CPSC 67 Lab #1: Spidering the Web

Due Wednesday, Jan. 29 (11:59pm)

The goal of this lab is to build a collection of web pages related to a set of musical artists. You will build this collection according to the following algorithm:

1. Given an artist (e.g. Jimi Hendrix), construct a query suitable for finding web pages about this artist on a search engine.
2. Given a specific search engine (e.g. Google), use your query to create a list of the URLs for the 50 most relevant web pages according to this search engine.
3. Download each of these 50 most relevant web pages to a local cache.
4. For each downloaded web page, strip the HTML markup and tokenize the words and punctuation that remain.

As a class, you will be provided with a common set of 20 musical artists that you will use in Step 1 above (provided on the class wiki). However, you will each be given a different search engine to use in Step 2. This will allow us, as a class, to compare the results of different search engines.

More importantly, by having each of you work on a different search engine, you will each be constructing a valuable piece of the larger music discovery engine that will become your final project. As such, an important component of this lab will be designing software that you and others will be comfortable maintaining later in the course, and writing software that complies with a specific API so that other students can make use of your code.

The search engine you will be using will be assigned in class at random. These search engines include Yahoo, Microsoft Live Search, Ask.com, and Cuil. A solution for using Google will be provided after the lab is completed.

While this document aims to answer many questions, it cannot possibly answer them all. Make use of the extensive help resources provided by Python or available through web searches.

1 Directory structure

Each of the components above will produce a file output that should be stored in a specific location. This will make it easier for you to share files with other students in later portions of the class.

You should create a directory for yourself in `/local` where you will store the output of this lab. This will allow you to download lots of web pages without worrying about your disk quota. Let's assume that the directory you've chosen is `/local/mystuff/`. You will want the following directory tree:

```
/local/mystuff/data/data.db        ; the database (more on this later)
/local/mystuff/data/search/        ; directory to cache part of Step 2
/local/mystuff/data/search/header/ ; directory to cache part of Step 2
/local/mystuff/data/raw/           ; directory to cache part of Step 3
/local/mystuff/data/raw/header/    ; directory to cache part of Step 3
/local/mystuff/data/clean/         ; directory to store Step 4
```

In later parts of the lab writeup, it is assumed that you will store the files in the appropriate places shown above. Note that `/local` is machine-specific, so if you start using one machine for this lab, you should use it for the entirety of the lab. Do not use `/scratch` due to the network high traffic this will generate.

2 Spidering

Although the algorithm above lists this as Step 3, from a design standpoint it makes sense for you to start here. You will write a Python class called `Spider` that must contain the following two functions as part of the API:

```
def __init__(self, cachedir, dbfile, tablename, cookiejar=None):
   """
   Initialize the Spider.
   The cachedir is the directory where cached pages will be stored.
   As shown above, this would be '/local/mystuff/data/raw/' (for
   Step 3), or '/local/mystuff/data/search/' (for Step 2).

   The dbfile is the name of the database file that will store the
   index of cached pages.  As shown above, this would be
   '/local/mystuff/data/data.db'.

   The tablename is the name of the table in the database that stores
   the index of cached pages.  For Step 3, this will be 'cached',
   and for Step 2, this will be 'searched'.  More on this later.

   The cookie jar is the name of the file that will store cookies.
   If cookiejar is None, the local file 'cookies.lwp' will be used.
   """

def fetch(url, usecache=True, offline=False):
   """
   Return the header information and web page specified by the url.

   If usecache is True, the cachedir is consulted to see if it
   already contains the information.  If the page is stored in the
   cache, it is returned.  If the page is not stored in the cache,
   the web page is downloaded and both the header and the web page are
   stored in the cache.

   If usecache is False, the cache is not consulted or written to.

   If offline is True, results will only be returned if the page has
   previously been cached.  If offline is True and usecache is False,
   no results will be returned.
   """

def cacheID(url):
   """
   Returns the integer ID of URL in the cache, or None if the page is
   not in the cache.
   """
```

