LECTURE 16

Serialization and
Data Persistence

SERIALIZATION

Serialization refers to the flattening of complex object hierarchies into a format that is easily stored, sent over a network, or shared with another program.

A good example of this is when you save in a video game. Your progress may be represented by a data structure which holds all of the necessary information about the game and character state at that moment.

To save, the data structure is serialized, or flattened, into a writeable format to be written to disk. When you want to pick up where you left off, the game will use the saved data to reconstruct the data structure which contains information about your game.

SERIALIZATION

There are a number of application-dependent methods for serializing data in Python, but the two most common (and probably the only ones you'll need) are:

- pickle and cpickle
- json

PICKLE

The pickle module is the main mechanism provided by Python for serializing python objects. Basically, pickle turns a Python structure into a bytestream representation which is easily sent or stored. Then, pickle can be used to reconstruct the original object.

The pickle module can serialize:

- All of Python's native datatypes: floating point numbers, Booleans, integers, etc.
- Lists, tuples, dictionaries, and sets containing any combination of native datatypes.
- Lists, tuples, dictionaries, and sets containing any combination of lists, tuples, dictionaries, and sets (and so on…).
- Functions, classes, and instances of classes.

CPICKLE

A faster pickle. The cpickle module is the exact same algorithm implemented in C instead of Python.

Using cpickle means pickling at speeds up to 1000x faster than pickle.

However, you cannot create custom pickling and unpickling classes with cpickle. But if customizability is not important for you, use cpickle.

The bytestreams produced by pickle and cpickle are identical so they can be used interchangeably on pickled data.

PICKLE

Advantages:

- Customizable.
- Can serialize pretty much any Python object.
- Space efficient – only stores multiply-used objects once.
- Easy for small uses.

Disadvantages:

- Slower than most other methods.
- Not secure: no protection against malicious data.
- Python specific. Can't communicate with non-Python code.

PICKLE

There are actually a number of data stream formats to choose from when pickling your data. You can specify the protocol, which defaults to 0.

- Protocol version 0 is the original ASCII protocol and is backwards compatible with earlier versions of Python.
- Protocol version 1 is the old binary format which is also compatible with earlier versions of Python.
- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of new-style classes.

PICKLE

As I said before, you should probably be using cpickle. Assume that I have the following import statement at the top of all of the example code:

```
try:
    import cPickle as pickle
except:
    import pickle
```

So, we'll alias cpickle as pickle when possible. Otherwise we default to pickle.

PICKLE

Let's take the following Python object as an example. We have a list object with a dictionary, list, and integer as elements.

```
obj = [{'one': 1, 'two': 2}, [3, 'four'], 5]
```

The simplest usage of pickle involves creating a `Pickler` instance and calling it's `dump()` method.

```
p = pickle.Pickler(file, protocol)
p.dump(obj)
```

PICKLE

The file argument can be any Python object with a `write()` method (file, socket, pipe, etc). The protocol is either 0, 1, or 2 depending on the protocol you'd like the bytestream to follow.

```
obj = [{'one': 1, 'two': 2}, [3, 'four'], 5]
p = pickle.Pickler(open("data.p", 'w'), 0)
p.dump(obj)
```

The contents of data.p are:

```
(lp0 (dp1 S'two' p2 I2 sS'one' p3 I1 sa(lp4 I3 aS'four' p5 aaI5 a.
```

Not super-readable but at least pickle knows what's going on. We could do some debugging if we had to.

PICKLE

We could also call the convenient `pickle.dump()` method.

```
pickle.dump(obj, file, protocol)
```

which is equivalent to

```
p = pickle.Pickler(file, protocol)
p.dump(obj)
```

PICKLE

Furthermore, we have (note the ‘s’)

```
pickle.dumps(obj, protocol)
```

which returns the pickled object instead of writing it to a file.

