `desc()` function wraps the specific column I want to sort in a descending manner as shown in Example 3-12.

Example 3-12. 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()` can also be used as a method on a column object, such as `cookies.c.quantity.desc()`. However, I find that a bit more confusing to read in long statements, and 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 actually 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, I only have time to make two batches of cookies today, and I want to know which two cookies I should make. I can use our ordered query from earlier and add a limit statement to get just two the two types of cookies I should make.

Example 3-13. Two fewest 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 I am using the interable capabilities of the ResultsProxy in a list comprehension.

Results in:

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

Now that I know what kind of cookies I need to be baking, I'm starting to get curious about how many cookies I have in my inventory now. 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.

Builtin 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 function we need to import the `sqlalchemy.sql.func` module where they are found. These functions are wrapped around the column or columns on which they are operating. So to get a total count of cookies, I would do something like Example 3-14.

Example 3-14. 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 use of scalar, which will return only the left most column in the first record

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 3-15. 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 result proxy

❷ The column name is auto generated and is commonly <func_name>_<position>.

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 I like to be in my naming, especially when surrounded with other Python code. Thankfully, SQLAlchemy provides a way to fix this via the `label()` function. Example 3-16, performs the same query as Example 3-15; however, it uses label to give us a more useful name to access that column.

Example 3-16. 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 I just use the `label()` function on the column object I want to change

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 3-17, we'll find a cookie named chocolate chip.

Example 3-17. 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 that will give me a list of columns and values.

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 3-18. 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)
```

Results in:

```
chocolate chip
dark chocolate chip
```

In the `.where()` statement of Example 3-18 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 3-18, we are taking advantage of the `like()` method that is available on ClauseElements. There are many other methods available, which are listed in Table 3-1 below. Each of these methods are analogous to a standard SQL statement construct. You'll find various examples of these used throughout the book.

Table 3-1. 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 column

Method	Purpose
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* (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 a one of the above methods, then we will have an operator in our where clauses. Most of the operators work as you might expect; however, we want to talk about 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 a `IS NULL` statement. Arithmetic operators (\+, `-`, `*`, `/`, and `%`) are also supported with additional capabilities for database independent string concatenation as shown in Example 3-19.

Example 3-19. String concatenation with `\+`

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

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 thing to usage of operators is to compute values from multiple columns. You'll often do this is applications and reports dealing with financial and statistic data. In Example 3-20, we are looking at a common inventory value calculation.

Example 3-20. 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 like 6.0000000000. So by casting it, we can make it look like currency. It is also possible to accomplish the same task in python as well 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 rename the column would be named `anon_1` since the operation doesn't result in a name.

Results in:

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

Boolean Operators

SQLAlchemy also provides for use of 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 affect. 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 do the following.

Example 3-21. 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 find cookies that we have between 10 and 50 of in inventory or that the name contains *chip*, we could do the following:

Example 3-22. 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)
```

Results in:

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

The `not_()` function works in a similar fashion to other conjunctions, and it simple is used to select records where a record do 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 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. I finally finished baking those chocolate chip cookies that we needed for our inventory. In Example 3-23, we are going to add them to our existing inventory with an update query, and then check to see how many we have currently in inventory.

Example 3-23. 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 where updated

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.

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).

Example 3-24. 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))
```

Returns:

```
1
0
```

Okay, let's take a break from all the learning, and load up some data using what we already learned for the users, orders, and line_items tables. You can copy my code below; however, consider taking this 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 SQLAlchemy Core, we learned how to define ForeignKeys and relationships; however, we've not used them yet 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, it's useful to know how many of what kind of cookies are ordered by the `cookiemon` user in order to fulfill its order. 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 do 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 3-25. 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

Results in:

```
    (u'wlk001', u'cookiemon', u'111-111-1111', u'chocolate chip', 2, Decimal('1.00'))
    (u'wlk001', u'cookiemon', u'111-111-1111', u'peanut butter', 12, Decimal('3.00'))
```

The SQL looks like:

```
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 user for all users, not just those with current orders. In order to do this, we have to use the `outerjoin()` method, and it requires a bit more care in the ordering of the join since the table we use the `outerjoin()` method on will be the one from which all results are returned.

Example 3-26. 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.

Results in:

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

Up to now, 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? In order 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 3-27. 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 non aggregated included column

Results in:

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

We've shown the generative building of statements through-out the previous examples, but I want to focus on that specifically for a moment.

Chaining

We've used chaining several times through out 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 3-28 below.

