MySQL\_for\_Python\_Albert\_c07

eating and Dropping

The secret to all effective programming lies in abstracting problems so we can

leverage the power of the computer to our benefit. The more we can abstract

a problem and still get the same results, the more we leverage the power of

the computer, and the less work the end user has to do. Abstraction provides

opportunity for automation.

In addition to allowing dynamic insertion and retrieval of information, MySQL

for Python allows us to automate database and table creation and removal. In this

chapter, we will see:

•How to create and delete both databases and tables in MySQL

•How to manage database instances with MySQL for Python

•Ways to automate database and table creation

At the end of the chapter, we will put these dynamics together into a web application

that will allow us to perform this kind of administration on MySQL remotely.

Creating databases

Creating a database in MySQL is as simple as declaring the name of the database that

you want to create. The syntax reads:

CREATE DATABASE <database name>;

An example of this is:

CREATE DATABASE csv;Creating and Dropping

As with all SQL statements, blank space is the token by which the command is

divided. We first need to tell MySQL that we want to create something by using that

keyword. Then we need to tell it what we want to create, a database. Finally, we

give it the name of the database followed by the requisite semi-colon (when in

MySQL itself).

Note that, in order to create databases in MySQL, the account you use

must have the CREATE privilege on the database.

CREATE statements are also sensitive to user privileges. If a user is only

granted CREATE privileges on a single database (for example, csv.\*),

then that user cannot create databases, but can create tables on that

specific database.

Unlike some commands in MySQL, database creation is case-sensitive. So the

following CREATE statements each create a different database:

CREATE DATABASE csv;

CREATE DATABASE Csv;

CREATE DATABASE CSV;

CREATE DATABASE C\_S\_V;;

Whether the CREATE DATABASE statement is case-sensitive ultimately

depends on the filesystem and configuration of your server. Unix-based

systems are usually case-sensitive by default. However, Windows and

Mac servers may not be.

While Mac OS X is derived from Unix, it uses the HFS+ filesystem.

So database names are not case-sensitive by default. To add the case

sensitivity feature for Mac OS X, one would need to use the UFS

filesystem.

To configure MySQL for case-sensitivity when it is not the default, you

need to set the variable lower\_case\_table\_names. More information

on that variable can be found at:

http://dev.mysql.com/doc/refman/5.5/en/server-system-

variables.html#sysvar\_lower\_case\_table\_names

Test first, create second

If a database already exists and you try to create one by the same name, an error will

be thrown.

mysql> CREATE DATABASE csv;

ERROR 1007 (HY000): Can't create database 'csv'; database exists

[ 184 ]Chapter 7

To avoid this error and any ensuing fallout, we can use the IF NOT EXISTS condition

in our CREATE statement. This clause immediately precedes the name of the database

to be created.

mysql> CREATE DATABASE IF NOT EXISTS csv;

Query OK, 0 rows affected, 1 warning (0.00 sec))

Note that the output you get may differ slightly from version to version. MySQL

5.1.42, for example, gives this output:

mysql> CREATE DATABASE IF NOT EXISTS csv;

Query OK, 1 row affected, 1 warning (0.00 sec)

It depends on the version and configuration of your database server.

CREATE specifications

While the preceding example illustrates the most basic database creation statement,

one can also add further specifications for the database immediately after declaring

the name of the database. MySQL supports two ways to further define the database:

•By character set

•By collation used

Specifying the default character set

A character set is a set of symbols and encodings used to represent and store the

information held in a database. If we wanted to declare the csv database to use 8-bit

Unicode by default, we would define it as follows:

CREATE DATABASE csv CHARSET=utf8;

Or better:

CREATE DATABASE csv CHARACTER SET = utf8;

If we then use that database and ask for its status, we will see how that setting

takes hold:

mysql> use csv;

Database changed

mysql> status;;

--------------

mysql Ver 14.12 Distrib 5.0.51a, for debian-linux-gnu (i486) using

readline 5.2

[ 185 ]Creating and Dropping

Connection id:10

Current database:tgv

Current user:skipper@localhost

SSL:Not in use

Current pager:stdout

Using outfile:''

Using delimiter:;

Server version:5.0.51a-3ubuntu5.4 (Ubuntu)

Protocol version:10

Connection:Localhost via UNIX socket

Server characterset:latin1

Dbutf8

characterset:

Client characterset:latin1

Conn.latin1

characterset:

UNIX socket:/var/run/mysqld/mysqld.sock

Uptime:4 hours 36 min 47 sec

Threads: 2 Questions: 112 Slow queries: 0 Opens: 49

Open tables: 43 Queries per second avg: 0.007

Flush tables: 1

--------------

Specifying the collation for a database

The second specification that can be made in a database creation statement is

collation. Where a character set is a set of symbols and encodings. A collation

is a system of rules that MySQL uses to work with the database for purposes of

comparison and matching.

Declaring collation

To express the collation rules for the csv database defined previously as Unicode, we

would use the following definition statement:

CREATE DATABASE csv CHARSET=utf8 COLLATE=utf8\_general\_ci;

Or you can use:

CREATE DATABASE csv CHARACTER SET = utf8 COLLATE = utf8\_general\_ci;

Note that the latter CREATE statement is technically more correct. Either one works,

however, and both may be seen in real-life.

[ 186 ]Chapter 7

In this statement, however, the collation definition is not necessary as every character

set has its own set of collations available to it. For the Unicode character set, the

default collation is Unicode. What this statement does do, however, is overtly define

the Unicode collation to be used.

