MySQL\_for\_Python\_Albert\_c03

Simple Insertion

The obvious complement to record retrieval is the insertion of data into a MySQL database. Data insertion is a matter of learning the syntax of the MySQL keyword for the task and applying it through MySQL for Python.

El complemento obvio a la recuperación de registros es la inserción de datos en una base de datos MySQL. La inserción de datos es cuestión de aprender la sintaxis de la palabra clave MySQL para la tarea y aplicarla a través de MySQL para Python.

As with retrieval, MySQL functions on the basis of parameter-based invocation and the returning of results in accordance with those parameters. All of this is again based on using MySQL for Python as an intermediary to that process to invoke MySQL, to log in, and to connect to our chosen database.

Al igual que con la recuperación, MySQL funciona sobre la base de la invocación basada en parámetros y la devolución de resultados de acuerdo con esos parámetros. Todo esto se basa nuevamente en el uso de MySQL para Python como intermediario de ese proceso para invocar MySQL, iniciar sesión y conectarse a nuestra base de datos elegida.

You will recall that, in Chapter 2, Simple Querying, we needed to validate user input consistently. Malformed input would have caused our program to throw an error without it. That caution goes doubly for insertion. Unqualified user input can corrupt a database and even give the malicious user access to all traffic on the server by granting him or her unwarranted administrative privileges.

Recordará que, en el Capítulo 2, Consultas simples, necesitábamos validar la entrada del usuario de manera consistente. Una entrada con formato incorrecto habría provocado que nuestro programa arrojara un error sin ella. Esa precaución se aplica doblemente a la inserción. La entrada de un usuario no calificado puede dañar una base de datos e incluso darle al usuario malicioso acceso a todo el tráfico en el servidor al otorgarle privilegios administrativos injustificados.

In this chapter, we will look at the following:

* Forming an insertion statement in MySQL
* Passing an insertion to MySQL
* User-defined variables in a MySQL insertion
* Passing metadata between databases
* Changing insertion statements dynamically, without user input

Each of these sections will be built into the project at the end of this chapter: Inserting user input into MySQL from the command-line without using the MySQL shell.

Cada una de estas secciones se incorporará al proyecto al final de este capítulo: Insertar la entrada del usuario en MySQL desde la línea de comandos sin usar el shell de MySQL.

**Forming a MySQL insertion statement**

As with record retrieval in the previous chapter, inserting data into MySQL through Python relies on understanding data insertion in MySQL itself. You will recall that the requirements of a computing language necessitate the use of as few words as possible to do anything. Ideally, there should be only one word as the Zen of Python reads:

Al igual que con la recuperación de registros en el capítulo anterior, la inserción de datos en MySQL a través de Python depende de comprender la inserción de datos en el propio MySQL. Recordará que los requisitos de un lenguaje informático requieren el uso de la menor cantidad de palabras posible para hacer cualquier cosa. Lo ideal sería que solo hubiera una palabra como dice el Zen de Python:

There should be one—and preferably only one—obvious way to do it.

Debería haber una manera obvia (y preferiblemente sólo una) de hacerlo.

For retrieval, we used the SELECT command. For putting data into the database, we use INSERT. So instead of saying "Put everything on the far table!" or "Stick everything over there!", MySQL needs specification such as:

Para la recuperación, utilizamos el comando SELECT. Para poner datos en la base de datos, utilizamos INSERT. Entonces, en lugar de decir "¡Pon todo en la mesa del fondo!" o "¡Pega todo ahí!", MySQL necesita especificaciones como:

**INSERT INTO far VALUES("everything");**

This is perhaps the most basic insertion statement that one can make for MySQL. You can tell from it that the basic syntax of MySQL's INSERT statement is as follows:

Esta es quizás la declaración de inserción más básica que se puede hacer para MySQL. Se puede ver que la sintaxis básica de la declaración INSERT de MySQL es la siguiente:

**INSERT INTO <some table> (<some column names>) VALUES("<some**

**values>");**

Now let's take this skeleton of a statement apart and see how MySQL compares to what we might use in English.

Ahora desmantelemos este esqueleto de declaración y veamos cómo se compara MySQL con lo que podríamos usar en inglés.

**INSERT**

It should be clear by now that the use of INSERT is for our benefit as humans. There is nothing special about the word other than the fact that the MySQL programmer used it. It is easier to remember, closer to being standard throughout English, and better reflects the action being called than, say, STICK. As you may know, put is currently used in other programming languages for much the same kind of functionality (for example, fputs in PHP, C, C++). The keyword consequently could have been PAPAYA if the MySQL programmers coded the database system to use that word instead of INSERT (of course, the usability of the system would have taken a sharp drop at that point). All that matters is that we use the word that the system requires in order to do the action that we desire.

A estas alturas ya debería quedar claro que el uso de INSERT es para nuestro beneficio como seres humanos. No hay nada especial en la palabra aparte del hecho de que el programador de MySQL la usó. Es más fácil de recordar, más cercano a ser estándar en todo el inglés y refleja mejor la acción que se realiza que, por ejemplo, STICK. Como sabrás, put se utiliza actualmente en otros lenguajes de programación para prácticamente el mismo tipo de funcionalidad (por ejemplo, fputs en PHP, C, C++). En consecuencia, la palabra clave podría haber sido PAPAYA si los programadores de MySQL codificaran el sistema de base de datos para usar esa palabra en lugar de INSERT (por supuesto, la usabilidad del sistema habría caído drásticamente en ese punto). Lo único que importa es que usemos la palabra que el sistema requiere para realizar la acción que deseamos.

It is worth noting that there is one other keyword that can be used for placing data into a MySQL database. REPLACE uses much the same syntax as INSERT.

Vale la pena señalar que existe otra palabra clave que se puede utilizar para colocar datos en una base de datos MySQL. REPLACE utiliza prácticamente la misma sintaxis que INSERT.

**REPLACE INTO <some table> SET("<some column name>" = "<some value>");**

As it is formed on analogy with SELECT, we will not discuss REPLACE much. However, you can read more about it on the MySQL manual page at: http://dev.mysql.com/doc/refman/5.1/en/replace.html

Como está formado por analogía con SELECT, no discutiremos mucho sobre REPLACE. Sin embargo, puede leer más al respecto en la página del manual de MySQL en: <http://dev.mysql.com/doc/refman/5.1/en/replace.html>

**INTO**

In a lot of ways, the MySQL database handles insertion like a postmaster. It will put mail anywhere you tell it as long as the box exists. So if we are going to tell MySQL to INSERT something, we must tell it where that something must go. To do that we use the complementary keyword INTO. This is the natural complement to the commands INSERT and REPLACE.

En muchos sentidos, la base de datos MySQL maneja la inserción como un administrador de correo. Pondrá el correo en cualquier lugar que usted le indique mientras exista el buzón. Entonces, si vamos a decirle a MySQL que INSERTE algo, debemos decirle dónde debe ir ese algo. Para ello utilizamos la palabra clave complementaria INTO. Este es el complemento natural de los comandos INSERTAR y REEMPLAZAR.

If you are new to computer programming, it may still seem reasonable to ask a computer to just do something. But computers are ultimately just machines, exceedingly fast and dumb. They will not reason unless they are explicitly, painstakingly, told how to reason by the programmer. They cannot guess unless told how. In the early days of modern computing, the 1970s and early 1980s, programmers would describe this dynamic of computing with the acronym GIGO—garbage in, garbage out. If you as the programmer don't tell it what to do, it won't know how to do it.

Si es nuevo en la programación de computadoras, aún puede parecer razonable pedirle a una computadora que simplemente haga algo. Pero, en última instancia, las computadoras son sólo máquinas, extremadamente rápidas y tontas. No razonarán a menos que el programador les diga explícita y minuciosamente cómo razonar. No pueden adivinar a menos que se les diga cómo. En los primeros días de la informática moderna, en los años 1970 y principios de los 1980, los programadores describían esta dinámica de la informática con el acrónimo GIGO: basura que entra, basura que sale. Si usted, como programador, no le dice qué hacer, no sabrá cómo hacerlo.