Example 3-28. 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')
```

Results in:

```
[(u'ol001', u'cakeeater', u'222-222-2222', u'chocolate chip', 24, Decimal('12.00')),
 (u'ol001', 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 function 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 3-29. 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_order_by_customer('cakeeater', shipped=True) ❸

get_orders_by_customer('cakeeater', shipped=False) ❹

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

❶ All Orders

❷ All Orders with details

❸ Shipped Orders Only

❹ Orders that haven't shipped yet

❺ Orders that haven't shipped yet with details

Results in:

```
    [(u'ol001', u'cakeeater', u'222-222-2222')]

    [(u'ol001', u'cakeeater', u'222-222-2222', u'chocolate chip', 24, Decimal('12.00')),
     (u'ol001', u'cakeeater', u'222-222-2222', u'oatmeal raisin', 6, Decimal('6.00'))]

    []

    [(u'ol001', u'cakeeater', u'222-222-2222')]

    [(u'ol001', u'cakeeater', u'222-222-2222', u'chocolate chip', 24, Decimal('12.00')),
     (u'ol001', 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.

Raw Queries

It is also possible to send execute raw SQL statements or use raw SQL in part of a SQLAlchemy Core query. It still returns a result proxy, and you can continue to interact with it just as you would a query build 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 3-30. Full Raw Queries

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

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. Here is an example of a raw SQL where clause using the `text()` function.

Example 3-31. Partial Text Query

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

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, update, read, and delete operations. This 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 4

Exceptions and Transactions

In the last 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 are going to purposely perform some actions incorrectly so that we can see the types of errors that occur and how we should respond to them. We're going to conclude the chapter with 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; however, I want to focus on the ones that are the most common: AttributeErrors and IntegrityErrors. By learning how to handle these common exceptions, you'll be able to understand how 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 SQLAlchemy Core into your shell. Example 4-1 contains those tables and connection again for reference.

Example 4-1. 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, and 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 a attribute that doesn't exist. The 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 not present on that object. You've probably run into this in normal python code. I'm singling it out because it's very easy to cause this error in SQLAlchemy and 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 4-2. 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 since we only queried the username column

The code above in Example 4-2 causes a 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 4-3. Error Output from Example 4-2

```
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 4-3, we have the typical format for an AttributeError in Python. It starts the line that says the type of error. Next there is a traceback showing us where the error occurred. Since 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 are doing something that would violate the constraints configured on a Column or Table. It is easy to encounter this type of error when you require something to be unique such as the username in our users table, and attempt to create two users with the same username. Example 4-4 shows some code that will cause such an error.

Example 4-4. 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 4-4 causes SQLAlchemy to create an IntegrityError. Let's look at the error output, and learn how to interpret what happened.

Example 4-5. 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 4-5 shows the typical format for an IntegrityError output in SQLAlchemy. It starts with the line that says the type of error. Next it includes the traceback details however, this is normal 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 constrain 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 it's compiled parameters like we looked at in Chapter 3. 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 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 above. The SQLAlchemy documentation contains information on the other types of errors.

In order for our programs not to crash whenever they encounter an error, we need to learn how to handle errors properly.

Handling Errors

To prevent an error form crashing or stopping our program, we need to handle errors cleanly. This is done with the same way it is done 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 4-6 has the details..

Example 4-6. 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 Catching the IntegrityError exception as `error` so we can access properties of the exception.

In Example 4-6, we are running the same statement as Example 4-4, 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 to return an error message to the our application user informing them that their operation failed. By handling the error with a try/except block, our application continues to execute and run.

While Example 4-6 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 4-1 shows the normal transaction workflow.

