Web Clients and Crawlers

1 Web Clients
   - alternatives to web browsers
   - opening a web page and copying its content

2 Scanning files
   - looking for strings between double quotes
   - parsing URLs for the server location

3 Web Crawlers
   - making requests recursively
   - incremental development, modular design of code

MCS 275 Lecture 34
Programming Tools and File Management
Jan Verschelde, 5 April 2017
Web Clients and Crawlers

1. **Web Clients**
   - alternatives to web browsers
   - opening a web page and copying its content

2. **Scanning files**
   - looking for strings between double quotes
   - parsing URLs for the server location

3. **Web Crawlers**
   - making requests recursively
   - incremental development, modular design of code
Web Clients
alternatives to web browsers

We do not really need Apache to host a web service. Recall testing ourwebserver.py last lecture. → the client is a browser, e.g.: Netscape, Firefox, ...

But we can browse the web using scripts.

Why do we want to do this?

1. more efficient: no overhead from GUI
2. in control: request only what we need → update most recent information
3. crawl the web: request recursively → operate like a search engine

How?

use urllib and urlparse modules
$ python3 forecast.py
opening http://tgftp.nws.noaa.gov/data/forecasts/state/il/ilz013.txt ...

<table>
<thead>
<tr>
<th>Today</th>
<th>Sat</th>
<th>Sun</th>
<th>Mon</th>
<th>Tue</th>
<th>Wed</th>
<th>Thu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apr 07</td>
<td>Apr 08</td>
<td>Apr 09</td>
<td>Apr 10</td>
<td>Apr 11</td>
<td>Apr 12</td>
<td>Apr 13</td>
</tr>
</tbody>
</table>

Chicago Downtown

<table>
<thead>
<tr>
<th>SUNNY</th>
<th>SUNNY</th>
<th>MOCLDY</th>
<th>PTCLDY</th>
<th>PTCLDY</th>
<th>SUNNY</th>
<th>MOCLDY</th>
</tr>
</thead>
<tbody>
<tr>
<td>/50</td>
<td>38/67</td>
<td>54/72</td>
<td>56/68</td>
<td>45/54</td>
<td>40/48</td>
<td>42/59</td>
</tr>
<tr>
<td>/00</td>
<td>00/00</td>
<td>00/10</td>
<td>30/30</td>
<td>30/10</td>
<td>10/00</td>
<td>20/40</td>
</tr>
</tbody>
</table>

Chicago O’hare

<table>
<thead>
<tr>
<th>SUNNY</th>
<th>SUNNY</th>
<th>MOCLDY</th>
<th>PTCLDY</th>
<th>PTCLDY</th>
<th>SUNNY</th>
<th>MOCLDY</th>
</tr>
</thead>
<tbody>
<tr>
<td>/55</td>
<td>37/67</td>
<td>54/73</td>
<td>57/69</td>
<td>44/56</td>
<td>39/56</td>
<td>41/60</td>
</tr>
<tr>
<td>/00</td>
<td>00/00</td>
<td>10/10</td>
<td>40/40</td>
<td>30/10</td>
<td>10/00</td>
<td>30/40</td>
</tr>
</tbody>
</table>
```python
from urllib.request import urlopen
HOST = 'http://tgftp.nws.noaa.gov'
FCST = '/data/forecasts/state'
URL = HOST + FCST + '/il/ilz013.txt'
print('opening ' + URL + ' ...
')
DATA = urlopen(URL)
while True:
    LINE = DATA.readline().decode()
    if LINE == '':
        break
    L = LINE.split(' ')
    if 'FCST' in L:
        LINE = DATA.readline().decode()
        print(LINE + DATA.readline().decode())
    if 'Chicago' in L:
        LINE = LINE + DATA.readline().decode()
        LINE = LINE + DATA.readline().decode()
        print(LINE + DATA.readline().decode())
```
Web Clients and Crawlers

1. Web Clients
   - alternatives to web browsers
   - opening a web page and copying its content

2. Scanning files
   - looking for strings between double quotes
   - parsing URLs for the server location

3. Web Crawlers
   - making requests recursively
   - incremental development, modular design of code
copying a web page to a file

Syntax:

```
urlretrieve( < URL >, < file name > )
```

Example:

```
from urllib.request import urlretrieve
urlretrieve('http://www.python.org','wpt.html')
```

Opening a web page with `urllib.request.urlopen`:

```
from urllib.request import urlopen
< object like file > = urlopen( < URL > )
data = < object like file >.read( < size > ).decode()
< object like file >.close()
```

