# SQLAlchemy quick reference


A Python-database interface

John W. Shipman

2013-04-11 12:57

**Abstract**

Summary of common functions in the SQLAlchemy object-relational database system.

This publication is available in Web form[1] and also as a PDF document[2]. Please forward any comments to **tcc-doc@nmt.edu**.

## Table of Contents



[1] http://www.nmt.edu/tcc/help/pubs/db/sqlalchemy/
[2] http://www.nmt.edu/tcc/help/pubs/db/sqlalchemy/sqlalchemy.pdf

# 1. Introduction

This document summarizes some of the more commonly used features of the SQLAlchemy system. The purpose of the SQLAlchemy system is to provide a connection between your Python program and any of several classic database systems built on top of SQL (Structured Query Language[3]) such as MySQL, Postgres, or Oracle.

## Warning

You should have general knowledge of relational databases, and the additional problems of object-relational mappers, or the current document will not make much sense to you.

Assuming you understand how a classical SQL database works, SQLAlchemy can make life much easier for these applications and many more:

- You can use SQLAlchemy to connect to applications in other languages, using classical SQL database engines to interchange data.
- You can use SQLAlchemy to provide a persistence layer, so that some of the objects within your Python application can be stored in external databases for later retrieval and translation back into live objects.

This document is not a complete reference. It is intended for daily reference to functions necessary to put up fairly simple applications. SQLAlchemy is a full-featured and highly tuneable system, and we will not cover here many of its useful features for improved performance and simplified programming. Please refer to the SQLAlchemy home page[4] for tutorials and detailed reference information.

- Section 9, "`reflector`: Script to show an existing database's structure" (p. 26) is a script that uses reflection to determine the structure of an existing database.
- For a sizeable working example, see *`pycbc`: A Python interface to the Christmas Bird Count database*[5].

[3] http://en.wikipedia.org/wiki/SQL
[4] http://www.sqlalchemy.org/
[5] http://www.nmt.edu/~shipman/z/cbc/pycbc/

# 2. Overall structure of SQLAlchemy

You will need to know about three layers of SQLAlchemy:

- At the bottom is the *engine*, which manages the sending of actual SQL queries to the underlying database and reporting the results. See Section 3, "Connecting to database engines" (p. 3).
- The structure of the SQL database may be used as-is, or you may instead specify how the database is to be structured. See Section 4, "Metadata: Describing the database schema" (p. 4).
- To use an engine, you will need one or more *sessions*. Each session is a completely independent environment in which to conduct SQL transactions. If you like, you can build your own SQL statements and send them to the session, but this is not recommended: it is both easier and more secure to use the abstractions in the higher layers.

  If your application is single-threaded, you will need only one session. However, for web sites, SQLAlchemy makes it easy to scale up the application to use multiple threads. For session operations, see Section 6, "Session operations" (p. 19).
- The *object-relational mapper* gives you access to an even higher level of abstraction. At this level, you work directly with objects that are automatically connected to the database. See Section 5, "The object-relational mapper" (p. 15).

# 3. Connecting to database engines

To create an engine connected to an SQL database, you will need this `import` statement:

```
from sqlalchemy.engine import create_engine
```

Once you have imported this function, use this calling sequence to create an engine instance:

```
create_engine('dsn' [,keyword=value] ...)
```

***dsn***
: A string that defines what database you are using. For allowable values, see Section 3.1, "DSN: The Data Source Name" (p. 4).

**`convert_unicode=False`**
: If `True`, any Unicode values that you send to the database will automatically be stored as raw byte values, and automatically converted back to Unicode on retrieval.

**`echo=False`**
: Use `echo=True` for debugging purposes, before your application is in production status. All generated SQL is written to the standard output stream. The default behavior is for messages to be sent to a log.

  If you would like result sets to be written as well, use "`echo='debug'`".

**`pool_recycle=`*`timeout`***
: By default, connections will never be recycled (closed). The value, if given, specifies how often (in seconds) SQLAlchemy should reconnect to the database. The default value is `-1`, meaning wait forever.

  However, MySQL will disconnect a connection after a time. If you are using MySQL, it is a good idea to use "`pool_recycle=3600`" so that SQLAlchemy will reconnect after an hour of idle time.

**pool_size=*nConnects***
SQLAlchemy manages a pool of connections. This argument specifies the number of connections to maintain in the pool. The default value is 5.

## 3.1. DSN: The Data Source Name

The string used in your call to `create_engine()` specifies what database system you want to use, and how you want to connect to it. It has this general form:

```
"protocol://username:password@host:port/dbname"
```

**protocol**
The type of database system. Supported values include:

| | |
|---|---|
| `mysql` | MySQL |
| `postgresql` | Postgres |
| `oracle` | Oracle |

**username**
The user name.

**password**
The user's database password.

**host**
The host name. For the TCC's educational server, this will be "`dbhost.nmt.edu`".

**port**
The "`:port`" part is optional, if you need to specify a TCP/IP port for the database server.

**dbname**
Name of the database.

Here's an example. Suppose your username is `murgatroyd`, your password is `shazam`, and you are using a database named `blortbase` on the TCC's MySQL server. The DSN would look like this:

```
mysql://murgatroyd:shazam@dbhost.nmt.edu/blortbase
```

# 4. Metadata: Describing the database schema

*Metadata* means data about data. In the database world, metadata refers to the *schema*, the structure of the database.

One significant complication is that the SQLAlchemy view of a database's structure may be different than the structure at the SQL level.

Accordingly, there are three ways to use metadata to work with an SQL database.

- If you are working with an existing SQL database, you can ask SQLAlchemy to use *reflection* to determine the structure of the database, then use it as it is. See Section 4.1, "Reflection: finding out the structure of an existing SQL database" (p. 5).
- Your program can use SQLAlchemy to create and maintain the underlying SQL structure.
- If you want to build a different conceptual structure on top of an SQL database, you can let SQLAlchemy take care of converting SQLAlchemy operations into SQL operations.

In any case, you will use instances of a `MetaData` class to manage the representation of the database schema at both the abstract (SQLAlchemy) and concrete (SQL) levels.

## 4.1. Reflection: finding out the structure of an existing SQL database

In some situations, you have no control over the structure of the underlying SQL database, but you will need to be query and modify that database.

To create a `MetaData` instance that represents the current structure of a database, you will need this `import`:

```
from sqlalchemy import schema, types
```

Then, use a constructor with this general form to create the `MetaData` instance:

```
schema.MetaData(bind=E, reflect=True)
```

where *E* is the `Engine` instance returned by the `create_engine` function described in Section 3, "Connecting to database engines" (p. 3).

For operations on the resulting instance, see Section 4.3, "Interface to `class MetaData`" (p. 6).

## 4.2. Building your own schema

This section describes how to construct a `MetaData` instance from scratch. You would, for example, do this if there is no existing SQL database, and you need to create one.

You will need this `import`:

```
from sqlalchemy import schema, types
```

Create a `MetaData` instance like this.

```
schema.MetaData(bind=None, reflect=False)
```

**`bind`**
: If you have an `Engine` instance *E*, pass it to this argument and your metadata will be connected to that engine.

  You don't have to bind your metadata to an existing engine. You can always connect later with a statement of this form:

  ```
  M.bind = E
  ```

  where *M* is your `MetaData` instance and `E` is the `Engine` instance.