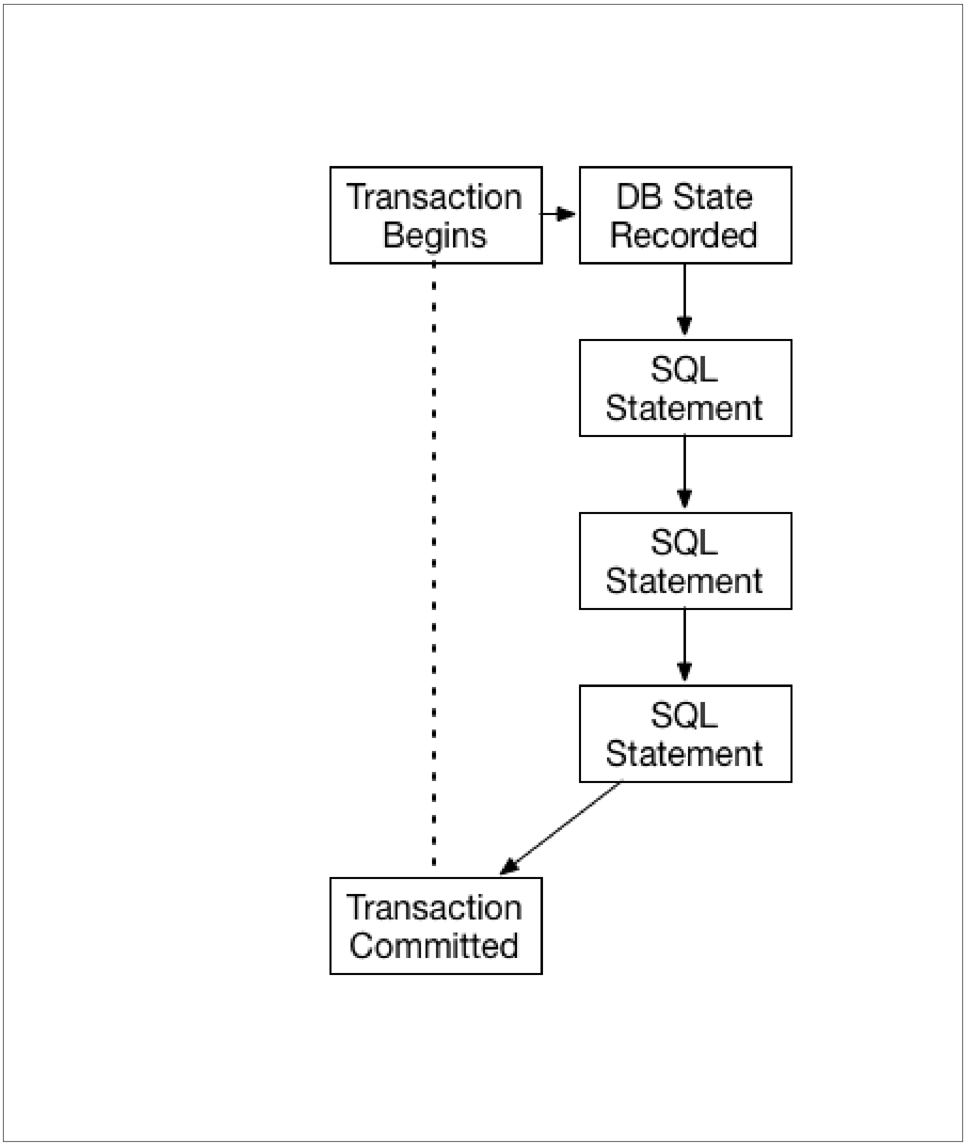


Figure 4-1. Successful Transaction Flow

However, if one or more of those statements fail we can catch that error and use the prior state to rollback back any statements that succeeded. Figure 4-2 shows an errored transaction workflow.

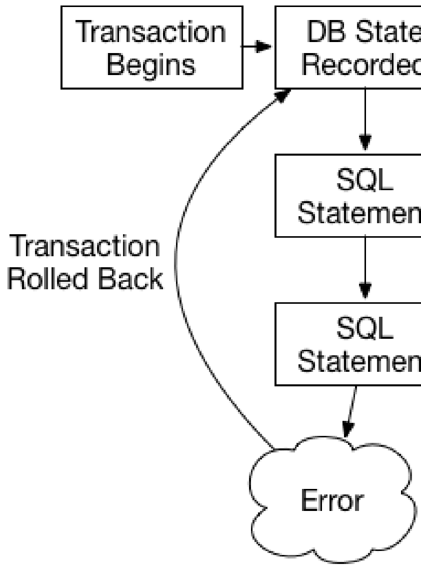


Figure 4-2. Failed Transaction Flow

A good example of when we might want to do this is actually present in our existing database. After a customer has ordered cookies from us, we will 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're gonna solve this with transactions.

We'll need a fresh python shell with the tables from Chapter 3; however, we need to add a CheckConstraint to the quantity column to ensure it can not go below 0, because we can't have negative cookies in inventory. Next recreate 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 4-7 shows how I setup the tables with the CheckConstraint, added the cookiemon user, and added the cookies.

Example 4-7. 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 one dark chocolate chip cookie and four regular chocolate chip cookies. We'll do this using the insert statements we learned in the previous chapter. Example 4-8 has the details.

Example 4-8. 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': 2,
        'extended_cost': 1.50
    },
    {
        'order_id': 2,
        'cookie_id': 1,
        'quantity': 9,
        'extended_cost': 4.50
    }
]
result = connection.execute(ins, order_items) ❷
```

❶ Adding the chocolate chip cookie order

❷ Adding the dark chocolate chip cookie order

That will give us all the order data we need to explore how transactions work, and now we need to define a function called ship_it. Our ship_it function will accept an order_id and remove the cookies from inventory and mark the order as shipped. Example 4-9 shows how this works.

Example 4-9. 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 4-10 shows how to do that.

Example 4-10. 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 on the first order_id

❷ Look at our cookie inventory

Running the code in Example 4-10 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 4-11).

Example 4-11. 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 4-10.

```
[(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 wanna ship whole orders to our customers. Using what you learned about try/except "Handling Errors" on page 49 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 transaction to safely execute our statements; Example 4-12 shows what to do.

Example 4-12. 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 4-10 to make sure that it didn't mess up our inventory with a partial shipment.

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

Excellent, our transactional function didn't mess up our inventory and didn't crash our application. We also didn't have to do a lot of coding to manually rollback 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 stop our application from crashing just because of a database statement error. We also looked at how transactions can help us avoid inconsistent databases, and application crashes in groups of statements. Now, we need to learn how to test our code to ensure it behaves the way we expect, and we'll do that in the next chapter.