→ process web pages like we handle files
def main():
    ""
    Prompts the user for a web page, a file name, and then starts copying.
    ""
    from urllib.request import urlopen
    print('making a local copy of a web page')
    url = input('Give URL : ')
    try:
        page = urlopen(url)
    except:
        print('Could not open the page.')
    return
    name = input('Give file name : ')
copypage(page, name)
a function to copy a web page to a file

```python
def copypage(page, file):
    
    """
    Given the URL for the web page, 
a copy of its contents is written to file. 
Both url and file are strings.
    """

    copyfile = open(file, 'w')
    while True:
        try:
            data = page.read(80).decode()
        except:
            print('Could not decode data."
            break
        if data == '':
            break
        copyfile.write(data)
    page.close()
    copyfile.close()
```
Web Clients and Crawlers

1. Web Clients
   - alternatives to web browsers
   - opening a web page and copying its content

2. Scanning files
   - looking for strings between double quotes
   - parsing URLs for the server location

3. Web Crawlers
   - making requests recursively
   - incremental development, modular design of code
scanning HTML files

Applications to scan an HTML file:
1. search for particular information,
2. navigate to where the page refers to.

Example (1): download all `.py` files from
http://www.math.uic.edu/~jan/mcs275/main.html

Example (2): retrieve all URLs the page
http://www.python.org refers to.

What is common between these two examples:

`.py` files and URLs appear between " and "

→ scan for all strings between double quotes
Problem Statement
scanning for double quoted strings

Input: a file, or object like a file.
Output: list of all strings between double quotes.

Recall that we read files with fixed size buffer:

```
..."..."..."..."...".....""".."""..""".....
```

For double quoted strings which run across two buffers we need another buffer.

Two buffers: one for reading strings from file,
one for buffering double quoted string.

→ Two functions:
1. read buffered data from file,
2. scan the data buffer for double quoted strings.
def quoted_strings(file):
    """
    Given a file object, this function scans the file and returns a list of all strings on the file enclosed between double quotes.
    """
    result = []
    buffer = '
    while True:
        data = file.read(80)
        if data == '':
            break
        (result, buffer) \
            = update_qstrings(result, buffer, data)
    return result
processing the buffers

We perform a buffered reading of the file. In `acc` we store the double quoted strings. In `buf` we buffer the double quoted strings.

In the example below, every `.` represents a character.

```
...."...."  acc = [], buf = "...."
."...."."  acc = ['....', ',', '.'], buf = 
........  acc = ['....', ',', '.'], buf = 
."...."."  acc = ['....', ',', '.', ',', '..'], buf = ""
.."...."..  acc = ['....', ',', '.', ',', '..', ',', '.', ',', '..'], buf = ""
```
prototype for \texttt{update\_qstrings}

In \texttt{quoted\_strings} we make the following call:

\begin{verbatim}
(result, buffer) \=
update_qstrings(result, buffer, data)
\end{verbatim}

The specification is defined as

\begin{verbatim}
def update_qstrings(acc, buf, data):
    """
    acc is a list of double quoted strings, 
    buf buffers a double quoted string, and 
    data is the string to be processed. 
    Returns an updated (acc, buf). 
    """
\end{verbatim}
def update_qstrings(acc, buf, data):
    newbuf = buf
    for char in data:
        if newbuf == '':
            if char == '"':
                newbuf = 'o'  # 'o' is for 'opened'
            else:
                if char != '"':
                    newbuf += char
                else:  # do not store 'o'
                    acc.append(newbuf[1:len(newbuf)])
                    newbuf = ''
        else:
            if char != '\"':
                newbuf += char
            else:  # do not store 'o'
                acc.append(newbuf[1:len(newbuf)])
                newbuf = ''
    return (acc, newbuf)
def main():
    """
    Prompts the user for a file name and scans the file for double quoted strings.
    """
    print('getting double quoted strings')
    name = input('Give a file name : ')
    file = open(name, 'r')
    strs = quoted_strings(file)
    print(strs)
    file.close()
Web Clients and Crawlers

1. Web Clients
   - alternatives to web browsers
   - opening a web page and copying its content

2. Scanning files
   - looking for strings between double quotes
   - parsing URLs for the server location

3. Web Crawlers
   - making requests recursively
   - incremental development, modular design of code
scanning web pages for URLs

Recall the second example application:
list all URLs referred to at http://www.python.org

def main():
    
    """
    Prompts the user for a web page, and prints all URLs this page refers to.
    """

    print('listing reachable locations')
    page = input('Give URL : ')
    links = httplinks(page)
    print('found %d HTTP links' % len(links))
    show_locations(links)
def httpfilter(strings):
    """ Returns from the list strings only those strings which begin with http. """
    result = []
    for name in strings:
        if len(name) > 4:
            if name[0:4] == 'http':
                result.append(name)
    return result
scanning a HTML file for HTTP strings

The first part, opening the URL:

def httplinks(url):
    
    """
    Given the URL for the web page, returns the list of all http strings.
    """
    from urllib.request import urlopen
    try:
        print('opening ' + url + ' ...')
        page = urlopen(url)
    except:
        print('opening ' + url + ' failed')
    return []
httplinks() continued ...

