MySQL Users Conference 2005, Santa Clara, CA

Python and MySQL

Andy Dustman
Office of Information Technology

Terry College of Business

University of Georgia

Andy Dustman
    <adustman@terry.uga.edu>
Terry College of Business
    http://www.terry.uga.edu/
University of Georgia
    http://www.uga.edu/

Python for the PyCurious

- interpreted (byte-code compiler)
- interactive (easy to test ideas)
- object-oriented (everything's an object)
- rapid development (5-10x C++, Java)
- fits your brain *[Bruce Eckel]*
- fits your wallet: free (OSI and GPL)
- fun!

Introductory Material on Python:
    http://www.python.org/doc/Intros.html

Types

The basic Python types and their mutability

	Mutable	**Immutable**
Sequence	list	tuple str, unicode
Number		int, long, float
Mapping	dict	
Other	object	

Basic type examples

```
>>> i=1 # an int
>>> j=2**64-1 # a long integer
>>> print j
18446744073709551615
>>> f=3.14 # float (C double)
>>> c=1-1j # complex (1j is imaginary)
>>> print c
(1-1j)
>>> s="welcome to python!"
>>> s.capitalize().split() # returns a list
['Welcome', 'to', 'python!']
>>> [ word.capitalize() for word in s.split() ]
['Welcome', 'To', 'Python!']
>>> a, b = 1, 2
>>> print (a,b) # a tuple
(1, 2)
>>> a, b = b, a
>>> print (a,b)
(2, 1)
```

Strings

```
>>> "Normal string literal isn't very interesting."
"Normal string literal isn't very interesting."
>>> 'Single quotes work "same as double".'
'Single quotes work "same as double".'
>>> """Triple-quoted strings are good for long strings
... which span multiple lines."""
'Triple-quoted strings are good for long strings\nwhich span multiple li
>>> r"Raw strings are useful for regexs, i.e. \w+ or \1"
'Raw strings are useful for regexs, i.e. \\w+ or \\1'
>>> u"Unicode strings work just like regular strings."
u'Unicode strings work just like regular strings.'
>>> u"\u72c2\n\u7009".encode('utf-8')
'\xe7\x8b\x82\n\xe7\x80\x89'
>>> print u"\u72c2\n\u7009".encode('utf-8')
狂
瀉
```

Strings

Lots of string methods and operators:

```
>>> "Split words into a list.".split()
['Split', 'words', 'into', 'a', 'list.']
>>> ' '.join(['Join', 'a', 'list', 'of', 'strings'])
'Join a list of strings'
>>> "Concatenate" + " " + "strings"
'Concatenate strings'
>>> "Multiplicity! " * 3
'Multiplicity! Multiplicity! Multiplicity! '
>>> "Parameter %s" % "substitution"
'Parameter substitution'
>>> d = dict(first_name="Vee", last_name="McMillen",
... company="O'Reilly")
>>> "Hello, %(first_name)s. How are things at %(company)s?" % d
"Hello, Vee. How are things at O'Reilly?"
```

Dictionaries

Python dictionaries are like perl hashes:

```
>>> d1={}
>>> d1['a']=1
>>> d1['b']=2
>>> d1['c']=3
>>> d1
{'a': 1, 'c': 3, 'b': 2}
>>> d2={'a': 1, 'c': 3, 'b': 2}
>>> d3=dict([('a',1),('b',2),('c',3)])
>>> d4=dict(a=1, b=2, c=3)
>>> d1 == d2 == d3 == d4
True
>>> len(d1)
3
```

Values can be any type, but keys must be immutable.

Sequences

```
>>> l = ['a','b','c','d','e']
>>> print l[0]
a
>>> print l[-1]
e
>>> print l[2:4]
['c', 'd']
>>> s='abcde'
>>> print s[2:4]
cd
>>> print s[::2]
ace
>>> print s[::-1]
edcba
>>> l.append(s)
>>> print l
['a', 'b', 'c', 'd', 'e', 'abcde']
```

Iterators

- `iter(object)` returns an iterator object
- `iterobj.next()` returns the next object
- `StopIteration` is raised when there are no more objects

  ```
  >>> # no normal person would do this
  >>> l = [1, 2, 3]
  >>> i = iter(l)
  >>> i.next()
  1
  >>> i.next()
  2
  >>> i.next()
  ```

```
    3
    >>> i.next()
    Traceback (most recent call last):
      File "", line 1, in ?
    StopIteration
```