**Table name**

Python helps with this process by offering high-level handles for a lot of common functionality, but there are still limits to that automation and elements of programming for which one must assume responsibility. Where MySQL sticks your data is one of them. The table value is yours to define. If you tell MySQL the correct place to put information, all is well. If it puts it in the wrong place, chances are you are to blame (unless someone is holding a strong magnet next to the CPU at the time). If MySQL does not know what to do with your data, it will throw an error—as we will see in the next chapter.

Python ayuda con este proceso al ofrecer identificadores de alto nivel para muchas funciones comunes, pero todavía existen límites para esa automatización y elementos de programación de los cuales uno debe asumir la responsabilidad. Donde MySQL guarda sus datos es uno de ellos. El valor de la tabla es suyo para definirlo. Si le dice a MySQL el lugar correcto para colocar la información, todo estará bien. Si lo coloca en el lugar equivocado, es probable que usted tenga la culpa (a menos que alguien esté sosteniendo un imán fuerte junto a la CPU en ese momento). Si MySQL no sabe qué hacer con sus datos, generará un error, como veremos en el próximo capítulo.

**Column names**

In this part of the statement, you indicate to MySQL the order in which you will pass the values later in the statement. These are dealt with like variable names and so are not set in quotes, single or double.

En esta parte de la declaración, le indica a MySQL el orden en el que pasará los valores más adelante en la declaración. Estos se tratan como nombres de variables y, por lo tanto, no se colocan entre comillas, ni simples ni dobles.

The column names that you must address here and in the value section of the statement are determined by the nature of the database. If we use the fish database from the previous chapter, we have the following dataset:

Los nombres de las columnas que debe abordar aquí y en la sección de valor de la declaración están determinados por la naturaleza de la base de datos. Si utilizamos la base de datos de peces del capítulo anterior, tenemos el siguiente conjunto de datos:

**mysql> select \* from menu;**

+----+----------------+-------+

| id | name

| price |

+----+----------------+-------+

|1 | tuna|7.50 |

|2 | bass|6.75 |

|3 | salmon|9.50 |

|4 | catfish|5.00 |

|5 | trout|6.00 |

|6 | haddock|6.50 |

|7 | yellowfin tuna | 12.00 |

|8 | sole

|

7.75 |

+----+----------------+-------+

8 rows in set (0.00 sec)

The definitions for this dataset are purposely poor for illustrative reasons.

Las definiciones de este conjunto de datos son deliberadamente deficientes por motivos ilustrativos.

mysql> **describe menu;**

+-------+--------------+------+-----+---------+----------------+

| Field | Type

| Null | Key | Default | Extra

|

+-------+--------------+------+-----+---------+----------------+

| id| int(11)| NO| PRI | NULL| auto\_increment |

| name| varchar(30)| YES|| NULL|

|

| price | decimal(6,2) | YES|| NULL|

|

+-------+--------------+------+-----+---------+----------------+

3 rows in set (0.00 sec)

As such, the only value that is required, 'that cannot be left blank', is the value for id, the primary key. This is already set by the system because it is automatically incremented. Therefore, we can get away with the following statement:

Como tal, el único valor requerido, "que no puede dejarse en blanco", es el valor de id, la clave principal. Esto ya está configurado por el sistema porque se incrementa automáticamente. Por lo tanto, podemos salirnos con la siguiente afirmación:

mysql> **insert into menu(name) values("shark");**

You will notice that we have left off the value for the price column. The effect is that it is now set to a NULL value:

Notarás que hemos omitido el valor de la columna de precio. El efecto es que ahora se establece en un valor NULL:

mysql> **select \* from menu;**

+----+----------------+-------+

| id | name

| price |

+----+----------------+-------+

|1 | tuna|7.50 |

|2 | bass|6.75 |

|3 | salmon|9.50 |

|4 | catfish|5.00 |

|5 | trout|6.00 |

|6 | haddock|6.50 |

|7 | yellowfin tuna | 12.00 |

|8 | sole|7.75 |

|9 | shark|NULL |

+----+----------------+-------+

9 rows in set (0.00 sec)

NULL values in themselves are not bad. All computing is data and code, but both code and data must be controlled by the programmer to affect a desired, controlled result. Otherwise, errors are sure to creep in along with aberrations, and compromises in security and effectiveness will be the result.

Los valores NULL en sí mismos no son malos. Toda informática es datos y código, pero tanto el código como los datos deben ser controlados por el programador para lograr un resultado controlado y deseado. De lo contrario, es seguro que aparecerán errores junto con aberraciones, y el resultado será comprometer la seguridad y la eficacia.

Any data on which a program depends for its execution should be required. If this were an enterprise database, you would probably want this hole closed and so would define the table differently. You would want to require a non-NULL value for as many columns as are necessary to ensure the security and serviceability of your database.

Cualquier dato del que dependa un programa para su ejecución debe ser requerido. Si se tratara de una base de datos empresarial, probablemente desearía cerrar este agujero y, por lo tanto, definiría la tabla de manera diferente. Le recomendamos que requiera un valor que no sea NULL para tantas columnas como sean necesarias para garantizar la seguridad y la capacidad de servicio de su base de datos.

**VALUES**

There are two keywords that you can use to introduce the data to be inserted at this point in the INSERT statement: VALUE or VALUES. Either one is correct; both can be used with either a single value or multiple values. There needs to be no consonance between the number of values being inserted and the number aspect of the keyword.

Hay dos palabras clave que puedes utilizar para introducir los datos a insertar en este punto de la sentencia INSERT: VALOR o VALORES. Cualquiera de los dos es correcto; ambos se pueden utilizar con un valor único o con varios valores. No es necesario que haya consonancia entre la cantidad de valores que se insertan y el aspecto numérico de la palabra clave.

VALUES("<some values>", "<some more values>", "<some other values>");

is to MySQL the same as:

VALUE("<some values>", "<some more values>", "<some other values>");

just like the following two phrases of an INSERT statement are the same:

VALUE("<some values>");

VALUES("<some values>");

All this keyword slot does is introduces the values in parentheses.

**<some values>**

The values that follow the VALUES keyword must appear in the same order as the column names. Otherwise, MySQL will try to place the data in the wrong location. If you do not verify the integrity of the data passed to MySQL, the data can quickly get out of hand. Consider the effect of this statement on the table menu.

Los valores que siguen a la palabra clave VALUES deben aparecer en el mismo orden que los nombres de las columnas. De lo contrario, MySQL intentará colocar los datos en la ubicación incorrecta. Si no verifica la integridad de los datos pasados a MySQL, los datos pueden salirse de control rápidamente. Considere el efecto de esta afirmación en el menú de la mesa.

mysql> **INSERT INTO menu(name, price) VALUES("13.00", "shark");**

Because of the discord between the order of column names and the order of values, this statement tells MySQL to insert the fields name and price with the following values:

Debido a la discordia entre el orden de los nombres de las columnas y el orden de los valores, esta declaración le dice a MySQL que inserte los campos nombre y precio con los siguientes valores:

•name = 13.00

•price = shark

The problem is that these values are not allowed by the definition of the table:

El problema es que estos valores no están permitidos por la definición de la tabla:

mysql> describe menu;

+-------+--------------+------+-----+---------+----------------+

| Field | Type

| Null | Key | Default | Extra

|

+-------+--------------+------+-----+---------+----------------+

| id| int(11)| NO| PRI | NULL| auto\_increment |

| name| varchar(30)| YES|| NULL|

|

| price | decimal(6,2) | YES|| NULL|

|

+-------+--------------+------+-----+---------+----------------+

3 rows in set (0.00 sec)

[ 64 ]Chapter 3

The field name is supposed to be a thirty character string. The field price is supposed to be a decimal value with up to six numbers to the left of the decimal point and up to two to the right. So what happens when the two are mixed up to the point of utter confusion? Disaster.