**`reflect`**
: If you use the `bind` argument to connect the metadata to an engine, and then pass `reflect=True`, the metadata will be initialized from that engine's database.

Once you have a `MetaData` instance, refer to Section 4.4, "`class Table`: Metadata for one table" (p. 6) to see how to add table definitions to it.

## 4.3. Interface to `class MetaData`

A `MetaData` instance is basically a container for `Table` instances that describe its constituent tables. For two different ways to call its constructor, see Section 4.1, "Reflection: finding out the structure of an existing SQL database" (p. 5) and Section 4.2, "Building your own schema" (p. 5).

Here are some of the attributes and methods of this class.

**`.create_all(tables=`*`tableList`*`, checkfirst=True)`**
Create the tables described by this metadata. If the `checkfirst` argument is `True` (the default value), no existing tables will be destroyed and recreated empty. To create only a specific set of tables, provide a list of the `Table` instances for those tables as the `tables` keyword argument.

**`.drop_all(tables=`*`tableList`*`, checkfirst=True)`**
Destroy (drop) tables. If the `checkfirst` option is `True`, SQLAlchemy will drop only existing tables. To drop a specific set of tables, provide as the `tables` argument a list of `Table` instances representing the tables to be dropped.

**`.reflect(only=None)`**
Load the metadata with the actual definitions in the database from the engine to which the metadata is bound. The default is to reflect all the tables. To reflect only specific tables, provide a list of their `Table` instances as the `only` argument.

**`.sorted_tables`**
This attribute is a list of the `Table` objects in the metadata. The tables are sorted in dependency order, that is, each table definition is preceded by the definitions of any tables that it references.

**`.tables`**
A dictionary whose keys are the names of the tables in this metadata, and each related value is the `Table` instance that has that name.

## 4.4. `class Table`: Metadata for one table

To add the definition of a table to a `MetaData` instance *M*, use a constructor of this general form:

```
schema.Table(name, M, col_1, col_2, ..., keyword=value, ...)
```

***`name`***
The SQL name of the table. If the name has any uppercase characters, that name will be treated as case-sensitive. Names with only lowercase characters will be case-insensitive.

***`M`***
The `MetaData` instance.

***$col_i$***
The remaining positional arguments specify the columns, and constraints if any, of the table.

- Each column is specified as a `Column` instance; see Section 4.9, "`class Column`: Metadata for one table column" (p. 10).
- To specify a primary key, especially when it comprises multiple columns, see Section 4.6, "`class PrimaryKeyConstraint`: Describing a table's primary key" (p. 9).
- To specify a foreign key constraint, see Section 4.5, "`class ForeignKeyConstraint`: Specifying a relation to another table" (p. 8).
- To specify a constraint that certain field values are unique within the table, see Section 4.7, "`class UniqueConstraint`: Describing a uniqueness constraint on a table" (p. 9).

## Note

If you would like to specify an additional index on a table, this is not done by the `Table()` constructor. See Section 4.8, "`class Index`: Specifying an index on a table" (p. 10).

**`autoload=False`**
If you specify `autoload=True`, the metadata for this table will be reflected, that is, the current definition of the table at the SQL level will be used.

**`include_columns=None`**
Normally, all the columns of the table will be visible at the SQLAlchemy level. If you would like to make only some columns visible, pass a sequence of column names to this keyword argument.

For example, suppose a table has columns named `first`, `last`, and `ssn`, but you don't want to make the `ssn` column visible. To accomplish this, use the argument "`include_columns=['first', 'last']`".

The `Table()` constructor is located in the `sqlalchemy.schema` module, so you will need to import it in one of these two ways:

- ```
  from sqlalchemy.schema import Table

  some_table = Table(...)
  ```
- ```
  from sqlalchemy import schema

  some_table = schema.Table(...)
  ```

Methods and attributes on a `Table` instance:

**`.append_column(C)`**
Append a `Column` instance *`C`* as the next column of the table.

**`.c`**
Synonym for `.columns`.

**`.columns`**
A `ColumnCollection` instance that describes the columns of this table.

You can iterate over this `ColumnCollection`, and it will generate each `Column` object in order. You can also use an attribute reference on this `ColumnCollection`.

For example, suppose a table named `colors` has three columns named `red`, `green`, and `blue`. This code would print the names of the columns:

```
    for col in colors.columns:
        print col.name
```

You could also retrieve the `Column` instance for the first column with this expression:

```
    colors.columns.red
```

Because retrieving columns in this way is quite common, the synonym `.c` makes this operation shorter, like this:

```
    colors.c.red
```

**`.create(checkfirst=False)`**
Creates the SQL table, unless `checkfirst` is `True` and the table does not already exist.

**`.drop(checkfirst=False)`**
Drops (destroys!) the SQL table. If `checkfirst=True` and the table does not exist, no drop operation will be issued.

**`.exists()`**
A predicate that returns `True` if the SQL table exists, `False` otherwise.

**`.foreign_key`**
If this table has any foreign key constraints, this attribute will describe them as a sequence of `ForeignKey` instances; see Section 4.11, "`class ForeignKey`: Specifying a column's foreign key relation" (p. 14).

**`.indexes`**
If this table has any additional indices, this attribute will describe them as a sequence of `Index` instances; see Section 4.8, "`class Index`: Specifying an index on a table" (p. 10).

**`.primary_key`**
If the table has a primary key, this attribute will be a `PrimaryKeyConstraint` instance that enumerates the column or columns that form the primary key. You can treat the value as a sequence of `Column` instances.

## 4.5. `class ForeignKeyConstraint`: Specifying a relation to another table

If a table has a foreign-key relation, there are two ways to specify it.

- If a single column of this table is related to a column in another table, you can pass a `ForeignKey` instance to the `Column()` constructor. See Section 4.11, "`class ForeignKey`: Specifying a column's foreign key relation" (p. 14).
- If one or more columns of this table are related to columns of another table, you can pass a `ForeignKeyConstraint` instance to the `Table()` constructor.

  This approach is the only way to describe a *composite key* relation, that is, a situation where the concatenation of two or more columns of this table are related to the concatenation of the same number of columns, all located in a foreign table.

To specify a foreign key relation by that second method, include an expression of this form in the argument list to the `Table()` constructor, after the column definitions:

```
schema.ForeignKeyConstraint(localColumns, foreignColumns
  [, name=localName])
```

***`localColumns`***
A sequence of the names of columns in the current table that are related to the foreign table.

***`foreignColumns`***
A sequence of the columns in the foreign table to which the columns in *`localColumns`* are related.

Each element of this sequence may be either: a name of the form "*`table.col_name`*"; or the actual `Column` instance in the foreign table.

***`localName`***
The name of this constraint, unique within the database.

For example, suppose the concatenation of three columns named `city`, `state`, and `nation` in the current table is a foreign key reference to columns named `city_code`, `region`, and `country` in a

table named `geo`. One way to describe this relation is to include this expression in the argument list to the `Table()` constructor:

```
schema.ForeignKeyConstraint(['city', 'state', 'nation'],
  ['geo.city_code', 'geo.region', 'geo.country'])
```

If `geo_table` is the `Table` instance for the foreign table, you could instead use its `.c` attribute and describe the relation by including this expression in the `Table()` argument list:

```
schema.ForeignKeyConstraint(['city', 'state', 'nation'],
    [geo_table.c.city_code, geo_table.c.region, geo_table.country])
```

## 4.6. `class PrimaryKeyConstraint`: Describing a table's primary key

There are two ways to describe the primary key of a table.

