Data Persistence

JSON for list/dict serialization

JSON is a text format that stores Python list and dict objects.

In a file called mystruct.json:

```
{
   "key1":  ["a", "b", "c"],
   "key2":  {
              "innerkey1": 5,
              "innerkey2": "woah"
            },
   "key3":  55.09,
   "key4":  "hello"
}
```

Python code to read mystruct.json

```
fh = open('mystruct.json')    # file contains JSON
mys = json.load(fh)           # load from a file
fh.close()

print mys['key2']['innerkey2']   # woah
```

Python code to write JSON to file

```
mys['key2']['innerkey2'] = 'oh yes'   # change struct

wfh = open('newstruct.json', 'w')  # open file for writing
json.dump(mys, wfh)                # write JSON to file
```

JSON format is stringent
Although it resembles Python structures, JSON notation is slightly less forgiving than Python: double quotes are required around strings, and no trailing comma is allowed after the last element in a dict or list.

Violating any JSON rules results in a failure to load the file, and a challenging error message:

```
{
  "key1":  "goodbye",
  "key4":  "hello",        # no trailing comma allowed
}
```

When I then tried to load it, the **json** module complained with an error location (although in a larger file this could take time to locate in the file):

```
ValueError: Expecting property name: line 3 column 18 (char 42)
```

pickle for object serialization

Any Python data object can be written to disk without modification.

```
import pickle
br = [('joe', 22, 'clerk', 33000.0), ('pete', 34, 'salesman', 51000.0)]
F = open('datafile.pickle', 'w')
pickle.dump(br, F)
F.close()

## later
G = open('datafile.txt')
J = pickle.load(G)
print J
    #  [('joe', 22, 'clerk', 33000.0), ('pete', 34, 'salesman', 51000.0)]
```

This saves the added step of rearranging data in an object into a structure that can be stored conventionally (for example in a relational database, CSV file or even JSON file).

pickle with complex data objects

For ease of use, custom-designed objects are superior. As they can behave like Python objects, they can be used in a variety of familiar situations. As they are custom-designed, they can be made to hold data in any form.

Since custom-designed objects can be serialized using pickle, they can be stored directly without using object<->relational or object<->JSON translations. pickle is slower than RDBS and JSON, but it is a convenient option for lightweight applications.

Consider this pair of data objects: one holds items and the other is an item:

```
import pickle

class ItemList(object):

    def __init__(self, name):
        self.name = name
        self.items = []
        self.index = -1

    def add(self, text):
        self.items.append(Item(text))

    def __iter__(self):
        return self

    def next(self):
        self.index += 1
        if self.index > len(self.items) - 1:
            raise StopIteration
        return self.items[self.index]

class Item(object):
    def __init__(self, text):
        self.text = text
```

ItemList is designed to store and recover **Item** objects, and **Item** objects are designed to hold a line of text (obviously each of these could be much more complex).

Since we designed these objects, they are easy to use:

```
this_list = ItemList('mylist')

this_list.add('line 1')
this_list.add('line 2')
this_list.add('line 3')
```

pickle makes object storage and recovery a snap:

```
wfh = open('picklefile.pickle', 'w')
pickle.dump(this_list, wfh)
wfh.close()

### THE NEXT DAY...

fh = open('picklefile.pickle')

resurrected_list = pickle.load(fh)

print 'items for {}'.format(resurrected_list.name)
for item in resurrected_list:
    print '  ' + item.text
```

But because we have designed the objects around certain built-in functionality, it's possible to plug them directly into an unrelated feature such as Jinja2 template, without performing any transformations:

Template show_items.html

```
<h1>Items for {{ resurected_list.name }}</h1>
{% for item in resurrected_list %}
  {{ item }}
{% endfor %}
```

Flask code for above

```
  fh = open('picklefile.pickle')
  thislist = pickle.load(fh)

  return render_template('show_items.html', resurrected_list=thislist)
```

SQLite

SQLite is a simple database, similar to MySQL. We can

```
def insert_db(fname, lname):
    sql = INSERT_QUERY.format(TABLE_NAME, fname, lname)
    execute_query(sql)

def get_db_data():
    sql = SELECT_QUERY.format(TABLE_NAME)
    c = execute_query(sql)
    return c.fetchall()
```

We have also defined a function that can execute any query provided to it. Our system creates a database connection when needed, and then stores it in the attribute of a special object called **flask.g**. This all-purpose storage object is

available through the module directly, so it has global scope.

```
def execute_query(sql):
    conn = get_db()
    c = conn.cursor()
    c.execute(sql)
    conn.commit()
    return c

def get_db():
    db = getattr(flask.g, '_database', None)
    if db is None:
        db = flask.g._database = sqlite3.connect(DATABASE)
    return db

@app.teardown_appcontext
def close_connection(exception):
    db = getattr(flask.g, '_database', None)
    if db is not None:
        db.close()
```

Also included here is a special function that Flask can invoke automatically when the program is exiting -- it closes the database connection we stored in **flask.g**.