Finding available character sets and collations

To see the character sets and default collations available on your system, use the

following from a MySQL shell:

mysql> show character set;

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

| Charset

| Description

| Default collation

| Maxlen |

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

| big5| Big5 Traditional Chinese| big5\_chinese\_ci|2 |

| dec8| DEC West European| dec8\_swedish\_ci|1 |

| cp850| DOS West European| cp850\_general\_ci|1 |

| hp8| HP West European| hp8\_english\_ci|1 |

| koi8r| KOI8-R Relcom Russian| koi8r\_general\_ci|1 |

| latin1| cp1252 West European| latin1\_swedish\_ci|1

Not all collations can be used with every character set. Rather every character set has

a group of collations with which it works. If one tries to use a collation that is not

available for a given character set, MySQL will raise an error like this one:

mysql> CREATE DATABASE csv CHARSET=utf8 COLLATE=latin2\_bin;

ERROR 1253 (42000): COLLATION 'latin2\_bin' is not valid

for CHARACTER SET 'utf8'

Removing or deleting databases

To remove or delete a database in MySQL, we use a DROP statement. This statement

is functionally the opposite of the basic CREATE statement used previously:

DROP DATABASE <database name>;

So, for csv, a DROP statement would look like this:

DROP DATABASE csv;

Note that the DROP statement not only deletes the structure of the database setup by

CREATE but also irrevocably drops all of the database data, as well.

[ 187 ]Creating and Dropping

Avoiding errors

As with the CREATE statement, MySQL's DROP statement also supports a test for

existence. If the database you wish to drop does not exist, MySQL will throw an

error. Therefore, it is good practice to use the IF EXISTS conditional as follows:

DROP DATABASE IF EXISTS <database name>;

For a database called foo, this statement would read:

DROP DATABASE IF EXISTS foo;

Preventing (illegal) access after a DROP

By dropping a database, one simply removes it from the list of available databases

that MySQL knows about.

Using DROP does not remove user privileges to that database.

Therefore, if one drops the database csv to scrub all of its data and then creates

another database of the same name, the same users who had access to the original

database will have the same access to the new one. This will occur regardless of

whether the table definitions are different between the two databases.

For a detailed discussion of the REVOKE statement, see:

http://dev.mysql.com/doc/refman/5.5/en/revoke.html

To avoid this, you must revoke user privileges on the database in question. The basic

syntax for REVOKE is:

REVOKE <privileges> ON <database.table> FROM <user>;

To revoke privileges, the user account that you use must have GRANT privileges as

well as any access that you are trying to revoke. With those privileges, one must then

state both the user and table from which you want privileges revoked.

So to revoke all access for user skipper to table filenames from the database csv, we

would use this statement:

REVOKE ALL ON csv.filenames FROM 'skipper' @' localhost';

Note that this does not remove the user and does not impact on the user's access to

other tables within the same database.

[ 188 ]Chapter 7

The easiest way to drop one database and create another is obviously to give

the second a slightly different name than the first and configure network calls

appropriately. However, this is not a commendable practice as it can lead to

confusion in myriad ways. As we will see later in this chapter, however, Python

and MySQLdb can be used to keep database connections in order.

Creating tables

In its basic structure, the MySQL statement for creating tables is very similar to

that for databases. Recall that database creation is involved in simply naming

the database:

CREATE DATABASE foo;;

Creating a table requires the definition of at least one column. The basic syntax looks

like this (note that the backticks are optional):

CREATE TABLE <table name> (`<column name>` <column specifications>);

In practice, definition of a table bar in database foo would be as follows:

CREATE TABLE bar (snafu varchar(30));

While this is basically the statement, it is also a very flawed way of defining the table

when using the MySQLdb module. Before we go into what is wrong, however, let's

cover what is right.

A comprehensive discussion of the options available when creating a

table can be found in the MySQL manual: http://dev.mysql.com/

doc/refman/5.4/en/create-table.html

Obviously, we declare a table named bar. MySQL requires at least one column to be

defined. A column definition at its most basic is the name of the column, the type of

data it will hold, and how long it will be. Here, snafu is defined as varchar (that is,

variable length of characters) and of 30 characters in length. If we go over that limit

on an INSERT statement, the data will almost always be truncated to 30 characters.

Consider the following:

mysql> INSERT INTO bar(snafu) VALUES('pi');

Query OK, 1 row affected (0.08 sec)

mysql> INSERT INTO bar(snafu) VALUES('supercalifragilisticexpialidocious

');

Query OK, 1 row affected, 1 warning (0.00 sec)

[ 189 ]Creating and Dropping

mysql> SELECT \* FROM bar;

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

| snafu

|

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

| pi

|

| supercalifragilisticexpialidoc |

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

2 rows in set (0.00 sec)

Covering our bases

Among the significant issues in this statement, however, is the fact that no default is

mentioned. We could therefore add blank values. Further, as there is no primary key,

we can add redundant values ad nauseam.

mysql> INSERT INTO bar(snafu) VALUES('pi');

Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO bar(snafu) VALUES('');

Query OK, 1 row affected (0.00 sec)

mysql> SELECT \* FROM bar;

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

| snafu

|

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

| pi

|

| supercalifragilisticexpialidoc |

| pi|

||

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

4 rows in set (0.00 sec)

To mitigate against these problems, we should refine our table definition. To define a

column value as mandatory, we use NOT NULL in the table definition:

CREATE TABLE bar (snafu varchar(30) NOT NULL);

[ 190 ]Chapter 7

This only ensures that the column has a value and, by default, forces the user to

input that value. If we want to ensure that the column has a value even if the user

does not define it, we need to set a default:

CREATE TABLE bar (snafu varchar(30) NOT NULL DEFAULT '');

Avoiding errors

Note, however, that we have no conditionality built into this statement. If the table

already exists, we will get an error. To be sure that we do not have this problem, we can

use the IF NOT EXISTS clause again:

CREATE TABLE IF NOT EXISTS bar (snafu varchar(30) NOT NULL DEFAULT

'');

Creating temporary tables

Finally, if we only need the table for some temporary work and want to scrap it after

we are done, we can use the TEMPORARY keyword in our definition. This creates the

database for as long as the current connection is maintained.

mysql> CREATE TEMPORARY TABLE IF NOT EXISTS bar (`snafu` varchar(30) NOT

NULL default '');

Note that temporary tables are only visible to the user session that created it. So

there can be no confusion on the part of other sessions on the same server. This is

helpful for creating temporary datasets for debugging.

