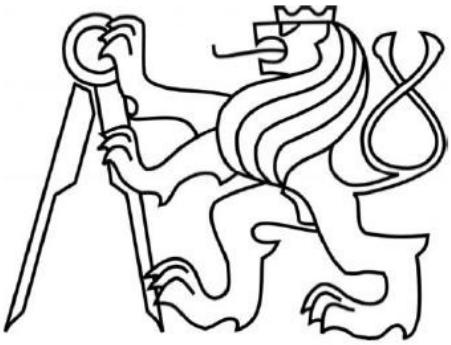
Czech Technical University in Prague

Faculty of Mechanical Engineering



Study of Python programming language for database systems

Ainur Duisenbayeva

Zhuldyz Assylova

May, 2012

Project Report

supervised by Ivo Bukovsky

Project Objectives

1. Analyze all the database servers that are supported by the Python Database API
2. Choose a database to work with in Python
3. Create a program that would operate with data from the database
4. Analyze the obtained results
5. Make the conclusion

Contents

1 Python and MySQL

This section reviews available resources and basic information about database modules that are known to be used with Python. Each review is accompanied by a demonstrative Python code that can help users to start using the modules in Python.

Basic introduction on Python and SQL modules can be retrieved from resources as [9] [10] and most relevant information can be retrieved as follows.

> *1. Python is a general-purpose, high-level programming language whose design philosophy emphasizes code readability. Python claims to combine "remarkable power with very clear syntax",and its standard library is large and comprehensive.Python is a programming language that lets you work more quickly and integrate your systems more effectively.Python runs on Windows, Linux/Unix, Mac OS X, and has been ported to the Java and .NET virtual machines. Python is free to use, even for commercial products, because of its OSI-approved open source license.Like other dynamic languages, Python is often used as a scripting language, but is also used in a wide range of non-scripting contexts. Using third-party tools, Python code can be packaged into standalone executable programs. Python interpreters are available for many operating systems.Python has a large standard library, commonly cited as one of Python's greatest strengths, providing pre-written tools suited to many tasks. This is deliberate and has been described as a "batteries included"Python philosophy. The modules of the standard library can be augmented with custom modules written in either C or Python. Boost C++ Libraries includes a library, Boost.Python, to enable interoperability between C++ and Python. Because of the wide variety of tools provided by the standard library, combined with the ability to use a lower-level language such as C and C++, which is already capable of interfacing between other libraries, Python can be a powerful glue language between languages and tools.The standard library is particularly well tailored to writing Internet-facing applications, with a large number of standard formats and protocols (such as MIME and HTTP) already supported. Modules for creating graphical user interfaces, connecting to relational databases, arithmetic with*

arbitrary precision decimals, manipulating regular expressions, and doing unit testing are also included.Some parts of the standard library are covered by specifications (for example, the WSGI implementation wsgiref follows PEP 333), but the majority of the modules are not. They are specified by their code, internal documentation, and test suite (if supplied). However, because most of the standard library is cross-platform Python code, there are only a few modules that must be altered or completely rewritten by alternative implementations.The standard library is not essential to run Python or embed Python within an application. Blender 2.49 for instance omits most of the standard library.For software testing, the standard library provides the unittest and doctest modules.Python has support for working with databases via a simple API. Modules included with Python include modules for SQLite and Berkeley DB. Modules for MySQL , PostgreSQL , FirebirdSQL and others are available as third-party modules. The latter have to be downloaded and installed before use. The package MySQLdb can be installed, for example, using the debian package "python-mysqldb".Some supported databases:

- *GadFly*
- *MySQL*
- *PostgreSQL*
- *Informix*
- *Interbase*
- *Oracle*

Five steps must be taken to make Python work in a database system:

1. Import the database module (**MySQLdb**, **phpmyAdmin**, **sqlite**, etc)
2. Use **module.connect(...)** to create a connection.
3. Use **connection.cursor()** to get a cursor. Cursors do all the work.
4. Use **cursor.execute(sql_query)** to run something.
5. Use **cursor.fetchall()** to get results.

1.1 Gadfly

First database module we review is Gadfly. We adopted most relevant information about Gadfly from resource [1] as follows.

Gadfly is a collection of python modules that provides relational database functionality entirely implemented in Python. It supports a subset of the intergalactic standard RDBMS Structured Query Language SQL.