- Within any of the `Column()` constructors that describe the table's columns, you can supply a `primary_key=True` argument.

  If you use this argument on only one column, that column becomes the primary key. If multiple columns use `primary_key=True`, the primary key will be the concatenation of those columns in the order they are defined.

- You can instead include an expression of this form in the argument list of the `Table()` constructor, after the columns:

  ```
  schema.PrimaryKeyConstraint(column, ...)
  ```

  where each *`column`* is the name of the column.

For example, suppose that the primary key of a table is the concatenation of two columns named `major_id` and `minor_id`. To define this primary key, include this expression in the argument list to the `Table()` constructor:

```
schema.PrimaryKeyConstraint('major_id', 'minor_id')
```

## 4.7. `class UniqueConstraint`: Describing a uniqueness constraint on a table

A uniqueness constraint on a table is a requirement that the values in a certain column, or the concatenation of the values from multiple columns, must be unique in each row.

There are two ways to specify a uniqueness constraint.

- To specify that the value of a single column must be unique in the table, you may use the `unique=True` keyword argument in the `Column()` constructor for the column.
- You may also supply an instance of the `UniqueConstraint` class in the argument list of the `Table()` constructor, after the columns. This is the only way to specify a uniqueness constraint on a combination of two or more columns.

Here is the general form of the `UniqueConstraint()` constructor:

```
schema.UniqueConstraint(column, ... [, name=indexName ])
```

***column***
Each column is specified by its name.

**name**
Each uniqueness constraint must have a name that is unique within the database. You can supply this name as the value of the `name=` keyword argument; if you omit this argument, SQLAlchemy will invent one.

Suppose, for example, that you have a table with two columns named `city` and `state`, and you want to insure that there is only one row in the table with any given pair of values for these two columns. You can specify this constraint by including this expression in the arguments to the `Table()` constructor, after the column definitions:

```
schema.UniqueConstraint('city', 'state', name='city_state_key')
```

## 4.8. `class Index`: Specifying an index on a table

If you would like to add an index to table, it must be done *after* the table is created as described in Section 4.4, "`class Table`: Metadata for one table" (p. 6).

A constructor of this general form adds an index to table:

```
schema.Index(indexName, column, ...[, keyword=value])
```

***indexName***
A name for this particular index. The name must be unique among all the indices in the same database.

***column***
Provide one or more `Column` instances in order to index on the concatenation of the column values in each row. Typically, each will be an expression of the form "*table*`.c.`*colName*", where *table* is the `Table` instance for which the index is to be created, and `colName` is the name of the column.

**unique=False**
By default, there is no requirement that each value in the index be unique. If you specify `unique=True`, only one occurrence of each value will be permitted in the table.

For example, suppose `town_table` is a `Table` instance with two rows named `state` and `city`. To add an index to this table whose values are unique, you would call the `Index()` constructor with these values:

```
schema.Index("town_x", town_table.c.state, town_table.c.city,
      unique=True)
```

Attributes of an `Index` instance include:

**.columns**
A sequence of the `Column` instances in this index, in order; see Section 4.9, "`class Column`: Metadata for one table column" (p. 10).

**.name**
The name of the index.

## 4.9. `class Column`: Metadata for one table column

To create a `Column` instance that describes one column of a table, use a constructor of this general form:

```
schema.Column(name, type, arg, ..., keyword=value, ...)
```

***name***
The name of the column.

**type**
The column type. You may pass either a class or an instance of a class, so long as the class is a subtype of `AbstractType`. For some of the common column types, see Section 4.10, "Column types" (p. 12).

**arg**
Following the column type, you may optionally include these items:
- Section 4.11, "`class ForeignKey`: Specifying a column's foreign key relation" (p. 14).
- Section 4.12, "`class Sequence`: Specifying an auto-incrementing column" (p. 15).

**default=*value***
Use *`value`* as the default value of this column when a new row is added, if no explicit value is provided.

**index=True**
Specify an index on the table. If this column is to be indexed, specify `index=True`.

To specify an index on multiple columns, see Section 4.8, "`class Index`: Specifying an index on a table" (p. 10).

**key=*columnKey***
Normally, the name of a column at the SQLAlchemy level is the same as its name in the SQL table. To use a different name throughout SQLAlchemy, supply that name as the value of this keywoard argument.

**nullable=True**
By default, a column may contain null values. To prohibit this, use `nullable=False`.

**primary_key=True**
If you provide this option, the column will be part of the primary key for the table. If only one column specifies this option, it will be the sole primary key column. To define a primary key composed of multiple columns, use this option on each of the columns in the key.

**unique=True**
By default, a column may contain multiple rows with the same value. To prohibit this, use `unique=True`.

Attributes of a `Column` instance include:

**.name**
The column's name.

**.nullable**
A `bool` value, `True` if this column may contain null values.

**.primary_key**
A `bool` value, `True` if this column is a primary key column.

**.type**
The column's type as an instance of one of the classes described in Section 4.10, "Column types" (p. 12).

**.unique**
A `bool` value, `True` if the values in this column must be unique within the table.

## 4.10. Column types

In order to specify SQLAlchemy column types, you will need to import the `types` module like this:

```
from sqlalchemy import types
```

When you call the `Column()` constructor, the second argument specifies the column type (see Section 4.9, "`class Column`: Metadata for one table column" (p. 10)).

This section enumerates some of the more commonly used SQLAlchemy classes that are used to specify column types. When you create a `Column`, you can pass either a class name or an instance as the type argument.

- For a number of generic, simple data types like `Integer` or `Date`, you'll generally specify the type as just a class name.
- For column types that need additional information, such as the size of a text field, you'll pass that additional information to the constructor, and use the resulting instance as the type argument to the `Column()` constructor. For example, if you want to store a text string up to 8 characters, you'll use "`Text(8)`" as the argument.

Here's an example. Suppose your database has a table named `horses` that describes horses. The first column is the horse's name, the primary key, up to 40 characters; and the second column is its birthdate. This code would add that table's description to a `MetaData` instance named `meta`; note that the type of the first column is an instance of the `Text` class, while the type of the second column is the name of a class (`Date`).

```
horse_table = schema.Table("horses", meta,
    schema.Column("name", Text(40), primary_key=True),
    schema.Column("born", Date))
```

SQLAlchemy's types have two faces: on the SQLAlchemy side, you will see a Python type, and on the SQL side is the actual type that appears in the SQL statements sent to the database engine. For example, in your Python code, an instance of SQLAlchemy's `Date` class looks like a `date` instance from Python's `datetime` library, but in the SQL world the representation will use SQL's `DATE` type.

Here are some of the more common data types. Types whose names contain lowercase letters, such as `DateTime`, are generic types; SQLAlchemy will pick an appropriate SQL type for you. Types whose names are all caps, such as `INTEGER`, are guaranteed to use the SQL type of the same name. Refer to the vendor's documentation[6] for the full set.

### 4.10.1. `Boolean` type

Represents a Python `bool` value, `True` or `False`.

### 4.10.2. `Date` type

Represents a calendar date as a `datetime.date` instance on the Python side.

### 4.10.3. `DateTime` type

Represents a date and time as a `datetime.datetime` instance; this corresponds to the SQL `TIMESTAMP` type.

