Python programming — databasing

Finn Årup Nielsen

DTU Compute
Technical University of Denmark
September 22, 2014

Overview

Pickle — simple format for persistence

Key/value store — Persistent storage of dictionary-like objects

SQL — Traditional relational databases

NoSQL — JSON-like storage

Mongo — A NoSQL database

CouchDB — Another NoSQL database

Persistence via `pickle`

String representation of an object (Downey, 2008, section 14.7):

```
>>> alist = [1, 2, 4]
>>> import pickle
>>> pickle.dumps(alist)                                  # 'dump' with 's'
'(lp0\nI1\naI2\naI4\na.'
```

Save the object to a file called `save.pickle`:

```
pickle.dump(alist, open('save.pickle', 'w'))      # 'dump' without 's'
```

In another Python session you can load the file:

```
>>> import pickle
>>> thelist = pickle.load(open('save.pickle'))
>>> thelist
[1, 2, 4]
```

. . . Persistence via `pickle`

`cPickle` module in Python 2 is faster than `pickle` module:

```
>>> import pickle, cPickle
>>> from time import time as t   # or import clock as t for CPU time
>>> a = range(100000)
>>> t1 = t(); pickle.dump(a, open('save.pkl', 'w')); t() - t1
0.65437793731689453
>>> t1 = t(); cPickle.dump(a, open('save.pkl', 'w')); t() - t1
0.101058599609375
>>> t1 = t(); cPickle.dump(a, open('save.pkl', 'w'), 2); t() - t1
0.030821800231933594
```

The last `cPickle` example uses a faster protocol for encoding.

Note: Not all Python code can be pickled directly

Why not use JSON for persistence?

For simple Python objects you can also use JSON

```
>>> import cPickle
>>> import json
>>> from time import time as t    # or import clock as t for CPU time
>>> a = range(100000)
>>> t1 = t(); cPickle.dump(a, open('save.pkl', 'w')); t() - t1
0.05009317398071289
>>> t1 = t(); json.dump(a, open('save.json', 'w')); t() - t1
0.09607791900634766
```

though in this case JSON is a bit slower, but other programs than Python can read it, and reading a apparently faster (for default protocol)

```
>>> t1 = t(); b = cPickle.load(open('save.pkl')); t() - t1
0.06250786781311035
>>> t1 = t(); c = json.load(open('save.json')); t() - t1
0.013430118560791016
```

Why not use JSON for persistence?

Insecure against erroneous or maliciously constructed data, e.g., Nadia Alramli's example on why pickle is dangerous:

```
import pickle
pickle.loads("cos\nsystem\n(S'ls ~'\ntR.") # This will run: ls ~
```

JSON is more secure, though cannot do (Martelli et al., 2005, sect. 7.4):

```
>>> class MyClass():
...     def __init__(self, v): self.value = v

>>> mc = MyClass(4)
>>> pickle.dump(mc, open('save.pickle', 'w'))
>>> reloaded = pickle.load(open('save.pickle'))
>>> reloaded.value
4
```

Serialized

JSON and Pickle can also be serialized and transmitted.

. . . Persistence via `pickle`

What is the result of this pickle: example.pickle? (Do not unpickle files, i.e., load pickle files, from the net unless you trust them!)

Persistent dictionary

Simple Python-based persistent dictionary with `shelve`:

```
>>> import shelve
>>> she = shelve.open('test.she', 'c')
>>> she['stupid'] = -3
>>> she['funny'] = 3
>>> she.close()
```

In another later Python session the stored items can be accessed:

```
>>> import shelve
>>> she = shelve.open('test.she', 'c')
>>> for k, v in she.items():
...     print k, v
funny 3
stupid -3
```

Other key/value store

Berkeley DB (here with *Evolution* addressbook):

```
>>> import bsddb                                    # Release 4
>>> filename = '.evolution/addressbook/local/system/addressbook.db'
>>> db = bsddb.hashopen(filename, 'r')              # Open for reading
>>> for k, v in db.iteritems(): print("%s --> %s" % (k, v))
pas-id-469F320F00000000 --> BEGIN:VCARD
VERSION:3.0
EMAIL;TYPE=WORK:fn@imm.dtu.dk
X-EVOLUTION-BLOG-URL:http://fnielsen.posterous.com
...
>>> db.close()                                      # Close the database
```

`shelve` manipulation

What is the sum of the values where the initial letter in the keys are ‘a’ in this shelve: example.shelve?

