Python exercises

Deep Dive with Apache Derby: Perl, PHP, and Python

OSCON

August 2, 2005

Dan Scott

dan.scott@acm.org / dan.scott@ca.ibm.com

Objectives:

- Introduce the Python DB-API 2.0
- Set up support for the pyDB2 module
- Create a Web page that pulls data from Apache Derby
- Create a Web page that inserts data into Apache Derby

Quick facts

Python was created by Guido van Rossum as an object-oriented scripting language. (In)famous for its use of indentation, rather than braces, to define code blocks. Some of the best-known Python projects are BitTorrent peer-to-peer, SpamBayes spam filtering, and MailMan mailing list manager.

Command line interface

Python includes a command line interface that is useful for testing code or language constructs on the fly. To start the command line interface, simply invoke the **python** executable. You will be greeted by a >>> prompt, where you can begin typing your Python statements. To close the command line, press **CTRL-Z**, followed by return.

1. Introducing the Python DB-API 2.0

The standard database interface for Python applications is the Python DB-API 2.0. Two implementations offer access to Apache Derby:

- pyDB2, written by Man-Yong Lee, is an LGPL module
- mxODBC, written by Marc-Andre Lemburg, is a commercial module

This section focuses on the pyDB2 module as it is truly free and offers most of the features defined by the Python DB-API 2.0.

2. Building and configuring the pyDB2 module

Prerequisites:

- Installed DB2 Runtime Client with application development support
- Installed Python 2.3.x or 2.4.x
- (Linux): Download pyDB2 1.1 source from ftp://people.linuxkorea.co.kr/pub/DB2/src/PyDB2-devel-v1.1.tar.gz
- (Linux): Application development tools and libraries such as python-devel, gcc, autoconf, automake, bison, and flex.

Configuring and compiling

On Windows, a binary module offering pyDB2 support is available from ftp://people.linuxkorea.co.kr/pub/DB2/win32/.

On Linux, you can compile pyDB2 support against the DB2 Runtime Client libraries by following these steps:

```
bash$ tar xzf PyDB2-devel-v1.1.tar.gz
bash$ cd v1.1
bash$ python setup.py build
bash$ su -c 'python setup.py install'
```

This compiles pyDB2 and installs the module in the Python module library.

Testing the installation

Test that you have successfully built and installed pyDB2 by issuing the following commands:

```
bash$ python
>>> import DB2
>>> print DB2
<module 'DB2' from 'DB2.py'>
>>> print DB2.connect
DB2.Connection
```

Cataloging the database

Currently, pyDB2 only supports *cataloged connections*. These are connections that use the DB2 Runtime Clients directory of database servers (nodes) and databases to avoid having to explicitly pass the hostname, port number, and protocol for the database to which you want to connect.

To catalog the database, open a command line and issue the following commands:

```
bash$ db2 catalog tcpip node DERBY remote LOCALHOST server 1527
bash$ db2 catalog database MYDB at node derby authentication server
```

The authentication server option is required, but introduces an additional requirement of running Apache Derby under a Java Runtime Environment that includes the IBM version of the Java Cryptographic Extensions (JCE) to support the encryption that this option uses.

3. Displaying data in a Web page

We'll start by creating a Python script that simply creates a connection to your Apache Derby database and either confirms the connection, or explains why the connection failed.

3a: Creating a function library

Create a new script named **`menu.py`** containing the following code:

```
#!/usr/bin/python
import DB2

def connect(database='MYDB', user='lynn', password='5tuff'):
    dbh = None
    dbh = DB2.connect(database, user, password)
    return dbh
```

3b: Connecting to the database through pyDB2

Create a new script named **`connect.py`** containing the following code:

```
#!/usr/bin/python
import sys
import DB2

# import the functions in menu.py within this script
import menu

# Create the connection with default parameter values
try:
    conn = menu.connect()

except Exception, e:
    sys.exit("Failed to connect: %s" % e)

# Here is where we would do real work; just print connection status
print "Connection succeeded!"

# Clean up the connection
conn.close()
```

Test the script from the command line to ensure the connection succeeds:

```
bash$ python connect.py
```

We will use the **`connect.py`** script as the basis for the rest of our PHP scripts by fleshing it out with real PHP code and HTML output.

