

Using Databases with Python

Steve Holden

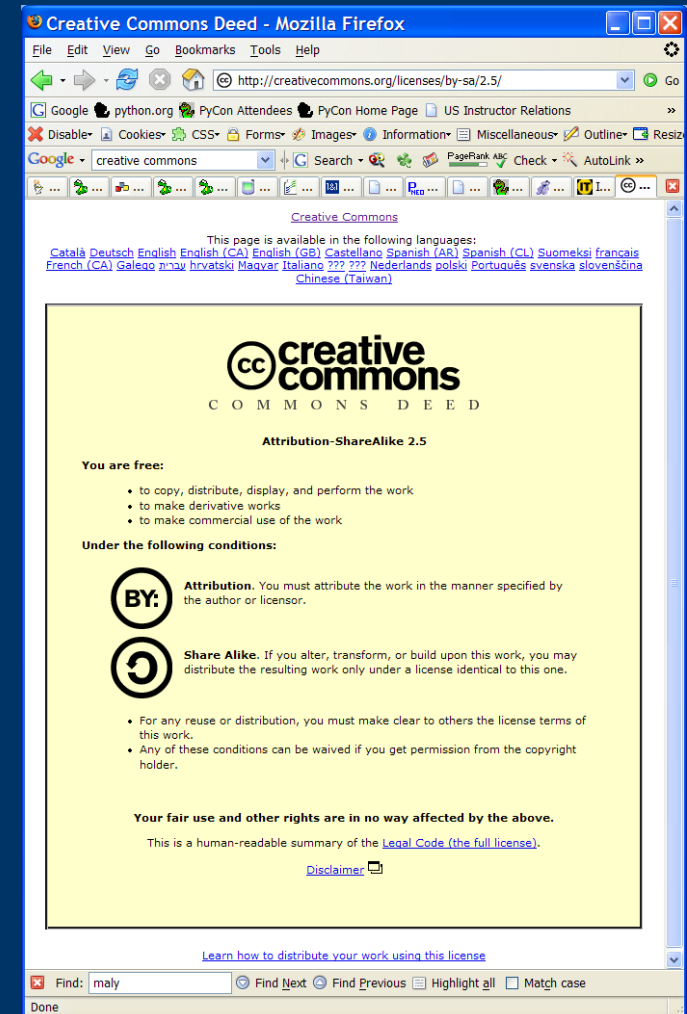
Holden Web LLC

PyCon TX 2006



Creative Commons License

- Attribution Share-Alike 2.5
SEE: <http://creativecommons.org/licenses/by-sa/2.5/>
- This course is copyright
©2006, Holden Web Ltd
SEE: <http://www.holdenweb.com/>
- You may use these materials
for courses and develop
them as long as everything
you develop remains
available in the same way



Your Instructor: Steve Holden

- Long-term interest in object-oriented programming
 - Since 1973 (SmallTalk)
- Python user since 1998
- Author of *Python Web Programming*
- *Lots* of computing experience
 - Five years as lecturer at Manchester University
 - 15 years as a professional instructor and consultant
 - Still learning ...



General Approach to Exercises

There are relatively few exercises
(we only have 3 hours!)

Load the **exNN.py** program into an editor.

Modify the program so it works.

Compare your program with sample
solution **solNN.py**

Exercises are intended to be completed quickly
All exercises are optional



The Relational Data Model

- Data are stored in *relations* (tables)
- Technically a relation is a *mapping*
 - From *primary key* values to *data tuples*
- Can *think of* relations as tables
 - The columns are *attributes*
 - The rows are *data tuples*
 - that include the key attributes
- Primary key values must be unique
 - Guarantees that each tuple is unique



The Relational Rules

- Relations may not contain duplicate tuples
- Attribute order in a relation is insignificant
 - They are identified by name
- Tuple order in a relation is insignificant
- Attribute values must be *atomic*
 - No arrays, lists or whatever*
- *Entity integrity*: no part of the primary key may be NULL

* Yes, I *do* know about PostgreSQL arrays – they are a pragmatic violation of this principle