SQL Python overview

Python can work with relational database management systems, such as MySQL, PostgreSQL (both client-server-based) and SQLite (lightweight)

The databases can be accessed by:

1. Specialized modules: `MySQLdb`, `psycocg` and `sqlite`. Because these modules implement the "DB API 2.0" standard the interfaces are similar.

2. Generic connectivity with ODBC: `pyodbc`

3. With an object-relational mapper (mediator between Python classes and SQL tables): `sqlobject` (simple) `sqlalchemy` (more complex) and "models" in the *Django* web framework (`from django.db import models`), . . . On top of SQLAlchemy: `elixir`.

DB API 2.0-like SQL connectivity steps

`import` database module

`connect` to the database

Acquire `cursor`. With `connection.cursor()`

`execute` SQL statement. With `execute()`, `executemany()` or `executescript()`

`fetch` the results. With `fetchone()`, `fetchmany()` or `fetchall()`

`commit` changes. `connection.commit()`

`close` connection. Use `connection.close()` or the `with` keyword.

DB API 2.0-like SQL connectivity

Import module and get a connection. For SQLite no host, user and password is necessary to get a connection, — the database is just a single file.

```
>>> from pysqlite2 import dbapi2 as db
>>> connection = db.connect("courses.sqlite")
```

Get `cursor` and create a table called "courses" in the SQL database with an SQL statement issued with the `execute` method:

```
>>> cursor = connection.cursor()
>>> cursor.execute("""CREATE TABLE courses (
        number CHAR(5) PRIMARY KEY,
        name CHAR(100),
        ects FLOAT);""")
```

Python SQL Insertion

Insert some explicit data into the "courses" table:

```
>>> cursor.execute("""INSERT INTO courses VALUES
        ("02820", "Python programming", 5);""")
>>> connection.commit()
```

Parameterized input with a Python variable (`courses`):

```
>>> courses = ("02457", "Nonlinear Signal Processing", 10)
>>> cursor.execute("INSERT INTO courses VALUES (?, ?, ?);", courses)
>>> connection.commit()
```

Parameterized insertion of multiple rows with `executemany`:

```
>>> courses = [("02454", "Introduction to Cognitive Science", 5),
               ("02451", "Digital Signal Processing", 10)]
>>> cursor.executemany("INSERT INTO courses VALUES (?, ?, ?);", courses)
>>> connection.commit()
```

Python SQL fetching data

Return one row at a time with `fetchone`:

```
>>> cursor.execute("SELECT * FROM courses;")
>>> cursor.fetchone()
(u'02820', u'Python programming', 5.0)
```

Or use the cursor as iterator:

```
>>> cursor.execute("SELECT * FROM courses;")
>>> for row in cursor: print(row)
# The 4 rows are printed
```

Get all returned data into a Python variable

```
>>> cursor.execute("SELECT * FROM courses ORDER BY number LIMIT 2;")
>>> c = cursor.fetchall()
[(u'02451', u'Digital Signal Processing', 10.0),
 (u'02454', u'Introduction to Cognitive Science', 5.0)]
```

Python SQL paramstyle differences

With `pysqlite`, tuple and question mark:

```
>>> param = {'ects': 10.0}
>>> cursor.execute("SELECT name FROM courses WHERE ects = ?",
                   (param['ects'],))
>>> cursor.fetchall()
[(u'Nonlinear Signal Processing',), (u'Digital Signal Processing',)]
```

Equivalent query with `pysqlite`, dictionary and named parameter:

```
>>> cursor.execute("SELECT name FROM courses WHERE ects = :ects", param)
```

With `MySQLdb`, tuple and "`%s`"

```
>>> cursor.execute("SELECT name FROM courses WHERE ects = %s",
                   (param['ects'],))
```

