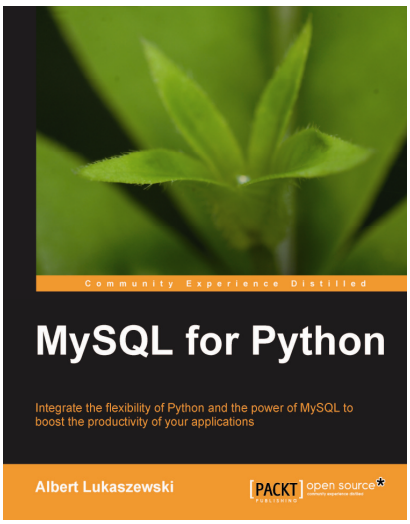
MySQL for Python

Albert Lukaszewski



Chapter No. 4
"Exception Handling"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.4 "Exception Handling"

A synopsis of the book's content

Information on where to buy this book

About the Author

Albert Lukaszewski is principal consultant for Lukaszewski Consulting Services in southeast Scotland. He has programmed computers for 30 years. Much of his experience has related to text processing, database systems, and **Natural Language processing (NLP)**. Currently he consults on database applications for companies in the financial and publishing industries.

In addition to MySQL for Python, Albert Lukaszewski has also written "About Python", a column for the New York Times subsidiary, `About.com`.

Many people had a hand in this work beyond my typing at the keyboard. Some contributed by their effort and others by their sacrifice. Thanks to the team at Packt for their consistent understanding and support. I am particularly thankful to Steven Wilding for help and support above and beyond the call of duty.

Thanks also to Andy Dustman, Geert Vanderkelen, and Swaroop for their helpful review of this book and for making so many significant and helpful recommendations. This book would be much the poorer were it not for their suggestions.

To Richard Goodrich, who first introduced me to Python, thank you for liberating me from bondage to that other P-language. Funny what a little problem can lead to.

My heartfelt thanks and appreciation go to my wife, Michelle, and my sons, Cyrus and Jacob. The latter was born during the writing of this book and consistently brightens even the darkest Scottish weather with his smile. I appreciate your sacrifice. I could not have written this book without your support.

Finally, my thanks to my brother, Larry, who first introduced me to the world of computing. I would probably not know anything about computer programming if you had not left me your TRS-80. So this is all your fault, and I am glad you did it.

MySQL for Python

Python is a dynamic programming language, which is completely enterprise ready, owing largely to the variety of support modules that are available to extend its capabilities. In order to build productive and feature-rich Python applications, we need to use MySQL for Python, a module that provides database support to our applications.

This book demonstrates how to boost the productivity of your Python applications by integrating them with the MySQL database server, the world's most powerful open source database. It will teach you to access the data on your MySQL database server easily with Python's library for MySQL using a practical, hands-on approach. Leaving theory to the classroom, this book uses real-world code to solve real-world problems with real-world solutions.

The book starts by exploring the various means of installing MySQL for Python on different platforms and how to use simple database querying techniques to improve your programs. It then takes you through data insertion, data retrieval, and error-handling techniques to create robust programs. The book also covers automation of both database and user creation, and administration of access controls. As the book progresses, you will learn to use many more advanced features of Python for MySQL that facilitate effective administration of your database through Python. Every chapter is illustrated with a project that you can deploy in your own situation.

By the end of this book, you will know several techniques for interfacing your Python applications with MySQL effectively so that powerful database management through Python becomes easy to achieve and easy to maintain.

What This Book Covers

Chapter 1, Getting Up and Running with MySQL for Python, helps you to install MySQL for Python specific software, how to import modules into your programs, connecting to a database, accessing online help, and creating a MySQL cursor proxy within your Python program. It also covers how to close the database connection from Python and how to access multiple databases within one program.

Chapter 2, Simple Querying, helps you to form and pass a query to MySQL, to look at user-defined variables, how to determine characteristics of a database and its tables, and program a command-line search utility. It also looks at how to change queries dynamically, without user input.

Chapter 3, Simple Insertion, shows forming and passing an insertion to MySQL, to look at the user-defined variables in a MySQL insertion, passing metadata between databases, and changing insertion statements dynamically without user input.

Chapter 4, Exception Handling, discusses ways to handle errors and warnings that are passed from MySQL for Python and the differences between them. It also covers several types of errors supported by MySQL for Python, and how to handle them effectively.

Chapter 5, Results Record-by-Record, shows situations in which record-by-record retrieval is desirable, to use iteration to retrieve sets of records in smaller blocks and how to create iterators and generators in Python. It also helps you in using `fetchone()` and `fetchmany()`.

Chapter 6, Inserting Multiple Entries, discusses how iteration can help us execute several individual INSERT statements rapidly, when to use or avoid `executemany()`, and throttling how much data is inserted at a time.

Chapter 7, Creating and Dropping, shows to create and delete both databases and tables in MySQL, to manage database instances with MySQL for Python, and to automate database and table creation.

Chapter 8, Creating Users and Granting Access, focuses on creating and removing users in MySQL, managing database privileges with MySQL for Python, automating user creation and removal, to GRANT and REVOKE privileges, and the conditions under which that can be done.

Chapter 9, Date and Time Values, discusses what data types MySQL supports for date and time, when to use which data type and in what format and range, and frequently used functions for handling matters of date and time.

Chapter 10, Aggregate Functions and Clauses, shows how MySQL saves us time and effort by pre-processing data, how to perform several calculations using MySQL's optimized algorithms, and to group and order returned data by column.

Chapter 11, SELECT Alternatives, discusses how to use HAVING clauses, how to create temporary subtables, subqueries and joins in Python, and the various ways to join tables.