Se supone que el nombre del campo es una cadena de treinta caracteres. Se supone que el campo precio es un valor decimal con hasta seis números a la izquierda del punto decimal y hasta dos a la derecha. Entonces, ¿qué sucede cuando los dos se mezclan hasta el punto de confundirse por completo? Desastre.

mysql> select \* from menu;

+----+----------------+-------+

| id | name

| price |

+----+----------------+-------+

|1 | tuna|7.50 |

|2 | bass|6.75 |

|3 | salmon|9.50 |

|4 | catfish|5.00 |

|5 | trout|6.00 |

|6 | haddock|6.50 |

|7 | yellowfin tuna | 12.00 |

|8 | sole|7.75 |

|9 | shark|NULL |

| 10 | 13.00|0.00 |

+----+----------------+-------+

10 rows in set (0.00 sec)

We get a fish called 13.00 that costs nothing! The value 13.00 can be a varchar string and is so interpreted by MySQL. However, shark cannot be interpreted as a decimal value in this context.

¡Conseguimos un pescado llamado 13.00 que no cuesta nada! El valor 13.00 puede ser una cadena varchar y así lo interpreta MySQL. Sin embargo, en este contexto, tiburón no puede interpretarse como un valor decimal.

It is worth noting that the reason shark cannot be a decimal value is because it is not defined as such. By passing it in double quotes, we indicate that it is a value, not a variable name. If, however, we had previously defined a variable shark as a decimal value, then we could use it accordingly.

Vale la pena señalar que la razón por la que tiburón no puede ser un valor decimal es porque no está definido como tal. Al pasarlo entre comillas dobles indicamos que es un valor, no un nombre de variable. Sin embargo, si previamente hubiéramos definido una variable tiburón como un valor decimal, entonces podríamos usarla en consecuencia.

Such a definition could be done in either MySQL or Python. In Python, we would use a simple variable assignment statement:

Esta definición podría realizarse en MySQL o Python. En Python, usaríamos una declaración de asignación de variable simple:

shark = 13.00

This would be truncated by Python to 13.00, but it would nonetheless preserve the integrity of the datatype (to insert 13.00, we would need to use a DECIMAL type for the column when we create the table). The second zero could later be reclaimed with a formatting convention.

Python truncaría esto a 13.00, pero aún así preservaría la integridad del tipo de datos (para insertar 13.00, necesitaríamos usar un tipo DECIMAL para la columna cuando creamos la tabla). El segundo cero podría recuperarse posteriormente mediante una convención de formato.

In MySQL, we would use the SET command. See the MySQL manual Section 8.4, for more:

<http://dev.mysql.com/doc/refman/5.1/en/user-variables.html>

**; (semicolon)**

As noted in the previous chapter, the semicolon is the line delimiter in MySQL. While necessary to indicate the end of any MySQL statement, it is not used when passing commands through MySQL for Python.

Como se señaló en el capítulo anterior, el punto y coma es el delimitador de línea en MySQL. Si bien es necesario indicar el final de cualquier declaración de MySQL, no se usa al pasar comandos a través de MySQL para Python.

**Helpful ways to nuance an INSERT statement**

Formas útiles de matizar una declaración INSERT

Like SELECT has other helpful quantifiers to weed through the data being returned, INSERT has ways of nuancing the origin of the data to be inserted as well as the timing and conditions of the insertion. The three most common ways of altering the way MySQL processes an INSERT statement are:

Al igual que SELECT tiene otros cuantificadores útiles para eliminar los datos que se devuelven, INSERT tiene formas de matizar el origen de los datos que se insertarán, así como el momento y las condiciones de la inserción. Las tres formas más comunes de alterar la forma en que MySQL procesa una declaración INSERT son:

•INSERT...SELECT...

•INSERT DELAYED...

•INSERT...ON DUPLICATE KEY UPDATE...

In the following section, we take each one in turn.

**INSERT...SELECT...**

Using INSERT...SELECT... we can tell MySQL to draw from different tables without having to draw them into Python or to set a variable in MySQL. It functions on the following syntactic template:

INSERT INTO <target table>(target column name) SELECT <source column

name> FROM <source table>;

By default, the SELECT phrase of the sentence is greedy and will return as many hits as it can. As with a generic SELECT statement, however, we can restrict the hits returned using WHERE. See the Other helpful quantifiers section in the previous chapter for more on this critical argument to SELECT.

To understand how to use this technique well, let us switch to the world database from MySQL that was mentioned in the previous chapter.

USE world;

The database has three tables. If you forget what they are, simply type:

SHOW TABLES;

You will then be rewarded with the following output:

mysql> show tables;

+-----------------+

| Tables\_in\_world |

+-----------------+

| City|

| Country|

| CountryLanguage |

+-----------------+

3 rows in set (0.00 sec)

In order to affect a statement using INSERT...SELECT..., it is necessary to understand the make-up of each database. Use DESCRIBE to get the definitions on each.

mysql> describe City;

+-------------+----------+------+-----+---------+----------------+

| Field

| Type

| Null | Key | Default | Extra

|

+-------------+----------+------+-----+---------+----------------+

| ID| int(11)

| Name| char(35) | NO

| PRI | NULL| auto\_increment |

|||

|

| NO|||

|

| District| char(20) | NO|||

|

| Population| int(11)|| 0|

|

| CountryCode | char(3)

| NO

| NO

+-------------+----------+------+-----+---------+----------------+

5 rows in set (0.00 sec)

[ 67 ]Simple Insertion

mysql> describe Country;

+----------------+-------------------------------------------------------

--------------------------------+------+-----+---------+-------+

| Field

| Type

| Null | Key | Default | Extra |

+----------------+-------------------------------------------------------

--------------------------------+------+-----+---------+-------+

| Code

| Name

| char(3)

| NO| PRI |||

| NO|||

| char(52)

|

| Continent

| enum('Asia','Europe','North America','Africa','Oceania

','Antarctica','South America') | NO

|

| Asia

|

|

| Region| char(26)

| SurfaceArea| float(10,2)

|

| NO|| 0.00||

| NO|| 0||

| YES|| NULL||

| YES|| NULL||

| YES|| NULL||

| NO||||

| NO||||

| YES|| NULL||

| YES|| NULL||

| NO||||

| int(11)

|| Population|| smallint(6)

| NULL

|

|

|| IndepYear

| YES || NO

| LifeExpectancy | float(3,1)

| GNP

| GNPOld

| LocalName

| float(10,2)

| float(10,2)

| char(45)

| GovernmentForm | char(45)

| HeadOfState

| Capital

| Code2

| char(60)

| int(11)

| char(2)

+----------------+-------------------------------------------------------

--------------------------------+------+-----+---------+-------+

15 rows in set (0.01 sec)

[ 68 ]Chapter 3

mysql> describe CountryLanguage;

+-------------+---------------+------+-----+---------+-------+

| Field

| Type

| Null | Key | Default | Extra |

+-------------+---------------+------+-----+---------+-------+

| CountryCode | char(3)| NO| PRI |||

| Language| char(30)| NO| PRI |||

| IsOfficial| enum('T','F') | NO|| F||

| Percentage| float(4,1)|| 0.0||

| NO

+-------------+---------------+------+-----+---------+-------+

4 rows in set (0.01 sec)

INSERT...SELECT... allows us to draw from each of the tables to form a new one. Let's say we wanted a table Combo that operated off the same identifier as City and incorporated the names for the first 999 countries listed in that database. We would begin by creating a MySQL table for the task. Creating a MySQL table is addressed in a later chapter, so here we assume the existence of a table Combo with the following definition:

+-------------+----------+------+-----+---------+----------------+

| Field

| Type

| Null | Key | Default | Extra

|

+-------------+----------+------+-----+---------+----------------+

| ID| int(11)

| Name| char(35) | NO

| CountryCode | char(3)

| NO

| NO

| PRI | NULL| auto\_increment |

|| NULL|

|

|| NULL|

|

+-------------+----------+------+-----+---------+----------------+

Having done that we can insert the desired data from City into Combo using the following INSERT command:

INSERT INTO Combo(ID, Name, CountryCode) SELECT ID, Name, CountryCode

FROM City WHERE ID < 1000;

A SELECT command to the database then shows the effect. For the sake of space, let's restrict ID to 10.

mysql> SELECT \* FROM Combo WHERE ID<=10;

+----+----------------+-------------+

| ID | Name

| CountryCode |

+----+----------------+-------------+

| 1 | Kabul

| AFG

|

| 2 | Qandahar

| AFG

|

| 3 | Herat

| AFG

|

| 4 | Mazar-e-Sharif | AFG

|

| 5 | Amsterdam

| NLD

|

| 6 | Rotterdam

| NLD

|

| 7 | Haag

| NLD

|

| 8 | Utrecht

| NLD

|

| 9 | Eindhoven

| NLD

|

| 10 | Tilburg

| NLD

|

+----+----------------+-------------+

10 rows in set (0.00 sec)

This significantly cuts down on I/O and therefore dramatically reduces processing time—whether perceived or real. It lightens the load on the network and makes it appear more responsive and able to handle more requests (all other dynamics being equal).

Note that even if your program is run on the same system as the database being queried, you will still have the dynamics of a network and therefore suffer lag if your program passes too many requests to MySQL too quickly.

Sluggishness on many systems is due to excessive data transfer between processes, not because of the speed at which those processes are executed.

More information on the INSERT...SELECT... functionality can be found in the MySQL manual at:

http://dev.mysql.com/doc/refman/5.1/en/insert-select.html

**INSERT DELAYED…**

The DELAYED argument to INSERT causes MySQL to handle the insertion in deference to other MySQL processes. When the server is sufficiently quiet, the INSERT command is executed. Until then, MySQL keeps it on hold.

The DELAYED argument simply follows the INSERT command. Otherwise, the syntax is the same:

INSERT DELAYED INTO <some table> (<some column names>) VALUES("<some

values>");

For finer details on the DELAYED argument to INSERT, see the MySQL manual at http://dev.mysql.com/doc/refman/5.1/en/insert-delayed.html

**INSERT...ON DUPLICATE KEY UPDATE...**

Whenever you insert a record into a large table, there is a chance of creating an identical record. If your INSERT statement would result in two identical records, MySQL will throw an error and refuse to create the record. The error you get will look something like this:

ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'

To mitigate against this error and the chance of submitted data not being inserted properly, MySQL offers this further argument to the INSERT command. The syntax is as follows:

INSERT INTO <some table>(<some column names>) VALUES ("<some values>")

ON DUPLICATE KEY UPDATE <change to make the data unique>

After UPDATE, simply include what you have MySQL do to the record that you would insert in order to ensure that it is no longer a duplicate. In practice, this means incrementing the Primary key identifier. So where we get an error with one statement, we can adapt the statement. In the following statement, we get an error due to a duplicate ID number:

mysql> INSERT INTO Combo(ID, Name, CountryCode) VALUES ("27",

"Singapore", "SGP");

ERROR 1062 (23000): Duplicate entry '27' for key 'PRIMARY'

Using the ON DUPLICATE KEY UPDATE... argument, we can insert the value and ensure that the record is unique:

mysql> INSERT INTO Combo(ID, Name, CountryCode) VALUES ("4078",

"Singapore", "SGP") ON DUPLICATE KEY UPDATE ID=ID+1;

Query OK, 1 row affected (0.00 sec)

Note that if there is no conflict in values, MySQL will process the statement as if you did not include the ON DUPLICATE KEY UPDATE... clause.

If we then run a quantified SELECT statement against the table, we see that we now have two unique records for Singapore:

mysql> select \* from Combo WHERE Name="Singapore";

+------+-----------+-------------+

| ID

| Name

| CountryCode |

+------+-----------+-------------+

| 3208 | Singapore | SGP|

| 4078 | Singapore | SGP|

+------+-----------+-------------+

2 rows in set (0.00 sec)

**Passing an insertion through MySQL for Python**

As you can see, inserting data into MySQL is a straightforward process that is largely based around ensuring that the database daemon knows where you want your data placed. Inserting data into MySQL may seem a bit more complicated than retrieving it but the previous discussion shows it is still logical, but just requires a few more keywords in order to be useful.

**Setting up the preliminaries**

Using INSERT with MySQL for Python is just as easy as using SELECT. As we saw in the previous chapter, we pass the command to MySQL using the execute() method of the database cursor object.

We will again use the fish database and the menu table as follows:

+----+----------------+-------+

| id | name

| price |

+----+----------------+-------+

|1 | tuna|7.50 |

|2 | bass|6.75 |

|3 | salmon|9.50 |

|4 | catfish|5.00 |

|5 | trout|6.00 |

|6 | haddock|6.50 |

|7 | yellowfin tuna | 12.00 |

+----+----------------+-------+

[ 72 ]Chapter 3

Once again, we need to set up the database objects in our Python session. If you are using the same Python terminal session as you did for the previous chapter, you may want to go through this process anyway to ensure that all names are set for the examples to come. Alternatively, close the session by pressing Ctrl+D and initiate a new one. Then import MySQL for Python, tend to the database login and create the cursor object as follows:

import MySQLdb

mydb = MySQLdb.connect(host = 'localhost',

user = 'skipper',

passwd = 'mysecret',

db = 'fish')

cur = mydb.cursor()

Now we are ready to insert data using Python.

**A simple INSERT statement**

Inserting data through MySQL for Python uses the same method as retrieving it using execute(). You will recall that data retrieval using MySQL for Python follows this formula:

results\_variable = cursor\_handle.execute('MySQL SELECT statement')

And so one gets a Python statement that looks like the following:

command = cur.execute('SELECT \* FROM menu')

The main difference in data insertion is that no values are being returned. Therefore, because we are not retrieving any data, we do not need to assign the value of returning data to a variable. Instead, we pass the insertion command as a stand-alone command.

The basic system call for the insertion command would follow this template:

cursor\_handle.execute('MySQL INSERT statement')

Using this template, we can pass the following MySQL INSERT statement:

INSERT INTO menu(name, price) VALUES("shark", "13.00");

Without worrying about validating the integrity of the data for the moment, we insert this statement through MySQL for Python as follows (using the cursor object cur as defined previously):

cur.execute("""INSERT INTO menu(name, price) VALUES("shark",

"13.00")""")

Of course, as with the SELECT statement in the previous chapter, this statement can become difficult to control rather quickly because of the number of quotation marks and parentheses. If this proves difficult to follow for you, simply break the statement down by defining the argument for execute(), the actual MySQL statement, in a separate line. As with elsewhere in Python, you can use triple quotes to assign a value verbatim. The preceding call could then be rewritten as follows:

statement = """INSERT INTO menu(name, price) VALUES("shark",

"13.00")"""

cur.execute(statement)

Using triple quotes is also helpful for handling more complex statements as they can bridge multiple lines. This makes it easier to format statements in a way that humans can read more easily. Therefore, to use the ON DUPLICATE KEY UPDATE... example from earlier in this chapter, we can define the statement:

INSERT INTO Combo(ID, Name, CountryCode) VALUES ("4078", "Singapore",

"SGP") ON DUPLICATE KEY UPDATE ID=ID+1;

As follows for better readability:

statement = """INSERT INTO Combo(ID, Name, CountryCode)

VALUES ("4078", "Singapore", "SGP")

ON DUPLICATE KEY UPDATE ID=ID+1;"""

As the Zen of Python reads:

Readability counts

The virtue of readability in programming is often couched in terms of being kind to the next developer who works on your code. There is more at stake, however. With readability comes not only maintainability but control.. If it takes you too much effort to understand the code you have written, you will have a harder time controlling the program's flow and this will result in unintended behavior. The natural consequence of unintended program behavior is the compromising of process stability and system security.