Bad and dangerous use of SQL!!!

```
>>> from pysqlite2 import dbapi2 as db
>>> connection = db.connect("test.sqlite")
>>> cursor = connection.cursor()
>>> cursor.execute("CREATE TABLE students ( password, name );")
>>> cursor.execute("""INSERT INTO students VALUES ("1234", "Finn");""")
>>> cursor.execute("""SELECT * FROM students;""")
>>> cursor.fetchall()
[(u'1234', u'Finn')]
>>> pw = 'dummy", (SELECT password FROM students WHERE name = "Finn")); --'
>>> name = "dummy"
>>> sql = 'INSERT INTO students VALUES ("%s", "%s");' % (pw, name)
>>> cursor.execute(sql)
>>> cursor.execute('SELECT * FROM students;')
>>> cursor.fetchall()
[(u'1234', u'Finn'), (u'dummy', u'1234')]  # Password revealed in name!
```

(And don't store literal passwords in databases — hash it)

sqlite manipulation

An sqlite file contains a table called “data” with columns “value” and “type”. What is the average of the “value”s conditioned on the “type” for this sqlite file: example.sqlite?

Object-relational mapping

One-to-one mapping between Python class and SQL table.

```
from sqlobject import *
import os
__connection__ = 'sqlite:' + os.path.abspath('politicians.sqlite')

class Politician(SQLObject):
    name = UnicodeCol(length=60, notNone=True)
    partyletter = UnicodeCol(length=1)
    votes = IntCol()
    birthday = DateCol()
```

Create table:

```
Politician.createTable()
```

Now the `politicians.sqlite` SQLite database file has been written.

Insert a couple of politicians in the database:

```
>>> Politician(name=u"Lars Løkke Rasmussen", partyletter="V",
               votes=21384, birthday="1964-05-15")
>>> Politician(name=u"Villy Søvndal", partyletter="F",
               votes=33046, birthday="1952-04-04")
>>> Politician(name=u"Pia Kjærsgaard", partyletter="O",
               votes=47611, birthday="1947-02-23")
```

There is an "`id INT PRIMARY KEY AUTO_INCREMENT`" column added. Simple data fetch based on the `id`:

```
>>> Politician.get(1).name
u'Lars L\xf8kke Rasmussen'
```

Select with condition on Politicians:

```
>>> [p.name for p in Politician.select(Politician.q.votes > 30000,
                                       orderBy=Politician.q.birthday)]
[u'Pia Kj\xe6rsgaard', u'Villy S\xf8vndal']
```

Another example with numerical aggregation (summing, averaging):

```
import os
from sqlobject import *

__connection__ = 'sqlite: + os.path.abspath('pima.db')

class Pima(SQLObject):
    npreg = IntCol()
    bmi = FloatCol()
    type = EnumCol(enumValues=["Yes", "No"])

Pima.createTable()
Pima(npreg=6, bmi=33.6, type="Yes")
Pima(npreg=1, bmi=26.6, type="No")
Pima(npreg=1, bmi=28.1, type="No")
```

```
>>> Pima.selectBy(type='No').avg('bmi')
27.35
```

More complex queries are not necessarily pretty:

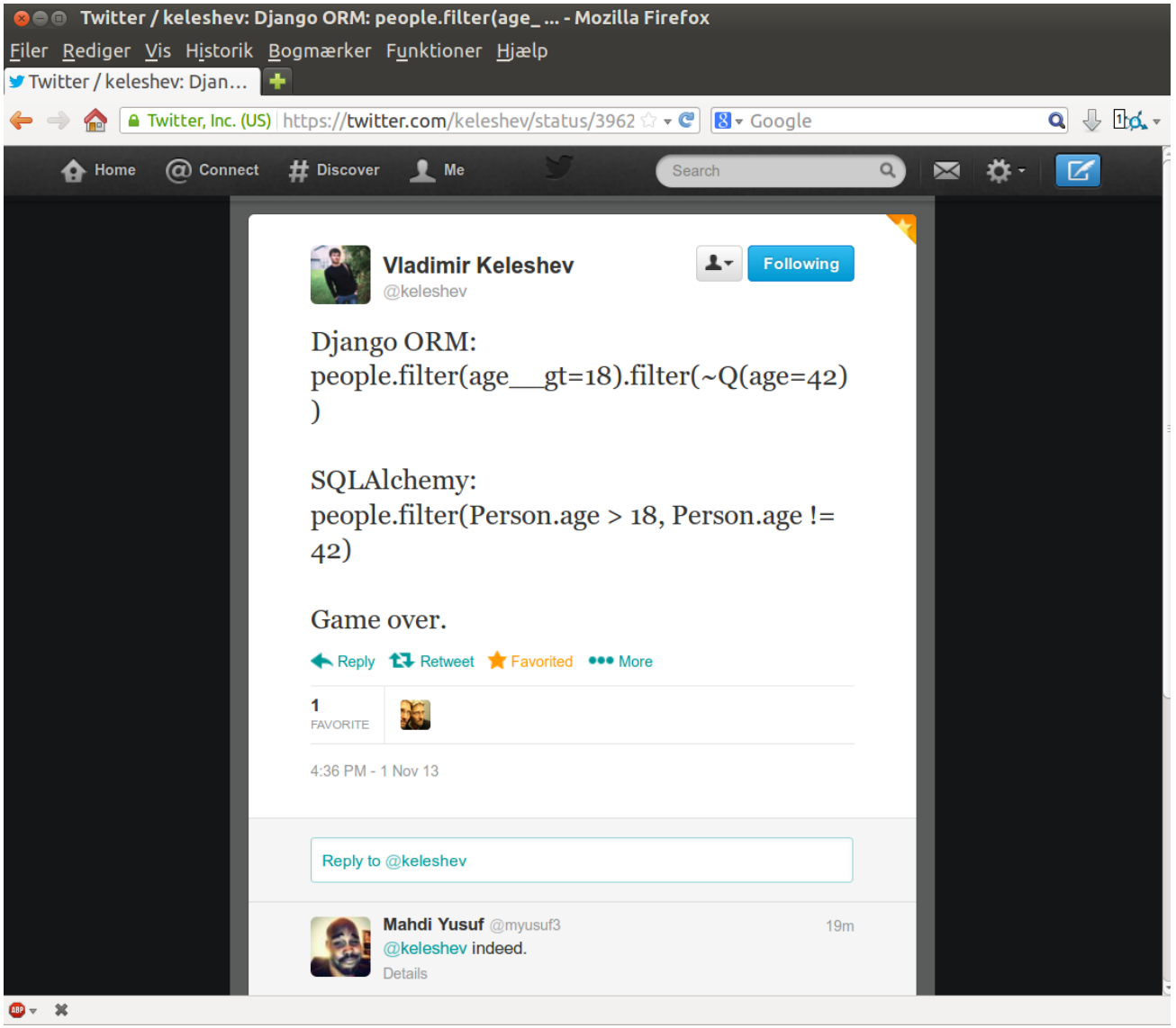
```
>>> c = Pima._connection
>>> c.queryAll("SELECT AVG(npreg), AVG(bmi), type FROM pima GROUP BY type")
[(1.0, 27.35, 'No'), (6.0, 33.6, 'Yes')]
```

The complex query using `sqlbuilder`:

```
>>> from sqlobject.sqlbuilder import *

>>> select = Select([func.AVG(Pima.q.npreg), func.AVG(Pima.q.bmi),
                     Pima.q.type],
                    groupBy=Pima.q.type)
>>> query = Pima.sqlrepr(select)
>>> Pima._connection.queryAll(query)
[(1.0, 27.35, 'No'), (6.0, 33.6, 'Yes')]
```

Which ORM?


Twitter / keleshev: Django ORM: people.filter(age_ ... - Mozilla Firefox
Filer Rediger Vis Historik Bogmærker Funktioner Hjælp
Twitter / keleshev: Djan...
Twitter, Inc. (US) https://twitter.com/keleshev/status/3962
Google
Home
Connect
Discover
Me
Search
Vladimir Keleshev
@keleshev
Following
Django ORM:
people.filter(age__gt=18).filter(~Q(age=42)
)
SQLAlchemy:
people.filter(Person.age > 18, Person.age !=
42)
Game over.
Reply
Retweet
Favorited
More
1
FAVORITE
4:36 PM - 1 Nov 13
Reply to @keleshev
Mahdi Yusuf @myusuf3
19m
@keleshev indeed.
Details

NoSQL

NoSQL = Databases with no SQL functionality

Examples: CouchDB, MongoDB, Cassandra, . . .