Chapter 12, String Functions, shows how MySQL allows us to combine strings and return the single, resulting value, how to extract part of a string or the location of a part, thus saving on processing, and how to convert cases of results.

Chapter 13, Showing MySQL Metadata, discusses the several pieces of metadata about a given table that we can access, which system variables we can retrieve, and how to retrieve user privileges and the grants used to give them.

Chapter 14, Disaster Recovery, focuses on when to implement one of several kinds of database backup plans, what methods of backup and disaster recovery MySQL supports, and how to use Python to back up databases

4
Exception Handling

Any application that is used by multiple users in a production environment should have some level of exception handling implemented.

In this chapter, we will look at the following:

- Why errors and warnings are a programmer's friend
- The difference between errors and warnings
- The two main kinds of errors passed by MySQL for Python
- The six kinds of `DatabaseError`
- How to handle errors passed to Python from MySQL
- Creating a feedback loop for the user, based on the errors passed

At the end of this chapter, we will use this information along with the knowledge from the preceding chapters to build a command-line program to insert, update, and retrieve information from MySQL and to handle any exceptions that arise while doing so.

Why errors and warnings are good for you

The value of rigorous error checking is exemplified in any of the several catastrophes arising from poor software engineering. Examples abound, but a few are particularly illustrative of what happens when bad data and design go unchallenged.

On 4 June 1996, the first test flight of the Ariane 5 rocket self-destructed 37 seconds after its launch. The navigation code from Ariane 4 was reused in Ariane 5. The faster processing speed on the newer rocket caused an operand error. The conversion of a 64-bit floating-point value resulted in a larger-than-expected and unsupported 16-bit signed integer. The result was an overflow that scrambled the flight's computer, causing too much thrust to be passed by the rocket itself, resulting in the crash of US$370 million worth of technology. Widely considered to be one of the most expensive computer bugs in history, the crash arose due to mistakes in design and in subsequent error checking.

On 15 January 1990, the American telecommunications company AT&T installed a new system on the switches that controlled their long-distance service. A bug in the software caused the computers to crash every time they received a message from one of their neighboring switches. The message in question just happened to be the same one that the switches send out when they recover from a system crash. The result: Within a short time, 114 switches across New York City were rebooting every six seconds, leaving around 60,000 people without long distance service for nine hours. The system ultimately had to be fixed by reinstalling the old software.

On the Internet, a lack of proper error-checking still makes it possible for a malformed ping request to crash a server anywhere in the world. The **Computer Emergency Response Team** (**CERT**) Advisory on this bug, CA-1996-26, was released in 1996, but the bug persists. The original denial-of-service attack has thus evolved into the distributed denial-of-service attack employing botnets of zombie machines worldwide.

More than any other part of a computing system, errors cost significantly more to fix later than if they were resolved earlier in the development process. It is specifically for this reason that Python outputs error messages to the screen, unless such errors are explicitly handled otherwise.

A basic dynamic of computing is that the computer does not let anyone know what is happening inside itself. A simple illustration of this dynamic is as follows:

```
x = 2
if x == 2:
    x = x + x
```

Knowing Python and reading the code, we understand that the value of `x` is now `4`. But the computer has provided us no indication of the value of `x`. What's more, it will not tell us anything unless we explicitly tell it to do so. Generally speaking, there are two ways you can ask Python to tell you what it's thinking:

- By outputting values to the screen
- By writing them to a file

Here, a simple print statement would tell us the value of `x`.

Output displayed on the screen or saved to a file are the most common ways for programs to report their status to users. However, the similar effect is done by indicator lights and other non-verbal forms of communication. The type of output is necessarily dependent on the hardware being used.

By default, Python outputs all errors and warnings to the screen. As MySQL for Python is interpreted by Python, errors passed by `MySQLdb` are no different. This naturally gives the debugging programmer information for ironing out the performance of the program—whether determining why a program is not executing as planned or how to make it execute faster or more reliably. However, it also means that any information needed for tracing the error, along with parts of the code, is passed to the user, whoever they may be.

This is great for debugging, but makes for terrible security. That is why the Zen of Python reads:

> *Errors should never pass silently.*
>
> *Unless explicitly silenced.*

One needs the error messages to know why the program fails, but it is a security hazard to pass raw error messages to the user. If one wants the user to handle an error message, it should be sanitized of information that may compromise the security of the system.

Handling exceptions correctly takes a lot of code. At the risk of sounding like a hypocrite, it should be noted that the exigencies of a printed book do not allow for the reproduction of constant, rigorous error-handling in the code examples such as this chapter espouses. Therefore, while I state this principle, the programming examples included in this book do not always illustrate it as they should. If they did, the book would be significantly thicker and heavier (and probably cost more too!).

Further, the more complicated an application, the more robust the error-handling should be. Ultimately, every kind of error is covered by one of the several types that can be thrown by MySQL for Python. Each one of them allows for customized error messages to be passed to the user.

With a bit of further coding, one can check the authentication level of the user and pass error messages according to their level of authorization. This can be done through a flag system or by using modules from the Python library. If the former is used, one must ensure that knowledge of the flag(s) used is guarded from unauthorized users. Alternatively, one can employ both systems by checking the authentication level of users or programs that pass a particular flag to the program.

Errors versus warnings: There's a big difference

As with Python in general, the main difference between errors and warnings is that warnings do not cause a program to terminate. Errors do. Warnings provide notice of something we should note; errors indicate the reason the program cannot continue. If not handled appropriately, warnings therefore pass process information to the user without interrupting the execution of the program. This lack of detectability makes warnings more dangerous to the security of an application, and the system in general, than errors. Consequently, the error-handling process of an application must account for both errors and warnings.

