CSC326 Persistent Programming

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.0	2011-09		JZ

Contents

1 Agenda

- File
- Exception
- Key-Value Store
- Client Server Key-Value Store
- Relational Database

2 Persistent Programming

- So far we have seen
  - Imperative programming constructs
  - operating on data structures
- Data structures are
  - Transient In memory
  - Lost if power is done
  - Not carried across different sessions of the same program
- Persistent programs
  - Run for a long time (all the time)
  - Keep at least some data in perment storage (disk, flash etc)
  - Examples: OS, web servers

3 File

- Opening file

```
>>> fout = open( 'output.txt', 'w' )
>>> print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

- Write file

```
>>> line1 = "This her's the wattle,\n"
>>> fout.write( line1 )
```

- Close file

```
>>> fout.close()
```

- Formating
  - argment of write has to be a string

- format operator %
- similar to printf in C, except in expression form

```
>>> camels = 42
>>> '%d' % camels
'42'
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

- If more than one, has to use tuple

```
>>> 'In %d years I have spotted %d %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camles.'
```

- Make sure to match types

4 File names and Path

```
>>> import os
>>> cwd = os.getcwd()
>>> print cwd
/home/jzhu
>>> os.path.abspath( 'memo.txt' )
'/home/jzhu/memo.txt'
>>> os.path.exists( 'memo.txt' )
True
>>> os.path.isdir( 'memo.txt' )
False
>>> os.path.isfile( 'memo.txt' )
False
>>> os.path.listdir( cwd )
['music', 'photos', 'memo.txt']
```

```
def walk(dir) :
    for name is os.listdir(dir) :
        path = os.path.join( dir, name )
        if os.path.isfile(path) :
           print path
        else
           walk(path)
```

5 Catching Exception

```
>>> fin = open( 'bad_file' )
IOError: [Errno 2] No such file or directory: 'bad_file'
>>> fin = open( '/etc/passwd', w )
IOError: [Errno 13] Permission deined: 'etc/passwd'
>>> fin = open( '/home' )
IOError: [Errno 21] Is a directory
```

```
try :
    fin = open( 'bad_file' )
    for line in fin :
      print line
    fin.clse()
except :
    print 'Something went wrong'
```

6 Key-Value Store

- Persistent dictionary

```
>>> import anydbm
>>> db = anydbm.open( 'captions.db', 'c' )
>>> db['cleese.png'] = 'Photo of John Cleese'
>>> print db['cleese.png'] = 'Photo of John Cleese'
Phote of John Cleese.
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> for key in db:
        print key
>>> db.close()
```

7 Pickling

- Both file and anydbm expect strings
- Using formating operator for arbitary data structure can be complicated
- pickle package helps

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps( t )
'lp0\nI1\naI2\naI3\na'
```

- Not really human readible
- But pickle can read it back

```
>>> import pickle
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps( t1 )
>>> t2 = pickle.loads(s)
>>> print t2
[1, 2, 3]
```

8 Client-Server Key-Value Store

- Problems
  - What if I need many applications simultaneously accessing the same database?

    ```
    Example: Horizontally scaled web farm with many different web
    servers running on separate machines, your web app can all call
    on and reference the same database instances
    ```

  - String as the only data type supported is two restrictive, need native data type support
  - Atomicity
- Solution
  - Client-Server architecture: database server
  - Redis
- Building Redis

```
$ wget http://redis.googlecode.com/files/redis-1.2.5.tar.gz
$ gunzip redis-1.2.5.tar.gz
$ tar -xvf redis-1.2.5.tar
$ cd redis-1.2.5
$ make
```

- Running Server

```
$ ./redis-server
```

- Installing Client

```
easy_install redis
```

- Data Structure: String

```
>>> import redis
>>> r_server = redis.Redis("localhost")
>>> r_server.set("name", "DeGizmo")
True
>>> r_server.get("name")
'DeGizmo'
```

- Data Structure: Integer

```
>>> r_server.set("hit_counter", 1)
True
>>> r_server.incr("hit_counter")
2
>>> r_server.get("hit_counter")
'2'
>>> r_server.decr("hit_counter")
1
>>> r_server.get("hit_counter")
'1'
```

- Data structure: Lists

Very similar to the built in Python list type, the redis list type has a few basic methods that combined can quite powerful. We are only covering a tiny portion of the commands available, you can find all the commands in the redis Command Reference.