It is worth noting that the dropping of temporary tables is logged

differently when the session ends rather than when they are overtly

dropped. Therefore, the best practice is to drop every temporary table

you create when you are done using it, even at the end of a session.

When the database is created, MySQL will report that nothing has been affected.

Since it is a temporary table, it does not show up in the list of tables. However, a

query against it will return results:

mysql> CREATE TEMPORARY TABLE foo (`snafu` varchar(30) NOT NULL default

'');

Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO foo(snafu) VALUES('cucumber');

Query OK, 1 row affected (0.00 sec)

[ 191 ]Creating and Dropping

mysql> show tables;

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

| Tables\_in\_csv |

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

| foo

|

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

1 row in set (0.00 sec)

mysql> select \* from foo;

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

| snafu

|

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

| cucumber |

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

1 row in set (0.00 sec)

Dropping tables

Dropping tables in MySQL follows the same pattern as dropping databases.

DROP TABLE <table name>;

We use the keyword DROP so MySQL knows what we want to do. We then indicate

that we want to drop a table and follow that with the name of the table to be deleted.

When the DROP command is executed, the table and its definition are

deleted unrecoverably from the database. You should therefore exercise

caution when using it.

It is worth noting that the user who passes the DROP statement to MySQL must have

the DROP privilege. Otherwise, MySQL will not execute the statement.

Playing it safe

If you create a temporary table and want to ensure that only that table is dropped,

use the TEMPORARY keyword:

DROP TEMPORARY TABLE <table name>;

[ 192 ]Chapter 7

So to drop the araba table defined previously, we would issue this command:

DROP TEMPORARY TABLE araba;

Of course, if we issue that command twice, MySQL will get confused. More on this is

mentioned in the following section Avoiding errors.

Avoiding errors

As with deleting databases, we should ask MySQL to ensure that the table exists

before trying to remove it. Otherwise, we can receive an error. To avoid this we use

the IF EXISTS conditional again:

DROP TABLE IF EXISTS <table name>;

So for the table bar created with this CREATE statement (from previous command):

CREATE TABLE IF NOT EXISTS bar (snafu varchar(30) NOT NULL DEFAULT

'');

The corresponding DROP command would be:

DROP TABLE IF EXISTS bar;

For the temporary table, we would change the DROP command accordingly:

DROP TEMPORARY TABLE IF EXISTS foo;

Removing user privileges

As with dropping databases (from previous secion), dropping a table does not

remove access to that table from a user's profile. Therefore, dropping it and

subsequently creating another table of the same name will automatically allow

the users of the first table to access the second. This will be done with the same

privileges, as well. To avoid this, use the REVOKE command as outlined under

Preventing (illegal) access after a DROP.

Doing it in Python

As you might expect, affecting the creation and deletion of databases and tables in

Python is very similar to MySQL when using MySQL for Python. There are some

differences as we shall see in this section.

[ 193 ]Creating and Dropping

For the following examples, we will work with the following code being assumed:

import MySQLdb

mydb = MySQLdb.connect('localhost', 'skipper', 'secret', 'csv')

cur = mydb.cursor()

Creating databases with MySQLdb

Where Cursor.execute() shows the number of affected rows in previous

commands, whether INSERT or SELECT, it always returns a 0 value for the CREATE

command of a database:

>>> cur.execute("""CREATE DATABASE foo""")

0L

Testing the output

Consequently, passing the output of the method to a variable will result in that

variable equating to 0:

>>> res = cur.execute("CREATE DATABASE foo")

>>> print res

0

The only time that this does not occur is if an error is thrown (that is, if you do not

include the conditional IF NOT EXISTS).

This is helpful when working with code that you did not write. By testing the

returned value, you can have greater control over what happens in your program.

This testing can be negative as follows:

>>> if res != 0: <do something>

Or can be positive:

>>> if res == 0: <do something>>

Naturally, if the statement does not execute as expected, you will want to catch the

exception as shown in Chapter 4, Exception Handling.

[ 194 ]Chapter 7

Dynamically configuring the CREATE statement

The CREATE statement can be dynamically constructed in Python. To do this, we use

a string formatting convention:

cur.execute("CREATE DATABASE IF NOT EXISTS %s" %('foo'))

Dropping databases with MySQLdb

Similarly to creating a database, dropping a database returns a 0 value.

>>> res = cur.execute("DROP DATABASE IF EXISTS foo")")

>>> print res

0

By leaving off the IF EXISTS clause, we can create a feedback mechanism:

try:

res = cur.execute("DROP DATABASE foo")

except:

print "Drop operation failed."

If the DROP is executed, program execution continues without comment. If the table

has already been dropped, we get output:

Drop operation failed.

and the program then continues to execute.

Creating tables in Python

As discussed earlier in this chapter, the MySQL syntax for creating a table is similar

to creating a database. The example of table bar from above reads:

CREATE TABLE IF NOT EXISTS bar (snafu varchar(30) NOT NULL DEFAULT

'');

To put this into Python, we pass the statement as an argument to Cursor.execute():

cursor.execute("""CREATE TABLE IF NOT EXISTS bar (snafu varchar(30)

NOT NULL DEFAULT '')""")

[ 195 ]Creating and Dropping

Once again, however, we are able to use Python's string formatting facilities to create

dynamic statements:

cursor.execute("""CREATE TABLE IF NOT EXISTS %s (%s varchar(30) NOT

NULL DEFAULT '')""" %('bar', 'snafu'))

As before this statement creates a table bar with a column snafu.

Verifying the creation of a table

Depending on the nature of your program, it is frequently good practice to

validate the creation of a table. To do so, we need to retrieve a listing of the

tables in the database.

cursor.execute("""SHOW TABLES""")

tables = cursor.fetchall()

The result here is to always get the tables. Of course, there is no need for the second

line if the table creation statement has failed and no tables are available. Therefore,

the better way to affect this is:

table\_no = cursor.execute("""SHOW TABLES""")

if table\_no != 0:

tables = cursor.fetchall()

Cursor.fetchall() returns a tuple of the tables available. To confirm the existence

of the table in the database, we need to search through the list of tables.

created\_table = 'bar'

for item in tables:

if item[0].count(created\_table) != 0:

print item[0]

This will print the table name if it matches the value of created\_table, bar.