If this is still too complex for you to follow with ease, it may be advisable to rework the value of statement by employing string formatting techniques as shown later in the chapter under the heading Using user-defined variables.

**More complex INSERT commands**

To pass the INSERT command with any of its optional arguments, simply include them in the statement. For example, where we had the following INSERT…

SELECT... command:

INSERT INTO Combo(ID, Name, CountryCode) SELECT ID, Name, CountryCode

FROM City WHERE ID < 1000;

One can simply pack all of that into the value of statement:

statement = """INSERT INTO Combo(ID, Name, CountryCode) SELECT ID,

Name, CountryCode FROM City WHERE ID < 1000;"""

The DELAYED argument can be passed similarly. The previous statement passed through execute() would look like this:

cur.execute("""INSERT DELAYED INTO Combo(ID, Name, CountryCode) SELECT

ID, Name, CountryCode FROM City WHERE ID < 1000;""")

Likewise, we could include the INSERT...ON DUPLICATE KEY UPDATE... argument as follows:

cur.execute("""INSERT INTO Combo(ID, Name, CountryCode) VALUES

("4078", "Singapore", "SGP") ON DUPLICATE KEY UPDATE ID=ID+1""")

It is not necessary to use triple quote marks when assigning the MySQL sentence to statement or when passing it to execute(). However, if you used only a single pair of either double or single quotes, it would be necessary to escape every similar quote mark. As a stylistic rule, it is typically best to switch to verbatim mode with the triple quote marks in

order to ensure the readability of your code.

**Using user-defined variables**

Just as in data retrieval, it is inevitable that you will want to utilize user input when inserting data into MySQL. MySQL for Python provides a consistent, Pythonic interface for this.

We use the same string conversion specifier as we did when incorporating user input into our SELECT statements in the previous chapter. Using the fish database, if we assume that the user gives us the name of the fish and the cost, we can code a user-defined INSERT statement as follows:

import MySQLdb, sys

mydb = MySQLdb.connect(host = 'localhost',

user = 'skipper',

passwd = 'secret',

db = 'fish')

cur = mydb.cursor()

fish = sys.argv[1]

price = sys.argv[2]

statement = """INSERT INTO menu(name, price) VALUES(%s, %s)""" %(fish,

price)

cur.execute(statement)

An alternative way of rendering the last two lines is to leave the value insertion to the execute() function. Instead of using %(fish, price) at the end of the first of the two lines, we can include the fish and price values as a second argument to execute():

statement = "INSERT INTO menu(name, price) VALUES (%s, %s)"

cur.execute(statement, (fish, price))

To make this program executable, you can preface this code with a shebang line, make the file executable (by changing the permissions on the file), and then call it as you would any other local executable that is not in your execution path. Alternatively, you can call it from the command-line by prefacing it with a call to your local Python interpreter. In either case, don't forget to supply the arguments for sys.argv[]. Here I have run it using the latter method:

python ./user-defined-data.py angel 7.00

This then appends the data to the database in real time.

mysql> SELECT \* FROM menu;

+----+----------------+-------+

| id | name

| price |

+----+----------------+-------+

|1 | tuna|7.50 |

|2 | bass|6.75 |

|3 | salmon|9.50 |

|4 | catfish|5.00 |

|5 | trout|6.00 |

|6 | haddock|6.50 |

|7 | yellowfin tuna | 12.00 |

|8 | sole|7.75 |

|9 | angel|7.00 |

+----+----------------+-------+

9 rows in set (0.01 sec)

As this is all within the Python API, you are not limited merely to %s, but can use the same string formatting techniques as you would anywhere else in Python.

**Using metadata**

On February 23, 2006, an American B-2 bomber crashed shortly after take-off in Guam due to bad data being fed to the airplane's flight control computers. A lack of data checking resulted in the loss of a $2.1 billion plane. As with any user interaction in programming, it is foolish to trust data without validating its integrity first.

One of the main ways of validating user input is to verify the data definition for the database. More often than not, the database definition will be known at the time of application development. You can then verify user input against a known specification. However, if you do not have this luxury, you will need to query the database for its definition and ensure the user's data does not run afoul of it.

**Querying the database for its structure**

If we are completely ignorant of a database's structure, we need to first retrieve a table listing. To affect that, we use SHOW TABLES.

statement = """SHOW TABLES"""

command = cur.execute(statement)

Be sure to follow the execute() call with fetchall() assigned to a variable to hold the tuple that is returned.

results = cur.fetchall()

The tuple results can then be accessed to give the user a choice.

print "Which table would you like to use?"

for i in xrange(0, len(results)): print i+1, results[i][0]

choice = input("Input number:")

As fish only has one table, the output of the for loop would simply be:

1 menu

But if we do the same for the world database, we get a more realistic selection:

1 City

2 Combo

3 Country

4 CountryLanguage

The user can then choose from the list. If we want to verify the user's data, we need to verify three things:

1. The value input by the user is only a number.

2. The numeric value is not outside the offered range.

3. The value is a whole number.

To validate the input as a number, we need to import the string module and use the isdigit() method of string objects.

import string

We would then use an if statement along the following lines:

if choice.isdigit() is True:

print "We have a number!" ## or do further checking

[ 78 ]Chapter 3

We then need to confirm that the input is within the given range. To verify that the value of results is greater than 0 but not greater than the number of given options:

if (choice<0) or (choice>len(results)):

print "We need a new number!" ## or do further checking

Within the previous range, however, we still run into problems with decimals. We currently have no protection against choice being equal to 3.5, for example. There are a couple of ways that we can protect against this at the validation stage:

•By checking the length of the input and telling the user that we need a single digit within the given range

•By stripping out all but the first digit and returning the results to the user for confirmation

To check the length of the input, we simply use Python's built-in len() function in a conditional loop:

if len(choicea) != 1:

print "We need a single digit within the given range, please."

This, however, is not the most user-friendly way to handle the data. In cases where there are a lot of choices, it can even leave the user confused. Better is to offer an alternative by way of confirmation. To do this, we convert the input to a string using Python's built-in str() function and then present the first element of the indexed string to the user for confirmation.

choice\_string = str(choice)

confirm = input("Did you mean %s?" %(choice\_string[0]))

If confirm is assigned a positive value by the user—whether it is yes, true, or just 1, we should then convert the value of choice\_string[0] to an integer. We do this with Python's built-in int() function.

real\_choice = int(choice\_string[0])

This has the benefit of handling input from users who either have poor typing skills or who may otherwise input gobbledygook after their initial, valid selection.

**Retrieving the table structure**

After validating the user's input, we have the choice of database to be used. We now need to give the user details on the fields being used by that table. Again, we use DESCRIBE.

table\_statement = """DESCRIBE %s""" %(results[real\_choice-1][0])

cur.execute(table\_statement)

table\_desc = cur.fetchall()

It is worth noting here that indices start at 0 but our choices to the user started at 1. Therefore, whatever choice the user makes must be reduced by one in order to synchronize it with the index of results.

Also, we do not want to pass the value of the entire tuple in the statement. We just want the value of the table to be queried. Therefore, we must subscript the results record with a 0.

In MySQL, the DESCRIBE statement returns a table. In MySQL for Python, we get another tuple. Each element in that tuple is a row in the table returned by MySQL.

So where MySQL would return the following.

mysql> DESCRIBE CountryLanguage;

+-------------+---------------+------+-----+---------+-------+

| Field

| Type

| Null | Key | Default | Extra |

+-------------+---------------+------+-----+---------+-------+

| CountryCode | char(3)| NO| PRI |||

| Language| char(30)| NO| PRI |||

| IsOfficial| enum('T','F') | NO|| F||

| Percentage| float(4,1)|| 0.0||

| NO

+-------------+---------------+------+-----+---------+-------+

4 rows in set (0.00 sec)

A prettified version of what Python returns is the following:

>>> for i in xrange(0, len(table\_desc)): print table\_desc[i]

...