The `Spider` class can then be used as follows to fetch a web page:

```
def main():
   url = "http://www.cs.swarthmore.edu"
   dir = '/local/mystuff/data'
   spider = Spider('%s/raw/' % dir, '%s/data.db' % dir, 'cached')
   (header, page) = spider.fetch(url)
```

2.1 User-agent and Cookies

Python's `urllib` module provides some basic functionality required to download web pages. Unfortunately, there are two issues that `urllib` does not handle very well. The first relates to the "User-agent". When Firefox (or any other web browser) requests a page from a web server, the browser identifies itself to the web server. This identification is known as the browser's User-agent. For example, the User-agent for version 3.05 of Firefox is `Firefox/3.05`. In contrast, Python's `urllib` module identifies itself as `Python-urllib/1.17`[1]. The problem, as far as spidering is concerned, is that some web sites return different pages depending on the User-agent. Since we'd

[1]As of the writing of this document. You can find the User-agent of your version of by first importing `urllib` and then printing `urllib.URLopener.version`.

like to download the version of the page we would see if we were using a standard web browser, we'd like to change the User-agent of `urllib` to something like `Firefox/3.05`.

Though it is possible to fix the User-agent using `urllib`, the solution is not particularly nice, and it doesn't address the second problem: cookies. Cookies are small pieces of information sent by the web server to your browser to perform various tasks, such as session tracking. For example, many web sites (such as cnn.com) use cookies to remember if you're interested in the U.S. version of the page or the international version. The web site may ask use this question once and then store your choice in a cookie. Some web browsers (including Firefox, if you set this option) will refuse cookies that are sent by a server. However, a web server can refuse to send you the web page if you do not accept the server's cookie. This is a problem because `urllib` does not accept server cookies.

The solution is to use two other Python modules in combination with `urllib` to fetch web pages: `urllib2` and `cookielib`. The `urllib2` module will allow us to easily set the User-agent, and the `cookielib` module will allow us to store cookies. In order to store cookies, we need a file (commonly called a "cookie jar") to store the cookies in. Since cookies are just text, the cookie jar is just a text file.[2] The following code sets up the cookie jar and instructs `urllib2` to use the cookie jar:

```
import os
import urllib2
import cookiejar

cookiefile = 'cookies.lwp'            #the name of the cookie file
cookiejar = cookielib.LWPCookieJar()  #create an empty cookie jar
if os.path.isfile(cookiefile):        #if the cookie jar file exists
    cookiejar.load(cookiefile)        #load cookies into cookiejar

#tell urllib2 to use cookiejar when fetching web pages:
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cookiejar))
urllib2.install_opener(opener)

...                                   #fetch web pages

cookiejar.save(cookiefile)            #save cookies into cookie file
```

In the snippet above, you should note that you only need to setup the cookie jar once, regardless of the number of web pages you need to fetch.

Now that the `urllib2` module knows what to do with cookies it receives, we can fetch pages using `urllib2`. However, before we do so, we should also fix the User-agent. We do this as follows:

```
#include User-agent information in your request for the url:
request = urllib2.Request(url, headers={'User-agent':'Firefox/3.05'})

response = urllib2.urlopen(request)     #get the server's response
page = response.read()                  #save the page (a string)
```

As written above, the two snippets comprise nearly everything you need to do to read web pages with the User-agent set to `Firefox/3.05` and accept server cookies. There are two small additions that you should make in order to handle web servers that do not respond (because they are down or have moved) or other errors related to fetching web pages. First, you should set a maximum amount of time that you will give a web server to respond:

```
import socket

socket.setdefaulttimeout(10)   #10 seconds
```

Second, you should catch the `URLError` exception that is raised whenever the web server returns an error or the socket timeout has been reached. Adapting the code above:

[2]We will store cookies in `libwww-perl` format which makes the cookies human-readable. By convention, the extension for `libwww-perl` cookie jars is `.lwp`, so the default name for your cookie jar is ``` ``cookies.lwp'' ```.

```
header = {'User-agent' : 'Firefox/3.05'}
request = urllib2.Request(url, None, header)
try:
    response = urllib2.urlopen(request)
    page = response.read()
except urllib2.URLError, e:    #socket timeout or web server error
    print 'Error: \%s' \% (e)
```