[6] http://www.sqlalchemy.org/docs/core/types.html

### 4.10.4. `Float` type

Represents a floating-point number as a Python `float`. If you use the class name `Float` to specify this type, you will get a column of the maximum precision. You may also specify an expression of this form to select the precision:

```
types.Float(B)
```

The value of *B* is the number of bits of precision in the mantissa. In practical terms, any value of *B* from 1 to 24 will give you a single-precision float, while values 25–53 will give you a double-precision column.

### 4.10.5. `Integer` type

Represents a Python `int` integer. Some engines may not support values that would require a Python `long`.

### 4.10.6. `Interval` type

Represents an interval of time, which will use a Python `datetime.timedelta` instance.

### 4.10.7. `LargeBinary` type

A large binary object, which will use an SQL `BLOB` or `BYTEA` type.

### 4.10.8. `Numeric` type

Represents a scaled integer. Use an expression of this general form to specify a field of this type:

```
types.Numeric(precision=None, scale=None)
```

The `precision` argument specifies the maximum number of decimal digits, and the `scale` argument specifies the scale factor. For example, a value of type `Numeric(5,2)` will hold numbers of up to five digits, but with an implied decimal point before the last two digits, that is, numbers up to 999.99. The default `scale` value is zero, and the default `precision` value depends on the underlying database engine.

### 4.10.9. `String` type

Represents a Python `str` or `unicode` value. For most engines, you must indicate the maximum length using a constructor call of this general form:

```
types.String(length)
```

For example, `String(37)` would translate to the SQL type `VARCHAR(37)`.

### 4.10.10. `Text` type

For Python `str` and `unicode` types. Comparisons are case-insensitive.

### 4.10.11. `Time` type

Values represent a time of day, and are represented on the Python side by an instance of the `time` class in the `datetime` module. The actual database will use an SQL `TIME` column.

### 4.10.12. `Unicode` type

Use this column type for Unicode strings. Passing `str` values to SQLAlchemy for this type is not recommended.

### 4.10.13. The SQL types

Use the types in this list to guarantee that the SQL type of the same name will be used. Use the class name (e.g., INTEGER) for types that are always the same; use the argument list shown to specify additional type information such as the capacity of a field.

- `BINARY(`*`length`*`)`, where *`length`* is in bytes.
- `BLOB`
- `BOOLEAN`
- `CHAR(`*`length`*`)`
- `DATE`
- `DATETIME`
- `DECIMAL([`*`precision`*`[, `*`scale`*`]])`: The values are the same as for Section 4.10.8, "`Numeric` type" (p. 13).
- `FLOAT([`*`precision`*`])`: The *`precision`* is in bits, as with Section 4.10.4, "`Float` type" (p. 13).
- `INT` or `INTEGER`
- `NUMERIC([`*`precision`*`[, `*`scale`*`]])`: The values are the same as for Section 4.10.8, "`Numeric` type" (p. 13).
- `SMALLINT`: A two-byte signed integer value in the range [-32768, 32767].
- `TEXT(`*`maxlen`*`)`: A variable-length text column whose maximum capacity is *`maxlen`* bytes.
- `TIME`
- `TIMESTAMP`
- `VARBINARY(`*`maxlen`*`)`: A binary string with a capacity of *`maxlen`* bytes.
- `VARCHAR(`*`maxlen`*`)`: A case-insensitive character string with a capacity of *`maxlen`* bytes.

## 4.11. `class ForeignKey`: Specifying a column's foreign key relation

To specify that a column is a foreign key—that is, its value is related to a column in a different table—add to the arguments of the `Column()` constructor an expression of this form:

```
schema.ForeignKey(foreignColumn[, name="keyName"])
```

***`foreignColumn`***
: A reference to the column in the foreign table. This may be either a string of the form "*`table.column`*", or a `Column` instance in the foreign table.

***`keyName`***
: A unique name for this key. If omitted, it defaults to a string of the form "`ix_`*`columnName`*". This name must be unique within the database.

Here's an example. Suppose you are defining an integer column named `part_no`, and values in this column are related to a column named `part_no` in a table named `part`. Here is one way to define the column:

```
  schema.Column("part_no", types.Integer,
    schema.ForeignKey("part.part_no"))
```

You may also provide as an argument the actual `Column` instance from the foreign table. Here is another way to define this column, assuming that `part_table` is the foreign table as a `Table` instance:

```
  schema.Column("part_no", types.Integer,
    schema.ForeignKey(part_table.c.part_no))
```

Attributes of a `ForeignKey` instance include:

**`.column`**
The column in the foreign table, as a `Column` instance. See Section 4.9, "`class Column`: Metadata for one table column" (p. 10).

**`.parent`**
The column in this table that refers to the foreign key, as a `Column` instance.

## 4.12. `class Sequence`: Specifying an auto-incrementing column

To specify that a column's value is automatically generated each time a new row is added, add an expression of this form to the argument list of the `Column()` constructor:

```
  schema.Sequence(name, start=None, increment=None, optional=False)
```

***`name`***
A name for this sequence column. This name must be unique within the database.

**`start`**
The value to be used for the first row added. Default is 1.

**`increment`**
The value to be added to the previous value every time a new row is added. Default is 1.

**`optional`**
If you are using Postgresql, use `optional=True` so that the engine's built-in auto-increment feature will be used.

Suppose you are defining an auto-increment column named `ticket_id`. You might use a definition like this:

```
  schema.Column("ticket_id", types.Integer, Sequence("ix_ticket_id"))
```

# 5. The object-relational mapper

Once you have defined a metadata object as described in Section 4, "Metadata: Describing the database schema" (p. 4), the next step is to define a set of Python classes called *mapped classes* that represent entities from the database while your program is working on them.

Generally each mapped class corresponds to a table in the SQL database, but you can also define mapped classes that correspond to information derived from multiple tables.

To work with SQLAlchemy's ORM (object-relational mapper), you will need an import like this:

```
from sqlalchemy import orm
```

For each class to be mapped to a table, perform these three steps:

1. Define the metadata for the table.
2. Define a Python class whose instances will represent rows of the table inside your Python application. See Section 5.1, "Defining a class to be mapped" (p. 16).
3. Use a call to `orm.mapper()` to connect the table and the class. See Section 5.2, "Using `orm.mapper()`" (p. 16).

## 5.1. Defining a class to be mapped

You must write declarations for each class to be mapped in the ORM. For a typical class that represents a row from a table, you will need these things in the class:

- The class must inherit from `object`; that is, it is a new-style Python class.
- It is good practice to write a `__repr__()` method so that you can display some meaningful information about the instance while debugging your program.
- A constructor is not strictly necessary if all instances of the class are generated by SQLAlchemy. However, if you are adding new data to the database, you will need a constructor that takes column values as arguments.

Here is a small example of a mapped class. Suppose that you are managing a collection of beads. Your database has a table named `bead_table` with three columns: `id`, an inside diameter in millimeters as a float; `material`, a string defining what the bead is made of; and `count`, an integer specifying the number on hand. Your class might look like this.

```
class Bead(object):
    def __init__(self, id, material, count):
        self.id = id
        self.material = material
        self.count = count

    def __repr__(self):
        return("<Bead(%s (%.1f),
```

## 5.2. Using `orm.mapper()`

Once you have defined a `Table` instance *T* for the table in your metadata, and written a class *C* to be mapped, relate them with a call of this general form:

```
orm.mapper(C, local_table=None[, keyword=value, ...])
```

Keyword arguments include:

**`local_table`**
The `schema.Table` instance to which *C* is being mapped.

**`order_by`**
By default, rows retrieved from the mapped table will be returned in order of the primary key. If you prefer a different order, use this keyword argument to specify a `Column` instance, or a list of `Column` instances, to be used as keys.

