i206 Spring 2013: Assignment 7

Name_____

Assignment: Write a Mini Search Engine

In this assignment you will put together some of the programming skills you've acquired to date to build a mini search engine. We'll be crawling and indexing a small set of simple web pages starting out from the ischool course pages. Before starting on this, I provide three exercises below to get you familiar with some python packages you'll need for this assignment. These are followed by the assignment questions.

Please complete Exercises 1-3 by Thursday March 20th, Questions 1-2 by Thursday April 4th, and Question 3 and any Extra Credit work by Thursday April 11th.

Exercise 1: Get Familiar with HTTP

In this exercise you will familiarize yourself with HTTP requests and responses. For your information, this Wikipedia page defines the components of message headers for HTTP requests and responses: https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

We will use the python urllib package for making http connections. This is built into the python 3 releases. (Note that you may see references to urllib2 on the web; that was present in python 2.7 but it has been rolled into urllib for python 3.) You'll want to use these packages:

- http://docs.python.org/py3k/library/urllib.request.html
- http://docs.python.org/py3k/library/urllib.parse.html
- http://docs.python.org/py3k/library/urllib.robotparser.html

I have written some starter code for you to show you how to request a response from a web server and look at the header for that response:

```
# use urllib to get the header information for a web page at location url
import urllib.request
import urllib.parse

def RequestResponse(url):
	try:
		response = urllib.request.urlopen(url)
		return response
	except urllib.error.URLError as err:
		print("Error opening url {} .\nError is: {}".format(url, err))
		return False

def ExamineResponseHeader(url):
	response = RequestResponse(url)
	if not response: return
	print("response url: {}".format(response.geturl()))	# check the url
	print("response headers: {}".format(response.getheaders()))
```

In web pages, a url can be fully specified (starts with http://) or it can be a relative link. In one case the relative link starts with "/" meaning that the domain name or host should be the prefix, or without a slash meaning that the file structure beneath the current url should be accessed. In the case where the relative path begins with "/", you can use the urlparse and urljoin functions to create the appropriate fully specified link as follows, assuming you checked the response url as in the code shown in ExamineResponseHeader.

```
        if onpage_url[0] == '/':
                parsed_url = urllib.parse.urlparse(response_url)
                hostname = "http://" + parsed_url.netloc
                new_full_url = urllib.parse.urljoin(hostname, onpage_url)
```

Exercise: Play around a bit with the header code above, running it or variations of it on some sample web pages to see what the headers look like.

Exercise 2: Get Familiar with an HTML Parser

We will also be using an excellent python software package for parsing HTML called BeautifulSoup. It's very easy to use (unlike the built in python HTMLparser), especially since you now know how to program tree structures, recursive structures, and regular expressions. Instructions about how to install it and a tutorial for how to use it can be found here: http://www.crummy.com/software/BeautifulSoup/bs4/doc/

I tested it with python 3.2 and it works fine if you use the latest version. The web page has installation instructions. I was able to install it without any problem on my mac by running going to the directory into which I had uncompressed BeautifulSoup and running this command as noted in the instructions above:

```
        %python setup.py install
```

Here is some sample code to get you familiar with BeautifulSoup, using the http code shown above:

```
# extract all the links ('a' tags) from a web page; print first 10
from bs4 import BeautifulSoup
def GetLinksFromPage(url):
        response = RequestResponse(url)
        if not response: return
        soup = BeautifulSoup(response.readall())
        links = soup.find_all('a')
        for link in links[0:10]
                print("Full link: {}\nAnchor text: {}".format(link, link.text))
```

Exercise: play around with the BeautifulSoup navigation and search functions on some web pages.

Exercise 3: Learn about Robots.txt

When doing web crawling it is important to be polite by checking the robots.txt file to see which web pages you are allowed to crawl and which you are not. You can view the robots.txt file for the ischool web pages at: http://www.ischool.berkeley.edu/robots.txt Below is python code that checks if a url within the ischool domain is legitimate to crawl:

```
# read the robots.txt file for this host
import urllib.robotparser
def SetRobotsChecker(robot_url):
        rp = urllib.robotparser.RobotFileParser()
        rp.set_url(robot_url)
        rp.read()
        return rp
```

```
# returns True if it is ok to fetch this url, else False
def OKToCrawl(rp, url):
      return rp.can_fetch("*", url)

# sample usage:
>>> rp = SetRobotsChecker("http://www.ischool.berkeley.edu/robots.txt")
>>> OkToCrawl(rp, "http://ischool.berkeley.edu/index.html")
True
>>> OkToCrawl(rp, "http://ischool.berkeley.edu/misc/index.html")
False
```

You also need to make sure that your crawler does not swamp the web server. For this reason, you should make your crawler program pause for 1 second between each web page access. You can do this with the time package

```
# make program sleep for 1 second
>>> import time
>>> time.sleep(1)
```

Exercise: test out this code on a few urls.