While Python handles warnings and exceptions differently by default, especially with regard to program execution, both are written to `stderr`. Therefore, one handles them the same way that one handles standard errors (see *Handling exceptions passed from MySQL* in the later sections).

Additionally, one can set warnings to be silenced altogether or to carry the same gravity as an error. This level of functionality was introduced in Python 2.1. We will discuss this more later.

The two main errors in MySQLdb

Python generally supports several kinds of errors, and MySQL for Python is no different. The obvious difference between the two is that `MySQLdb`'s errors deal exclusively with the database connection. Where `MySQLdb` passes warnings that are not MySQL-specific, all exceptions are related to MySQL.

The MySQL-specific exceptions are then classified as either warnings or errors. There is only one kind of warning, but `MySQLdb` allows two categories of errors—`DatabaseError` and `InterfaceError`. Of the former, there are six types that we will discuss here.

DatabaseError

When there is a problem with the MySQL database itself, a `DatabaseError` is thrown. This is an intermediate catch-all category of exceptions that deal with everything from how the data is processed (for example, errors arising from division by zero), to problems in the SQL syntax, to internal problems within MySQL itself. Essentially, if a connection is made and a problem arises, the `DatabaseError` will catch it.

Several types of exceptions are contained by the `DatabaseError` type. We look at each of these in the section *Handling exceptions passed from MySQL*.

InterfaceError

When the database connection fails for some reason, `MySQLdb` will raise an `InterfaceError`. This may be caused from problems in the interface anywhere in the connection process.

Warnings in MySQL for Python

In addition to errors, MySQL for Python also supports a `Warning` class. This exception is raised for warnings like data truncation when executing an `INSERT` statement. It may be caught just like an `error`, but otherwise will not interrupt the flow of a program.

Handling exceptions passed from MySQL

MySQL for Python takes care of the nitty-gritty of communication between your program and MySQL. As a result, handling exceptions passed from MySQL is as straightforward as handling exceptions passed from any other Python module.

Python exception-handling

Python error-handling uses a `try...except...else` code structure to handle exceptions. It then uses `raise` to generate the error.

```
while True:
    try:
        x = int(raw_input("Please enter a number: "))
        break
    except:
        print "That is not a valid number.  Please try again..."
```

While this is the textbook example for raising an error, there are a few points to keep in mind.

```
while True:
```

This sets up a loop with a condition that applies as long as there are no exceptions raised.

```
    try...break
```

Python then tries to execute whatever follows. If successful, the program terminates with break. If not, an exception is registered, but not raised.

```
    except
```

The use of `except` tells Python what to do in the event of an exception. In this case it prints a message to the screen, but it does not raise an exception. Instead, the `while` loop remains unbroken and another number is requested.

Catching an exception from MySQLdb

All exceptions in MySQL for Python are accessed as part of `MySQLdb`. Therefore, one cannot reference them directly. Using the `fish` database, execute the following code:

```
#!/usr/bin/env python

import MySQLdb

mydb = MySQLdb.connect(host ='localhost',
                       user = 'skipper',
                       passwd = 'secret',
                       db = 'fish'
cur = mydb.cursor()

# Note the use of '7a' instead of '7'
statement = """SELECT * FROM menu WHERE id=7a"""

try:
    cur.execute(statement)
    results = cur.fetchall()
    print results
except Error:
    print "An error has been passed."
```

The preceding code will return a `NameError` from Python itself. For Python to recognize the exception from `MySQLdb`, change each instance of `Error` to read `MySQLdb.Error`. The `except` clause then reads as follows:

```
except MySQLdb.Error:
    print "An error has been passed."
```

The resulting output will be from the `print` statement.

```
An error has been passed.
```

Raising an error or a warning

An exception is only explicitly registered when `raise` is used. Instead of the `print` statement used in the `except` clause previously, we can `raise` an error and update the `print` statement with the following line of code:

```
raise MySQLdb.Error
```

Instead of the friendly statement about an error passing, we get a stack trace that ends as follows:

```
_mysql_exceptions.Error
```

Remember that `MySQLdb` is a macro system for interfacing with `_mysql_` and, subsequently, with the C API for MySQL. Any errors that pass from MySQL come through each of those before reaching the Python interpreter and your program.

Instead of raising an actual error, we can raise our own error message. After the `MySQLdb.Error` in the raise line, simply place your error message in parentheses and quotes.

```
raise MySQLdb.Error("An error has been passed.  Please contact your
system administrator.")
```

As shown here, the exact error message is customizable. If `raise` is simply appended to the preceding code as part of the `except` clause, the usual stack trace will be printed to `stdout` whenever the `except` clause is run. Note also that the flow of the program is interrupted whenever `raise` is executed.

The same process applies when raising a warning. Simply use `MySQLdb.Warning` and, if necessary, also use a suitable warning message.

Making exceptions less intimidating

For many users, program exceptions are a sign on par with Armageddon and tend to elicit the anxiety and mystery that accompany the usual view of that occasion. In order to be more helpful to users and to help users be more helpful to their IT support staff, it is good practice to give error messages that are explanatory rather than merely cryptic. Consider the following two error messages:

- Exception: `NameError` in line 256 of `someprogram.py`.
- The value you passed is not of the correct format. The program needs an integer value and you passed a character of the alphabet. Please contact a member of IT staff if you need further clarification on this error and tell them the error message is: "Unknown column '`7a`' in '`where` clause' on line 256 of `someprogram.py`".