For example, suppose your mapped `Reptiles` class is related to a table `reptile_table` with columns named `section` and `id_number`, and you want rows to be returned in order by the concatenation of those two columns. You would use this call:

```
orm.mapper(Reptiles, reptile_table,
    order_by=(Reptiles.section, Reptiles.id_number))
```

**`properties`**
A dictionary whose keys are the names of properties of the mapping. See Section 5.3, "Properties of a mapping" (p. 17).

## 5.3. Properties of a mapping

When you map a class to a table with `orm.mapper()`, you can specify additional *properties* of the class. These properties act like attributes of an instance, but they perform database operations that can retrieve related data from other tables.

The `properties` keyword argument to `orm.mapper()` is a dictionary whose keys define additional attributes of the mapped class, and each related value is an `orm.relation` instance that defines how that column is derived from the database.

There are a number of different kinds of relations. We'll discuss the two most common cases.

- Section 5.3.1, "One-to-many relations" (p. 17): Where each row of a parent table is related to multiple rows of a child table.
- Section 5.3.2, "Many-to-many relations" (p. 18): Where each row of one table may be related to many rows of another table, and vice versa.

### 5.3.1. One-to-many relations

For a one-to-many (parent-child) relation, in your call to `orm.mapper()`, include an expression of this general form for the value of the `properties=` keyword argument:

```
childcol: orm.relation(Other, backref=backname)
```

This will create a one-to-many relation between the mapped class and some class *`Other`*. Once you have done this:

- A reference to the *`childcol`* attribute on the mapped class will produce a list of the related children.
- In each child instance, the *`backname`* attribute will point at the parent instance.

Here is an example. Suppose that the `Part` class is mapped to a table `parts_table` that relates part numbers (column name `partNo`) to descriptions (column name `desc`). Further suppose that there is an `Inventory` class that specifies how many (column `partsCount`) of a given part (whose part number is in that class's `.partNo` attribute) are on hand. Here is the call to `orm.mapper()` for this relation:

```
orm.mapper(Part, parts_table,
    properties={'inventories': orm.relation(Inventory, backref='part')})
```

For any instance *`P`* of class `Part`, attribute *`P`*`.inventories` will be a list of `Inventory` instances that have the same part number.

Also, for any instance *`I`* of class `Inventory`, attribute *`I`*`.part` will be the `Part` instance that has the same part number.

### 5.3.2. Many-to-many relations

Suppose you are putting up a blog, and you want to be able to associate short tags with your articles. An article might have no tags, or it might have several tags such as "basketball", "Vegemite", and "lakes".

But the opposite relation is also the case. For a tag like "Vegemite", there may be no articles, one article, or several articles.

This is an example of a many-to-many relationship. To manage such a relation, you will need to create a new *secondary table* with two columns, such that each column is related to one of the two tables in the many-to-many relation.

To continue our example, every row of the secondary table represents one association of a tag with an article. This table has two columns: one column containing a foreign key to the articles table, and the other column containing a foreign key to the tags table.

To map such a table, your call to `orm.mapper()` for one of the two primary tables will need a `properties` argument of this general form:

```
othercol: orm.relation(OtherTable,
    secondary=secondaryTable,
    backref=backname)
```

- *`otherCol`* is the name of an attribute that will be added to the primary table being mapped.
- *`OtherTable`* is the other primary table as a class name.
- *`secondaryTable`* is the `Table` instance for the secondary table.
- *`backname`* is the name of an attribute that will be added to the other primary table.

Once you have done this:

- Any instance of the first primary table will have an attribute that iterates over the related rows of the other primary table. This instance name was specified by *`otherCol`*.
- Any instance of the second primary table will have an attribute that iterates over the related rows of the first primary table. This instance name was specified by *`backname`*.

To continue our example, suppose the `Article` class, whose primary key is its `article_id` column, has a many-to-many relation with the `Tag` class, whose primary key is its `tag_id` column. The secondary table metadata will look like this:

```
article_tag_table = schema.Table('article_tag', metadata,
    schema.Column('article_id',
      schema.ForeignKey('articles.article_id')),
    schema.Column('tag_id',
      schema.ForeignKey('tags.tag_id')))
```

To set up this relation in the mapper:

```
orm.mapper(Article, articles_table,
    properties={ 'tags': orm.relation(Tag, secondary=article_tag_table,
                                      backref='articles') })
```

Once this mapping is in place, every instance of the `Article` class will have a `.tags` attribute that iterates over all the related `Tag` instances. Conversely, every instance of the `Tag` class will have an `.articles` attribute that will iterate over the articles with that tag.

# 6. Session operations

To operate on your database, you will need a *session*. A session is a mechanism for performing one or more operations on the database: finding data, adding data, modifying data, and deleting data.

A session is also a container for a set of related operations that make up a *transaction*. Your program may perform a number of operations on a session, but nothing is saved to the database until you *commit* the transaction. You may instead *rollback* the transaction, which cancels all the operations within the transaction.

Within the session there may be any number of objects that represent items in the database. At any given time, these objects may have different values from the database. The process of copying any changes into the database is called *flushing*. Flushing is not the same as committing: you may flush the session several times before committing a transaction. Furthermore, even after flushing the session, you can still rollback the transaction, which undoes all the changes, whether they were ever flushed or not.

For simple applications, you can create a single session and use it for all your operations. However, in some situations such as web frameworks, there may be multiple sessions all operating on the same database using the same engine.

Before you can create a session, you must import the `orm` module from the `sqlalchemy` module, like this:

```
from sqlalchemy import orm
```

To create a session is a two-step process.

1. Use the `sessionmaker()` function from the `orm` module to create a new class. In our examples, we will call this class `Session`.
2. Use the constructor of this new class to create a session. The arguments that were passed to the `sessionmaker()` function determine what kind of session will be created.

Here are the arguments to `sessionmaker()`, and their default values:

```
Session = orm.sessionmaker(bind=engine, autoflush=True, autocommit=False,
                           expire_on_commit=True)
```

**`bind`**
The engine returned by `create_engine()` (see Section 3, "Connecting to database engines" (p. 3)).

**`autoflush`**
If this argument is `True`, every transaction commit will flush the changes from the session into the database before performing the commit.

**`autocommit`**
This option affects whether flushing the session also causes a commit. Generally it is good practice to use the default value, `autocommit=False`, and commit explicitly. If this option is `True`, however, whenever the session is flushed, SQLAlchemy also performs a commit.

**`expire_on_commit`**
By default, this option is `True`, which means that after every commit, objects in the session expire, so the next transaction will fetch the latest values from the database. This option is strongly recommended for multi-threaded applications.

Here's an example of session creation. Let's suppose that module `model.py` contains a variable named `metadata` that defines the schema. First we create the `Session` class:

```
import model

engine = create_engine("mysql://alice:rabbit@dbhost.nmt.edu/alicedb")
model.metadata.bind = engine
Session = orm.sessionmaker(bind=engine)
```

Creation of a session is straightforward. To continue this example:

```
session = Session()
```

## 6.1. Operations on a session

Here are some of the common methods on a session instance *S*.