3c. Retrieving data from Apache Derby

So far we have been working strictly with the Connection object defined in the Python DB-API 2.0 specification. To interact with SQL objects, however, you need to create a Cursor object. The Cursor object issues all SQL statements and provides the interface for retrieving all results from the database.

Python comes with a base module named **`cgi`** that implements the functionality required by the Common Gateway Interface specification. We will use this in our script to retrieve the values of the form input fields, then build our SQL statement and retrieve the results.

Create a new script named **`search.py`** containing the following code, copying the contents of the search form **`search.html`** you created during the PHP exercises into the **`header()`** method . Modify the form to call search.py:

```
#!/usr/bin/python
import sys
import cgi
import menu
import re

def header(title):
  header = "Content-type: text/html\n\n"
  header = header + """<html>
  <head><title>""" + title + """</title></head>
```

```
  <body><h1>""" + title + '</h1>'

  # add modified search code here
  header = header + """<form action='search.py' method='POST'>
      <label for='sinput'>Please enter the search string:</label>
      <input type='text' name='search' id='sinput' size='30'/><br />
      <input type='submit'/><input type='reset'/>
    </form>
  """

  return header

def footer():
  return "</body></html>"

if __name__ == '__main__':
  search = None
  form = cgi.FieldStorage()

  if (form.has_key('search')):
    alphabet = re.compile('^([^\W_]+)$')
    matches = alphabet.search(form['search'])
    if (matches):
      search = form['search']

  if (not search):
    print header('Search for food')
    print footer()
    sys.exit()

  print header('Searching for food...')
  try:
    conn = menu.connect()

    # create a Cursor object
    curs = conn.cursor()

    sql = """SELECT name, description
      FROM menu.food
      WHERE name LIKE ?"""

    # issue the SQL statement
    curs.execute(sql, search)

    # fetch the first row from the result set
    result = curs.fetchone()

    if result:
      while result:
        print("<p>Name: %s <br />Description: %s</p>\n") % (result[0], result
[1])
        result = curs.fetchone()
    else:
      print "Failed to find any search results for that string."

  except Exception, e:
    print "SQL failure: %s" % e
```

```
print footer()
```

Load **`search.py`** in your Web browser and ensure that it returns the expected results for known searches. Test your filtering code to ensure it does not allow unwanted characters.

Improve the error-handling to issue an error message when illegal characters have been passed to the search script without introducing a new vulnerability.

4. Demonstrating the use of SQL triggers

Apache Derby supports *triggers* -- SQL objects that defines a set of actions to be performed when specific delete, update, or insert events occur in a specified table in the database. In the following example, we create a new table **`menu.counter`** that contains a single row to track the number of items within the **`menu.food`** table.

```
CREATE TABLE menu.counter (id INTEGER, num INTEGER);
INSERT INTO menu.counter(id, num) values (1, 1);
CREATE TRIGGER incrementCount
    AFTER INSERT
    ON menu.food FOR EACH ROW MODE DB2SQL
    UPDATE menu.counter SET num = num + 1 WHERE id = 1;
CREATE TRIGGER decrementCount
    AFTER DELETE
    ON menu.food FOR EACH ROW MODE DB2SQL
    UPDATE menu.counter SET num = num - 1 WHERE id = 1;
```

4a. Testing the SQL triggers

Issue a number of INSERT and DELETE statements against the **`menu.food`** table, then check the value of the **`menu.counter`** table. If you issue a single INSERT statement that inserts multiple rows, is the **`menu.counter`** table incremented appropriately?

4b. Inserting and deleting data

Copy and modify the **`search.py`** script so that it enables a user to add or remove food items from the online menu. Notice that none of the modifications to the data actually stay in the database -- this is because Python automatically sets the database to be in transaction mode, and automatically rolls back the transaction if the application does not explicitly call the **`commit()`** or **`rollback()`** methods on the Connection object.