Modeling the Real World

- Data models support applications
 - They (supposedly) describe the *real world* !
 - But only those aspects of interest to the application
- Applications query the model to discover information about the real world
 - Notionally easier and/or more efficient than examining the real world directly
- Must update the model as the real world changes
 - Otherwise the model yields inaccurate information




Modeling the World Relationally

- Each relation describes one type of thing
 - Often referred to as the *entity-type*
- Each occurrence of the entity-type is a row in the corresponding relation
 - We may *say*
“each row in the table describes a single student”
 - What we actually *mean* is
“each tuple in the relation contains the attribute values for a single occurrence of entity-type *student*”
 - We identify an occurrence by its *primary key value*



A Description of a Zoo



Animal		
<u>ANAME</u>	AFAMILY	WEIGHT
Jeremy	Jackal	135
Timmy	Tiger	400
Guido	Groundhog	20
Barry	Buffalo	1250
Fred	Fox	45

* Primary key attribute(s) are normally indicated by underlining

Handling Multi-Valued Attributes

~~ANIMAL-FOOD~~

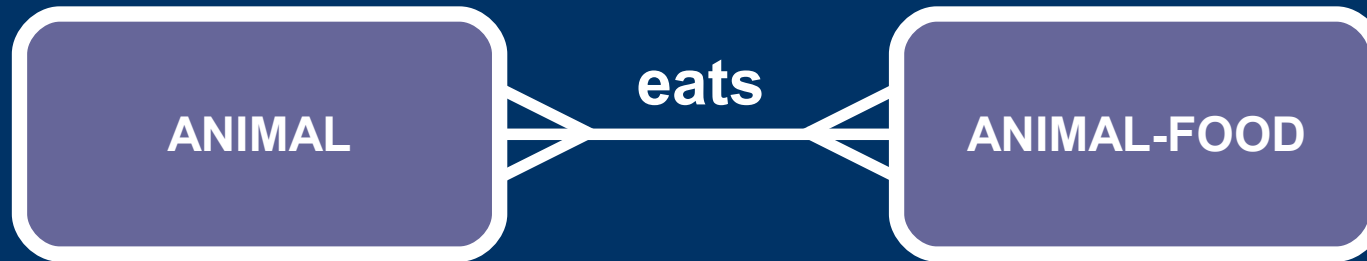
ANAME	FOOD
Barry	Grass, Leaves
Fred	Meat
Guido	Grass, Leaves, Flowers
Jeremy	Meat
Timmy	Meat

The representation above requires us to decompose the FOOD attribute values to answer the question “Who eats leaves?”

ANIMAL-FOOD

<u>ANAME</u>	<u>FOOD</u>
Barry	Grass
Barry	Leaves
Fred	Meat
Guido	Grass
Guido	Leaves
Guido	Flowers
Jeremy	Meat
Timmy	Meat

Tables Can Be Related



- Relationships appear as *foreign key* attributes
 - value *refers to* an occurrence of the related entity
- *Referential integrity requirement:*
the value of a foreign key attribute **MUST** occur as a primary key value in the related table
 - *e.g.* **ANIMAL-FOOD.ANAME** must specify some value of **ANIMAL.ANAME** (attribute *names* aren't significant here)

SQL In a Nutshell

- SQL is actually three separate sub-languages
- DDL: *Data Definition Language*
 - Used to create and maintain database structures
- DCL: *Data Control Language*
 - Used to specify who gets which access to what
- DML: *Data Manipulation Language*
 - Day-to-day query and update of the data model
 - Probably about 99.9% of what we do with databases!