Admittedly, the first takes up less space and takes less time to type. But it also is guaranteed to compromise the usefulness of your program for the user and to increase the number of phone calls to the IT helpdesk. While the second may be a bit longer than necessary, the user and helpdesk will benefit from a helpful message, regardless of its verbosity, more than an overly technical and terse one.

To accomplish a user-friendly error message that nonetheless provides the technical information necessary, catch the exception that Python passes. Using the previous `if...except` loop, we can catch the error without the traceback as follows:

```
#!/usr/bin/env python


import MySQLdb

mydb = MySQLdb.connect(host ='localhost',
                       user = 'skipper',
                       passwd = 'secret',
                       db = 'fish')
cur = mydb.cursor()

statement = """SELECT * FROM menu WHERE id=7a"""

try:
    cur.execute(statement)
    results = cur.fetchall()
    print results
except MySQLdb.Error, e:
    print "An error has been passed. %s" %e
```

Now when the program is executed, we receive the following output:

```
An error has been passed. (1054, "Unknown column '7a' in 'where clause'")
```

This could easily be revised to similar wording as the second of the two error examples seen just now.

Catching different types of exceptions

It is typically best practice to process different types of exceptions with different policies. This applies not only to database programming, but to software development in general. Exceptions can be caught with a generic `except` clause for simpler implementations, but more complex programs should process exceptions by type.

In Python, there are 36 built-in exceptions and 9 built-in warnings. It is beyond the scope of this book to go into them in detail, but further discussion on them can be found online.

For exceptions see:

`http://python.about.com/od/pythonstandardlibrary/a/lib_exceptions.htm`

For warnings visit:

`http://python.about.com/od/pythonstandardlibrary/a/lib_warnings.htm`

The Python documentation also covers them at:

`http://docs.python.org/library/exceptions.html`

Types of errors

The following are the six different error types supported by MySQL for Python. These are all caught when raising an error of the respective type, but their specification also allows for customized handling.

- `DataError`
- `IntegrityError`
- `InternalError`
- `NotSupportedError`
- `OperationalError`
- `ProgrammingError`

Each of these will be caught by using `DatabaseError` in conjunction with an `except` clause. But this leads to ambiguous error-handling and makes debugging difficult both for the programmer(s) who work on the application as well as the network and system administrators who will need to support the program once it is installed on the end user's machine.

DataError

This exception is raised due to problems with the processed data (for example, numeric value out of range, division by zero, and so on).

IntegrityError

If the relational integrity of the database is involved (for example a foreign key check fails, duplicate key, and so on), this exception is raised. To illustrate this, save the following code into a file for inserting data into the `fish` database that we have used in previous chapters:

```
#!/usr/bin/env python

import MySQLdb, sys

mydb = MySQLdb.connect(host ='localhost',
                       user = 'skipper',
                       passwd = 'secret',
                       db = 'fish')
cur = mydb.cursor()

ident = sys.argv[1]
fish = sys.argv[2]
price = sys.argv[3]

statement = """INSERT INTO menu(id, name, price) VALUES("%s", "%s",
"%s")""" %(ident, fish, price)
print "Data has been inserted using the following statement: \n",
statement
cur.execute(statement)
```

Change the database login information as necessary. Then call it with an existent value for the identifier. For example:

```
> python temp.py 2 swordfish 23
```

The type of error follows a multiple line traceback:

```
_mysql_exceptions.IntegrityError: (1062, "Duplicate entry '2' for key
'PRIMARY'")
```

InternalError

This exception is raised when there is an internal error in the MySQL database itself (for example, an invalid cursor, the transaction is out of sync, and so on). This is usually an issue of timing out or otherwise being perceived by MySQL as having lost connectivity with a cursor.

NotSupportedError

MySQL for Python raises this exception when a method or database API that is not supported is used (for example, requesting a transaction-oriented function when transactions are not available. They also can arise in conjunction with setting characters sets, SQL modes, and when using MySQL in conjunction with **Secure Socket Layer** (**SSL**).

OperationalError

Exception raised for operational errors that are not necessarily under the control of the programmer (for example, an unexpected disconnect, the data source name is not found, a transaction could not be processed, a memory allocation error occurrs, and so on.). For example, when the following code is run against the `fish` database, MySQL will throw an `OperationalError`:

```
#!/usr/bin/env python


import MySQLdb, sys
mydb = MySQLdb.connect(host = 'localhost',
                       user = 'skipper',
                       passwd = 'secret',
                       db = 'fish')
cur = mydb.cursor()

statement = """SELECT * FROM menu WHERE id=7a"""
cur.execute(statement)
results = cur.fetchall()
print results
```

The error message reads as follows:

```
SELECT * FROM menu WHERE id=7a
Traceback (most recent call last):
  File "temp.py", line 54, in <module>
    cur.execute(statement)
  File "/usr/local/lib/python2.6/dist-packages/MySQL_python-1.2.3c1-
py2.6-linux-i686.egg/MySQLdb/cursors.py", line 173, in execute
  File "/usr/local/lib/python2.6/dist-packages/MySQL_python-
1.2.3c1-py2.6-linux-i686.egg/MySQLdb/connections.py", line 36, in
defaulterrorhandler
_mysql_exceptions.OperationalError: (1054, "Unknown column '7a' in 'where
clause'")
```

ProgrammingError

Exception raised for actual programming errors (for example, a table is not found or already exists, there is a syntax error in the MySQL statement, a wrong number of parameters is specified, and so on.). For instance, run the following code against the `fish` database:

```
#!/usr/bin/env python

import MySQLdb

mydb = MySQLdb.connect(host ='localhost',
                       user = 'skipper',
                       passwd = 'secret',
                       db = 'fish')
cur = mydb.cursor()
fish = sys.argv[1]
price = sys.argv[2]
statement = """INSERTINTO menu(name, price) VALUES("%s", "%s")"""
%(fish, price)
print "Data has been inserted using the following statement: \n",
statement
cur.execute(statement)
```