May have good capabilities for replication, high availability, sharding ☺

May lack more than simple query mechanisms ☹

(You can have NoSQL functionality in SQL databases)

MongoDB

MongoDB, http://www.mongodb.org/, stores JSON-like objects ☺

2GB limitation on 32-bit platforms ☹

```
$ mongod &              # The database server
$ mongo                 # The database client ('test' database default)
> use test
> book = { title : "Kongens Fald", author : "Johannes V. Jensen" }
> db.books.save(book)
> book = { title : "Himmerlandshistorier", author : "Johannes V. Jensen" }
> db.books.save(book)
> book = { title : "Eventyr", author : "H. C. Andersen" }
> db.books.save(book)
```

MongoDB query

```
> db.books.find()     # Find all
{ "_id" : ObjectId("4c82a97c6600b82f5cdcb93b"),
    "title" : "Kongens Fald", "author" : "Johannes V. Jensen" }
{ "_id" : ObjectId("4c82a98c6600b82f5cdcb93c"),
    "title" : "Himmerlandshistorier", "author" : "Johannes V. Jensen" }
{ "_id" : ObjectId("4c82a9996600b82f5cdcb93d"),
    "title" : "Eventyr", "author" : "H. C. Andersen" }

> db.books.find({ author : "Johannes V. Jensen" }) # Find a specific
{ "_id" : ObjectId("4c82a97c6600b82f5cdcb93b"),
    "title" : "Kongens Fald", "author" : "Johannes V. Jensen" }
{ "_id" : ObjectId("4c82a98c6600b82f5cdcb93c"),
    "title" : "Himmerlandshistorier", "author" : "Johannes V. Jensen" }
```

MongoDB with Python

```
$ sudo pip install pymongo    # Installation with "pip"
                              # Installation requires "python-dev"
$ mongod &                    # Starting server
$ python                      # and into Python

>>> import pymongo
>>> connection = pymongo.Connection()     # help(pymongo.Connection)
>>> db = connection.test
>>> books = db.books          # "books" is a "collection"
>>> book = {"title": "Kongens Fald", "author": "Johannes V. Jensen" }
>>> books.insert(book)
>>> morebooks = [{"title": "Himmerlandshistorier",
                  "author": "Johannes V. Jensen" },
                 {"title": "Eventyr", "author": "H. C. Andersen"},
                 {"title": "Biblen"}]
>>> books.insert(morebooks)
```

MongoDB with Python

Getting back information from the Mongo server:

```
>>> for book in books.find():
...     print(book.get("author", "Missing author"))
Johannes V. Jensen
Johannes V. Jensen
H. C. Andersen
Missing author
```

Updating a field:

```
>>> books.update({"title": "Himmerlandshistorier"},
                 {"$set": {"isbn": "8702040638"}})
```

Twitter, Python and MongoDB (outdated)

```
import getpass, pymongo, simplejson, urllib
password = getpass.getpass()       # get password for "fnielsen2" user
url = "http://fnielsen2:" + password + \
      "@stream.twitter.com/1/statuses/sample.json"
connection = pymongo.Connection()
db = connection.twitter
tweets = db.tweets
for tweet in urllib.urlopen(url):
    oid = tweets.insert(simplejson.loads(tweet))
    print(tweets.count())
```

As Twitter returns JSON we can directly convert it to a Python structure and further to an entry in Mongo. (Note: Twitter is changing its authentication)

```
>>> tweets.find_one().keys()
[u'favorited', u'retweet_count', u'in_reply_to_user_id',
u'contributors', u'truncated', u'text', u'created_at', u'retweeted',
u'coordinates', u'entities', u'in_reply_to_status_id', u'place',
u'source', u'in_reply_to_screen_name', u'_id', u'geo', u'id', u'user']

>>> tweets.find_one()['user'].keys()
[u'follow_request_sent', u'profile_use_background_image', u'id',
u'verified', u'profile_sidebar_fill_color', u'profile_text_color',
u'followers_count', u'profile_sidebar_border_color', u'location',
u'profile_background_color', u'listed_count', u'utc_offset',
u'statuses_count', u'description', u'friends_count',
u'profile_link_color', u'profile_image_url', u'notifications',
u'show_all_inline_media', u'geo_enabled',
u'profile_background_image_url', u'screen_name', u'lang',
u'profile_background_tile', u'favourites_count', u'name', u'url',
u'created_at', u'contributors_enabled', u'time_zone', u'protected',
u'following']
```