Data Definition Language

```
CREATE TABLE spclass
(
    spcid serial NOT NULL,
    spcname varchar(10),
    spccost int4,
    CONSTRAINT "spc_PK" PRIMARY KEY (spcid)
);

CREATE TABLE message
(
    msgid serial NOT NULL,
    msgname varchar(10),
    msgsubject varchar(80),
    msgbody varchar,
    CONSTRAINT message_pkey PRIMARY KEY (msgid),
    CONSTRAINT message_msgname_key UNIQUE (msgname)
);

DROP TABLE sometable;
```



Data Control Language

```
GRANT SELECT, UPDATE (name, address)  
ON employee TO steve
```

- This is almost exclusively used by the DBA
- Not covered in the remainder of this tutorial



Data Manipulation Language (1)

- The everyday SQL we use to manipulate and retrieve our data
- Allows us to work with *sets of rows*
 - Different from file processing, where we work with one record at a time
- A declarative rather than a procedural language
 - We say what we want rather than how to get it
 - The RDBMS is expected to optimize performance using advanced techniques



Data Manipulation Language (2)

```
SELECT orgname, spcName, orgInvNo, spcCost, cntName, cntEmail  
FROM organization  
    JOIN contact ON orgCntID = cntID  
    RIGHT JOIN spClass ON orgspcid=spcID  
WHERE orgcntid IS NOT NULL  
ORDER BY spcid, orgSpriDate
```

- SELECT verb retrieves information from tables
 - Can join several tables in one statement
 - WHERE clause specifies conditions that retrieved data must meet
 - ORDER BY specifies the sequence of the results



Data Manipulation Language (3)

```
INSERT INTO CONTACT (cntEmail, cntName, cntOrgID)
VALUES ('steve@somewhere.com', 'Steve Holden', 19)
```

- INSERT verb creates new rows
 - Can only insert one row into one table per statement

```
DELETE FROM CONTACT WHERE cntName = 'Steve Holden'
```

- DELETE verb removes rows from tables
 - WHERE clause specifies rows to delete
 - *CAREFUL: no WHERE clause deletes all rows!*



Data Manipulation Language (4)

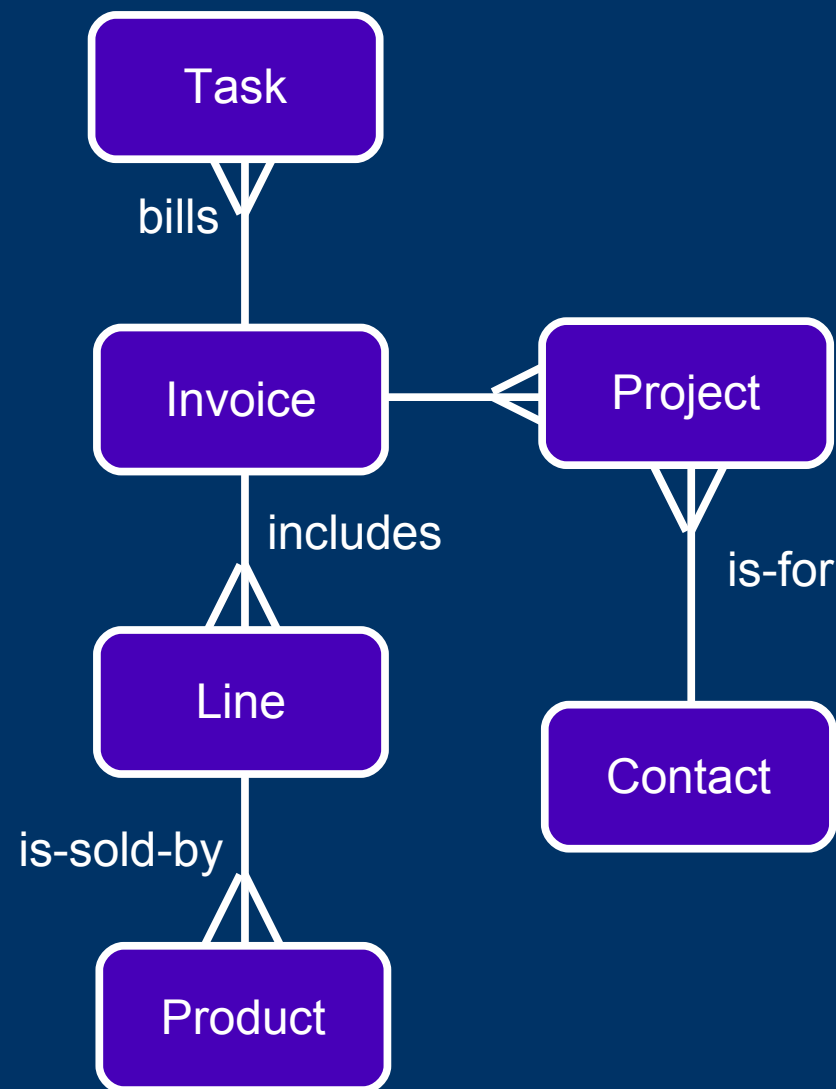
```
UPDATE employee  
    SET empSalary = empSalary * 1.1,  
        empLastRaise = '28-Feb-2005'  
WHERE empDeptNo = 20
```