The second part, scanning for double quoted strings:

\[
\begin{align*}
\text{(result, buf)} & = ([] , '') \\
\text{while True:} & \\
\text{try:} & \\
\quad & \text{data} = \text{page.read(80).decode()}
\end{align*}
\]

except:
    print(‘could not decode data’)  
    break

if data == ‘’:
    break

\[
\begin{align*}
\text{(result, buf)} & \backslash \\
\quad & = \text{update_qstrings(result, buf, data)} \\
\text{result} & = \text{httpfilter(result)} \\
\text{page.close} & ()
\end{align*}
\]

return result
showing only server locations

An URL consists of 6 parts

protocol://location/path:parameters?query#frag

**Given URL** u, the `urlparse(u)` returns a 6-tuple.

```python
def show_locations(links):
    
    Shows the locations of the URLs in links.
    
    from urllib.parse import urlparse
    for url in links:
        pieces = urlparse(url)
        print(pieces[1])
```
Web Clients and Crawlers

1. Web Clients
   - alternatives to web browsers
   - opening a web page and copying its content

2. Scanning files
   - looking for strings between double quotes
   - parsing URLs for the server location

3. Web Crawlers
   - making requests recursively
   - incremental development, modular design of code
web crawlers – making requests recursively

Scanning HTML files and browsing:
1. given a URL, open a web page,
2. compute the list of all URLs in the page,
3. for all URLs in the list do:
   1. open the web page defined by location of URL,
   2. compute the list of all URLs on that page.
→ continue recursively, *crawling* the web

Things to consider:
1. remove duplicates from list of URLs,
2. do not turn back to pages visited before,
3. limit the levels of recursion,
4. some links will not work.

Similar to finding a path in a maze, but now we are interested in all intermediate nodes along the path.
running the crawler

$ python webcrawler.py
crawling the web ...
Give URL : http://www.uic.edu
give maximal depth : 2
opening http://www.uic.edu ...
opening http://maps.uic.edu ...
could not decode data
opening http://maps.google.com ...
opening http://maps.googleapis.com ...
opening http://maps.googleapis.com failed
opening http://fimweb.fim.uic.edu ...

.. it takes a while ..

total #locations : 3954

in 2010: total #locations : 538
Web Clients and Crawlers

1. Web Clients
   - alternatives to web browsers
   - opening a web page and copying its content

2. Scanning files
   - looking for strings between double quotes
   - parsing URLs for the server location

3. Web Crawlers
   - making requests recursively
   - incremental development, modular design of code
modular design of a web crawler

from scanhttplinks import httpplinks

Still left to write:

1. management of list of server locations,
2. recursive function to crawl the web.
def new_locations(links, visited):
    """
    Given the list links of new URLs and the list of already visited locations, returns the list of new locations, locations not yet visited earlier.
    """

    from urllib.parse import urlparse
    result = []
    for url in links:
        parsed = urlparse(url)
        loc = parsed[1]
        if loc not in visited:
            result.append(loc)
    return result
unparsing URLs

Recall that we only store the server locations.

To open a web page we also need to specify the protocol.

We apply `urlparse.urlunparse`

```python
>>> from urlparse import urlunparse
>>> urlunparse(('http','www.python.org','','','',''))
'http://www.python.org'
```

We must provide a 6-tuple as argument ...
def main():
    
    Prompts the user for a web page, and prints all URLs this page refers to.

    print('crawling the web ...')
    page = input('Give URL : ')
    depth = int(input('give maximal depth : '))
    locations = crawler(page, depth, [])
    print('reachable locations :', locations)
    print('total #locations :', len(locations))
def crawler(url, k, visited):
    
    Returns the list visited updated with the list of locations reachable from the given url using at most k steps.
    
    from urllib.parse import urlunparse
    links = http_links(url)
    newlinks = new_locations(links, visited)
    result = visited + newlinks
    if k == 0:
        return result
    else:
        for loc in newlinks:
            url = urlunparse(('http', loc, '', '', '', ''))
            result = crawler(url, k-1, result)
        return result
Assignments:

1. Write script to download all .py files from http://www.math.uic.edu/~jan/mcs275/main.html

2. Limit the search of the crawler so that it only opens pages within the same domain. For example, if we start at a location ending with edu, we only open pages with locations ending with edu.

3. Adjust webcrawler.py to search for a path between two locations. The user is prompted for two URLs. Crawling stops if a path has been found.

4. Write an iterative version for the web crawler.

5. Use the stack in the iterative version of the crawler from the previous exercise to define a tree of all locations that can be reached from a given URL.