2.2 Server header

One final piece of information: you will want to save the header returned by the server in addition to the page itself. The header might look something like this:

```
Date: Thu, 15 Jan 2009 18:19:35 GMT
Server: Apache/2.2.3 (Debian)
Last-Modified: Tue, 02 Dec 2008 14:34:03 GMT
ETag: "708206-17d0-380c4cc0"
Accept-Ranges: bytes
Content-Length: 6096
Connection: close
Content-Type: text/html; charset=ISO-8859-1
```

This includes information such as the time the web page was fetched, the last time the page was modified, the length of the page, the type of page (`text/html`) and the character set used to encode the page (`ISO-8859-1`). This information will be handy later, so be sure to save it as described in Section 2.3. To retrieve the server header, you can use the following line (that can be inserted immediately after `page = response.read()`:

```
header = str(response.info())
```

2.3 Caching results

Once a web page has been downloaded, you would like to avoid downloading it a second time by storing a copy of the page in a local cache. When you download a page, you will save a copy of the page to the directory specified by the `cachedir` directory. For example, you might download `www.cs.swarthmore.edu` and cache the page in a file called `000001.html`. On subsequent requests to download `www.cs.swarthmore.edu`, you will instead return the page stored in `000001.html`.

In addition to saving the downloaded web page, you will also want to save the downloaded server header (Section 2.2). If `cachedir/raw/000001.html` is the cached page, the cached header will be `cachedir/raw/header/000001`.

You will save the mapping of the filename to the URL stored in the file in an **sqlite3** database. To manipulate data in the sqlite3 database, you need to use SQL, a language used in relational databases. For the purposes of this assignment, the SQL you need to learn is very minimal and will be shown in a series of examples below.

2.3.1 Connecting to a database

To use an sqlite3 database, you must import the sqlite3 modules and then "connect" to the file that stores your database. Once you have the connection established, you need a "cursor" that can execute SQL queries. Using the example file name given above:

```
import sqlite3

connection = sqlite3.connect('/local/mystuff/data/data.db')
cursor = connection.cursor()
```

2.3.2 Creating a table

We desire a table that relates integers to URLs. In SQL, this means we'll need a **table** with two columns: one with type `INTEGER` and one with type `VARCHAR`. The `INTEGER` will be unique (that is, no two rows in the table will have the same integer) and so we designate it to be our `PRIMARY KEY`. The `VARCHAR` column stores a variable-length string. For example, two rows of the table might look like this:

INTEGER id	VARCHAR url
1	http://web.cs.swarthmore.edu/~turnbull/cs67/s09/
2	http://en.wikipedia.org/wiki/SQL

The SQL needed to create this table is:

```
CREATE TABLE IF NOT EXISTS cached (
 id  INTEGER PRIMARY KEY,
 url VARCHAR
);
```

This creates a table called `cached` with two columns, `id` and `url`, unless the table already exists, in which case nothing happens.

To execute this SQL query, we use the `cursor` we created above. The triple quotes in the example below are used by Python to indicate a multi-line string:

```
cursor.execute("""CREATE TABLE IF NOT EXISTS cached (
 id  INTEGER PRIMARY KEY,
 url VARCHAR
);""")
```

After executing any SQL, you need to commit the changes to the database. In Python, you say:

```
connection.commit()
```

2.3.3 insert

To insert rows into the database, we can fully specify the `id` and the `url` that we want to insert. For example, we could say:

```
INSERT INTO cached (id, url) VALUES (1, "http://www.cnn.com")
```

However, usually we will want to insert a `url` into the table and have SQL tell us the unique `id` that was assigned to it. We do this by inserting only the `url` into the table, and then querying the value of the last row entered into the table (shown in Python):

```
cursor.execute('INSERT INTO cached (url) VALUES ("http://www.cnn.com")')
lastrow = cursor.lastrowid
connection.commit()
```

Here, `lastrow` will be equal to the `id` of `"http://www.cnn.com"`.