- UPDATE verb can change several rows at a time
 - WHERE clause specifies the rows that will be updated
 - SET clause specifies the columns to be changed
 - Column names on RHS refer to values before update
 - Column names on LHS will be changed
 - $col = col * 1.1$ adds 10% to existing value
 - Similar to Python assignment



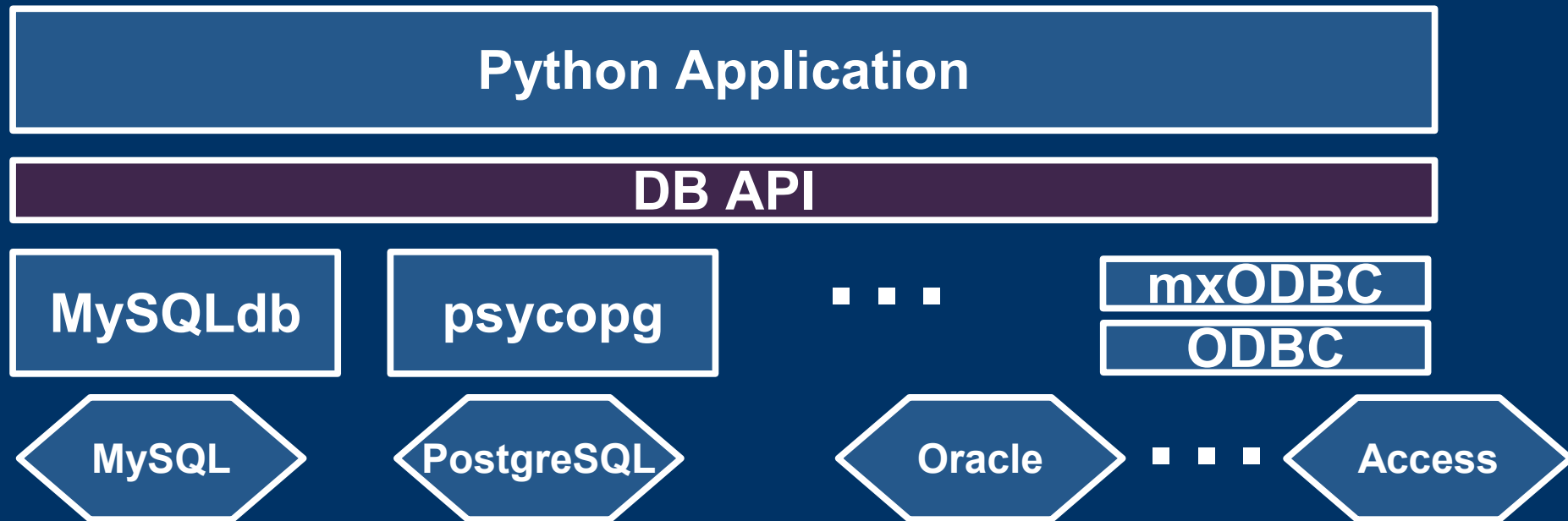
The Sample Database

- Simple billing database
- Company runs projects for contacts and invoices periodically/on demand
 - All completed tasks
 - Sales of products
- For simplicity, the relationships are not included in the definition



The Python DB API

- Designed to give Python programs access to RDBMS features using standard calls
- Each compliant module interfaces to a particular RDBMS engine