Common iterator usage

```
>>> l = [1, 2, 3]
>>> for item in l:
...     print item
...
1
2
3
>>> d = dict(a=1, b=2, c=3)
>>> for key in d:
...     print key, d[key]
...
a 1
c 3
b 2
```

Exceptions

```
f = open("myfile", 'r')
try:
    try:
        for line in f:
            try:
                process(line)
            except TypeError:
                line = mangle(line)
                try:
                    process(line)
                except TypeError:
                    raise FoobarError, line
    except IOError, message:
        print "Error reading:", message
    except FoobarError:
        print "This file is totally munged."
    except:
        print "Something inexplicable happened:"
        raise # re-raise original exception
finally:
    f.close()
```

Odds and ends

- Code blocks are delimited by indentation
  - You probably do this already
  - Space or tabs, your call; just be consistent
  - No need for curly braces

- Less cluttered, easier to read
- End-of-line is a statement separator (so is `;`)
- No type enforcement
  - Argument types are not checked
  - Function return types are not checked
  - Type checking makes your code less flexible
  - If you still want it, you can add it cleanly with decorators
- Operator overloading for user-defined classes
- *Everything* is a reference (pass by reference)
- `None` object for null/missing values (equivalent to `NULL`)

Odds and ends

- Member access with `.` operator
  - `instance.method()`
  - `instance.attribute`
  - `instance.attribute.another`
- Functions/methods are not the only things that are callable
- Decorators apply a callable to a function at creation time:

  ```
  @g
  def f(x):
      ...
  ```

  is equivalent to:

  ```
  def f(x):
      ...
  f = g(f)
  ```

The Python DB-API

- Standard API for database access
- PEP 249: http://www.python.org/peps/pep-0249.html
- By convention, module name ends with "db", i.e. MySQLdb
  - Module Interface
  - Connection Objects
  - Cursor Objects
  - DBI Helper Objects
  - Type Objects and Constructors
  - Implementation Hints
  - Major Changes from 1.0 to 2.0

Module Interface

connect(...)
: Constructor for creating a connection to the database. Returns a Connection Object.

apilevel
: String constant stating the supported DB API level.

threadsafety

Integer constant stating the level of thread safety the interface supports.

SQL parameter placeholders

paramstyle
: String constant stating the type of parameter marker formatting expected by the interface.

  'qmark'
  : Question mark style, e.g. '...WHERE name=?'

  'numeric'
  : Numeric, positional style, e.g. '...WHERE name=:1'

  'named'
  : Named style, e.g. '...WHERE name=:name'

  'format'
  : ANSI C printf format codes, e.g. '...WHERE name=%s'

  'pyformat'
  : Python extended format codes, e.g. '...WHERE name=%(name)s'

  MySQLdb 1.0 and 1.2 uses format and pyformat; 2.0 may also support qmark.

Exceptions

- StandardError
  - Warning
  - Error
    - InterfaceError
    - DatabaseError
    - DataError
    - OperationalError
    - IntegrityError
    - InternalError
    - ProgrammingError
    - NotSupportedError

Connection Object

.close()
: Close the connection now

.commit()
: Commit any pending transaction to the database. Auto-commit off by default.

.rollback()
: Rollback any pending transaction.

.cursor()
: Return a new Cursor Object using the connection.

exceptions
: The standard exception classes; simplfies error handling in some cases

.messages
: list of error/warning messages since last method call

Cursor Object

.description
    A sequence of sequences, each of which describe a column of the result.
.rowcount
    Number of rows affected by last query.
.callproc(procname[,parameters])
    Call a stored database procedure with the given name.
.close()
    Close the cursor now.
.execute(operation[,parameters])
    Prepare and execute a database operation (query or command). Parameters: sequence or mapping.
.executemany(operation,seq_of_parameters)
    Prepare a database operation (query or command) and then execute it against a sequence of parameters.