One of the most compelling aspects of Gadfly is that it runs where ever Python runs and supports client/server on any platform that supports the standard Python socket interface. Even the file formats used by Gadfly for storage are cross-platform -- a gadfly database directory can be moved from Win95 to Linux using a binary copying mechanism and gadfly will read and run the database.

It supports persistent databases consisting of a collection of structured tables with indices, and a large subset of SQL for accessing and modifying those tables. It supports a log based recovery protocol which allows committed operations of a database to be recovered even if the database was not shut down in a proper manner (ie, in the event of a CPU or software crash, [but not in the event of a disk crash]). It also supports a TCP/IP Client/Server mode where remote clients can access a Gadfly database over a TCP/IP network (such as the Internet) subject to configurable security mechanisms.

Because it lacks (at this time) true concurrency control, and file-system based indexing it is not appropriate for very large multiprocess transaction based systems.

Since Gadfly depends intimately on the kwParsing package it is distributed as part of the kwParsing package, under the same generous copyright.

Creating a new database

```
import gadfly
connection = gadfly.gadfly()
cursor = connection.cursor()
cursor.execute("create table ph (nm varchar, ph varchar)")
cursor.execute("insert into ph(nm, ph) values ('arw', '3367')")
cursor.execute("select * from ph")
for x in cursor.fetchall():
  print x
# prints ('arw', '3367')
connection.commit()
```

1.2 MySQL

We retrieved relevant information about MySQL module from resource [2] as follows.

MySQL is a leading open source database management system. It is a multi user, multithreaded database management system. MySQL is especially popular on the web. It is one of the parts of the very popular LAMP platform. Linux, Apache, MySQL, PHP. Currently MySQL is owned by Oracle. MySQL database is available on most important OS platforms. It runs under BSD Unix, Linux, Windows or Mac. Wikipedia and YouTube use MySQL. These sites manage millions of queries each day. MySQL comes in two versions. MySQL server system and MySQL embedded system.

The codes on this module are demonstrated in section **Error! Reference source not found.**.

1.3 PostgreSQL

Another reviewed module is PostgreSQL and the most relevant information is reviewed from [5] as follows.

PostgreSQL is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness. It runs on all major operating systems, including Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), and Windows. It is fully ACID compliant, has full support for foreign keys, joins, views, triggers, and stored procedures (in multiple languages). It includes most SQL:2008 data types, including INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL, and TIMESTAMP. It also supports storage of binary large objects, including pictures, sounds, or video.

PostgreSQL runs stored procedures in more than a dozen programming languages, including Java, Perl, Python, Ruby, Tcl, C/C++, and its own PL/pgSQL, which is similar to Oracle's PL/SQL. Included with its standard function library are hundreds of built-in functions that range from basic math and string operations to cryptography and Oracle compatibility. Triggers and stored procedures can be written in C and loaded into the database as a library, allowing great flexibility in extending its capabilities. Similarly, PostgreSQL includes a framework that allows developers to define and create their own custom data types along with supporting functions and operators that define their behavior. As a result, a host of advanced data types have been created that range from geometric and spatial primitives to network addresses to even

ISBN/ISSN (International Standard Book Number/International Standard Serial Number) data types, all of which can be optionally added to the system.

There are several Python drivers for PostgreSQL:

Software	License	Platforms	Python versions	DB API 2.0	Native (uses libpq)	Notes
Psycopg	LGPL	Unix, Win32	2.4-3.2	yes	yes	Most popular libpq-based driver
PyGreSQL	BSD	Unix, Win32	2.3-2.6	yes	yes	
ocpgdb	BSD	Unix	2.3-2.6	yes	yes	PG8.1+
py-postgresql	BSD	any (pure Python)	3.0+	yes	no	pure Python with optional C accelerator modules, extensive custom API
bpgsql	LGPL	any (pure Python)	2.3-2.6	yes	no	labeled alpha
pg8000	BSD	any (pure Python)	2.5+ / 3.0+	yes	no	new maintenance fork (2012)

1.4 Oracle

Oracle modules for Python are reviewed mainly from resources [3][4].