The values you pass as arguments do not matter. The syntactic problem in the MySQL statement will cause a `ProgrammingError` to be raised:

```
_mysql_exceptions.ProgrammingError: (1064, 'You have an error in your
SQL syntax; check the manual that corresponds to your MySQL server
version for the right syntax to use near \'INSERTINTO menu(name, price)
VALUES("jellyfish", "27")\' at line 1')
```

Customizing for catching

Each of the previous types can be caught with the `DatabaseError` type. However, catching them separately allows you to customize responses. For example, you may want the application to fail softly for the user when a `ProgrammingError` is raised but nonetheless want the exception to be reported to the development team. You can do that with customized exception handling.

Catching one type of exception

To catch a particular type of exception, we simply include that type of exception with the `except` clause. For example, to change the code used for the `OperationalError` in order to catch that exception, we would use the following:

```
#!/usr/bin/env python


import MySQLdb, sys
mydb = MySQLdb.connect(host = 'localhost',
                       user = 'skipper',
                       passwd = 'r00tp4ss',
                       db = 'fish')
cur = mydb.cursor()

statement = """SELECT * FROM menu WHERE id=7a"""

try:
    cur.execute(statement)
    results = cur.fetchall()
    print results
except MySQLdb.OperationalError, e:
    raise e
```

After the traceback, we get the following output:

```
_mysql_exceptions.OperationalError: (1054, "Unknown column '7a' in 'where clause'")
```

You can similarly catch any of MySQL for Python's `error` types or its `warning`. This allows much greater flexibility in exception-handling. We will see more of this in the project at the end of the chapter.

Catching different exceptions

To customize which error is caught, we need different `except` clauses. The basic structure of this strategy is as follows:

```
try:
    <do something>
except ErrorType1:
    <do something>
except ErrorType2:
    <do something else>
```

To combine the examples for the `OperationalError` and `ProgrammingError` that we just saw, we would code as follows:

```
#!/usr/bin/env python


import MySQLdb, sys
mydb = MySQLdb.connect(host ='localhost',
                       user = 'skipper',
                       passwd = 'secret',
                       db = 'fish')
cur = mydb.cursor()

identifier = sys.argv[1]

statement = """SELECT * FROM menu WHERE id=%s""" %(identifier)

try:
    cur.execute(statement)
    results = cur.fetchall()
    print results
except MySQLdb.OperationalError, e:
    raise e
except MySQLdb.ProgrammingError, e:
   raise e
```

After writing this into a file, execute the program with different arguments. A definite way to trigger the `OperationalError` is by passing a bad value like `7a`. For the `ProgrammingError`, try an equals sign.

One can do more than simply `print` and `raise` exceptions in the `except` clause. One can also pass system calls as necessary. So, for example, one could pass programming variables to a function to send all errors to a set address by a protocol of your choosing (SMTP, HTTP, FTP, and so on.). This is essentially how programs such as Windows Explorer, Mozilla Firefox and Google's Chrome browsers send feedback to their respective developers.

Combined catching of exceptions

It is not uncommon to want to handle different errors in the same way. To do this, one simply separates the errors by a comma and includes them within parentheses after `except`. For example, the preceding program could be rewritten as follows:

```
#!/usr/bin/env python


import MySQLdb, sys
mydb = MySQLdb.connect(host ='localhost',
                       user = 'skipper',
                       passwd = 'secret',
                       db = 'fish'))
cur = mydb.cursor()

identifier = sys.argv[1]

statement = """SELECT * FROM menu WHERE id=%s""" %(identifier)

try:
    cur.execute(statement)
    results = cur.fetchall()
    print results
except (MySQLdb.OperationalError, MySQLdb.ProgrammingError), e:
    raise e
```

As with most parts of Python, enclosing the two error types in parentheses tells Python to accept either one as the error type. However, only one will be output when `e` is raised.

Raising different exceptions

Python will raise whatever valid type of error you pass to the `raise` statement. It is not particularly helpful to raise the wrong exceptions. However, particularly when debugging or auditing code that has been copied from another project, one should be aware that it can be done (as in the story of Ariane 5, at the beginning of this chapter).

To illustrate this using the code listing previously, change the `except` clause to read as follows:

```
except (MySQLdb.OperationalError, MySQLdb.ProgrammingError), e:
    raise ValueError
```

Then pass bad arguments to the program like `7a` or `?`.

Note that any use of `raise` will provide a stack trace. If you do not want a stack trace printed, then you should simply use a `print` statement to output the error message, as shown earlier in this chapter.