2.3.4 select

Given a `url`, you may want to know the associated `id`:

```
url = "http://www.cnn.com"
res = cursor.execute('SELECT id FROM cached WHERE url="%s"' % url)
resultList = res.fetchall()
# no need to commit since this was only a read operation
```

The `resultList` variable stores a list containing appropriate matches. If the table contained a row `(3, "http://www.cnn.com")`, then `resultList` would be `[(3, )]`. Here, the `resultList` is a list of 1-element tuples containing the `id` of the matching row.

2.3.5 Notes

When you are attempting to retrieve a url, you will first want to check if the url is in the database (using a `SELECT` statement). If so, you'll want to retrieve the appropriate file from disk. When you download a new url that isn't in the cache, you'll want to insert the url into the database using the `INSERT` statement. To turn an `id` into a filename, you will want to do something like this:

```
filename = '%06d.html' % (urlid)
```

The name of the table storing the cache from Step 3 should be `cached` (as in the above examples) and the table storing the cache from Step 2 should be `searched`.

For debugging or other purposes, it might be helpful for you to write a routine that deletes a single row. The following deletes the row whose `id` is 3:

```
DELETE FROM cached WHERE id=3
```

3 Searching

Once you have completed Spidering (Section 2), you can return to Steps 1 and 2 from our initial algorithm. This part of the assignment is less structured than the previous section because you have each been given a specific search engine to work with. You will all likely face the same challenges, but your specific solution to these challenges will be different than the specific solution of someone working on a different search engine. None of you have been assigned to using Google as your search engine; therefore, the solutions shown below should only be used as examples: it will not suffice for you to use the solution below to complete this section.

Below, Google will be used as the running example. You should substitute your search engine (and its peculiarities) for the examples given here. The general strategy is for you to use the search engine you have been assigned (here, Google) to download search results for a particular query.

Your API should consist of a class named for your search engine, and it should include a method called `search` as described below:

```
class Google(object):
  def __init__(self, spider):
     """
     Initialize the Google search engine class.

     spider is a Spider object which, according to the examples shown
     so far, should have its cachedir set to '/local/mystuff/data/search',
     its dbfile set to '/local/mystuff/data/data.db', and its db table
     set to 'searched'
     """

  def search(self, query, hits=10):
     """
     Given a search query and the number of hits to be returned,
     return a list of links proposed by the Google search engine.
     """
```

You are encouraged to create additional functions to assist in performing the search function, but those functions are not specified.

Given the API above, the following code should return a list of the top 50 URLs for the query `"Jimi Hendrix" music`:

```
query = '"Jimi Hendrix" music'
dir = '/local/mystuff/data'
```

```
spider = Spider('%s/search/' % dir, '%s/data.db' % dir, 'searched')
google = Google(spider)
urlList = google.search(query, 50)
```

3.1 Query formation

Begin by visiting the web site for the search engine you have been assigned and do some searches. Examine the URLs generated by the search engine in response to your searches. This should give you insight into how you should construct suitable URLs for your queries. For example, in making the query `"Jimi Hendrix" music` to Google, the URL generated is:

```
http://www.google.com/search?hl=en&q="Jimi+Hendrix"
```

Trying another query, `artist "Snow Patrol"`, the resulting URL is:

```
http://www.google.com/search?hl=en&q=artist+"Snow+Patrol"
```

You should now be able to derive the URL for an arbitrary query. You may find that the documentation for `urllib` is helpful with constructing these URLs.

3.2 Fetching the search result

Since you have already completed the `Spider` class, fetching the result for a query is straightforward.