```
>>> r_server.rpush("members", "Adam")
True
>>> r_server.rpush("members", "Bob")
True
>>> r_server.rpush("members", "Carol")
True
>>> r_server.lrange("members", 0, -1)
['Adam', 'Bob', 'Carol']
>>> r_server.llen("members")
3
>>> r_server.lindex("members", 1)
'Bob'
```

1. With the r_server object again we call the method *rpush* which will add the value *Adam* to the newly created list *members*
2. We add *Bob* to the same list
3. Finally we'll add *Carol*
4. With the lrange method we are asking redis to return all the objects in *members*. lrange takes 3 arguments: key, start index in list, end index in list. We are getting the objects from the key *members* starting at index 0 and ending at -1 (which is technically the -1, or last index in the list, which will return everything)
5. The llen method asks redis to return the length of the list at the key *members* which now has 3 objects
6. lindex method tells redis that we want the object from the key *members* at the index position of 1 (remember lists start at index 0), so redis returns *Bob*

We've got some elements in the list at the key *members*; now lets get remove some elements.

```
>>> r_server.rpop("members")
'Carol'
>>> r_server.lrange("members", 0, -1)
['Adam', 'Bob']
>>> r_server.lpop("members")
'Adam'
>>> r_server.lrange("members", 0, -1)
['Bob']
```

1. With the method rpop (right pop) we remove the element in the list on the right side (tail), which is *Carol*
2. Now when we ask for the list *members* from redis again (from the start of 0, and the end of -1 which returns everything) we see our list now doesn't have *Carol*
3. We now lpop (left pop) an element from the list *members*. This will remove the far left element from the list, *Adam*
4. Now the entire list only contains *Bob*
   - Data structure: Set

Again, sets perform identically to the built in Python set type. Simply, sets are lists but, can only have unique values. In the above example if we added the value Adam (r_server.lpush(*members*, *Adam*)) 20 times our list would contain 20 values all containing the value *Adam*. In a set, all elements are unique.

```
>>> r_server.delete("members")
True
>>> r_server.sadd("members", "Adam")
True
>>> r_server.sadd("members", "Bob")
True
>>> r_server.sadd("members", "Carol")
True
>>> r_server.sadd("members", "Adam")
False
>>> r_server.smembers("members")
set(['Bob', 'Adam', 'Carol'])
```

1. First off we delete the old list in the key *members* so we can use it as a set
2. Then we sadd (set add) the value *Adam* to the key *members*
3. Do the same for the value *Bob*
4. Do the same for the value *Carol*
5. Now we try to add the value *Adam* to the key *members* again but, this time it returns *False* since we are working on a set, and a set only has unique values. There already is a value *Adam* present in this set
6. The method smembers returns all the members of the set in the Python type set

An example of a use of sets in a web application would be for *upvotes* on a reddit, or hacker news type website. We want to keep track of who up votes a story but, you should only be able to up vote a story once.