All of this presumes a knowledge of the total number of tables in the database prior

to your creating the last one. To ascertain whether a table has been created without

counting tables, you can ask MySQL for the creation statement of the table:

try:

cursor.execute("""SHOW CREATE TABLE %s""" %('bar'))

except:

print "The table has not yet been created."

raise

else:

print "The table has been created."

The SHOW CREATE statement is addressed in more detail in Chapter 13, Showing

MySQL metadata.

[ 196 ]Chapter 7

Another way to verify table creation

As with creating a database, creating a table returns a 0 value if successful. Therefore,

we can again test for that. If we use the IF NOT EXISTS conditional, the statement will

always return 0. Therefore, we need to leave this off in order to use this method.

Note that many MySQL installations will issue a warning if you use

IF NOT EXISTS when attempting to create a table that already exists.

However, it is not good practice to rely on these warnings.

Consider the following:

>>> cursor.execute("""CREATE TABLE %s (%s varchar(30) NOT NULL DEFAULT

'')""" %('barge', 'snafu'))

0L

>>> cursor.execute("""CREATE TABLE IF NOT EXISTS %s (%s varchar(30) NOT

NULL DEFAULT '')""" %('barge', 'snafu'))

0L

>>> cursor.execute("""CREATE TABLE %s (%s varchar(30) NOT NULL DEFAULT

'')""" %('barge', 'snafu'))

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "/var/lib/python-support/python2.5/MySQLdb/cursors.py", line 166,

in execute

self.errorhandler(self, exc, value)

File "/var/lib/python-support/python2.5/MySQLdb/connections.py", line

35, in defaulterrorhandler

raise errorclass, errorvalue

\_mysql\_exceptions.OperationalError: (1050, "Table 'barge' already

exists")")

So we want to try to create the table and test the results:

>>> try: attempt = cursor.execute("""CREATE TABLE %s (%s varchar(30) NOT

NULL DEFAULT '')""" %('barge', 'snafu'))

... except: print "Houston, we have a problem"

...

Houston, we have a problem

[ 197 ]Creating and Dropping

Dropping tables with MySQLdb

Dropping a table through MySQLdb is very straightforward.

cursor.execute("DROP TABLE %s" %('barge'))

Of course, for the sake of feedback, it is worth sandwiching this in a try...except

structure. As with the DROP TABLE statement in MySQL, you can usually just pass

this statement with an IF EXISTS conditional and be done with it:

execution = cursor.execute("DROP TABLE IF EXISTS %s" %('barge'))

Note that, as before, the conditional element causes MySQL to always return a 0

value for successful execution. So a test for the value of execution will always test

true for 0. If you want an exception if the table does not exist, use the other DROP

statement in a try...except statement.

Project: Web-based administration of

MySQL

The project for this chapter will set the groundwork for the next several projects. We

will write a program for administering MySQL remotely through a web interface.

To be sure, more sophisticated applications like PHPMyAdmin exist. The value

of creating one of your own is that you can extend it and change it in the future

depending on your needs. Just creating the application is a good exercise as it leaves

you with code that you can import into other applications at will and gives you a

better understanding of the processes involved.

By the end of this project, we want a web application that will have the

following aspects:

•Ability to create MySQL statements for the following commands: CREATE,

DROP, INSERT, UPDATE, and SELECT

•Execute the created statement

•Output the results of any queries and confirm the successful execution of

other commands

•Be written in a modular structure that allows different functions or methods

to be used independently of the others

•Use CGI to call the Python program

[ 198 ]Chapter 7

It is worth noting that this implementation is fairly rudimentary due to the

exigencies of book media. Nevertheless, when we are done, you will have the basis

for a full web application that you can develop into administrative facilities for your

web-oriented MySQL database.

CGI vs PHP: What is the difference?

CGI stands for Common Gateway Interface. As such, it is not a language like

PHP but a protocol. This means that it can be implemented in just about any

programming language, and implementations of it exist in C and C++. CGI scripts

are often written in Perl but may be written in other languages.

PHP stands for Pre-Hypertext Processor and is a language developed specifically

to output HTML with speed. As such, it is optimized for web scripting and is

known as one of the lightest technologies on the Web. Where heavier technologies

become sluggish and can lose connectivity in the face of latency, PHP tends to be

more robust.

The following sections address some of the strengths and weaknesses of each option.

For this project, we will be using CGI to pass arguments to our Python program.

However, with the proper coding, one could also use PHP.

Depending on a variety of factors from language of implementation to

the configuration of the web server on which it runs, CGI is reported to

be either considerably slower or even faster than PHP. The performance

benefits of each implementation strategy are heavily dependent on

other environmental variables. Given the ubiquity of shared hosting and

the fact that web developers typically do not have control over system

variables in such circumstances, it is helpful to be conversant with both.

CGI and PHP usually come as standard with a shared hosting solution.

You can nevertheless find information on each consecutively at:

http://docs.python.org/library/cgi.html

http://www.php.net/manual/en/install.php

Note that this discussion does not address the matter of persistence. No matter which

way we implement our web application, the only way that we can preserve an object

between web pages is to pickle it. Pickling means converting the Python object to a

byte stream for writing to a space on the server's hard disk. It also introduces a layer

of complexity that we are leaving out of this project. However, if you would like

to look into it further, you can find information on Python's pickle module at

http://docs.python.org/library/pickle.html.

[ 199 ]Creating and Dropping

Not every Python object can be pickled. For help on determining whether something

can be pickled, see What can be pickled and unpickled? in the following Python

documentation module:

http://docs.python.org/library/pickle.html#what-can-be-pickled-and-

unpickled.

Pickling preserves an object as a byte stream. Alternatively, we could save the

session data into a database.

Basic CGI

The form dialogue of HTML requires us to include the program to which the data

should be passed. For a Python program myprogram.py, a form line would look

like this:

<form action="myprogram.py" method="POST">

Whether you use POST or GET depends on your development needs.