*The **Oracle Database** (commonly referred to as Oracle RDBMS or simply as Oracle) is an object-relational database management system(ORDBMS) produced and marketed by Oracle Corporation.*

***The Oracle RDBMS** stores data logically in the form of table spaces and physically in the form of data files ("datafiles"). Tablespaces can contain various types of memory segments, such as Data Segments, Index Segments, etc. Segments in turn comprise one or more extents. Extents comprise groups of contiguous data blocks. Data blocks form the basic units of data storage.*

Before you can access a database, you need to install one of the many available database modules. One such module is cx_Oracle.

***cx_Oracle** is a Python extension module that allows access to Oracle databases and conforms to the Python database API specification. The cx_Oracle module must be imported as it's not part of the core Python language.*

Connection of Oracle:

```
connection = cx_Oracle.connect("uid/pwd@database")
cursor = connection.cursor()
cursor.execute("SELECT COUNT(*) FROM User_Tables")
count = cursor.fetchall()[0][0]
cursor.close()
connection.close()
```

1.5 Informix

We studied about Informix module and retrieved relevant facts mainly from resource [6].

Informix database is known for its outstanding capabilities:

1. *Impressive Transaction Performance*
2. *High Reliability and First Class Support*
3. *Rich, sophisticated Extensibility Features*
4. *Backward Compatibility and Painless Version Upgrades*
5. *Powerful Replication Solutions inside the DBMS Kernel*
6. *Hardware gentle, scalable Multithreading Architecture*
7. *Extremely low Administration Overhead*

Example of using Informix by means of Python:

```
#!/usr/bin/python

import sys
import informixdb  # import the InformixDB module

# -------------------------------------
# open connection to database 'stores'
# -------------------------------------
conn = informixdb.connect("stores")

# ---------------------------------
# allocate cursor and execute select
# ---------------------------------
cursor1 = conn.cursor(rowformat = informixdb.ROW_AS_DICT)
cursor1.execute('select code, sname from state')

# --------------------------
# prepare 'delete' statement
# --------------------------
cursor2 = conn.cursor()
```

```
# ---------------------
# fetch thru result set
# ---------------------
for row in cursor1:

    # ------------------------------------------
    # delete row if column 'code' begins with 'C'
    # ------------------------------------------
    if row['code'][0] == 'C':
        cursor2.execute('delete from state where code = ?', (row['code'],))
        print "DELETED: %-2s %-15s" % (row['code'], row['sname'])
        continue

   # --------
   # show row
   # --------
   print "%-2s %-15s" % (row['code'], row['sname'])

# ---------------------------------------
# commit transaction and close connection
# ---------------------------------------
conn.commit()
conn.close()

sys.exit(0);
```

Demonstr

2 Demonstration Example

2.1 Connecting Python with the database

A table resides within a database. This is particularly true for MySQL. To create a table, a database must be created first, or at least a database must be present. So to retrieve data from a

table, a connection to the database must be established. This is done by using the connect() method. In other words, connect is the constructor of the phpMyAdmin. The parameters are as follows:

- host is the name of the system where the MySQL server is running. It can be a name or an IP address. If no value is passed, then the default value used is localhost.
- user is the user id, which must be authenticated. In other words, this is the authentic id for using the services of the Server. The default value is the current effective user. Most of the time it is either 'nobody' or 'root'.
- passwd -- It is by using a combination of the user id and a password that MySQL server (or for that matter any server) authenticates a user. The default value is no passwords. That means a null string for this parameter.
- db is the database that must be used once the connection has been established with the server. However, if the database to be used is not selected, the connection established is of no use. There is no default value for this parameter.

2.2 Creation of the cursor

In the terminology of databases, cursor is that area in the memory where the data fetched from the data tables are kept once the query is executed. In essence it is the scratch area for the database.

MySQL does not support cursors. But it is easy to emulate the functionality of cursors. That's what the phpMyAdmin does. To get a cursor, the cursor() method of connection object has to be used. There is only one parameter to this method -- the class that implements cursor behavior. This parameter is optional. If no value is given, then it defaults to the standard Cursor class. If more control is required, then custom Cursor class can be provided. To obtain a cursor object the statement would be:

cursor= db.cursor()

Once the above statement is executed, the cursor variable would have a cursor object.