('CountryCode', 'char(3)', 'NO', 'PRI', '', '')

('Language', 'char(30)', 'NO', 'PRI', '', '')

('IsOfficial', "enum('T','F')", 'NO', '', 'F', '')

('Percentage', 'float(4,1)', 'NO', '', '0.0', '')

[ 80 ]Chapter 3

The differences between xrange() and range() are often overlooked or even ignored. Both count through the same values, but they do it differently. Where range() calculates a list the first time it is called and then stores it in memory, xrange() creates an immutable sequence that returns the next in the series each time it is called. As a consequence, xrange() is much more memory efficient than range(), especially when dealing with large groups of integers. As a consequence of its memory efficiency, however, it does not support functionality such as slicing, which range() does, because the series is not yet fully determined.

Each element of each row is then available by a further subscript for the column that you want to access:

>>> print table\_desc[0][0]

CountryCode

>>> print table\_desc[0][1]

char(3)

>>> print table\_desc[1][3]

PRI

So to offer the user the format of the table columns, we could use the following code:

print "The records of table %s follow this format:" %(results[choice-

1][0])

for i in xrange(0, len(table\_desc)):

print table\_desc[i][0]

The output is as follows:

The records of table CountryLanguage follow this format:

CountryCode

Language

IsOfficial

Percentage

[ 81 ]Simple Insertion

We can also walk through this data to give the user the format of each field for each record in the table. The fields of information for each field in any MySQL table remains constant and follow this order:

•Field: The name of the column

•Type: The data type allowed in that column, along with its length

•Null: Whether a null value is allowed

•Key: Whether this value is a primary key for the table

•Default: What the default value is, if no value is entered as input for this column

•Extra: Any additional information about this column

To access a particular column, one simply appends the appropriate column number as a subscript, as shown previously.

Knowing this, one can code in helps options for each field in turn. This can be a JavaScript pop-up or a manual page for each column. For the sake of space, however, this is left here as an exercise for yourself.

**Changing insertion values dynamically**

Just because user input is not valid, it does not mean we should scrap it and ask the user for a new input. Rather, we can accept an entire statement, assign what values will fit and come back to the user to correct data that will not.

To do this, we need to import the string module and define three functions to do the following:

1. Validate the name column.

2. Validate the price column.

3. Query the user for a correction.

After defining the functions, we pass the user values to the first two functions, we then pass the user values in turn to the first two functions which then calls the third function only if the data does not check out.

Simply append string to the import line for the program or add the following line below it:

import string

Now, let's define the three functions.

**Validating the value of name**

This function needs to do the following:

1. Receive the value of the name column.

2. Check whether it is all letters—for example, no fish has numbers in its

market name.

3. Call query() if necessary.

4. Return the name value to the main program, corrected if necessary.

The first is accomplished with the definition line:

def valid\_name(name):

We then accomplish the rest of the specified functionality with an if...else statement:

if name.isalpha() is False:

fish = query("name", name, "alpha")

else:

fish = name

Finally, we return the value of fish to the main program:

return(fish)

**Validating the value of price**

Validating the value of price requires similar functionality. Here is the function that we need for the task:

def valid\_price(price):

if price.isdigit() is False:

price = query("price", price, "digit")

else:

price = price

return(price)

[ 83 ]Simple Insertion

**Querying the user for a correction**

As you can tell from the calls in the preceding functions, this function will take three arguments and query the user for a correction according to them. Our definition thus begins:

def query(column, value, kind):

For kind, we will use two different possible values: alpha and digit. Depending on which one is used, this function will behave differently. For validating the alpha character of name value, we use the following if... clause:

if kind == "alpha":

print "For %s, you input %s. This is not a valid value for

column %s. Please enter the name of the fish in the appropriate

format." %(column, value, column)

new\_value = raw\_input("New name: ")

new\_value = valid\_name(new\_value)

return (new\_value)

If type is not alpha but digit, we use an elif clause to continue with the user query:

elif kind == "digit":

print "For %s, you input %s. This is not a valid price.

Please enter the price in the appropriate format." %(column, value)

new\_price = raw\_input("New price: ")

new\_price = valid\_price(new\_price)

return (new\_price)

Finally, because this function interacts with the user, we want to ensure that it cannot be called from another program or with any other values for type other than alpha or digital. To affect this in the shortest amount of code possible, we use a simple else statement.

else:

return -1

We return the value -1 here, in an effort to ensure that the erroneous call does not go unnoticed.

**Passing fish and price for validation**

Having defined the three functions, we now need to call them and to pass to them the values for fish and price. We therefore put this code just after assigning the values of sys.argv[1] and sys.argv[2] to fish and price, respectively.

fish = valid\_name(fish)

price = valid\_price(price)

**Essentials: close and commit**

In programs that interface with multiple databases or otherwise persist beyond the database connection that you have initiated, you will find a need to use a couple of MySQL commands that we have not yet discussed: close and commit.

**In need of some closure**

When one is finished with a database, it is good practice to close the cursor proxy. This ensures the cursor is not used again to refer to that database connection and also frees up resources. To close a cursor connection in MySQL for Python, simply issue the method call to your cursor object:

cur.close()

**What happened to commit?**

If you are experienced with using the MySQL shell or perhaps programming interfaces with MySQL using different APIs, you may wonder what has happened to the commit call that one normally would make at the end of every transaction to render changes permanent.

MySQL for Python ships with an autocommit feature. Therefore, when the connection is closed, the changes are committed. However, if you are programming to several databases and want to ensure one is closed before another is opened, MySQL for Python still supports a commit() function. You simply call it with the handle of the database.

mydb.commit()

After committing the changes to the database, one typically closes the database connection. To do this, use the database object's close() method:

mydb.close()

**Why are these essentials non-essential?**

Unless you are running several database threads at a time or have to deal with similar complexity, MySQL for Python does not require you to use either commit() or close(). Generally speaking, MySQL for Python installs with an autocommit feature switched on. It thus takes care of committing the changes for you when the cursor object is destroyed.

Similarly, when the program terminates, Python tends to close the cursor and

database connection as it destroys both objects.

**Project: A command-line insertion utility**

We can now put together the elements of database programming that we have covered in this chapter to form a robust command-line insertion utility. For this project we want to create a program with the following functionality:

•Runs from the command-line

•Uses a flag system allowing for the -h flag for help

•Allows the user to define the database being used

•Allows the user to designate which user and password combination to use

•Allows the user to ask for the tables available in a given database

•Provides the user with the column structure of the table on demand

•Validates user input for the given table of the selected database

•Builds the database INSERT statement on-the-fly

•Inserts the user input into the chosen table of the selected database

**The necessary modules**

Before we jump into coding, let us first assess which modules we need to import. The modules we need are listed next to our required functionality as follows. The need for MySQLdb is understood.

•Flag system: optparse

•Login details: getpass

•Build the INSERT statement: string

Our import statement thus looks like this:

import getpass, MySQLdb, optparse, string

In addition to these, we will also use the PrettyTable module to provide the user with the column structure of the table in a neat format. This module is not part of the standard library, but is easily installed using the following invocation from the command-line:

easy\_install prettytable

If this does not work for you or you prefer to install the module manually, you will benefit from the instructions at the PrettyTable site: <http://code.google.com/p/> prettytable/wiki/Installation

If you prefer not to install PrettyTable, you will obviously want to modify the code according to your preferences when we get to printing out the database definition table to the user.

**The main() thing**

In this project, we will have several functions. In order to ensure that we can call those functions from other programs, we will code this project using a main function. The other functions will be inserted before the main() function in the program, but starting with the main() function and coding the others as needed helps us to keep from losing the plot of the program. So let's define main():

def main():

In any size of program, using a main() function is good practice and results in a high degree of readability. Ideally, main() should be among the smallest of the functions in a program. The point is that main() should be the brains of the program that coordinates the activity of the

classes and functions.