More information on HTML's form tag and its attributes can be found at:

http://www.w3schools.com/TAGS/tag\_form.asp

The difference between GET and POST is detailed in RFC 2616:

http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

Passing values to a Python program through CGI is affected by importing two

modules and instantiating an object. The modules used are cgi and cgitb. The role

of the cgi module is self-evident; importing cgitb gives us helpful error messages if

we need to debug the program.

import cgi

import cgitb

The cgi module provides a class FieldStorage that provides a method getvalue()

for accessing CGI input.

import cgi, cgitb

form = cgi.FieldStorage()

name = form.getvalue('firstname')

address = form.getvalue('surname')

phone = form.getvalue('phoneno')

[ 200 ]Chapter 7

After receiving and processing the input from CGI, the output of one's program is

sent to the web client through the server. If the output is in plain text, no formatting

is included. Therefore, one must be certain to include HTML formatting with it.

More detailed discussions on using CGI in Python can be found at:

http://docs.python.org/library/cgi.html

http://python.about.com/od/cgiformswithpython/ss/

pycgitut1.htm

HTML tutorials abound on the Internet, but one of the best venues for

learning HTML and other web-related technologies is W3Schools.

http://www.w3schools.com

Simple HTML output would look like this:

print """

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

<head>

<title>Successful Input</title>

</head>

<body>

<center>Information received successfully.</center>

</body>

</html> """

One of the main downsides of using CGI is the problem of data persistence. Without

using magic URLs, passing data from one page to another becomes a challenge.

A magic URL is a Uniform Resource Locator (URL) that includes unique

identifying information, so the server-side program can associate it with

saved data. One of the most common magic URLs is the session ID used

by many news sites to track reader habits. Magic URLs are one of the

most frequent security issues for web applications as web servers are not

designed to tell two users apart if they use the same identifier. When one

user purposely appears as another user, she/he is said to spoof that user.

The alternative would be to pass information through flags. But CGI does not

support a secure way to do this. Any values that one would want to persist from one

page to another would have to be embedded in the code, which is viewable and can

therefore be spoofed by any user who knows that information.

[ 201 ]Creating and Dropping

Using PHP as a substitute for CGI

Depending on the configuration of the server and system variables, PHP may be a

faster way to pass input to a Python program. To do this, we use the PHP command:

shell\_exec():

<?php

$somevar = $\_POST["somevar"];

$escapees = array("&", ";", "\")

$replacements = array("<ampersand>", "<semicolon>", "<backslash>")

$safevar = str\_replace($escapees, $replacements, $somevar)

$results = shell\_exec ("./myprogram.py " . $somevar) ;

print $results;

?>

This calls a Python program myprogram.py with the arguments contained in

somevar and assigns the output to $results. Before handing the value of somevar

to myprogram.py, we process out any known characters that could be used to break

out of the execution environment. Ampersands are used to background processes on

Unix-based machines. Semi-colons are used to pass multiple commands on the same

line. Backslashes are used to escape out of a formatted environment. You can (and

should) add to them according to the execution environment you use before using

this code for any projects beyond the scope of this book. Having replaced some of the

unsavory characters, we can then print $results or print the HTML formatting with

the output intermingled appropriately.

If the previous PHP script were called myprogram.php, the calling form line

would be:

<form action="myprogram.php" method="POST">

Aside from the aforementioned benefits of processing speed, using PHP has the

benefit of introducing heterogeneity into the system. Many security issues arise

because computing systems use the same technologies in the same way throughout

an organization. This makes the system easy to manage, but it also makes it

predictable. Using CGI, the program to be called is always shown in the HTML

source code. One therefore tips one's hand and shows that the system processing

the data is in Python. By using PHP to hand the data off to Python, one introduces

a hidden layer of complexity that makes cracking the system more difficult.

Finally, as shown in the previous example, PHP allows us to pass command-line

arguments discretely to our program. The form data, which the user already has, is

passed overtly. But any special settings can be reserved in the PHP script and passed

as an argument to the Python program at runtime.

[ 202 ]Chapter 7

It should be noted that the use of PHP over CGI does not impact on the HTML FORM

syntax at all except to change the name of the file. PHP serves as an intermediate

layer between the web page and the processing of data.

CGI versus PHP: When to use which?

CGI has been used for many years and therefore is well proven. PHP offers a cleaner

and more flexible implementation than CGI for passing data between a web page

and a Python program. However, depending on the implementation, it can easily

result in security issues. Setting aside personal familiarity as a criterion, which you

use (CGI or PHP) will largely be determined by the environment in which you

are deploying.

As already mentioned, the cgitb module provides for web-based error messages.

These are very helpful in the case of shared hosting, when one frequently does not

have shell access. Unless one is relying on system variables, however, using PHP is

still cleaner in this regard as it does not require a web server to work.

Where CGI requires a web server for testing, PHP can be run from the command-line.

Therefore, one can create a local copy of one's remote directory structure and not have

to run a local server. It is best practice to have such a copy for administering a website,

anyway. One can then test the PHP-Python system in that context before posting it.

For help on using PHP from the command line, see http://php.net/manual/en/

features.commandline.php.

A simple tutorial on using PHP in lieu of CGI can be found at:

http://python.about.com/od/cgiformswithpython/ss/

phpjscgi.htm

Some general considerations for this program

As already noted in this chapter, we are not going to persist data or objects on the

server between pages. Therefore, we do not need a class like the MySQLStatement

class that we used in Chapter 4. Instead, we will use functions for all MySQL

operations. The only class will be for our HTML output.

Program flow

For our purposes, the program starts as soon as a user accesses the first page and

enters their login information. We attempt to create a connection with the user's

login credentials. If they fail, program execution breaks.

[ 203 ]Creating and Dropping

If successful, we process the user's choice of actions. If the user does not give us all

the data we need for a chosen action, they receive an error message and program

execution terminates.