Creating a feedback loop

Being able to follow different courses of action based on different exceptions allows us to tailor our programs. For a `DataError` or even a `ProgrammingError`, we may want to handle the exception behind the scenes, hiding it from the user, but passing critical information to the development team. For `Warning` or non-critical errors, we may choose to pause execution of the program and solicit more information from the user. To do this, we would use a `raw_input` statement as part of the `except` clause. In order to be concise, the following program treats errors and warnings the same way, but they could easily be separated and treated with greater granularity.

```
#!/usr/bin/env python


import MySQLdb, sys

mydb = MySQLdb.connect(host ='localhost',
                       user = 'skipper',
                       passwd = 'secret',
                       db = 'world')
cur = mydb.cursor()

identifier = sys.argv[1]
statement = """SELECT * FROM City WHERE ID=%s""" %(identifier)

while True:
    try:
        print "\nTrying SQL statement: %s" %(statement)
        cur.execute(statement)
        results = cur.fetchall()
        print "The results of the query are:"
        print results
        break
    except (MySQLdb.Error, MySQLdb.Warning):
        new_id = raw_input("The city ID you entered is not valid.
Please enter a valid city ID:  ")
        print "Using the new city ID value '%s'" %(new_id)
        statement = """SELECT * FROM City WHERE ID=%s""" %(new_id)
```

Project: Bad apples

Bad apples come in all shapes and sizes. Some are users; some are staff. Either is capable of giving the computer bad data or the wrong commands. MySQL, on the other hand, validates all data against the database description. As mentioned earlier in this book, all statements have to be made according to a set syntax. If there is a mismatch along the way, an exception is thrown.

For this project, therefore, we will implement a program to do the following:

- Insert and/or update a value in a MySQL database table
- Retrieve data from the same table
- Handle MySQL errors and warnings
- Notify the appropriate staff

Exactly how these elements are implemented will naturally differ depending on your local dynamics. Further, there are plenty of other checks beyond these that one could include. For example, if you have a whitelist or blacklist against which you can check input data, it would follow to include that check with what we implement here. The point of this project is to handle any error that MySQL can throw and to do so appropriately without resorting to generic exception-handling. Depending on how you code it, not all errors need to be handled because not all of them will ever be thrown. However, here we aim for comprehensiveness if merely for the exercise.

For this project, we will use the following functions:

- `connection()`: To create the database connection
- `sendmail()`: To send error messages to different maintainers (for example, database administrator)

Additionally, we will have a class `MySQLStatement` with the following methods and attributes (`__init__` naturally being assumed):

- `type`: Attribute of the instance to indicate what kind of statement is being processed
- `form()`: Method to form the MySQL statement
- `execute()`: Sends the SQL statement to MySQL and receives any exceptions

All of these will once again be controlled by `main()`.

The preamble

Before coding the functionality mentioned, we need to attend to the basics of the `shebang` line and `import` statements. We therefore start with the following code:

```
#!/usr/bin/env python

import MySQLdb
import optparse
import sys
```

After this, we need to include support for options:

```
# Get options
opt = optparse.OptionParser()
opt.add_option("-i", "--insert", action="store_true", help="flag
request for insertion - only ONE of insert, update, or select can be
used at a time", dest="insert")
opt.add_option("-u", "--update", action="store_true", help="flag
request as an update", dest="update")
opt.add_option("-s", "--select", action="store_true", help="flag
request as a selection", dest="select")
opt.add_option("-d", "--database", action="store", type="string",
help="name of the local database", dest="database")
opt.add_option("-t", "--table", action="store", type="string",
help="table in the indicated database", dest="table")
opt.add_option("-c", "--columns", action="store", type="string",
help="column(s) of the indicated table", dest="columns")
opt.add_option("-v", "--values", action="store", type="string",
help="values to be processed", dest="values")
opt, args = opt.parse_args()

# Only one kind of statement type is allowed.  If more than one is
indicated, the priority of assignment is SELECT -> UPDATE -> INSERT.
if opt.select is True:
    statement_type = "select"
elif opt.update is True:
    statement_type = "update"
elif opt.insert is True:
    statement_type = "insert"
```

Then, as a matter of style, we assign the option values to more generic variable names for ease of handling:

```
database = opt.database
table = opt.table
columns = opt.columns
values = opt.values
```

Making the connection

The first function we define is `connection()`. This is called with the name of the database as specified by the user. For security reasons, we do not allow the user to specify the host. Also, for reasons of simplicity, we will hardwire the login credentials into the function.

```
def connection(database):
    """Creates a database connection and returns the cursor.  Host is
hardwired to 'localhost'."""
    try:
        mydb = MySQLdb.connect(host ='localhost',
                               user = 'skipper',
                               passwd = 'secret',
                               db = '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
```

If the connection is made successfully, the function returns the cursor to the calling function. All errors at this point are fatal. We print a message to guide the user and then exit. All warnings are passed silently.

Sending error messages

Next is a very simple SMTP server that we use to send messages through localhost. To our import statements at the beginning of the program file, we need to add:

```
import smtplib
```

Then we can call it in a function as follows:

```
def sendmail(message, recipient):
    """Sends mail through localhost.  Takes error message and intended
recipient as arguments."""



    fromaddr = "pythonprogram@someaddress.com"
    toaddrs  = recipient + "@someaddress.com"

    # Add the From: and To: headers at the start!
    msg = ("From: %s\r\nTo: %s\r\n\r\n" %(fromaddr,toaddrs)
```

```
    msg = msg + str(message[0]) + message[1]

    server = smtplib.SMTP('localhost')
    server.set_debuglevel(1)
    server.sendmail(fromaddr, toaddrs, msg)
    server.quit()
```

Before running this code, you will naturally want to change the value of the SMTP server to be used from `localhost` if you do not have a mail server on your local system. Also note that many ISPs verify the existence of domains prior to forwarding a message, so you should also check your values in that regard before using this function.

By allowing the function to receive both the message and the recipient, we can reuse this code for different exceptions. As noted in the Python documentation on `smtplib`, this is a simple example and does not handle all of the dynamics of RFC822. For more on RFC822, see: `http://www.faqs.org/rfcs/rfc822.html`.