Cursor Object

.fetchone()
    Fetch the next row of the result set as a sequence, or *None* if there are no more rows.
.fetchmany(*[size=cursor.arraysize]*)
    Fetch a sequence of up to *size* rows; may be fewer. Zero length sequence indicates end of result set.
.fetchall()
    Fetch all remaining rows as a sequence of rows.
.nextset()
    Skip to the next result set. Returns a true value if there is another result set; *None* (false) if not.
.arraysize
    Default number of rows to return with cursor.fetchmany(). Default: 1.

Cursor Object

.rownumber
    Current index into result set
.connection
    The Connection object for this cursor
.scroll(value*[,mode='relative']*)
    Scroll to a new position in the result set (relative or absolute).
.messages
    List containing warning/error messages since last method call (except the .fetch*XXX*() methods).
.next()
    Fetches one row (like fetchone()) or raises `StopIteration` if no rows left. *Iterator protocol*
.lastrowid
    Row id of the last affected row (i.e. inserting `AUTO_INCREMENT` columns)

MySQL for Python

- MySQL-python project on SourceForge: http://sourceforge.net/projects/mysql-python
- Current best version: 1.2.0

  - Python-2.3 and newer (and maybe 2.2)
  - MySQL-3.23, 4.0, and 4.1 (and maybe 5.0)
  - Prepared statements not supported yet
- Older version: 1.0.1
  - Python-1.5.2 (very old) and newer
  - MySQL-3.22, 3.23, and 4.0 (not 4.1 or newer)
  - Don't use if you can use 1.2.0
- Vaporware version: 2.0
  - Python-2.4 and newer
  - MySQL-4.0, 4.1, and 5.0
  - Prepared statements will be supported
  - Return all text columns as `unicode` by default

Architecture

`_mysql`

- C extension module
- transliteration of MySQL C API into Python objects
- If you use the C API, this should be very familiar
- Deprecated API calls not implemented
- Not everything (particularly fields) is exposed
- SQL column type to Python type conversions handled by a dictionary

`MySQLdb`

- Adapts `_mysql` to DB-API
- Many non-standard C API calls are exposed
- Relatively light-weight wrapper
- Implements cursors
- Defines default type mappings; easily customizable

Opening a connection

`connect()` takes the same options as `mysql_real_connect()`, and then some.

```
import MySQLdb

# These are all equivalent, for the most part
db = MySQLdb.connect("myhost", "myuser", "mysecret", "mydb")
db = MySQLdb.connect(host="myhost", user="myuser",
                     passwd="mysecret", db="mydb")
auth = dict(user="myuser", passwd="mysecret")
db = MySQLdb.connect("myhost", db="mydb", **auth)
db = MySQLdb.connect(read_default_file="/etc/mysql/myapp.cnf")
```

- `compress=1` enables gzip compression
- `use_unicode=1` returns text-like columns as `unicode` objects
- `ssl=dict(...)` negotiates SSL/TLS

Simple query example

```
import MySQLdb

db = MySQLdb.connect(read_default_file="/etc/mysql/myapp.cnf")
c = db.cursor()
c.execute("""SELECT * FROM users WHERE userid=%s""", ('monty',))
print c.fetchone()
```

Notes

- `('monty',)` is a 1-tuple; comma required to distinquish from algebraic grouping
- Good reasons not to use `*`
    - How many columns are being returned?
    - What is the order of the columns?
- Good reasons to use `*`
    - Table/database browser
    - Lazy

Multi-row query example

```
c = db.cursor()
c.execute("""SELECT userid, first_name, last_name, company
FROM users WHERE status=%s and expire>%s""",
(status, today))
users = c.fetchall()
```

Notes

- We know what the columns are
- Could use some object abstraction

A simple User class

```
class User(object):

    """A simple User class"""

    def __init__(self, userid,
                 first_name=None, last_name=None,
                 company=None):
        self.userid = userid
        self.first_name = first_name
        self.last_name = last_name
        self.company = company

    def announce(self):
        """Announce User to the world."""
        name = "%s %s" % (self.first_name, self.last_name)
        if self.company:
            return "%s of %s" % (name, self.company)
        else:
            return name

    def __str__(self):
        return self.announce()
```