Some Well-Known Database Modules

<u>Module</u>	<u>Database</u>
MySQLdb	MySQL
cx_Oracle	Oracle
psycopg	PostgreSQL
psycopg2	PostgreSQL
mxODBC	Any RDBMS with ODBC drivers
kinterbasdb	Firebird/Interbase
adodbapi	Windows ADO data sources
pydb2	DB/2



DB API Module Attributes

`apilevel` Should always be '2.0' nowadays

`threadsafety` 0: module not sharable
1: module sharable between threads
2: connections are shareable
3: cursors are shareable

`paramstyle` How query parameters are specified

<code>'qmark'</code>	<code>... WHERE col=?</code>
<code>'numeric'</code>	<code>... WHERE col=:1</code>
<code>'named'</code>	<code>... WHERE col=:name</code>
<code>'format'</code>	<code>... WHERE col=%s</code>
<code>'pyformat'</code>	<code>... WHERE col=%(name)s</code>



Connecting to the Database

```
import dbmodule
```

```
conn = dbmodule.connect(...)
```

- Connection parameters can be module-specific
 - This makes database-independence problematic
 - This isn't all the API's fault, as vendor SQL variants have always made DB application portability difficult
- The connection object is the primary interface for accessing database functionality



Connect() Parameter Guidelines

- Should implement keyword parameters:
 - dsn Data source name as string
 - user User name *
 - password User's password *
 - host Hostname or IP address *
 - database Database name *
- You need the specifics for each module you use

* PEP suggests these should be optional



Connection Examples (1)

```
import psycopg2 as db

conn = db.connect(database="pycon",
                  user="steve", password="password",
                  host="localhost", port=5432)
```

```
import adodbapi as db

conn = db.connect(
    r"Provider=Microsoft.Jet.OLEDB.4.0;Data \
    Source=C:\Steve\northwind.mdb;")
```



Connection Examples (2)

```
import MySQLdb as db  
  
conn = db.connect(host="127.0.0.1",  
                  user="webuser", passwd="useweb", db='test')
```

```
import mx.ODBC.Windows as db  
  
conn = db.DriverConnect("Driver={SQL  
    Server};Server=dellboy;Database=GPWeb;Uid=sa;  
    Pwd=secret;")
```

The **DriverConnect()** function is specific to mx.ODBC, allowing connection to databases specified as ODBC data sources



Exercise 1: Installing a Sample DB

The samples directory contains the sample database in several formats (Access, MySQL and PostgreSQL 8)

Modify your environment and the db.py module so the module runs without error, opening the database

You need access to the database for the remaining examples to make sense (though you can just do them later if you prefer)



Connection Methods

- `.cursor()` Returns a new cursor object
- `.commit()` Commits any pending transaction
- `.rollback()` Restores database, undoing transaction changes
- `.close()` Closes connection: database becomes unusable
- The connection is the unit of transactions:
 - `commit()` commits changes made by *all* cursors created from the connection



Cursor Methods (1): Executing Queries

- This is where the real business happens!
- `.execute(stmt[, params])` executes a SQL statement with (optional) parameters
 - Queries should be parameterized
 - *Don't* try to build entire statements yourself
 - How `params` are provided depends on `.paramstyle`
- `.executemany(stmt, paramseq)` is like

```
for params in paramseq:
    .execute(stmt, params)
```



Cursor Methods (2): Retrieving Results

- `.fetchone()`
 - returns a single row from a query
 - Result is a tuple
 - Inefficient to retrieve large result sets one at a time
- `.fetchmany(size)`
 - returns a sequence of **size** row tuples
- `.fetchall()`
 - Returns all remaining row tuples
 - Can be heavy on memory



Cursor Methods (3): Calling Stored Procedures

- `.callproc(procname[,parameterseq])`
 - Calls procedure `procname`
 - Provides parameters as a sequence
 - Return value is a modified copy of the input sequence
 - Input-only arguments remain unchanged
 - Output and input/output arguments take new values
 - Procedure may also generate a result set
 - Handle as usual with `.fetch*()` methods
- Not all modules implement `.callproc()`