From here, the flow of the main() function will follow this logic:

1. Set up the flag system.

2. Test the values passed by the user.

3. Try to establish a database connection.

4. If successful, show the user the tables of the designated database.

5. Offer to show the table structure, and then do so.

6. Accept user input for the INSERT statement, column-by-column.

7. Build the INSERT statement from the user input and execute it.

8. Print the INSERT statement to the user for feedback.

9. Commit changes and close all connections.

[ 87 ]Simple Insertion

**Coding the flag system**

As we did in the previous chapter, we need to tell Python which flags should be supported and to which variables the values should be assigned. The code looks like this:

opt = optparse.OptionParser()

opt.add\_option("-d", "--database", action="store", type="string",

dest="database")

opt.add\_option("-p", "--passwd", action="store", type="string",

dest="passwd")

opt.add\_option("-u", "--user", action="store", type="string",

dest="user")

opt, args = opt.parse\_args()

If you don't understand this, see the relevant section under the project listing from the previous chapter. For simplicity's sake, we then pass the values to simpler variable names:

database = opt.database

passwd = opt.passwd

user = opt.user

If you have trouble with the program after you code it, here is a point for blackboxing. Simply insert the following loop to show what the computer is thinking at this point:

for i in (database, passwd, user): print "'%s'" %(i)

Blackboxing is jargon in the IT industry and simply means to isolate the parts of a problem so that each piece can be tested separately of the others. With this for loop, we can ensure that Python has properly assimilated the flagged input from the user.

**Testing the values passed by the user**

Next, we need to ensure that the user has not passed us empty or no data. If the user has, we need to ask for a new value.

while (user == "") or (user == None):

print "This system is secured against anonymous logins."

user = getpass.getuser()

while (passwd == "") or (passwd == None):

print "You must have a valid password to log into the

database."

passwd = getpass.getpass()

while (database == "") or (database == None):

database = raw\_input("We need the name of an existing database

to proceed. Please enter it here: ")

Note that we are not using if. If we had, we would have needed to set up a loop to consistently check the value of the data. Using while saves us the trouble.

**Try to establish a database connection**

Having checked the login data, we can now attempt a connection. Just because the user data has checked out does not mean that the data is valid. It merely means that it fits with our expectations. The data is not valid until the database connection is made. Until then, there is a chance of failure. We therefore should use a try...

except... structure.

try:

mydb = MySQLdb.connect(host = 'localhost',

user = user,

passwd = passwd,

db = database)

cur = mydb.cursor()

quit = 1

except:

print "The login credentials you entered are not valid for

the database you indicated. Please check your login details and try

again."

quit = 0

Here we use quit as a token to indicate the success of the connection. One could just as easily use connected or is\_connected. A successful connection is not made until a cursor object is created.

Within the except clause, it is important to tell the user why the program is going to terminate. Otherwise, he or she is left in the dark, and the program can effectively become useless to them.

**Showing the tables**

Next, we cull out the tables from the database and show them to the user. We only do this if a successful connection has been made.

if quit == 1:

get\_tables\_statement = """SHOW TABLES"""

cur.execute(get\_tables\_statement)

tables = cur.fetchall()

print "The tables available for database %s follow below:"

%(database)

for i in xrange(0, len(tables)):

print "%s. %s" %(i+1, tables[i])

table\_choice = raw\_input("Please enter the number of the table

into which you would like to insert data. ")

For the sake of formatting, we increment the number of the table by one in order to use the natural number system when presenting the options to the user.

Upon receiving the number of table\_choice from the user, we must validate it. To do so, we stringify the number and pass it to a function valid\_table(), which we will create later in the development process. For now, it is enough to know that the function needs the user's choice and the number of tables in the designated database.

For simplicity, we pass the list of tables.

table\_choice = str(table\_choice)

table\_no = valid\_table(table\_choice, tables)

Once the number chosen is validated, we must decrement the number to synchronise it with the whole number system used by Python.

table = tables[table\_no-1][0]

**Showing the table structure, if desired**

The next step is to show the user the data structure of the table, if desired. We affect this with a raw\_input statement and an if... clause:

show\_def = raw\_input("Would you like to see the database

structure of the table '%s'? (y/n) " %(table))

Before launching into the if... statement, , we can economize on our code. Regardless of whether the user wants to see the table format, we will need the column headers later to affect the insertion. We can take care of retrieving them now so that the information is available in the if... statement as well as out, both for the price of one MySQL statement.

def\_statement = """DESCRIBE %s""" %(table)

cur.execute(def\_statement)

definition = cur.fetchall()

If the user chooses y to the input at show\_def, then we run the following if loop:

if show\_def == "y":

from prettytable import PrettyTable

tabledef = PrettyTable()

tabledef.set\_field\_names(["Field", "Type", "Null", "Key",

"Default", "Extra"])

for j in xrange(0, len(definition)):

tabledef.add\_row([definition[j][0], definition[j][1],

definition[j][2], definition[j][3], definition[j][4],

definition[j][5]])

tabledef.printt()

As mentioned when discussing the modules for this project, here we import PrettyTable from the module prettytable. This merely allows us to output a nicely formatted table similar to MySQL's own. It is not required for the program to work as long as you convey the value of the six tabular fields for each row. While this example is quite serviceable, you can find more information on how to use the PrettyTable module at: http://code.google.com/p/prettytable/

Note that, if show\_def equals anything other than a simple y, the if loop will not execute.

**Accepting user input for the INSERT statement**

We next need to ask the user for the values to be inserted. To guide the user, we will prompt them for the value of each column in turn:

print "Please enter the data you would like to insert into

table %s" %(table)

columns = []

values = []

for j in xrange(0, len(definition)):

column = definition[j][0]

[ 91 ]Simple Insertion

value = raw\_input("Value to insert for column '%s'?"

%(definition[j][0]))

columns.append(str(column))

values.append('"' + str(value) + '"')

columns = ','.join(columns)

values = ','.join(values)

print columns

print values

The lists columns and values obviously correspond to the respective parts of the MySQL INSERT statement. It is important to remember that the column headers in a MySQL statement are not set in quotes, but that the values are. Therefore, we must format the two lists differently. In either case, however, the items need to be separated by commas. This will make it easier when building the next INSERT statement.

If you encounter difficulty in coding this project, this is another good point for blackboxing. Simply print the value of each list after the close of the for loop to see the value of each at this point.

**Building the INSERT statement from the user input and executing it**

Having the user's values for insertion, we are now at a point where we can build the MySQL statement. We do this with string formatting characters.

statement = """INSERT INTO %s(%s) VALUES(%s)""" %(table,

columns, values)

We then execute the statement. For extra security against malformed data, you could couch this in a try...except... structure.

cur.execute(statement)

If the execute statement was processed without a problem, it is a good idea to give the user some feedback. An appropriate output here would be the statement that was processed.

print "Data has been inserted using the following statement:

\n", statement

[ 92 ]Chapter 3

**Committing changes and closing the connection**

Finally, we can commit the changes and close the connection. This is not typically necessary for a single run database program such as this, but it is not a bad habit to maintain.

cur.close()

mydb.commit()

mydb.close()

It is worth noting that committing before closing is not wholly necessary. The one implies the other. However, commit() allows us to commit changes to the database without closing the database connection, so we can commit changes at regular intervals.

**Coding the other functions**

We are not done yet. With main() finished, we now have to fill out the program with the auxiliary functions that were called in the course of main().

•valid\_digit()

•valid\_string()

•valid\_table()

•query()

The other functions must be inserted before the main() function is called, otherwise Python will throw a NameError.