This code is derived from the documentation on `smtplib`, which can be found at `http://www.python.org/doc/2.5.2/lib/SMTP-example.html`

This example assumes that you are running an SMTP server on your local machine and so directs `smtplib`'s `SMTP` class to use `localhost`. However, you can easily adapt this to any SMTP server by using the `login` method of the `SMTP` class. For example, the function could read:

```
fromaddr = "me@example.com"
toaddr = "you@example.com"
msg = ("From: %s\r\nTo: %s\r\b\r\n From me to you" %(fromaddr,
toaddr))

server = smtplib.SMTP('mail.example.com')
server.login('me@example.com', 'secretpass')
server.set_debuglevel(0)
server.sendmail(fromaddr, toaddr, msg)
server.quit()
```

If you need to use a particular port on the server, simply append it to the server address you use to instantiate the SMTP object:

```
server = smtplib.SMTP('mail.example.com:587')
```

An example of how to do this with Google's email service can be found at `http://www.nixtutor.com/linux/send-mail-through-gmail-with-python/`.

For reasons of spam and other security issues, you will want to hardwire as many variables as is reasonably possible into this function. For example, you would not want to allow the domain name of either the recipient or the sender to be dynamically set. If you had to send an error message to someone who is not on the domain, it is better to set up mail-forwarding from an address on that domain than to allow anyone with access to the module to send messages to and from random domains while using your server.

Finally, while you are developing the program, it is helpful to keep `server.set_debuglevel` switched on and set to 1. When the program is moved into regular use, however, you will want to change it to 0, so the user does not see the debugging messages output by the server.

The statement class

Next we need to write the `MySQLStatement` class. Instances of this class will have as an attribute what kind of class they are. Further, they will have methods to form and execute an SQL statement that incorporates the user's input in the appropriate kind of statement.

The __init__ method

First, we need to give the class a conscience. We do this with the `__init__()` method:

```
class MySQLStatement:
    def __init__(self):
        """Creates an instance to form and execute a MySQL
statement."""
        self.Statement = []
```

You will notice that we are starting to use **docstrings**. As the code gets more complex and there is an increased likelihood that we will reuse it, we will start using docstrings with greater frequency.

If you are unclear on what docstrings are and why you should use them, see this rationale for their use: `http://www.python.org/dev/peps/pep-0257/#rationale`.

We have no need to specify otherwise, so every instance of MySQLStatement will inherit the generic characteristics of a Python object.

Storing the statement type

The `statement` type is the only attribute that we will have in this class. Simply put, it stores the value that we pass to it. We could store the same value in a variable within the `main()` function and pass it as an argument to the object methods, but that would not leave us with re-usable code. Having the type as an attribute of the object ensures that we can access this same object as a module import instead of calling the entire program directly.

The code is as follows:

```
    def type(self, kind):
        """Indicates the type of statement that the instance is.
Supported types are select, insert, and update.  This must be set
before using any of the object methods."""
        self.type = kind
```

Here we do not challenge the value of kind when this attribute is invoked. Rather, we allow any value to be passed to the `main()` function. We will talk more about verifying the type in *Room to grow*, later in this chapter.

Forming the statement

The first method that we will code forms the SQL statement from the user's data. The code for this method is as follows:

```
    def form(self, table, column, info):
        """Forms the MySQL statement according to the type of
statement, using table as the tablename, column for the fields, and
info for values"""

        data = info.split(',')
        value = "'" + data[0]
        for i in xrange(1, len(data)):
            value = value + "', '" + data[i]
        value = value + "'"

        if self.type == "select":
            statement = """SELECT * FROM %s WHERE %s=%s""" %(table,
column, value)
            return statement
        elif self.type == "insert":
            statement = """INSERT INTO %s(%s) VALUES(%s)""" %(table,
column, value)
            return statement
        elif self.type == "update":
            statement = """UPDATE %s SET %s=%s WHERE %s=%s""" %(table,
column, data[0], column, data[1])
            return statement
```

If the same value is given for the two columns, the update will effectively be a replacement. If they differ, the first will be where the update is applied and the second will indicate the condition under which it is to be affected.

As usual in MySQL for Python, we leave off the semicolon at the end of the statement.

Different MySQL statements allow for comma-delimited values to be passed. Some don't. While comma-separated values without quotes are fine for the column names, the values must have quotes to have meaning (otherwise, they are read as variable names).

Even if we insisted the user pass values in quotes, we would still have the problem of getting the `optparse` module to recognize all of them as such. Therefore, we create a small routine to split the user's values on the comma and to insert quotes around each value.

Depending on which type of statement is held in the type attribute, we either form a `SELECT`, `INSERT`, or `UPDATE` statement. To process these options, we use a series of `if...elif...elif`.

Up to now, we have not covered the MySQL `UPDATE` statement. Up to now, we have not covered the MySQL `UPDATE` statement. `UPDATE` is similar to `REPLACE` in that it changes values that are already entered into the database. Where `REPLACE` affects changes in old rows and otherwise functions like `INSERT`, `UPDATE` impacts multiple rows at a time, wherever the given condition is met. As seen here, the basic syntax of `UPDATE` is:

```
UPDATE <table> SET <column>='<new value>' WHERE <column>='<old
value>';
```

Execute the MySQL statement

The final method of the class is `execute()`. As the name implies, this method accepts the statement to be processed. It also includes the table of the database specified by the user as well as the cursor returned by `connection()`. Here we include the majority of our exception-handling:

```
    def execute(self, statement, table, cursor):
        """Attempts execution of the statement resulting from
MySQLStatement.form()."""
        while True:
            try:
                print "\nTrying SQL statement: %s\n\n" %(statement)
                cursor.execute(statement)

                if self.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"
                    return results

                elif self.type == "insert":
                    results = "Your information was inserted with the
following SQL statement: %s" %(statement)
                    return results
                elif self.type == "update":
                    results = "You updated information in the database
with the following SQL statement: %s" %(statement)
                    return results
```