Query Parameterization

```
name = raw_input("Search for: ")
```

- NO:

```
stmt = "SELECT * FROM tbl WHERE name='%s'" % name  
curs.execute(stmt)
```

- YES:

```
stmt = "SELECT * FROM tbl WHERE name='%s'"  
curs.execute(stmt, (name, ))
```

- Parameterization avoids SQL injection exploits
- Also usually more efficient when repeated



SQL Injection: the Problem

- Pathological inputs can be extremely dangerous:

```
Search for: ' ; DROP TABLE tbl --
```

- Creates disastrous SQL statement sequence:

```
SELECT * FROM tbl WHERE name=' ' ; DROP TABLE tbl --'
```

- First statement becomes inconsequential
- Second statement removes table from database!
 - Clearly DB permissions offer *some* protection
 - But not a happy situation – why take the risk?



Parameterization Increases Efficiency

- Database goes through a complex process to prepare a SQL statement for execution
 - This process is independent of parameter values
- If SQL statement is retained (*e.g.* in a cache) the RDBMS can avoid repeating the preparation



You Can't Parameterize Everything

- Table and column names must be fixed!

```
>>> curs.execute("SELECT * FROM %s", ("contact", ))
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
psycopg.ProgrammingError: ERROR:  syntax error at or
near "'contact'" at character 15

SELECT * FROM 'contact'
>>>
```

- Parameter markers replaced with SQL literals
 - Table and column names *aren't* literals!



Exercise 2: Simple Data Access



Start with `ex02.py`

Write a program that accepts a contact ID (a number) and displays the IDs and dates of all invoices that customer has been sent



Portability Issues

- Need to accommodate `connect()` differences
- Need to accommodate `paramstyle` differences
 - This is only possible to a degree
 - 'qmark' and 'format' can be interchanged
 - As can 'named' and 'pyformat'
 - Both 'format' and 'pyformat' should use `%s` substitution for this to work
 - Most 'format' modules will accept `%s` for all types
- SQL differences are most troublesome



Dynamic SQL

- Useful techniques generate SQL statements on the fly
- Requires metadata which can be
 - Configured into the application code
 - Obtained by introspection on the database



Dynamic Queries (1)

- Suppose you want to access certain columns from a table

```
COLS = "cstID cstName cstAddr".split()
```

- A statement to retrieve those columns is easy:

```
sql = "SELECT %s FROM customer" % ', '.join(COLS)
```

- 'format' modules conflict with such substitutions
 - Suppose we'd wanted to add
WHERE cstRegion=' %s '
to select a parameterized region?



Dynamic Queries (2)

- One possibility is to *substitute the parameter mark* during statement construction

```
pmark = "%s" # or "?" - only works for positionals
sql = """SELECT %s FROM customer WHERE cstRegion=%s""" % (
    ', '.join(COLS), pmark)
curs.execute(sql, 'NORTH')
```

- This works quite well, though it can be tedious to set complex queries up
- Set *pmark* to '?' for paramstyle qmark (e.g. mx.ODBC)
 - This adds a further degree of portability



Exercise 3: Dynamic Queries

Start with `ex03.py`

Write a program that accepts a project name and then shows all the work that's been done on this project (grouping the data from each invoice together)



Query Results

- The result of `cursor.execute()` is undefined
 - Though *some* database modules give a row count
- The `.fetch*()` cursor methods retrieve the query result(s)
 - Each result row is returned as a tuple
 - Not convenient for symbolic access to columns
 - Fortunately we can solve this problem – this is Python :-)
- `.fetchone()` returns a single tuple
- `.fetchmany()`, `.fetchall()` return lists of tuples



Handing Results Naïvely

```
curs.execute("""\nSELECT cntName, cntAddr FROM customer WHERE\n      cstRegion='NORTH' """)
```