PICKLE

To deserialize, we create an `Unpickler` object and call its `load()` method.

```
u = pickle.Unpickler(file)
obj = u.load()
```

To reconstruct the object represented in data.p:

```
f = open("data.p", 'rb')
u = pickle.Unpickler(f)
obj = u.load()
print obj # Output: [{'two': 2, 'one': 1}, [3, 'four'], 5]
```

PICKLE

Like pickling, unpickling allows for some convenience functions. The method `load()` simply deserializes the contents of the file argument.

```
pickle.load(file)
```

The method `loads()` deserializes the string passed into it as an argument.

```
pickle.loads(pickled_string)
```

REDIS EXAMPLE

For this example, we'll use Redis, an in-memory database. Assuming Redis is installed, we'll use Python's redis module to push and pop request items from a queue.

redis_sender.py

```
import redis
import uuid
import pickle
conn = redis.Redis('localhost', 6379)
request = {'request_id': uuid.uuid4(), 'event': 'Taylor Swift',
           'location': 'Orlando'}
pickled_request = pickle.dumps(request, 0)
conn.rpush('queue', pickled_request)
```

uuid.uuid4() generates a unique identifier

REDIS EXAMPLE

For this example, we'll use Redis, an in-memory database. Assuming Redis is installed, we'll use Python's redis module to push and pop request items from a queue.

redis_receiver.py

```
import redis
import uuid
import pickle

conn = redis.Redis('localhost', 6379)
pickled_request = conn.lpop('queue')
request = pickle.loads(pickled_request)
print "ID: ", request['request_id']
print "Event: ", request['event']
print "Location: ", request['location']
```

REDIS EXAMPLE

Output from redis_receiver.py:

```
ID:  7e7b55d7-b08d-44c2-8385-9d67eb988660
Event:  Taylor Swift
Location:  Orlando
```

JSON

The json module provides an interface similar to pickle for converting Python objects into JavaScript Object Notation.

Advantages:

- Suitable for use with other languages – not Python-specific.
- Always text-based. No guessing.
- More readable than the pickle style.
- Not as dangerous as pickle.
- Faster than cpickle.

Disadvantage:

- Not all Python types supported.

JSON

We'll be using the same sample Python object.

```
import json
obj = [{'one': 1, 'two': 2}, [3, 'four'], 5]
print obj
data = json.dumps(obj)
print data
```

The output is

```
[{'two': 2, 'one': 1}, [3, 'four'], 5]
[{"two": 2, "one": 1}, [3, "four"], 5]
```

JSON

The serialization methods are:

- `json.dump(obj, file, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, encoding="utf-8", default=None, sort_keys=False, **kw)`
- `json.dumps(obj, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, encoding="utf-8", default=None, sort_keys=False, **kw)`

JSON

Serialization options:

• If `skipkeys` is True (default: False), then dict keys that are not of a basic type will be skipped instead of raising a TypeError.

• If `indent` is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level.

• If `separators` is an (item_separator, dict_separator) tuple, then it will be used instead of the default (', ', ': ') separators. (',', ':') is the most compact JSON representation.

• `encoding` is the character encoding for str instances, default is UTF-8.

• `cls` is a custom JSONEncoder class to use. Default to None.

JSON

Adding deserialization to our example,

```
import json obj = [{'one': 1, 'two': 2}, [3, 'four'], 5]
print obj
data = json.dumps(obj)
print data
print json.loads(data)
```

The output is:

```
[{'two': 2, 'one': 1}, [3, 'four'], 5]
[{"two": 2, "one": 1}, [3, "four"], 5]
[{u'two': 2, u'one': 1}, [3, u'four'], 5]
```

JSON

The deserialization methods are:

- `json.load(fp[, encoding[, cls[, object_hook[, parse_float[, parse_int[, parse_constant[, object_pairs_hook[, **kw]]]]]]]])`
- `json.loads(s[, encoding[, cls[, object_hook[, parse_float[, parse_int[, parse_constant[, object_pairs_hook[, **kw]]]]]]]])`

JSON

Deserialization options:

- `parse_float`, if specified, will be called with the string of every JSON float to be decoded. By default, this is equivalent to `float(num_str)`.
- `parse_int`, if specified, will be called with the string of every JSON int to be decoded. By default, this is equivalent to `int(num_str)`.
- `cls` specifies a custom JSONDecoder class.