The following six template variants correspond to the types of statements that we

will program:

•CREATE DATABASE <database name>

•DROP DATABASE <database name>

•CREATE TABLE<table name>

•DROP TABLE <table name>

•SELECT \* FROM table WHERE <column> = <value>

•INSERT INTO table (columns) VALUES (values)

These have all been covered in previous chapters. If anything looks unfamiliar, be

sure to revisit the relevant chapter before going on.

The basic menu

For this program, we will use two web pages at first. The second page will be

automatically generated by the program, but the first will be static.

The first page is a basic dialogue that asks the user to select the action that they want.

We want to give the user the choice of creating, dropping, querying, or inserting data

into a database.

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN" "http://

www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"

xml:lang="en"

lang="en"

dir="ltr">

<head>

<title>PyMyAdmin 0.001</title>

<meta http-equiv="Content-Type" content="text/html;

charset=utf-8">

</head>

<body>

<h1>PyMyAdmin Menu</h1>

<form name="input" action="./pymyadmin.py" method="post">

<div>AUTHENTICATION</div>

Login: <input type="text" name="user" value=""><br>

Password: <input type="password" name="password" value="">

<br><br>

<div>DATABASES</div>

<input type="radio" name="dbact" value="create"> CREATE<br>

[ 204 ]Chapter 7

<input type="radio" name="dbact" value="drop"> DROP<br>

Database name: <input type="text" name="dbname" value="">

<br><br>

<div>TABLES</div>

<input type="radio" name="tbact" value="create"> CREATE<br>

<input type="radio" name="tbact" value="drop"> DROP<br>

Database name: <input type="text" name="tbdbname" value=""><br>

Table name: <input type="text" name="tbname" value="">

<br><br>

<div>QUERIES</div>

<input type="radio" name="qact" value="select"> SELECT<br>

<input type="radio" name="qact" value="insert"> INSERT<br>

Database name: <input type="text" name="qdbname" value=""><br>

Table name: <input type="text" name="qtbname" value=""><br>

Columns (comma-separated): <input type="text" name="columns"

value=""><br>

Values (comma-separated): <input type="text" name="values"

value=""><br>

<input type="submit" value="Submit">

</form>

</body>

</html>

When put into a browser, the page should look like this:

[ 205 ]Creating and Dropping

Obviously, the web design aspect of the page has been left as an exercise.

Authorization details

As you can see from this dialogue, we first ask for the authorization details. Note

that the password dialogue is of input type password, to ensure it is hidden on entry.

In the program, this data will be first used to verify the authenticity of the user's

credentials before any MySQL statement is formed.

Three operational sections of the dialogue

In the subsequent three sections of the dialogue, we offer the user facilities to affect

databases, tables, and insertion or retrieval. Depending on which radio button the

user chooses, we will expect different parts of the form to be completed.

The variables

The web page will send us several variables, only some of which we will need for

any given operation. In the order that they appear on the previous page, these are:

1. user

2. password

3. dbact

4. dbname

5. tbact

6. tbdbname

7. tbname

8. qact

9. qdbname

10. qtbname

11. columns

12. values

The first two are of obvious import. The next two, dbact and dbname, are used to

create or drop a database. dbact is designed to indicate either CREATE or DROP as no

other values are allowed. The value of dbname can be anything.

[ 206 ]Chapter 7

For table creation and dropping, we use the next three variables along with columns

and values. To affect anything related to tables, we need tbact, whether to CREATE

or DROP, and tbdbname, the database to be used, and tbname, the name of the

table itself. Additionally, for creating a table, we need the columns and types, here

indicated by columns and values.

Finally, for INSERT and SELECT statements, we use qact, qdbname, qtbname, and

the last two variables. As you can guess from their names, qact is to indicate either

INSERT or SELECT. qdbname is the name of the database to be used, and qtbname is

the name of the table, into or out of which to process data.

Planning the functions

Some of the functions we will use will be used for multiple purposes. So we will

not have createdb() and dropdb(). Rather, we will use the variable prefixes as

a key to the function names. We will call dbaction() and pass the type of action

as an argument. Similarly, we will use tbaction() for table administration and

qaction() for queries.

We will naturally need to connect to the database. However, we will sometimes need

to connect without declaring a database. So we will have connection functions that

handle both.

Finally, we will need to execute the statement and return the appropriate values.

This role will be performed by execute().

Code of each function

The following subsections show the code for each function. Having planned them

out gives us a bird's eye view of the entire process.

Connecting without a database

The database argument on MySQLdb.connect() is optional. Therefore, we can create

a connection with it in order to validate user credentials and to administer databases

without knowing anything about which ones are available.

def connectNoDB(user, password):

"""Creates a database connection and returns the cursor.

hardwired to 'localhost'."""

try:

host = 'localhost'

mydb = MySQLdb.connect(host, user, password)

cur = mydb.cursor()

return cur

[ 207 ]

Host isCreating and Dropping

except MySQLdb.Error:

print "There was a problem in connecting to the database.

Please ensure that the database exists on the local host system."

raise MySQLdb.Error

except MySQLdb.Warning:

pass

Note that this function is hard-wired to work with the localhost. Without much

work at all, one could make all the values of this function to be dynamic.

Connecting with a database

Obviously, we need to declare a database to do anything with a table. We could

technically get by without declaring a database at first, but that would require a

locution of code to declare the database eventually. It is best to do it upfront.

For this, we will pull in the connection() function from previous projects.

def connection(user, password, database):

"""Creates a database connection and returns the cursor. Host is

hardwired to 'localhost'."""

try:

host = 'localhost'

mydb = MySQLdb.connect(host, user, password, database)

cur = mydb.cursor()

return cur

except MySQLdb.Error:

print "There was a problem in connecting to the database.