- Naïve methods are unsatisfactory in varying degrees:

```
for row in curs.fetchall():\n    print "name", row[0], "address", row[1]
```

```
for name, addr in curs.fetchall():\n    print "name", name, "address", addr
```

```
for row in curs.fetchall():\n    print "name %s address %s" % row
```



Creating More Amenable Results

```
curs.execute("""\nSELECT cntName, cntAddr FROM customer WHERE\n      cstRegion='NORTH' """)
```

- Dictionaries allow symbolic references:

```
for row in curs.fetchall():\n    row = dict(zip(COLS, row))
```

- Dictionary can be used to update an object:

```
class Row:\n    def __init__(self, COLS, row):\n        self.__dict__.update(**dict(zip(COLS, row)))\n\nfor row in curs.fetchall():\n    row = Row(COLS, row)\n    print "name", row.cntName,\n          "address", row.cntAddr
```

Exercise 4: Dynamic Queries

Start with ex04.py

Write a program like the one in Exercise 3
BUT ...

Use the predefined prTask() function to
print the details you retrieve



Convenience Techniques: `db_row`

```
from db_row import IMetaRow
Row = IMetaRow(COLS)
for row in curs.fetchall():
    r = Row(row)
    print r['cstID'], r[1], r.fields.cstAddr
```

- `IMetaRow()` returns a class
- Instances give dict, indexed and symbolic access to table columns
- See <http://opensource.theopalgroup.com/>
 - Similar to Greg Stein's `dtuple` module



Cursors Describe Query Results

- After a query the cursor's `.description` attribute describes the result
 - Sequence of 7-element tuples, one per column
 - Name, Type code are mandatory
 - Display size, Internal size, Precision, Scale, Null OK
 - may be provided or may contain **None**
 - You can use this to introspect column names:

```
curs.execute('SELECT * FROM tbl WHERE 1=0')  
COLS = [d[0] for d in curs.description]
```



Checking Uniqueness Constraints

- Relatively easy if uniqueness only required of individual columns
 - Trickier when required over sets of columns
 - But not impossible - use ANDed column groups
 - This example shows single columns in **UCOLS**

```
sql = ("SELECT COUNT(*) FROM %s WHERE (%s)" %
      (self.table,
       " OR ".join("%s=%s" % (fld, pmark) for fld in UCOLS)))
data = tuple(row[f] for f in UCOLS)
if id: # Match on an existing key doesn't count
    sql = "%s AND %s<>%s" % (sql, self.keyfield, pmark)
    data = data + (id, )
curs.execute(sql, data)
ok = curs.fetchone()[0] == 0
```



Handling Updates

```
sql = ("UPDATE %s SET %s WHERE %s=%s" %  
      (table, ", ".join("%s=%s" % (f, pmark) for f in COLS),  
      keyfield, pmark))  
data = tuple([row[f] for f in cols] + (ID, ))  
curs.execute(sql, data)  
conn.commit() # most important!
```

- Here **row** is assumed to be a dict
 - Otherwise use **data = tuple(row) + (ID,)**
 - But in that case column ordering *is* significant
- Note that primary key (ID) isn't being updated
 - Best practice – relationships make changes expensive



Handling Insertions

```
sql = ("INSERT INTO %s (%s) VALUES (%s)" %  
      (table, ", ".join(COLS+[keycol]),  
        ", ".join([pmark] * (len(COLS)+1))))  
data = tuple([row[f] for f in cols] + (ID, )  
curs.execute(sql, data)  
conn.commit() # most important!
```

- Generates a SQL statement like this:

```
INSERT INTO Customer  
      (cstID, cstName, cstAddr, ID)  
VALUES (%s, %s, %s, %s)
```

- If you have a sequence of data tuples:
 - Use `.executemany(sql, list_of_tuples)`



Object Relational Mappers

- Relational and object paradigms have similarities
 - Though you have to be careful not to push the comparison too far ...