Question 1: Write a Mini Web Crawler

Write a mini web crawler. You should use the following tools:

- The python url libraries to access web pages
- BeautifulSoup to parse the retrieved web pages
- A queue or a stack to keep track of which pages to crawl

Start with this url: http://courses.ischool.berkeley.edu Note that this is an alias for the actual underlying hostname; that's why checking the response url can be useful.

For this phase, just visit the pages; we'll do indexing in a following question. Your crawler should have the following properties:

- It should be polite, meaning check urls against the robots.txt file and pausing for 1 second between each web page access.
- It should avoid loops. A web crawler can get into an endless cycle if, say, page A links to B links to C links to A. To avoid this problem, be sure to write code for your web crawler that does not queue up a link if it has been seen before. It's best to use fully specified urls for this purpose.
- It only crawls web pages within the ischool.berkeley.edu domain. It should check to be sure that no pages it accesses are outside this domain.
- It should put new urls on the stack or queue and after accessing 40 unique web pages it should stop. You can choose the order in which the pages are accessed.
- If a link starts with a '/' then use the relative link resolver described above. If it is fully specified you can use that link directly. If it is any other kind of relative link, don't crawl it.

i206 Spring 2013: Assignment 7

Question 2: Write a Simple Indexer

Modify your web crawler of Question 1 so that as it encounters web pages, it records important information about those pages that would be useful to index and/or display in search results. This should include the following at minimum, although you can include more if you like:

- A unique ID for this web page
- The title of the web page (you need this to display in the search results)
- The url of the web page (to display in search results)
- The visible text of the web page (for searching over). You'll want to be able to show snippets of text in the search results.

The unique ID is for your programming convenience. From these you'll need to create an inverted index of the visible text and text of the page titles. (You don't have to capture every bit of visible text from every page, but get the bulk of it.) That is, for each word, you'll need to record which web pages you found that word in. You may want to write the results out to disk after you've processed all 40 web pages, to be read in by the search engine part of the code. You can write it out in whatever format is easy to read in again. Or you can keep it all in memory without writing it out.

Question 3: Write a Simple Query Processor and Results Display

Write a simple search engine that takes as input a query consisting of one or more words and produces a simple results lists in a web page. The results list should show:

- What the query was
- For each search hit:
    - The title of the page, hyperlinked to the appropriate url
    - A snippet of text from the web page. This does not have to contain query words; you can just extract some text from near the beginning of the page if you want and show this text every time this document is in the search results

The query should be an implicit conjunction (AND) over all of the query terms. That is, for a page to be relevant, all the relevant words must occur somewhere within the web page, but the words do not have to all be present in the snippet that you show. You can use whatever ranking function you like. You only have to return up to 10 results although you can show more if you like. You do not have to support phrase queries (adjacent words in quotes) or Boolean queries.

You do not need to create a web page that accepts a query in a query box; instead you can just write the query yourself, pass the query in to your function, compute the search results, construct a search results page, and display that page. Python provides you with a very easy way to display a webpage, with the webbrowser package. If you run the following code, a web browser opens up for you showing the specified page:

```
>>> import webbrowser
>>> webbrowser.open("http://ischool.berkeley.edu")
```

If you write your search results web page out to a local file in your computer, you can use the webbrowser command to display it as follows (on a mac anyhow):

```
>> webbrowser.open("file:///Users/name/Documents/search_results.html")
```

The web page does not have to be beautiful but you should use a simple CSS file to style it. Demonstrate the results on at least 3 queries; these queries should contain 1, 2, and 3 search terms in them. Be sure to handle the case where there are no matches.

i206 Spring 2013: Assignment 7

Bonus Points

There are many ways to earn bonus points in this assignment. In each case, be sure to clearly explain what you did in your writeup as well as show the code and its output.

- Make your web crawler more sophisticated in some manner; be sure to describe what you did.
- Use an appropriate disk-based data structure for storing and retrieving the crawled information (instead of just writing the results out to disk and reading them in again) such as a b-tree or a disk-based hash table. You need to write the code to use these yourself but can use existing software packages for the data structures.
- Show real search snippets containing all of the search words, with those words bolded or highlighted in some other way. If this requires a long stretch of text then use ellipses to shorten it. You only need to show one matching string of text and you can choose which one to show.
- Include stemming in your search algorithm (for instance, searching on “course” returns hits on “courses” as well; the stemming can’t be arbitrary though. You can use the Porter stemmer if you like.
- Support wildcards in the user query; for instance, searching on “cour*” matches “courses” and “course”.
- Integrate anchor text from linked pages into your ranking algorithm. This means that if a query word Q falls within anchor text in page B that points to page A, then you can retrieve page A as matching Q even if A does not contain the word Q explicitly. You’ll need to indicate this in the search results in some manner.
- Integrate a more sophisticated ranking algorithm into your code; you may make use of existing algorithms for this but you need to write the code yourself rather than get it from elsewhere. This will probably require you to create more complex data structures than you used for question 3.
- Build a standard web-based interface for the search engine using web forms and CGI or some other means to create a search form and search results cycle so that the user can type in a query into an entry form box, see search results, and repeat.