3.3 Parsing the search results

Now that you have downloaded the search page, you need to be able to extract the 50 most relevant pages. If you view the downloaded page, you will see that it is full lots of irrelevant junk. However, for a particular search engine, there is a particular template that the search engine uses to display the data, and this template can be exploited to extract the relevant information. For example, looking at the Google page results in Firefox, you can see that top result is from `last.fm`. Opening the cached page (since your Spider cached the search result on disk), you can look for the `last.fm` link in source of the HTML. Below is a snippet of text around this result:

```
<h3 class=r><a href="http://www.last.fm/music/Jimi+Hendrix" class=l
onmousedown="return clk(this.href,'','','res','1','')"><em>Jimi Hendrix</em>
 --- Listen free and discover <em>music</em> at Last.fm</a></h3>
```

From this, you might think that the search results are always stored in this template. However, you will want to look at other results to be sure that this extracts all results (we say that it has 100% *recall*), and that it doesn't extract any links you didn't want (we say that it has 100% *precision*). After some fiddling around, you might find that a pattern similar to this appropriate:

```
<h3 class=r><a href="link" ......</a></h3>
```

The easiest way to extract this information is to use Python's regular expression library, `re`. It is not necessary to know how to use regular expressions to extract the necessary data: various string functions can get you the data you need. It will be up to you to figure out how to extract the link information for your specific web site.

You need to download the top 50 most relevant links according to your search engine, but this query to Google only gave you the top 10. However, using the same technique as above, you can find out how Google encodes the next page of results. You might find that Google encodes the results 11-20 in the following URL:

```
<a href="/search?hl=en&amp;q=%22Jimi+Hendrix%22+music&amp;start=10&amp;sa=N">
```

And results 21-30 in this URL:

```
<a href="/search?hl=en&amp;q=%22Jimi+Hendrix%22+music&amp;start=20&amp;sa=N">
```

This can help you build queries for retrieving 50 total links.

4 Cleaning

You have now completed Steps 1, 2, and 3 from the algorithm listed at the start of the assignment, leaving only Step 4: extracting the words from the web page. Since we would like a common standard for extracting the words, and since this functionality has already been built into lynx[3] and NLTK[4], this part of the assignment is trivial. Given the contents of a web page, stored as a string in the variable `page`, a list of the words for the page will be obtained as follows:

```
import nltk

(header, page) = spider.fetch(url)
lynxed = lynx_clean(page)     #see wiki for source code for this function
wordlist = nltk.wordpunct_tokenize(lynxed)
```

Once you have a cleaned HTML page, you should write out the words, one per line, to a file. If the HTML page is cached as `000001.html`, you would save this to the `clean/` directory (as specified in Section 1) under `000001.txt`.

5 Putting it all together

The following `main` function should suffice to execute the entirety of the project (with appropriate modifications for your scratch directory and search engine, and assuming you write the `writeClean` function shown below). This code is also available on the class wiki.

```
def main():
    basedir = '/local/mystuff/data'
    hits = 50
    srchSpider = Spider('%s/search/' % basedir, '%s/data.db' % basedir, 'searched')
    engine = Google(srchSpider)

    fetchSpider = Spider('%s/raw/' % basedir, '%s/data.db' % basedir, 'cached')
    for artist in open('artists.txt'):
        print 'Processing %s' % (artist.rstrip())
        query = '"%s" music' % (artist.rstrip())
        urlList = engine.search(query, hits)
        for link in urlList:
            print "Fetching %s" % link
            (header, page) = fetchSpider.fetch(link)
            lynxed = lynx_clean(link, page)
            wordlist = nltk.wordpunct_tokenize(lynxed)
            writeClean('%s/clean' % basedir,wordlist,fetchSpider.cacheID(link))
```

6 Results

Once you have successfully downloaded 50 pages each for 20 different artists, answer the following question, reporting results for all 20 artists cumulatively.

1. For each artist, what percentage of the pages have the following keywords: 'alternative', 'indie, 'rock', 'country', 'jazz'.

To count the occurrences of 'jazz' (for example), `cd` into the `raw/` directory and run:

```
grep -lw jazz *.html | wc -l
```

Report your results on the wiki as described in class. If you cannot register for the wiki, please send us an email.

[3]http://lynx.isc.org/
[4]http://www.nltk.org