Python MongoDB: updating a field

```
import sys
reload(sys)
sys.setdefaultencoding('utf-8')

# Load a dictionary with words scored according to how English
# -3 not English, +3 English, 0 unknown or might or might not be English
filename = '/home/fn/fnielsen/data/Nielsen2010Responsible_english.csv'
englishwords = dict(map(lambda (k,v): (k,int(v)),
    [line.split() for line in open(filename)]))

for tweet in tweets.find({"delete": {"$exists": False}}):
  englishness = sum(map(lambda word: englishwords.get(word.lower(),0),
      tweet['text'].split()))
  tweet['englishness'] = englishness          # Add a new field
  oid = tweets.save(tweet)                    # Overwrite the element
```

MongoDB queries in Python

```
>>> tweets.find_one({"englishness": {"$gt": 15}})['text']
u'@ItsAnnaJayce HEY babs! could you please take one min to follow
@kyleinju5tice it would be appreciated thankyouuu:) &lt;3 #LLWFH'

>>> tweets.find_one({"englishness": {"$lt": -15}})['text']
u'eu quero sorvete e shampoo mas minha mae fica enrrolando na cama e
at\xe9 agora nao foi comprar .-.'

>>> tweets.find_one({"englishness": 0})['text']
u"@rafiniits  axo mtmtmt boom '-' UYSAGDYGAYSUG'"
```

See also http://www.mongodb.org/display/DOCS/Advanced+Queries

CouchDB and Python

CouchDB http://couchdb.apache.org/ RESTful, JSON ☺

```
$ couchdb
$ python
>>> import couchdb, couchdb.schema, urllib
>>> urllib.urlopen('http://localhost:5984/').read()     # RESTful
'{"couchdb":"Welcome","version":"0.10.0"}\n'
>>> server = couchdb.client.Server()   # Default http://localhost:5984
>>> db = server.create('books')
>>> urllib.urlopen('http://localhost:5984/_all_dbs').read()
'["books"]\n'
```

Inserting three books:

```
>>> did = db.create({"author": "H. C. Andersen", "title": "Eventyr"})
>>> did = db.create({"title": "Biblen"})
>>> did = db.create({"author": "Johannes V. Jensen",
                     "title": "Kongens Fald"})
```

Query CouchDB with Python

Simple getting of fields:

```
>>> for book in db: print(db[book].get("author", "Missing author"))
H. C. Andersen
Johannes V. Jensen
Missing author
```

Query via a map function written as a JavaScript function:

```
>>> fun = """function(book) { if (book.author == 'H. C. Andersen')
...                emit(book._id, book) }"""
>>> db.query(fun).rows[0]['value']['title']
'Eventyr'
```

CouchDB updating a field with Python

```
>>> fun = "function(b) { if (b.title == 'Kongens Fald') emit(b._id, b)}"
>>> result = db.query(fun).rows[0]
>>> did, book = result["key"], result["value"]
>>> book["isbn"] = "9788702058239"
>>> db[did] = book
```

More information

Introduction to SQLAlchemy, video tutorial with Mike Bayer from Pycon 2013.

Summary

Pickle is the primary format for storing and serializing Python objects, and you can use `pickle` module or the faster `cPickle` module for dumping and loading.

Problems with pickle is that it is for Python only and that you can have dangerous code in it.

There are interfaces to database management systems

Key/value store: shelve, Berkeley DB

Relational databases: Either "directly" (e.g., by MySQLdb), or generic (pyodbc), or ORM.

NoSQL access also available, e.g., MongoDB, CouchDB, ...

Usually a good idea to use ORM, but beware that ORM can yield large number of SQL queries from one ORM query.

References

Downey, A. (2008). *Think Python*. Green Tea Press, Needham, Massachusetts, version 1.1.15 edition.

Martelli, A., Ravenscroft, A. M., and Ascher, D., editors (2005). *Python Cookbook*. O'Reilly, Sebastopol, California, 2nd edition.