Add an explicit **`commit()`** call to commit the transaction.

Python Syntax

Variables

Variables in Python have no special signifier characters, and (like Perl and PHP) are dynamically typed. The basic data types Python supports are:

Boolean

A simple true or false value, where the initial letter must be capitalized.

```
>>> x = True
>>> y = False
```

Numeric (integer, long integer, float, and complex)

```
>>> integer = 6
>>> longinteger = 20L
>>> float_ = 3.5
>>> complex = 10.6 + 9.2J
>>> float = 3.5
```

String

```
>>> string = """This is a multiline
string, it keeps on going until you hit the closing
triple double quotation marks"""
>>> ipolate = "An integer: %d - and a string: %s" % (integer, "boo")
>>> print ipolate
An integer: 6 - and a string: boo
```

Sequence (tuples and lists)

A equence is a 0-indexed set of objects that you create by assigning a comma-delimited collection of elements to a variable. Tuples are immutable (cannot be changed), while lists are mutable.

Yes, they are very similar. The difference is that, to create a tuple, you can enclose the collection of elements in parentheses ().

To create a list, you must enclose the collection of elements in braces [].

```
>>> # Tuple -- immutable sequence
>>> tuple = (1, 2, 4, 3)
>>> print tuple
(1, 2, 4, 3)
>>> print tuple[0]
1
>>> print len(tuple)
4
>>> # List -- mutable sequences
>>> list = [4, 3, 2, 1]
>>> print list
[4, 3, 2, 1]
>>> print list[0]
4
>>> list[0] = 66
>>> print list
```

```
[66, 3, 2, 1]
```

Dictionary

A dictionary represents a set of key / value pairs -- much like a sequence object, except with named keys instead of simple integer positions. In a dictionary, the keys are seprated from their values by a colon :, each key/value pair is delimited by a comma, and the whole set of key / value pairs is enclosed in curly braces {}.

```
>>> cat = {'hair' : 'black', 'name' : 'Spook', 'weight' : 3.2}
>>> print cat['hair']
black
```

Defining functions and classes

Functions

You can define your own function by using the def keyword to identify the name of the function, a function signature in parentheses, a colon, and finally your indented function body.

```
>>> def tagit (tagname, text):
      html = "<%s>%s</%s>" % (tagname, text, tagname)
      return html

>>> print tagit("strong", "My name is Susan.")
<strong>My name is Susan.</strong>
>>>
```

Classes

You can define your own class by declaring the class name using the class keyword, followed by an optional list of classes from which your new class will inherit, a colon, and your indented class definition.

Define class methods in the same way that you define a function, except pass the **`self`** keyword as the first parameter to each method to represent the instance of the object. To define a constructor for the class, define a class method named **`__init__`** that sets the initial values for the instance variables.

Within the class definition, refer to instance variables using the **`self.`** prefix.

```
>>> class cat:
      def __init__(self, hair, name, weight):
            self.hair = hair
            self.name = name
            self.weight = weight
      def purr(self, volume):
            return volume * self.weight


>>> spook = cat('black', 'Spook', 3.2)
>>> print spook.purr(3)
9.6
```

Operators

while loops

A while loop is a control structure that loops while the specified condition is true:

```
>>> x = 0
>>> while x < 5:
      x = x + 1
      print x,


1 2 3 4 5
```

Notice that the numbers are all printed on the same line. Can you guess why?

if / elif / else

Standard condition tests, with the `else` keyword providing a default action if no other test succeeds. Only the `if` test is required.

```
>>> if spook.hair == 'white':
      print "Winter cat"
elif spook.hair == 'black':
      print "Night cat"
else:
      print "Calico cat"


Night cat
```

for iterators

The `for` operator iterates over each element of a sequence or other object that implements an iterator interface.

```
>>> cats = 'Spook', 'Mitten', 'Snowball'
>>> for name in cats:
      print name,


Spook Mitten Snowball
```

Resources

- Python DB-API 2.0 specification: http://www.python.org/peps/pep-0249.html
- Python Web site: http://python.org