**valid\_digit() and valid\_string()**

The first three functions should validate input as: digital, alpha character string, or as a valid number on the table menu presented to the user. If the input does not check out, then each should call query() to ask the user for new input, providing the type of valid input required as an argument with the value input by the user. The first two require only the value to be validated. They therefore look like this:

def valid\_digit(value):

if value.isdigit() is not True:

value = query(value, "digit")

else:

value = value

return value

[ 93 ]Simple Insertion

def valid\_string(value):

if value.isalpha() is not True:

value = query(name, "alpha")

else:

value = value

return value

**valid\_table()**

To validate the table selection, you will remember that we passed the number of the user's selection and the table names to valid\_table(). This function compares the user's selection to the number of tables available and calls query() if there is a problem. The function is therefore coded as follows:

def valid\_table(choice, tables):

valid\_choice = valid\_digit(choice) # Check whether the choice is a

valid number

valid\_choice = int(valid\_choice)

while 0 <= valid\_choice <= len(tables) :

# Ensure the choice is

among the valid options

print "Your selection is outside the bounds of possible

choices."

valid\_choice = query(valid\_choice, "digit")

return valid\_choice

**query()**

The function query() uses an if...elif...else structure to alert the user to the malformed data, asks for new input, and validates it. It therefore calls valid\_digit() and valid\_string() as necessary for the last task. Querying and testing is of two kinds: digit and alpha. Within the present program, only these two are called, but there is a chance that this function could be called wrongly from another Python program. Therefore, we try to fail softly by returning 1 if the wrong argument is passed for type.

def query(value, type):

if type == "alpha":

print "The value you entered ('%s') is not correct.

enter a valid value." %(value)

new\_value = raw\_input("New value: ")

valid\_string(new\_value)

return new\_value

elif type == "digit":

[ 94 ]

PleaseChapter 3

print "The value you entered ('%s') is not correct.

enter a valid value." %(value)

new\_value = raw\_input("New value: ")

valid\_digit(new\_value)

return new\_value

Please

else:

## if type is neither "alpha" nor "digit"

return 1

**Calling main()**

As the program stands now, it does nothing. It is simply a bunch of functions. While they call each other, there is nothing that instigates the execution process if the file is fed to the Python interpreter. We need to call main().

As every Python program can also function as a module, it is necessary for us to ensure that main() is only executed if the program is called directly. Now that all the functions are defined, we can use an if statement to verify how the program is called and to launch main() accordingly.

if \_\_name\_\_ == '\_\_main\_\_':

main()

Calling main() as a result of this if statement at the very end of the program is like not connecting the power to an electric circuit until you are ready to test it. It helps us to avoid lots of possible problems.

Now the program can be called at the command-line. You can either give it a shebang line at the top and ensure the file is set for execution, or call it as an argument to the Python interpreter:

python ./project-ch3.py -d world -u skipper -p secret

For reference, the full code of the program to follow is here. First, the shebang line and the imported modules:

#!/usr/bin/env python

import getpass, MySQLdb, optparse, string

[ 95 ]Simple Insertion

Next, we define all supporting functions:

def valid\_digit(value):

if value.isdigit() is not True:

value = query(value, "digit")

else:

value = value

return value

def valid\_string(value):

if value.isalpha() is not True:

value = query(name, "alpha")

else:

value = value

return value

def query(value, type):

if type == "alpha":

print "The value you entered ('%s') is not correct.

enter a valid value." %(value)

new\_value = raw\_input("New value: ")

valid\_string(new\_value)

return new\_value

elif type == "digit":

print "The value you entered ('%s') is not correct.

enter a valid value." %(value)

new\_value = raw\_input("New value: ")

valid\_digit(new\_value)

return new\_value

Please

Please

else:

## if type != "alpha" and type != "digit":

return 1

def valid\_table(choice, tables):

valid\_choice = valid\_digit(choice) # Check whether the choice is a

valid number

valid\_choice = int(valid\_choice)

while (valid\_choice <= 0) or (valid\_choice > len(tables)):

#

Ensure the choice is among the valid options

print "Your selection is outside the bounds of possible

choices."

valid\_choice = query(valid\_choice, "digit")

return valid\_choice

[ 96 ]Chapter 3

We then need to define main(), the master function. First we parse the options and assign their values:

def main():

opt = optparse.OptionParser()

opt.add\_option("-d", "--database", action="store", type="string",

dest="database")

opt.add\_option("-p", "--passwd", action="store", type="string",

dest="passwd")

opt.add\_option("-u", "--user", action="store", type="string",

dest="user")

opt, args = opt.parse\_args()

database = opt.database

passwd = opt.passwd

user = opt.user

Next, we validate the input, asking the user for clarification as necessary:

while (user == "") or (user == None):

print "This system is secured against anonymous logins."

user = getpass.getuser()

while (passwd == "") or (passwd == None):

print "You must have a valid password to log into the

database."

passwd = getpass.getpass()

while (database == "") or (database == None):

database = raw\_input("We need the name of an existing database

to proceed. Please enter it here: ")

Then we try to connect to the database with the credentials that the user passed to the program. If we fail, we print a simple error message:

try:

mydb = MySQLdb.connect(host = 'localhost',

user = user,

passwd = passwd,

db = database)

cur = mydb.cursor()

quit = 1

except:

print "The login credentials you entered are not valid for

the database you indicated. Please check your login details and try

again."

quit = 0

[ 97 ]Simple Insertion

If we successfully make a connection, we carry on showing the tables of the database and further interacting with the user to form the INSERT statement:

if quit == 1:

get\_tables\_statement = """SHOW TABLES"""

cur.execute(get\_tables\_statement)

tables = cur.fetchall()

print "The tables available for database %s follow below:"

%(database)

for i in xrange(0, len(tables)):

print "%s. %s" %(i+1, tables[i])

table\_choice = input("Please enter the number of the table

into which you would like to insert data. ")

table\_choice = str(table\_choice)

table\_no = valid\_table(table\_choice, tables)

table = tables[table\_no-1][0]

show\_def = raw\_input("Would you like to see the database

structure of the table '%s'? (y/n) " %(table))

def\_statement = """DESCRIBE %s""" %(table)

cur.execute(def\_statement)

definition = cur.fetchall()

if show\_def == "y":

from prettytable import PrettyTable

tabledef = PrettyTable()

tabledef.set\_field\_names(["Field", "Type", "Null", "Key",

"Default", "Extra"])

for j in xrange(0, len(definition)):

tabledef.add\_row([definition[j][0], definition[j][1],

definition[j][2], definition[j][3], definition[j][4],

definition[j][5]])

tabledef.printt()

print "Please enter the data you would like to insert into

table %s" %(table)

columns = ''

values = '"'

for j in xrange(0, len(definition)):

column = definition[j][0]

value = raw\_input("Value to insert for column '%s'?"

%(definition[j][0]))

columns = columns + str(column)

values = values + str(value)

[ 98 ]Chapter 3

if j < len(definition)-1:

columns = columns + ", "

values = values + '", "'

else:

values = values + '"'

We then form the INSERT statement and execute it. It is always a good idea to give the user feedback about what data has just been processed:

statement = """INSERT INTO %s(%s) VALUES(%s)""" %(table,

columns, values)

cur.execute(statement)

print "Data has been inserted using the following statement:

\n", statement

The next bit of code is necessary only if you have switched off auto-commit in MySQL for Python. Otherwise, you can skip this part.

cur.close()

mydb.commit()

mydb.close()

Finally, we need to check whether the program has been called directly. If the program is imported as a module into another Python program, main() is never run.

if \_\_name\_\_ == '\_\_main\_\_':

main()

**Room to grow**

We have fulfilled the specification we set out for this project. However, there are several points that you might consider for further development:

•Set the host name dynamically while validating the data. This will require you either to create a whitelist of hosts or to do some network programming in order to validate the existence not just of the host, but of a running MySQL server on it.

•Validate the success of the INSERT statement by running a SELECT statement afterward. Ideally, you will need to close one connection and open another one to be sure that the data is there.

•Validate the user's data more fully than we have here.

•Abstract the database connection and/or the table selection to a function. See how small you can make main().

**Summary**

In this chapter, we have covered the MySQL INSERT command and how to implement it in Python. In particular, we have seen how to form INSERT statements and pass them to MySQL through MySQLdb. Using Python, we also looked at how to use user-defined variables in MySQL statements, changing the INSERT statement dynamically. In the next chapter, we will look at how to handle MySQL errors.