Handling any fallout

If there is a failure along the way in the process, we need to handle the exceptions that are raised. To do so, we use a series of `except` clauses to process the different exceptions accordingly. In the following code, where the different exceptions are noted is indicated by comments:

```
            # OperationalError
            except MySQLdb.OperationalError, e :
                sendmail(e, "pythondevelopers")
                print "Some of the information you have passed is not
valid.  Please check it before trying to use this program again.  You
may also use '-h' to see the options available."
                print "The exact error information reads as follows:
%s" %(e)
                raise

            # DataError
            # ProgrammingError
            except (MySQLdb.DataError, MySQLdb.ProgrammingError), e:
                sendmail(e, "pythondevelopers")
                print "An irrecoverable error has occured in the way
your data was to be processed.   This application must now close.  An
error message describing the fault has been sent to the development
team.  Apologies for any inconvenience."
                raise
```

```
            # IntegrityError
            except MySQLdb.IntegrityError, e:
                sendmail(e, "dba")
                print "An irrecoverable database error has occurred
and this process must now end.  An error message describing the fault
has been sent to the database administrator.  Apologies for any
inconvenience."
                raise

            # InternalError
            # NotSupportedError
            except (MySQLdb.InternalError, MySQLdb.NotSupportedError),
e:
                sendmail(e, "dba")
                sendmail(e, "pythondevelopers")
                print "An irrecoverable error has occurred and this
process must now end.  An error message describing the fault has been
sent to the appropriate staff.  Apologies for any inconvenience."
                raise

            except MySQLdb.Warning:
                pass
```

Some errors can be handled more or less the same way. Instead of creating separate `except` clauses for each, we group them. Similarly, different exceptions may require addressing by more than one team. So in the last `except` clause, processing internal and not supported errors, we send the error message to both the database administrator as well as the team who maintains the program. Finally, warnings are passed over.

The main() thing

As usual, the `main()` function is the brains of our program. As options are set earlier in the program, `main()` is left to instantiate the `MySQLStatement` object and pull the puppet strings to get the functions and methods to form the appropriate statement and execute it. If there is a failure along the way, we want to field it accordingly.

Try, try again

The main actions of `main()` begin as follows:

```
    request = MySQLStatement()
    try:
        request.type(statement_type)
        phrase = request.form(table, columns, values)
        cur = connection(database)
        results = request.execute(phrase, table, cur)
        print "Results:\n", results
        cur.close()
```

We first create an object `request`. This is the only part of `main()` that is definite; the rest is performed under the caveat of `try`. If there is a failure along the way, the process is scrubbed and an exception is processed.

Within the `try` clause, we first set the type of statement to be formed. Note that the way we coded the `type` attribute, allows any value to be set. Similarly here, the value of `type` is not validated.

Following on from setting `type`, we pass the table name, the column(s), and value(s) to be used to the `form()` method. Depending on the value of type, `form()` will return one of the three supported statements. This is stored in `phrase`.

Next, we need a cursor. For this, we call the `connection()` function. The cursor is then returned and given the rubric `cur`.

We then pass `phrase` and `cur`, along with the name of the table in question, to the execute method of our object. `MySQLStatement.execute()` returns the output of a selection or otherwise returns a positive statement if the process has been successful. The results are then printed to screen.

If all else fails

If there should be a failure and the `try` statement does not succeed, we can pick up the pieces and move on with the following `except` clauses:

```
    except MySQLdb.Error, e:
        sendmail(e, "pythondevelopers")
        print "The values you entered are not valid.  Please check the
information you are using before trying again."
        print "The SQL statement that was tried reads as follows: %s"
%(statement)
        raise MySQLdb.Error

    except MySQLdb.Warning:
        pass
```

Room to grow

The program discussed previously will process data given in a set format and do one of three things with it (`SELECT` information from the `fish` database, `INSERT` new data, or `UPDATE` old data to new.). Using various forms of exception-handling, it also process any error that MySQL throws.

Despite all this code, however, there is quite a bit that it does not do. Some points for you to consider when further developing this code are as follows:

- How would you implement handling of Python-specific exceptions (for example, `NameErrors`, `KeyErrors`, and so on)?
- Should you modify the `type` attribute to be self-validating? If so, how would you do it?
- Currently, warnings are passed silently; how would you handle them more securely?
- How would you change the `UPDATE` feature to handle more than one column at a time (that is, change the value in column `price` according to the value of column `name`)?
- How would you implement new features such as support for `REPLACE` or `DELETE`?
- The e-mail messages can serve as a makeshift log of the different errors, but there is currently no central listing. How would you implement a database for logging exceptions and their given messages?
- Currently, the error messages that are sent are still pretty vague. What kind of information would you want to pass if this were a web application? What if it were a desktop application? How would you gather that information and send it in a feedback message?
- The program currently prints the results of a `SELECT` query to the screen without column headers. How would you affect them using the `PrettyTable` module mentioned in the previous chapter?

Summary

In this chapter, we have covered how to handle errors and warnings that are passed from MySQL for Python and the differences between them. We have also looked at how to pass errors silently and when, the several types of errors supported by MySQL for Python, and how to handle errors properly.

In the next chapter, we will look at how to retrieve single and multiple records efficiently, ensuring we do not use resources needlessly or for longer than necessary.

Where to buy this book

You can buy MySQL for Python from the Packt Publishing website:
`https://www.packtpub.com/mysql-for-python-database-access-made-easy/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/mysql-for-python-database-access-made-easy/book