**`S.add(item)`**
Add an instance of a database object to the session.

**`S.add_all([item, item, …])`**
Add multiple instances to the session.

**`S.commit()`**
Commit the pending transaction in *S* to the database.

**`S.delete(item)`**
Mark the *item* to be deleted. Actual deletion will occur only when you call `.flush()`, or right away if auto-flush is enabled.

**`S.flush()`**
This method forces the database to be updated using any records within the session that are to be added, modified, or deleted.

**`S.query(class)`**
Return a new `Query` instance that can retrieve data from some mapped *class*. See Section 7, “Queries” (p. 20).

**`S.rollback()`**
Rollback the transaction in *S*.

Here's an example. Suppose you have a `Session` instance named `session`; a class `Horse` is mapped to a table in your database. These line would create a new instance and commit it into the database:

```
seabiscuit = Horse("Seabiscuit", datetime.date(2008, 3, 14))
session.add(seabiscuit)
session.commit()
```

# 7. Queries

You can initiate a query against a particular mapped class in the database by using the `query()` method on a `Session` instance (see Section 6, “Session operations” (p. 19)).

Here is the general form of the expression that creates a `Query` instance, given a `Session` instance *S*:

```
S.query(item[, item, ...])
```

Each *item* may be either:

- A mapped table.
- A reference to a column of a mapped table. For example, if table `Moon` has a `diameter` column, you can refer to it as `Moon.c.diameter`.

Once you have a `Query` instance, you can perform these operations on it:

- You can use it as an iterator.
  - If you passed a single table argument to the `Query`constructor, it will generate a sequence of instances of that table.
  - If you passed a single column reference to the constructor, the resulting iterator will generate a sequence of the values for that column.
  - If you pass multiple arguments to the constructor, the resulting iterator will generate a sequence of special *result tuples*. See Section 7.12, "Result tuples from a `Query`" (p. 25) for the operations on these results.
- You can call various methods on the `Query` instance. Many of these methods return a modified `Query` instance, so you can stack multiple method calls after the original. The result of all these calls will be another `Query` instance that is modified to include the various parameters of the query that are applied by these method calls. See the sections below for a list of some of the common operators and methods on a `Query` instance.

## 7.1. Slicing a `Query`

You can use the slicing operator on a `Query` instance. For example, given a `session` instance, this query would display the second, third, and fourth rows from table `Tchotchke`:

```
for row in session.query(Tchotchke)[1:4]:
    print row
```

## 7.2. `Query.all()`

To return a list of all the objects from a query, use this method. (If this list is very large, you may wish to avoid this method, since it makes all the objects present in memory at once.)

For example, suppose you have a session instance named `session`, and a mapped class named `model.Kennel`. This query would display all the rows in the related table.

```
q = session.query(model.Kennel)
for row in q.all():
    print row
```

## 7.3. `Query.count()`

This method returns, as an integer, the number of items that are retrieved by a query.

For example, this would return the number of rows in the mapped class `model.Kennel`:

```
print session.query(model.Kennel).count()
```

## 7.4. `Query.delete()`

> **Warning**
>
> This method can delete entire tables! Use it with great caution. Back up your data outside the database if you are concerned about losses.
>
> Also, if you delete rows that are the subject of a foreign-key relation in another table, the operation may fail. It is best to delete first the records that depend on other rows.

To delete all the rows that match a given `Query` object *`q`*:

```
q.delete()
```

## 7.5. `Query.filter()`

This method returns a `Query` instance that is modified to include only certain rows.

The argument of this method may take any of the following forms, where *`column`* is a `Column` within the table being queried.

***`column == value`***
Return only those rows for which *`column`* equals the given *`value`*.

***`column != value`***
Return only those rows for which *`column`* *does not* equal the given *`value`*.

**Relational operators**
The usual relational operators are allowed in expressions passed to `.filter()`. For example, this would return a `Query` instance that produces all the rows from the `People` table whose `age` column is greater than or equal to 18:

```
q = session.query(People).filter(People.c.age >= 18)
```

***`column`*`.like(`*`wildcard`*`)`**
The `wildcard` argument is a string with one or more "`%`" characters match any string. For example, this query would return all the rows from the `Tree` table whose `kind` column contains the string `"elm"` anywhere in the column:

```
elmList = session.query(Tree).filter(Tree.c.kind.like("%elm%")).all()
```

***`column`*`.in_(`*`rowList`*`)`**
Returns a modified `Query` that includes only those rows found in *`rowList`*.

The *`rowList`* argument may be either a regular Python `list` containing column values, or a `Query` instance whose rows contain the given column. For example, this query displays rows of the `Things` table whose `category` column is either `"animal"` or `"vegetable"`.

```
for row in session.query(Things).filter(
    Things.c.category.in_( ['animal', 'vegetable']))
```

**`~`*`column`*`.in_(`*`rowList`*`)`**
To produce a modified `Query` containing rows whose column values are *not* found in a given sequence or query, use the "`~`" operator before the call to the `.in_()` method. For example, here is a query that excludes rows of the `Bull` table whose `state` column is Texas or Oklahoma.

```
session.query(Bull).filter( ~ Bull.c.state.in_(["TX", "OK"]))
```

***column* == None**
Returns a new `Query` that includes only rows where the specified *`column`* has a null value.

***column* != None**
Returns a new `Query` that includes only rows where the specified *`column`* *does not* have a null value.

**and_(*$C_1$*, *$C_2$*)**
Return only rows that satisfy both condition *$C_1$* and condition *$C_2$*. Each condition is described using any of the expressions that can be passed to `.filter()`. You will need to import it like this:

```
from sqlalchemy import and_
```

For example, this query would return rows from the `Date` table whose `marital` column is `'single'` and whose `sex` column is `'F'`.

```
q = session.query(Date).filter(and_(
         Date.c.marital=='single',
         Date.c.sex=='F'))
```

**or_(*$C_1$*, *$C_2$*)**
Similar to the `and_()` method, but returns rows for which either condition *$C_1$* or condition *$C_2$* is true. You will need to import it like this:

```
from sqlalchemy import or_
```

## 7.6. Query.filter_by()

This method produces a modified `Query` instance that includes only rows with certain column values. Here is the general form, given a `Query` instance *`q`*:

```
q.filter_by(colname=value[, colname=value, ...])
```

Each *`colname`* is the name of a column in the queried table. The returned query selects only rows for which each of the named columns has the given value.

For example, here is an example query from Section 7.5, "`Query.filter()`" (p. 22) redone using `.filter_by()`:

```
q = session.query(Date).filter_by(marital='single', sex='F')
```

## 7.7. Query.first()

This method takes no arguments, and attempts to produce the first result from the query.

- If the query produces one or more results, this method returns the first result.
- If the query produces no results, this method returns `None`.

## 7.8. `Query.get()`

This method attempts to fetch a single result row. If the table has a single-column primary key, use the value of that column. If the table's primary key has multiple columns, use a tuple containing the values of those columns, in the order they were declared when the table was created.

For example, suppose the `Gliders` table has a primary key `tail_number`. This query would return the row whose key is `"NCC-1701"`:

```
eprise = session.query(Gliders).get("NCC-1701")
```

Suppose you have a table named `Reptiles` with a composite primary key containing the columns `genus` and `species`. To query this table for a row whose `genus` is *Crocodylus* and its `species` *porosus*:

```
bigCroc = session.query(Reptiles).get(('Crocodylus', 'porosus'))
```

## 7.9. `Query.join()`

If you want to qualify your query according to the values of columns in a related table, use a method call of this general form:

```
.join(table_1, table_2, …)
```

Each $table_i$ is a mapped attribute of the table you are querying.