JSON

The json module also provides Encoder and Decoder classes which can be used for extra functionality with native data types or to create custom subclasses.

- `json.jsonEncoder()`
  - `encode(obj)` – return a JSON representation of the Python object *obj*.
  - `iterencode(obj)` -- Encode the given *obj* and yield each string representation as available.
- `json.jsonDecoder()`
  - `decode(s)` – return the Python representation of *s*.

JSON

The JSONEncoder iterencode method provides an iterable interface for producing "chunks" of encoded data. This is more convenient for sending large amounts of serialized data over a network.

```
import json

encoder = json.JSONEncoder()
data = [ { 'a':'A', 'b':(2, 4) } ]

for part in encoder.iterencode(data):
    print part
```

```
[
{
"a"
:
"A"
,
"b"
:
[2
, 4
]
}
]
```

JSON

Here's an example of a custom JSON encoder.

We implement the default method in a subclass of JSONEncoder such that it returns a serializable object for o, or calls the base implementation (to raise a TypeError).

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         return json.JSONEncoder.default(self, obj)
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[', '2.0', ', ', '1.0', ']']
```

SHELVE

A “shelf” is a persistent, dictionary-like object which can store arbitrary Python objects as values. The keys themselves must be strings.

Use a shelf anytime a relational database would be overkill. Keep in mind that shelf does not support concurrent writes and because it uses pickle, it’s also vulnerable to the same security problems as pickle.

Do not “unshelf” python objects that you do not trust!

SHELVE

The simplest usage of a shelf just involves the `open()` and `close()` methods.

```
import shelve
s = shelve.open('test_shelf.db')
s['key1'] = {'an_int': 8, 'a_float':3.7, 'a_string': 'Hello!'}
s.close()
```

The shelve module pickles the values to be stored in test_shelf.db.

SHELVE

Retrieving data from a shelf is as simple as opening the file and accessing its key.

```
import shelve
s = shelve.open('test_shelf.db')
val = s['key1']
print val
print val['a_string']
```

Output:

```
{'a_float': 3.7, 'an_int': 8, 'a_string': 'Hello!'}
Hello!
```

DATABASES

Commonly, Python applications will need to access a database of some sort.

As you can imagine, not only is this easy to do in Python but there is a ton of support for various relational and non-relational databases.

•Databases for which there is module support include:

- MySQL
- PostgreSQL
- Oracle
- SQLite
- Cassandra
- MongoDB
- etc…

DATABASES

Even for a certain database, there are a number of module options. For example, MySQL alone has the following interface modules:

- MySQL for Python (import MySQLdb)
- PyMySQL (import pymysql)
- pyODBC (import pyodbc)
- MySQL Connector/Python (import mysql.connector)
- mypysql (import mypysql)
- etc …

Yes, for every combination of my, py, and sql, there is someone out there with a "better" implementation of a MySQL module.

DATABASE API SPECIFICATION

So which module do you choose? Well, as far as code-writing goes, it probably won't make that much of a difference…

Python Enhancement Proposal 249 provides the API specification for modules that interface with databases. You can access the specification here.

The majority of database modules conform to the specification so no matter which kind of database and/or module you choose, the code will likely look very similar.

DATABASE API SPECIFICATION

The module interface is required to have the following:

- `connect(args)` – a constructor for `Connection` objects, through which access is made available. Arguments are database-dependent.
- Globals `apilevel` (DB API level 1.0 or 2.0), `threadsafety` (integer constant indicating thread safety status), `paramstyle` (string constant indicating query parameter style).
- A number of exceptions, including `IntegrityError`, `OperationalError`, `DataError`, etc.

DATABASE API SPECIFICATION

A little more about `paramstyle`: the defined string constants are shown below along with an example of each.

Constant	Meaning
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

DATABASE API SPECIFICATION

So assuming `conn = connect(args)` yields a `Connection` object, we should be able to manipulate our connection via the following methods:

- `conn.close()` – close connection.
- `conn.commit()` – commit pending transaction.
- `conn.rollback()` – if supported by db, roll back to start of pending transaction.
- `conn.cursor()` – return a `Cursor` object for the connection.

DATABASE API SPECIFICATION

So `c = conn.cursor()` should yield a `Cursor` object. We can have multiple cursors per connection, but they are not isolated from one another. The following attributes should be available:

- `c.description` – a description of the cursor with up to seven fields.
- `c.rowcount` – number of rows produced by last execute method.

DATABASE API SPECIFICATION

So `c = conn.cursor()` should yield a `Cursor` object. We can have multiple cursors per connection, but they are not isolated from one another. The following methods should be available:

- `c.execute[many](`*`op`*`, `*`[params]`*`)` – prepare and execute an operation with parameters where the second argument may be a list of parameter sequences.
- `c.fetch[one|many|all]([`*`s`*`])` – fetch next row, next *s* rows, or all remaining rows of result set.
- `c.close()` – close cursor.
- and others.

DATABASE API SPECIFICATION

There are a number of optional extensions such as the `rownumber` attribute for cursors, which specifies the current row of the result set.

There are also additional implementation requirements that are not necessary to be familiar with as a user of the module.

So now we basically understand how most of Python's database modules work.

MYSQLDB