Multi-row query with User object

```
users = []
c = db.cursor()
c.execute("""SELECT userid, first_name, last_name, company
FROM users WHERE status=%s and expire>%s""",
(status, today))

for userid, first_name, last_name, company in c.fetchall():
    u = User(userid, first_name, last_name, company)
    print u
    users.append(u)
```

might produce output like:

```
Tim O'Reilly of O'Reilly Media, Inc.
Monty Widenius of MySQL AB
Carleton Fiorina
Guido van Rossum of Elemental Security
```

Cursors are iterators

Not necessary to use `c.fetchall()`

```
users = []
c = db.cursor()
c.execute("""SELECT userid, first_name, last_name, company
FROM users WHERE status=%s and expire>%s""",
(status, today))

for userid, first_name, last_name, company in c:
    u = User(userid, first_name, last_name, company)
    print u
    users.append(u)
```

Under certain conditions, this is more efficient than `fetchall()`, and no worse.

Dictionaries as parameters

Python classes typically store attributes in `__dict__`, so you can get away with this:

```
u = User(...)
c = db.cursor()
c.execute("""INSERT INTO users
(userid, first_name, last_name, company)
VALUES (%(userid)s, %(first_name)s,
%(last_name)s, %(company)s)""", u.__dict__)
db.commit()
```

Multi-row INSERT

```
# users is a list of (userid, first_name, last_name, company)
c = db.cursor()
c.executemany("""INSERT INTO users
(userid, first_name, last_name, company)
VALUES (%s, %s, %s, %s)""", users)
db.commit()
```

In MySQLdb, this is converted internally to a multi-row INSERT, which is reported to be 2-3 orders of magnitude faster. Also works for REPLACE.

Multi-row INSERT with dictionaries

```
# users is a list of Users
c = db.cursor()
c.executemany("""INSERT INTO users
(userid, first_name, last_name, company)
VALUES (%(userid)s, %(first_name)s,
%(last_names, %(company)s)""",
[ u.__dict__ for u in users ])
db.commit()
```

This builds the parameter list with a list comprehension.

Never do this

Biggest MySQLdb newbie mistake of all time: Seeing `%s` and thinking, "I should use the `%` operator here."

```
users = []
c = db.cursor()
c.execute("""SELECT userid, first_name, last_name, company
FROM users WHERE status='%s' and expire>'%s'""" %
(status, today))

for userid, first_name, last_name, company in c:
    u = User(userid, first_name, last_name, company)
    print u
    users.append(u)
```

Note use of `%` operator to insert parameter values. This does **not** provide proper quoting (escaping of `'`, `NULL`/`None`, or `\0`). Passing them separately (as the second parameter) ensures they are quoted correctly. However, `%` **is** necessary if you have to insert arbitrary SQL such as column or table names or `WHERE` clauses.

To buffer, or not to buffer...

`mysql_store_result()`

Stores all rows of result set in client
Large result sets can chew up a lot of memory
Size of result set known immediately
Result set is seekable
Can issue another query immediately
Used for standard MySQLdb cursor

`mysql_use_result()`

Sends result set row by row
Consumes resources on server
Must fetch all rows before issuing any other queries
Size of result set unknown until finished
Not seekable
Can be used with MySQLdb's `SSCursor`

Optional cursor classes

`DictCursor` causes `fetchXXX()` methods to return mappings instead of sequences, with column names for keys.

```
users = []
c = db.cursor(MySQLdb.cursors.DictCursor)
c.execute("""SELECT userid, first_name, last_name, company
FROM users WHERE status=%s and expire>%s""",
(status, today))

for row in c:
    u = User(**row)
    print u
    users.append(u)
```

Note that column names happen to match `User` member names in this case.

Type objects and constructors

- Constructors
  - Date(year,month,day)
  - Time(hour,minute,second)
  - DateFromTicks(ticks)
  - TimeFromTicks(ticks)
  - TimestampFromTicks(ticks)
  - Binary(string)
- Types
  - STRING
  - BINARY
  - NUMBER
  - DATETIME
  - ROWID

These are not often used with MySQLdb.

Embedded server

1. Build with embedded server support:

   ```
   $ export mysqlclient=mysqld
   $ python setup.py build
   # python setup.py install
   ```

2. `_mysql.server_start()`
3. Use normally
4. `_mysql.server_end()`

Questions?

- http://sourceforge.net/projects/mysql-python
- http://www.terry.uga.edu/
- http://www.uga.edu/