This method effectively adds all the columns from the referenced tables to each row returned by the query, so you can use `.filter_by()` or `.filter()` or `.order_by()` methods to select or sort rows.

For example, suppose you have a table of inventory records that specify how many of a given part are on hand in a column named `on_hand`. This table is mapped to a class named `Inventory` that has a many-to-one relation to a mapped class named `Part` that describes various parts; the `Inventory` class has a mapped attribute named `.part` that refers to the related `Part` row. The `Part` table has a column named `obsolete` that contains 0 if the part is current, 1 if it is obsolete, and it also has a mapped attribute named `.inventories` that iterates over all the related inventory records for that part.

You want to formulate a query of all the inventory records for obsolete parts that have an `on_hand` column greater than zero. You might code it like this:

```
q =(session.query(Inventory)
        .join(Inventory.part)
        .filter(Inventory.on_hand > 0)
        .filter_by(obsolete=1))
```

The query will return `Inventory` instances because that was the argument you passed to the `.query()` method. The `.join()` method call effectively adds the columns from the `Part` table to each row of the query, so that you can refer to the `obsolete` column from the `Part` table in the `.filter_by()` method call.

## 7.10. `Query.one()`

Use this method when you expect a query to return exactly one record, and you want to be notified otherwise. It takes no arguments.

- If the query returns exactly one row, it returns that instance.
- If the query returns no rows, this method raises exception `orm.exc.NoResultFound`.

- If the query returns more than one row, the method raises exception `orm.exc.MultipleResultsFound`.

## 7.11. `Query.order_by()`

To specify the ordering of rows returned by a query, apply this method to it; the method returns a modified `Query` instance.

```
.order_by(column, ...)
```

For example, suppose you have a mapped class named `Addresses` with columns named `country`, `state`, and `town`. Here is a query that uses country as the major key, state as a secondary key, and town as a tertiary key:

```
q = session.query(Addresses).order_by(Addresses.country,
        Addresses.state, Addresses.town)
```

## 7.12. Result tuples from a `Query`

If you pass multiple arguments to the `.query()` method of a session, the resulting instance can be used as an iterator that generates special *result tuples*.

You can access the components of a result tuple in either of two ways.

- It acts like a normal tuple, so you can use an index expression to extract one of the values. For example, if `t` is a result tuple, `t[2]` is the third value, corresponding to the third argument passed to the `.query()` call.
- You can use an attribute reference. For components of the result tuple corresponding to column names passed to `.query()`, this will be a column name. For components corresponding to class names, this will be the class name.

  For example, suppose you have a mapped class named `Island` that has columns named `name`, `area`, and `population`. You might set up the query like this:

  ```
  q = session.query(Island, Island.name, Island.area)
  ```

  For a tuple `t` produced by iterating over `q`, you can access the value of the `name` column as `t.name`. You can access the entire `Island` instance as `t.Island`.

# 8. Exceptions

Here is a list of exceptions that may be thrown by SQLAlchemy. They are defined in module "`sqlalchemy.exc`".

- `ArgumentError`
- `CircularDependencyError`
- `CompileError`
- `ConcurrentModificationError`
- `DBAPIError`
- `DataError`
- `DatabaseError`
- `DisconnectionError`
- `FlushError`

- `IdentifierError`
- `IntegrityError`
- `InterfaceError`
- `InternalError`
- `InvalidRequestError`
- `NoReferenceError`
- `NoReferencedColumnError`
- `NoReferencedTableError`
- `NoSuchColumnError`
- `NoSuchTableError`
- `NotSupportedError`
- `OperationalError`
- `ProgrammingError`
- `SADeprecationWarning`
- `SAPendingDeprecationWarning`
- `SAWarning`
- `SQLAlchemyError`
- `SQLError`
- `TimeoutError`
- `UnboundExecutionError`
- `UnmappedColumnError`

# 9. `reflector`: Script to show an existing database's structure

Here is a script that connects to an existing database and prints a report showing its schema. This script uses lightweight literate programming[7] to explain the script, and its design uses the Cleanroom development methodology[8].

To use this script, use these command line options:

```
reflector protocol username passfile host[:port] dbname
```

***`protocol`***
Database protocol, one of: `postgresql`, `mysql`, or `oracle`.

***`username`***
Your database user name.

***`passfile`***
The path name of a file that contains your database password by itself on the first line. We require that you do things this way so that your database password will not be visible on the command line.

***`host[:port]`***
The database server's hostname, optionally followed by a colon and a port number.

**`dbname`**
The name of your database.

[7] http://www.nmt.edu/~shipman/soft/litprog/
[8] http://www.nmt.edu/~shipman/soft/clean/

## 9.1. Prologue

This script starts with the usual "pound-bang line" to make it self-executing under Linux, followed by a pointer back to this documentation.

reflector

```
#!/usr/bin/env python
#===============================================================
# reflector: Report on an existing database schema.
#   For documentation, see:
#     http://www.nmt.edu/tcc/help/pubs/db/sqlalchemy/
#---------------------------------------------------------------
# Command line options:
#   reflector PROTOCOL USERNAME PWDFILE HOST[:PORT] DBNAME
# where:
#   PROTOCOL     {postgresql|mysql|...}
#   USERNAME     Database username
#   PWDFILE      Name of a file containing the password
#   HOST[:PORT]  Host name with optional port
#   DBNAME       Database name
#---------------------------------------------------------------
```

Next come module imports. We need `sys` to access the standard I/O streams.

reflector

```
#===============================================================
# Imports
#---------------------------------------------------------------

import sys
```

From `sqlalchemy`, we will need the `engine` module to connect to the database engine, and the `schema` module for the `MetaData` class to reflect the database's schema. The `exc` module defines SQLAlchemy's exceptions.

reflector

```
from sqlalchemy import schema, engine, exc
```

The constant `URL_FORMAT` is used to assemble the various command line arguments and the password into a URL suitable for connecting to the database engine.

reflector

```
#===============================================================
# Manifest constants
#---------------------------------------------------------------
URL_FORMAT = "%s://%s:%s@%s/%s"
#             ^    ^  ^  ^  ^
#             |    |  |  |  +-- Database name
#             |    |  |  +-- Host name
#             |    |  +-- Password
#             |    +-- User name
#             +-- Protocol
```

## 9.2. `main()`: `reflector`'s main program

reflector

```
# - - - - -   m a i n

def main():
    '''Report on the schema of an existing database.

      [ if (the command line arguments are valid) and
        (we can connect to the database specified by those arguments) ->
          sys.stdout  +:=  a report on the schema of that database
        else ->
          sys.stderr  +:=  usage and error messages ]
    '''
```

The program is broken into three steps:

- Section 9.3, "`checkArguments()`: Process the command line arguments" (p. 29).
- `engine.create_engine()` opens the engine, and the `schema.MetaData()` constructor probes the schema of the database.
- Section 9.6, "`showTable()`: Report on one table" (p. 30).