```
import MySQLdb

db = MySQLdb.connect("localhost","username", "password", "EmployeeData")
cursor = db.cursor()
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE) VALUES ('%s', '%s', '%d')" %
       ('Caitlin', 'Carnahan', 24)

try:
    cursor.execute(sql)
    db.commit()
except:
    db.rollback()
db.close()
```

PSYCOPG2

```
import psycopg2

db = psycopg2.connect(database="mydatabase", user="uname", password="pword")
c = db.cursor()
c.execute ("SELECT * FROM versions")
rows = c.fetchall()
for i, row in enumerate(rows):
    print "Row", i, "value = ", row
c.execute("DELETE FROM versions")
c.execute ("DROP TABLE versions")
c.close()
db.close()
```

SQLITE3

To get a feel for database usage in Python, we'll play around with the sqlite3 module, which is a part of the standard library.

SQLite is a lightweight C-based relational database management system which uses a variant of the SQL language. The data is essentially stored in a file which is manipulated by the functions of the C library that implements SQLite.

SQLITE3

The very first thing we'll need is to build an sqlite3 database to mess around with.

Building off of our Blackjack application, let's build a database for tracking most winningest (yep) sessions.

When a user plays a session, they'll be able to record their name and number of winning games in the database. We'll then also allow them the option to see the top high scorers after they are finished playing.

SQLITE3

We'll start by creating our SQLite database with the sqlite3 command line tool.

The database file is highscores.db.
The table is called Scores.
We have two columns: name and wins.

Now, I just need to modify my blackjack program with the functionality to add and views records in the database.

Also check out the Firefox SQLite manager.

```
$ sqlite3 highscores.db
SQLite version 3.7.7 2011-06-23 19:49:22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> create table Scores(name varchar(10),
          wins smallint);
sqlite> insert into Scores values ('Caitlin', 3);
sqlite> insert into Scores values ('Ben', 4);
sqlite> insert into Scores values ('Melina', 2);
sqlite> select * from Scores;
Caitlin|3
Ben|4
Melina|2
sqlite>
```

DB_INTERFACE.PY

We're going to add to our application a python module responsible for connecting to the SQLite database and inserting/grabbing data.

```
import sqlite3

def top_scores():
    # Returns top three highest winners
    conn = sqlite3.connect("highscores.db")
    c = conn.cursor()
    c.execute("SELECT * FROM Scores ORDER BY wins DESC;")
    result_rows = c.fetchmany(3)
    conn.close()
    return result_rows

def insert_score(name, wins):
    # Inserts a name and score
    conn = sqlite3.connect("highscores.db")
    c = conn.cursor()
    c.execute("INSERT INTO Scores VALUES (?, ?);",
                (name, wins))
    conn.commit()
    conn.close()

if __name__ == "__main__":
    for s in top_scores():
        print s
```