Please ensure that the database exists on the local host system."

raise MySQLdb.Error

except MySQLdb.Warning:

pass

Database action

Forming the database-related statements is done by dbaction().

def dbaction(act, name, cursor):

if act == "create":

statement = "CREATE DATABASE IF NOT EXISTS %s" %(name)

output = execute(statement, cursor, 'create-db')

elif act == "drop":

statement = "DROP DATABASE IF EXISTS %s" %(name)

output = execute(statement, cursor, 'drop-db')

else:

output = "Bad information."

return output

[ 208 ]Chapter 7

No matter what happens, dbaction() gives us output. Using the conditional clause

in both the CREATE and DROP statements, we should never have a problem with

execution. Nevertheless, we include an else clause to ensure that we are covered.

Table action

The function affecting tables is more complicated than the preceding database

function. This is because we need to handle column names and values as well as the

basic information used for databases. The function looks like this:

def tbaction(act, db, name, columns, types, user, password):

cursor = connection(user, password, db)

if act == "create":

tname = name + "("

columns = columns.split(',')

types = types.split(',')

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

col = columns[i].strip()

val = types[i].strip()

tname = tname + col + " " + val

if i == len(columns)-1:

tname = tname + ")"

else:

tname = tname + ", "

statement = "CREATE TABLE IF NOT EXISTS %s" %(tname)

results = execute(statement, cursor, 'create-tb')

elif act == "drop":

statement = "DROP TABLE IF EXISTS %s" %(name)

results = execute(statement, cursor, 'drop-tb')

return results

As requested in the initial web page, we expect columns and types to come in a

comma-delimited format. Consequently, we need to split each and join them

up so they make sense to MySQL. This is done by the for loop in the middle of

the function.

[ 209 ]Creating and Dropping

Query action

For queries, we assume that the user will provide us with the following information:

•

•

•

•

•

•

•

Type of action

Database name

Table name

Columns involved

Values used

Username

Password

The function then looks like this:

def qaction(qact, db, tb, columns, values, user, password):

cursor = connection(user, password, db)

tname = tb + "("

columns = columns.split(',')

values = values.split(',')

cols = ""

vals = ""

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

col = columns[i].strip()

val = values[i].strip()

cols = cols + col

vals = vals + "'" + val + "'"

if i != len(columns)-1:

cols = cols + ", "

vals = vals + ", "

if qact == "select":

statement = "SELECT \* FROM %s WHERE %s = %s" %(tb, cols, vals)

results = execute(statement, cursor, 'select')

elif qact == "insert":

statement = "INSERT INTO %s (%s) VALUES (%s)" %(tb, cols,

vals)

results = execute(statement, cursor, 'insert')

return results

Once again, we need to break apart columns and values in order to use them

appropriately for MySQL.

[ 210 ]Chapter 7

execute()

Finally, we need a function to execute the MySQL statements that are formed by

either dbaction, tbaction, or qaction. The execute() function, as you may note in

the preceding code listings, takes the statement to be executed, the cursor to be used,

and the type of statement to be processed. The last is essential in allowing execute()

to handle the data returned by MySQL appropriately.

The function is as follows:

def execute(statement, cursor, type):

"""Attempts execution of the statement resulting from

MySQLStatement.form()."""

while True:

try:

cursor.execute(statement)

if type == "select":

# Run query

output = cursor.fetchall()

results = ""

data = ""

for record in output:

for entry in record:

data = data + '\t' + str(entry)

data = data + "\n"

results = results + data + "\n"

elif type == "insert":

results = "Your information was inserted with the

following SQL statement: %s;" %(statement)

elif type == "create-db":

results = "The following statement has been processed

to ensure the database exists: %s;" %(statement)

elif type == "create-tb":

results = "The following statement has been processed

to ensure the table exists: %s;" %(statement)

elif type == "drop-db":

results = "The following statement has been processed

to ensure the removal of the database: %s;" %(statement)

elif type == "drop-tb":

results = "The following statement has been processed

to ensure the removal of the table: %s;" %(statement)

return results

valid.

except MySQLdb.Error, e:

print "Some of the information you have passed is not

Please check it before trying to use this program again."

print "The exact error information reads as follows: %s"

%(e)

[ 211 ]Creating and Dropping

raise

except MySQLdb.Warning:

pass

except Warning:

pass

For purposes of feedback, the execute() function always returns the statement

processed. If the statement is a query, it returns the results in a basic form.

As noted in previous chapters, the error-handling used here is generic for reasons of

space. In real-life deployment, it should be more robust.

The HTML output

In addition to processing data, we need to return some feedback to the user. In order

to prettify the page, we should enclose the output in some HTML formatting.

For this purpose, we will use a class HTMLPage. This will ensure that we can

standardize the HTML output of the program, only changing what is necessary and

not having to include the header and footer code more than once.

HTMLPage has a single attribute, message, which holds the output passed from

execute(). Otherwise, it has four methods beyond \_\_init\_\_():

•header(): Returns a standard HTML header with a generic page title

•body(): Compiles a HTML body that includes the value of HTMLPage message

•footer(): Returns a standard HTML closing code

•page(): Coordinates the three methods to form and return the HTML output

Basic definition

The definition of HTMLPage begins as follows:

class HTMLPage:

def \_\_init\_\_(self):

"""Creates an instance of a web page object."""

self.Statement = []

Naturally, the class needs a way to reference itself. For our purposes, HTMLPage is a

regular Python object, so we do not need to declare any inheritance.

[ 212 ]Chapter 7

The message attribute

HTMLPage.message simply receives the value and assigns it to itself.

def message(self, message):

self.message = message

Essentially, message is a means of holding the output from execute() as an attribute

of the class object.

Defining header()

The header() method, as discussed, simply returns a HTML header. By defining it

distinctly, we can readily reuse this code and modify it as needed.

def header(self):

"""Prints generic HTML header with title of application."""

output = """

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN" "http://

www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"

dir="ltr">

<head>

<title>PyMyAdmin 0.001</title>

<meta http-equiv="Content-Type" content="text/html;

charset=utf-8" />

</head>

<body>

"""

return output

As this is straightforward HTML code, we could have made it an attribute. However,

by using a method, we are ready to render the code dynamically if needed. For

example, we could not change the page title dynamically if header() were an

attribute. As it is, we can render it dynamically with ease using this function.