reflector

```
    #-- 1 --
    # [ if the command line arguments are valid ->
    #     engineURL  :=  an URL that will open the database
    #                    with those arguments
    #   else ->
    #     sys.stderr  +:=  usage and error messages
    #     stop execution ]
    engineURL = checkArguments()

    #-- 2 --
    # [ if engineURL specifies a useable database connection ->
    #     engine  :=  an sqlalchemy.engine instance representing
    #                 that connection
    #     metadata  :=  an sqlalchemy.schema.MetaData instance
    #                   reflecting the schema of that database
    #   else ->
    #     sys.stderr  +:=  error message
    #     stop execution ]
    try:
        db = engine.create_engine(engineURL)
        metadata = schema.MetaData(bind=db, reflect=True)
    except exc.SQLAlchemyError, detail:
        fatal("Could not connect: %s" % detail)
```

The metadata attribute `.tables` is a dictionary whose keys are the table names, and each related value is the corresponding `Table` instance. We will present the table reports in ascending order by table name.

reflector

```
    #-- 3 --
    # [ sys.stdout  +:=  report showing the tables in metadata in
    #       ascending order by table name ]
    for tableName in sorted(metadata.tables.keys()):
```

```
        #-- 3 body --
        # [ table is an sqlalchemy.schema.Table instance ->
        #       sys.stdout  +:=  report showing the structure of table ]
        showTable(metadata.tables[tableName])
```

## 9.3. `checkArguments()`: Process the command line arguments

reflector

```
# - - -   c h e c k A r g u m e n t s

def checkArguments():
    '''Process the command line arguments.

      [ if the command line arguments are valid ->
          return those arguments as a database engine URL
        else ->
          sys.stderr  +:=  usage and error messages
          stop execution ]
    '''
```

There must be exactly five arguments. The third argument is the name of the password file, so we must try to go and read the password from that file. Errors are reported by Section 9.4, "`usage()`: Usage message" (p. 30) and Section 9.5, "`fatal()`: Fatal error message" (p. 30).

reflector

```
    #-- 1 --
    # [ if there are exactly five command line arguments ->
    #     protocol, username, passFileName, host, dbName  :=
    #         those arguments
    #   else ->
    #     sys.stderr  +:=  usage and error messages
    #     stop execution ]
    argList = sys.argv[1:]
    if len(argList) != 5:
        usage("Incorrect number of arguments.")
    protocol, username, passFileName, host, dbName = argList

    #-- 2 --
    # [ if passFileName names a readable, nonempty file ->
    #       password  :=  first line from that file, with trailing
    #                     whitespace removed
    #   else ->
    #       sys.stderr  +:=  (usage and error messages)
    #       stop execution ]
    try:
        passFile = file(passFileName)
        password = passFile.readline().rstrip()
        passFile.close()
    except IOError, detail:
        fatal("Could not open password file '%s': %s" %
              (passFileName, detail))
```

Finally, assemble the pieces into a URL using the format string defined in Section 9.1, "Prologue" (p. 27).

reflector

```
    #-- 3 --
    return(URL_FORMAT %
           (protocol, username, password, host, dbName))
```

## 9.4. `usage()`: Usage message

reflector

```
# - - -   u s a g e

def usage(*L):
    '''Write a usage message and terminate.

      [ L is a list of strings ->
          sys.stderr  +:=  (usage message) + (concatenated elements of L)
          stop execution ]
    '''
    fatal ("***Usage:\n"
        "***   %s PROTOCOL USERNAME PASSFILE HOST[:PORT] DBNAME\n"
        "*** where PROTOCOL is one of: postgresql|mysql|oracle\n"
        "***       USERNAME is your database username\n"
        "***       PASSFILE is the name of a file containing your database
 "
                        "password\n"
       "***      HOST[:PORT] is the host name and optional port number\n"

        "***      DBNAME is the name of the database\n"
        "*** Error: %s\n"
        % (sys.argv[0], ''.join(L)))
```

## 9.5. `fatal()`: Fatal error message

reflector

```
# - - -   f a t a l

def fatal(*L):
    '''Write a message and stop execution.

      [ L is a list of strings ->
          sys.stderr  +:=  the concatenated elements of L
          stop execution ]
    '''
    print >>sys.stderr, ''.join(L)
    raise SystemExit
```

## 9.6. `showTable()`: Report on one table

reflector

```
# - - -   s h o w T a b l e
```

```
def showTable(table):
    '''Display the schema for one table.

      [ table is a schema.Table instance ->
          sys.stdout  +:=  report showing the structure of table ]
    '''
```

In general, we display up to five sections for a table:

- The table's name as a heading.
- Primary key(s), if any.
- Foreign key(s), if any.
- Indices, if any.
- Descriptions of the columns.

reflector

```
    #-- 1 --
    print "\n\n================ %s ================" % table.name
```

The `.primary_key` attribute of a table is a sequence of `Column` objects; we display only the column names.

reflector

```
    #-- 2 --
    # [ if table has at least one primary key ->
    #     sys.stdout  +:=  primary key report
    #   else -> I ]
    if len(table.primary_key):
        print "Primary keys:",
        for k in table.primary_key:
            print k.name,
        print
```

Similarly, the `.foreign_keys` attribute is a sequence of `ForeignKey` instances, each of which has the local column as a `.parent` attribute, and the foreign column as a `.column` attribute.

reflector

```
    #-- 3 --
    # [ if table has any foreign keys ->
    #     sys.stdout  +:=  report on those keys
    #   else -> I ]
    if len(table.foreign_keys):
        print  "Foreign keys:"
        for fkey in table.foreign_keys:
            print "  (%s -> %s)" % (fkey.parent.name, fkey.column)
```

If the table has any indices, the `.indexes` attribute is a sequence of `Index` instances, each with its name as the `.name` attribute and its `.columns` attribute a sequence of `Column` instances.

reflector

```
    #-- 4 --
    # [ if table has an indices ->
    #     sys.stdout  +:=  report on those indices
    #   else -> I ]
    if len(table.indexes):
        print "Indices:"
        for index in table.indexes:
```

```
            print "  %s:" % index.name,
            for col in index.columns:
                print col.name,
            print
```

Column display is handled by Section 9.7, "`showColumn()`: Report on one column" (p. 32).

reflector

```
    #-- 5 --
    # [ sys.stdout  +:=  report on table's columns ]
    print "Columns [P=primary key; N=nullable; U=unique]:"
    for column in table.c:
        showColumn(column)
```

## 9.7. `showColumn()`: Report on one column

reflector

```
# - - -   s h o w C o l u m n

def showColumn(column):
    '''Display one column's definition.

      [ column is a schema.Column instance ->
          sys.stdout  +:=  report on column ]
    '''
```

To save space in the report, we represent the primary key, nullable, and unique properties as single-letter codes dispayed in brackets after the column name.

reflector

```
    #-- 1 --
    # [ if column is a primary key, nullable, or unique ->
    #     notes  :=  codes for those properties in brackets
    #   else ->
    #     noset  :=  '' ]
    L=[]
    if column.primary_key:
        L.append("P")
    if column.nullable:
        L.append("N")
    if column.unique:
        L.append("U")
    notes =("[%s]" % ''.join(L)
            if L
            else "")

    #-- 2 --
    # [ sys.stdout  +:=  column.name + notes + column.type ]
    print("  %s%s: %s" %
          (column.name, notes, column.type))
```

## 9.8. Epilogue

These lines invoke the main program, provided it is run as a script.

```
#=================================================================
# Epilogue
#-----------------------------------------------------------------
if __name__ == '__main__':
    main()
```