```
>>> r_server.sadd("story:5419:upvotes", "userid:9102")
True
>>> r_server.sadd("story:5419:upvotes", "userid:12981")
True
>>> r_server.sadd("story:5419:upvotes", "userid:1233")
True
>>> r_server.sadd("story:5419:upvotes", "userid:9102")
False
>>> r_server.scard("story:5419:upvotes")
3
>>> r_server.smembers("story:5419:upvotes")
set(['userid:12981', 'userid:1233', 'userid:9102'])
```

I added a little twist in here with the name of the key: *story:5419:upvotes* but, it's easy to explain. Redis is *flat* with it's keyspace. So if we have many different stories we use a fixed key naming convention for easy reference in redis. For this example our key is broken down like this: *object type* : *id* : *attribute*. So, we have an object of type *story* with an ID of *5419* with an attribute *upvotes*. Redis has no idea what any of this means it just knows the key is *story:5419:upvotes* but, it doesn't matter, we know what it means and we can divide up our objects into this name space to make it easier to work with and keep from *losing* things. The value being added to the key is divided up in the same way. *userid* is the type and *9102* is the value or the ID for that user voting on the story.

1. Just like before we are adding the value *userid:9102* to the key *story:5419:upvotes*
2. Now we are adding the value *userid:12981*
3. Finally adding the valud *userid:1233*
4. Now, the user with the ID 9102 tries to up vote the story with the ID 5419 again, and redis returns False since that user has already up votes this story before and you can't up vote a story twice!
5. The method *scard* is asking redis for the cardinality of the set at key *story:5419:upvotes* or how many things are in this set, and redis returns 3. 6. Finally we return the list of userid's that we have stored in the set.

- Data structure: Ordered sets

The last data structure we are going to talk about today is an ordered (or sorted) set. This operates just like a set but, has an extra attribute when we add something to a set called a *score*. This score determines the order of the elements in the set. We will stick with the concept for this final example

```
>>> r_server.zadd("stories:frontpage", "storyid:3123", 34)
True
>>> r_server.zadd("stories:frontpage", "storyid:9001", 3)
True
>>> r_server.zadd("stories:frontpage", "storyid:2134", 127)
True
>>> r_server.zadd("stories:frontpage", "storyid:2134", 127)
False
>>> r_server.zrange("stories:frontpage", 0, -1, withscores=True)
[('storyid:9001', 3.0), ('storyid:3123', 34.0), ('storyid:2134', 127.0)]
>>> frontpage = r_server.zrange("stories:frontpage", 0, -1, withscores=True)
>>> frontpage.reverse()
>>> frontpage
[('storyid:2134', 127.0), ('storyid:3123', 34.0), ('storyid:9001', 3.0)]
```

Quick namespace explanation like before. For the key we are going to be referring to *stories:frontpage* which is going to be a set of stories slated for the front page of our website. We are storing in that key the value of *storyid:3123* which is the ID of some story on the site and then a score, which in our case is going to be the number of votes on a story.

1. First we add the value *storyid:3123* to *stories:frontpage*, and *storyid:3123* in our example is going to have 34 votes.
2. Then add *storyid:9001* with 3 votes
3. Then add *storyid:2134* with 127 votes
4. We are going to try to add *story:2134* to the set again but, we can't since it already exists.
5. Now we are going to ask redis for all the elements in *stories:frontpage* from index 0 to index -1 (the end of the list) with all associated scores (withscores=True)
6. We've got the scores but, they are in ascending order, we want them in descending order for our website, so we are going to store the results in the variable *frontpage*
7. Then reverse it (which is an in place operation in Python)
8. Now print out the front page!

In conclusion let's do a quick example of a *view* in an application in which a user will vote of a story using redis as a storage engine

```
#given variables
#r_server   = our redis server
#user_id    = the user who voted on the story
#story_id   = the story which the user voted on
if r_server.sadd("story:%s" % story_id, "userid:%s" % user_id):
    r_server.zincrby("stories:frontpage", "storyid:%s" % story_id, 1)
```

2 lines of code' This is might compact but, once we unravel it we can see how it makes sense and how powerful redis can be. Let's start with the if statement.

```
if r_server.sadd("story:%s" % story_id, "userid:%s" % user_id):
```

We know the command *sadd* already. This will add an element to a set at a key. The key in this case is

```
"story:%s" % story_id
```

If story_id is 3211, then the resulting string will be *story:3211*. This is the key in redis which contains the list of users that has voted on the story.

The value to be inserted at this key is

```
"userid:%s" % user_id
```

Just like with story, if the user_id is 9481 then the string to be inserted into the set at *story:3211* will be *user_id:9481*

Now the redis command *sadd* will return False if that element is already present in the set. So if a user has already voted on this story before we don't execute the statement under the if. But, if it is added, then we have a brand new vote and we have to increment the votes for the front page.