2.3 Execution of the SQL statement

The steps enumerated until now have done the job of connecting the application with the database and providing an object that simulates the functionality of cursors. The stage has been set for execution of SQL statements. Any SQL statement supported by MySQL can be executed using the execute() method of the Cursor class. The SQL statement is passed as a string to it. Once the statement is executed successfully, the Cursor object will contain the result set of the retrieved values. For example, to retrieve all the rows of a table named USER_MASTER the statement would be:

cursor.execute("select * from USER_MASTER")

Once the above statement is executed, the cursor object would contain all the retrieved. This brings us to the fourth step, fetching of the resultset. Before moving on to the next step, there is one point you must understand. The execute() function accepts and executes any valid SQL statement, including DDL statements such as delete table, alter table, and so on. In the case of DDL statements, there is no fifth step (i.e. iteration over the results fetched).

2.4 Fetching the result set

The flexibility of Python comes to the fore in this step also. In the real world, fetching all the rows at once may not be feasible. MySQLdb answers this situation by providing different versions of the fetch() function of Cursor class. The two most commonly used versions are:

- fetchone(): This fetches one row in the form of a Python tuple. All the data types are mapped to the Python data types except one -- unsigned integer. To avoid any overflow problem, it is mapped to the long.
- fetchall(): This fetches all the rows as tuple of tuples. While fetchone() increments the cursor position by one, fetchall() does nothing of that kind. Everything else is similar.

The subtleties will become clear from the following example. To fetch one row at a time and display the result, the block would be:

numrows = int(cursor.rowcount) #get the count of total rows in the #resultset

```
# get and display one row at a time
for x in range(0,numrows):
    row = cursor.fetchone()
    print row[0], "-->", row[1]
```

The above result can be achieved by using fetchall() as shown below:

```
result = cursor.fetchall()

# iterate through resultset
for record in result:
    print record[0] , "-->", record[1]
```

The iteration is through the core Python APIs only. As the returned data structure is tuple, no extra API is required.

Resulting program:

```
#!/usr/local/python26
# -*- coding: utf-8
#!/usr/local/mysql5

import MySQLdb
import string

# connecting with a database
db = MySQLdb.connect(host="localhost", user="ab", passwd="root", db="ab", charset='utf8')
# forming a cursor
cursor = db.cursor()

# request to a db
sql = """SELECT * FROM a """
# performing the request
cursor.execute(sql)

# getting a result of the request
data =  cursor.fetchall()
# looking through notes
for reca in data:
    # extracting the data - in the same order as in SQL-request
    id, per = reca
    # printing the information
    # print id, per
    print reca[1]

# conneting with a database
db = MySQLdb.connect(host="localhost", user="ab", passwd="root", db="ab", charset='utf8')
# forming a cursor
cursor = db.cursor()

# request
sql = """SELECT * FROM b """
# performing the request
cursor.execute(sql)

# getting a result
data =  cursor.fetchall()
# looking through notes
for recb in data:
    # extracting the data - in the same order as in SQL-request
    id, per = recb
    # printing
    # print id, per
    print recb[1]


print reca[1]+recb[1]

# closing the connection
db.close()
```

That covers all the steps required to access MySQL. What we have discussed up to now is "the approach of tackling a problem of the type database connectivity." However, in real life, reusability plays a more important role than it has in what you have seen so far. Hence in the next section we will be using the steps discussed until now to create a generic class for accessing MySQL.

3 Conclusion

During the work on our project we tried to analyze all the database servers in order to find the most suitable one. After a careful consideration we chose MySQL Server since it has many

appropriate characteristics to be implemented in Python. Python is one of the most known advanced programming languages, which owns mainly to its own natural expressiveness as well as to the bunch of support modules that helps extend its advantages, that's why Python fits perfectly well when it comes to developing a stable connection between the program and the database.

References

[1] **http://gadfly.sourceforge.net/gadfly.html**

[2] **http://zetcode.com/databases/mysqlpythontutorial/**

[3] **http://www.itsabacus.com/oracle.html**

[4] **http://www.orafaq.com/wiki/Python**

[5] **http://www.postgresql.org/**

[6] **http://informix-zone.com/informix-script-dbapi**

[7] **http://clientes.netvisao.pt/luiforra/ib/**

[8] **http://docs.python.org/tutorial/modules.html**

[9] **http://en.wikipedia.org/wiki/Python_(programming_language)**

[10] **http://www.manuinfo.com/standard-library-of-python**