Defining footer()

To match header(), we use footer():

def footer(self):

"""Print generic HTML footer to ensure every page closes

neatly."""

output = """

</body>

</html>

"""

return output

[ 213 ]Creating and Dropping

Normally, this would be an attribute. However, there are circumstances when this

should be rendered as a function. Some web-tracking scripts rely on JavaScript with

unique identifiers to be issued at the end of a page. To form such pages dynamically,

one would need to use a method instead of an attribute.

Defining body()

The body() method combines a title with the output of execute().

def body(self):

output = ""

title = "<h1>PyMyAdmin Results</h1>"

output = output + title + "<br>" + self.message

return output

Defining page()

Finally, we define a method to coordinate the methods of HTMLPage to form a web

page for output.

def page(self):

"""Creates webpage from output."""

header = self.header()

body = self.body()

footer = self.footer()

output = header + body + footer

return output

Getting the data

Obviously, none of this data processing counts for anything if we cannot get the

data that the user sends to us. Depending on whether you use CGI or PHP to call

the Python program will determine how you accept the information.

Using CGI

To receive the information through CGI, we import cgi and cgitb, as discussed

previously. We then assign the variables from the cgi.FieldStorage() object

we create.

#!/usr/bin/python

import MySQLdb

import cgi, cgitb

[ 214 ]Chapter 7

form = cgi.FieldStorage()

user = form.getvalue('user')

password = form.getvalue('password')

dbact = form.getvalue('dbact')

dbname = form.getvalue('dbname')

tbact = form.getvalue('tbact')

tbdbname = form.getvalue('tbdbname')

tbname = form.getvalue('tbname')

qact = form.getvalue('qact')

qdbname = form.getvalue('qdbname')

qtbname = form.getvalue('qtbname')

columns = form.getvalue('columns')

values = form.getvalue('values')

Note that CGI always requires absolute paths. We cannot therefore use a shebang

line of #!/usr/bin/env python.

Using PHP

For PHP, we do not use any of the CGI modules. Instead, because PHP calls the

program as from a command-line, we can use the optparse module to weed

through the options.

#!/usr/bin/env python

import MySQLdb

import optparse

opt = optparse.OptionParser()

opt.add\_option("-U", "--user",

action="store",

type="string",

help="user account to use for login",

dest="user")

opt.add\_option("-P", "--password",

action="store",

type="string",

help="password to use for login",

dest="password")

opt.add\_option("-d", "--dbact",

action="store",

[ 215 ]Creating and Dropping

type="string",

help="kind of db action to be affected",

dest="dbact")

opt.add\_option("-D", "--dbname",

action="store",

type="string",

help="name of db to be affected",

dest="dbname")

opt.add\_option("-t", "--tbact",

action="store",

type="string",

help="kind of table action to be affected",

dest="tbact")

opt.add\_option("-Q", "--tbdbact",

action="store",

type="string",

help="name of database containing table to be

affected",

dest="tbdbname")

opt.add\_option("-N", "--tbname",

action="store",

type="string",

help="name of table to be affected",

dest="tbname")

opt.add\_option("-q", "--qact",

action="store",

type="string",

help="kind of query to affect",

dest="qact")

opt.add\_option("-Z", "--qdbname",

action="store",

type="string",

help="database to be used for query",

dest="qdbname")

opt.add\_option("-Y", "--qtbname",

action="store",

type="string",

help="table to be used for query",

dest="qtbname")

opt.add\_option("-c", "--columns",

action="store",

type="string",

help="columns to be used in query",

dest="columns")

[ 216 ]Chapter 7

opt.add\_option("-v", "--values",

action="store",

type="string",

help="values to be used in query",

dest="values")

opt, args = opt.parse\_args()

Defining main()

With the variables input into the program and the functions and class defined, we

can then code the main() function to orchestrate the program's execution.

def main():

"""The main function creates and controls the MySQLStatement

instance in accordance with the user's input."""

output = ""

while 1:

try:

cursor = connectNoDB(opt.user, opt.password)

authenticate = 1

except:

output = "Bad login information. Please verify the

username and password that you are using before trying to login

again."

authenticate = 0

if authenticate == 1:

errmsg = "You have not specified the information necessary

for the action you chose. Please check your information and specify

it correctly in the dialogue."

if opt.dbact is not None:

output = dbaction(opt.dbact, opt.dbname, cursor)

elif opt.tbact is not None:

output = tbaction(opt.tbact, opt.tbdbname, opt.tbname,

opt.columns, opt.values, opt.user, opt.password)

elif opt.qact is not None:

output = qaction(opt.qact, opt.qdbname, opt.qtbname,

opt.columns, opt.values, opt.user, opt.password)

else:

output = errmsg

printout = HTMLPage()

[ 217 ]Creating and Dropping

printout.message(output)

output = printout.page()

print output

break

First, we check whether the user's credentials are acceptable to MySQL. If they are

not, we simply issue a statement that their login information is bad. Otherwise, we

continue. Based upon what kind of action is specified, we call different functions. If

no radio button is ticked, the user receives the error message errmsg.

After all processing has been done, we form the page. The HTMLPage object is

printout. We set the value of HTMLPage.message to whatever output we generated

in the if...elif...else clause. After the page is formed by HTMLPage.page(), we

output it and break the while loop.

Room to grow

This project forms the basis for further development, some of which will be done in

the upcoming chapters. In writing this program and later extending its functionality,

be wary of trying to formulate too complicated of statements before you have good

processes in place for less complex commands.

Some places where this project could (and should) be developed further before

deploying it in real-life scenarios are:

•Implementing UPDATE and DELETE statements in addition to SELECT

and INSERT

•Using DESCRIBE to offer the user information about tables

•Implementing fuller exception-handling

Summary

In this chapter, we have covered how to create and remove databases and tables with

MySQL for Python. We have seen:

•How to use MySQLdb to create and delete both databases and tables

•How we can manage database instances with MySQL for Python

•Ways to automate database and table creation

In the next chapter, we will look at regulating MySQL user access within Python.

[ 218 ]Creatin