```
r_server.zincrby("stories:frontpage", "storyid:%s" % story_id, 1)
```

We have an ordered set at the key *stories:frontpage* and we are going to increment the element *storyid:%s* % story_id (*story:3211*) by 1.

And now we're done! We've made sure the user hasn't voted on this story before and then we've incremented the number of votes for this story on the front page!

9 Relational Database

```python
#!/usr/bin/env python
from sqlite3 import *
conn = connect('sample.db')
curs = conn.cursor()
```

This creates an empty SQLite database in the current directory, or connects to one previously created. The cursor - curs-, is simply the object instance which directs your query-like methods to the particular database.

Now let us create a table called "item":

```python
# Create Item table
curs.execute('''create table item
  (id integer primary key, itemno text unique,
        scancode text, descr text, price real)''')
```

This statement causes a table to be created with a primary key, a unique item no., and three more fields. If you were to interact directly with SQLite from the command line, it would look something like this:

```
$ sqlite3 sample.db
SQLite version 3.3.13
Enter ".help" for instructions
sqlite>  create table item (id integer primary key, itemno text unique,
   ...>          scancode text, descr text, price real);
```

As you can see, our sample Python code passes a "create table" statement, via the execute() method. The argument to the execute method is the string that we would key in interactive mode, without the semicolon.

Now let us add some data to the item table:

```python
curs.execute("insert into item values            (NULL,0001,32187645,'Milk',2.50)")
curs.execute("insert into item values            (NULL,0002,45321876,'Beer',4.50)")
curs.execute("insert into item values            (NULL,0003,18764532,'Bread',1.50)")
conn.commit()
```

Notice I gave the value NULL to the primary key. This will cause the next integer value to be assigned to the key. And the commit() method will force the changes to be updated. It would not be terribly inefficient to commit every time you make a change to the database, and might be safer.

```
sqlite> select * from item;
1|1|32187645|Milk|2.5
2|2|45321876|Beer|4.5
3|3|18764532|Bread|1.5
```

Now let us do our select statement in Python, and run it:

```
curs.execute("select * from item")
for row in curs:
    print row
(3, u'3', u'18764532', u'Bread', 1.5)
(1, u'1', u'32187645', u'Milk', 2.5)
(2, u'2', u'45321876', u'Beer', 4.5)
```

As you can see, the select causes curs to look like a list of rows of columns. The rows are tuples.

Now let us create an item/vendor file and populate it.

```
curs.execute('''create table itemvendor
   (id integer primary key, itemno text, vendor text)''')
curs.execute("insert into itemvendor values            (NULL,0001,2345)")
curs.execute("insert into itemvendor values            (NULL,0002,6789)")
curs.execute("insert into itemvendor values            (NULL,0001,0543)")
conn.commit()
```

Now, just for fun, let us do a simple join, and show the results:

```
curs.execute("select * from item LEFT OUTER JOIN itemvendor ON item.itemno = itemvendor. ↩
    itemno")
for row in curs:
    print row
(1, u'1', u'32187645', u'Milk', 2.5, 1, u'1', u'ABC Food')
(2, u'2', u'45321876', u'Beer', 4.5, 2, u'2', u'Joes Beer')
(3, u'3', u'18764532', u'Bread', 1.5, 3, u'3', u'Sysco')
```

Get a list of the interactive commands with > .help. SQLite maintains a journal. Here is the output of the interactive .dump:

```
sqlite> .dump
BEGIN TRANSACTION;
CREATE TABLE item
   (id integer primary key, itemno text unique,
         scancode text, descr text, price real );
INSERT INTO "item" VALUES(1,'1','32187645','Milk',2.5);
INSERT INTO "item" VALUES(2,'2','45321876','Beer',4.5);
INSERT INTO "item" VALUES(3,'3','18764532','Bread',1.5);
COMMIT;
```

10 Recap

- Need for persistent programming
- Files: raw form
- Key-Value Stores:

  - Extension of Dictionary
  - Extension of named data structures
- Relational Database: tables and calculus on them