Class
Instance
Attribute
Method

Relation
Occurrence
Attribute
Erm ... *

- We'll look at SQLObject as a recent project

* Yes, but stored procedures aren't attributes of relations



SQLObject Basics

- Module is designed to allow
`from SQLObject import *`
- Database connections include module selection *e.g.*:
`mysql://host/database?debug=1`
`postgres://user@host/database?debug=&cache=`
- Currently supported databases:
sqlite, mysql, postgres, firebird, interbase,
maxdb, sapdb, sybase, mssql
- Author (Ian Bicking) admits the API doesn't offer quite as much flexibility as SQL in joining tables



SQLObject Table Description

- Each relation is a subclass of SQLObject
 - Class variables define the columns
 - A range of convenience classes assist you in specifying the column types
 - Column names are usually camelCase
 - Normally converted to underscore_separated in database
 - This behavior *can* be overridden
- The primary key is assumed, and by default called ID



SQLObject Table Definitions

- This defines the Invoice table from the sample:

```
from SQLObject import *
conn = ... # Establish connection
sqlhub.processConnection = conn # Default it

class Invoice(SQLObject):
    invCntID = IntCol()
    invStartDate = DateCol()
    invEndDate = DateCol()
    invDate = DateCol()
    invOrdNo = StringCol(default=None)
    invCustRef = StringCol(default=None)
    invNotes = StringCol(default=None)
```



Simple Data Retrieval

- Use the class's `.get()` method
 - Argument should be row's primary key value

```
>>> inv = Invoice.get(122)
```

- Column names can be used like attributes

```
>>> inv.invOrdNo  
'QTG 40009'
```

- SQLAlchemy only keeps one copy of database rows
 - All retrievals give a reference to the existing object



More Complex Data Retrieval

- Use the class's `.select()` method

```
invoices = Invoice.select(  
    Invoice.q.invCntID = person.ID)
```

- Query columns are attributes of class's `q` attribute
- will also take a SQL query as a single argument
- Result is actually a generator
 - Can be sliced, counted and ordered as required
- Join helpers allow tables to be joined in a query



Insertion – Create a New Object!

```
class Product(SQLObject):  
    PrdDesc = StrCol()  
    PrdPrice = CurrencyCol
```

```
p = Product(  
    PrdDesc = 'PyCon Regular Registration',  
    PrdPrice = 185.00)
```

- Creating the new instance immediately inserts the new object into the underlying table



Update – Change Attribute Values!

```
>>> inv.invOrdNo = ''  
>>> inv.set(invCustref='',  
...      invNotes='Thanks for the order')
```

- The .set() method batches attribute updates
- Each attribute change causes a SQL UPDATE
 - To avoid this set class variable `_lazyupdate = True`
 - Force updates by calling `.syncUpdate()` or `.sync()` methods
 - `.sync()` method re-reads, `.syncUpdate()` does not



Other Features

- SQLAlchemy can pick up table definitions:

```
class Contact(SQLObject):  
    _fromdatabase = True
```

- Uses introspection on RDBMS metadata
- SQLBuilder helps build SQL queries
- *Can* use non-integer keys
 - but you must manage keys completely yourself
- **_style** class attribute allows naming choices
 - MixedCaseStyle would be needed for our sample DB



Other Object-Relational Mappers

- As always the Python world is well supplied
 - *DejaVu* – aims to be lightweight, with the ability to integrate multiple back-ends with various interfaces
 - *PyDO/PyDO2* – tables are dict subclasses
 - Allows you to work easily with subsets of table columns
 - *SQLalchemy* – decouples database and application
 - allows selection from joins, subqueries, unions &c
 - *MiddleKit* – early system from WebWare
 - somewhat cumbersome code generation process



The End ...

- ... or more likely just the beginning!
- If you have unanswered questions
 - now would be a *great* time to ask them
 - *much* better than saying “He didn't talk about *X*” later
- This was a *large* topic to cover in 3 hours
 - We have touched on the highlights
 - You should now be equipped for further research
- Feel free to keep in touch:

steve@holdenweb.com



Putting It All Together

This class deliberately has relatively few exercises based on sample code

While hands-on experience is important we have focused on advancing your understanding

Your instructor will now discuss either or both of

- a) some of the instructor's own code
- b) some further database samples

