Parsing HTML and Web Crawlers

1. Web Clients
   - alternatives to web browsers
   - opening a web page and copying its content
2. Course Listings at UIC
   - grabbing course offerings
   - the `HTMLParser` module
   - overriding methods in HTMLParser
   - filtering attributes of tags
3. Web Crawlers and Scrapers
   - making requests recursively
   - scraping with `BeautifulSoup`

MCS 507 Lecture 24
Mathematical, Statistical and Scientific Software
Jan Verschelde, 16 October 2023

Parsing HTML and Web Crawlers

1. Web Clients
   - alternatives to web browsers
   - opening a web page and copying its content
2. Course Listings at UIC
   - grabbing course offerings
   - the `HTMLParser` module
   - overriding methods in HTMLParser
   - filtering attributes of tags
3. Web Crawlers and Scrapers
   - making requests recursively
   - scraping with `BeautifulSoup`

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

the weather forecast for Chicago

```
Lec24> python forecast.py
opening http://tgftp.nws.noaa.gov/data/forecasts/state/il/ilz013.txt ...

   Today     Tue       Wed       Thu       Fri       Sat       Sun
   Oct 16    Oct 17    Oct 18    Oct 19    Oct 20    Oct 21    Oct 22

   Chicago Downtown
   Shwrs     Ptcldy    Mocldy    Shwrs     Ptcldy    Ptcldy    Sunny
     /56     45/59     47/66     54/60     50/58     47/56     43/56
      /50     10/00     00/20     50/50     50/40     30/30     30/10

   Chicago O'Hare
   Mocldy    Ptcldy    Mocldy    Shwrs     Ptcldy    Ptcldy    Sunny
     /58     43/60     45/67     52/61     47/59     44/56     40/57
      /30     00/00     00/20     50/50     50/40     30/30     20/10
```

Executed on Monday 16 October 2023, in a Windows PowerShell.

the script `forecast.py`

```
from urllib.request import urlopen
HOST = 'http://tgftp.nws.noaa.gov'
FCST = '/data/forecasts/state'
URL = HOST + FCST + '/il/ilz013.txt'
print('opening ' + URL + ' ...\n')
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())
```

Parsing HTML and Web Crawlers

1. Web Clients
   - alternatives to web browsers
   - opening a web page and copying its content
2. Course Listings at UIC
   - grabbing course offerings
   - the `HTMLParser` module
   - overriding methods in HTMLParser
   - filtering attributes of tags
3. Web Crawlers and Scrapers
   - making requests recursively
   - scraping with `BeautifulSoup`

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

attempting to open a web page

```
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

```
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()
```

Parsing HTML and Web Crawlers

1. Web Clients
   - alternatives to web browsers
   - opening a web page and copying its content
2. Course Listings at UIC
   - grabbing course offerings
   - the `HTMLParser` module
   - overriding methods in HTMLParser
   - filtering attributes of tags
3. Web Crawlers and Scrapers
   - making requests recursively
   - scraping with `BeautifulSoup`

storing web data

One quick way to create a database of considerable size is to load it with data available on web pages.

An application: a database of courses at UIC

- data is already structured
- readily available at UIC's web pages

Stages in this project:

1. grab the data from the web into a file
2. format data on file into data tuples
3. insert data tuples into a database.

running `courselistings.py`

```
$ python3.5 courselistings.py
Give a subject : lat
Give the year (4 digits) : 2016
Spring, Fall, or Summer (0, 1, or 2) : 0
opening http://osss.uic.edu/ims/classschedule/S2016/LAT.htm ...
courses on http://osss.uic.edu/ims/classschedule/S2016/LAT.htm :
LAT102 Elementary Latin II
LAT104 Intermediate Latin II
LAT299 Independent Reading
$
```

Note: ran with version 3.5 of Python,
because later versions of Python complain about the certificate of the page ...

The work around is to use

```
context = ssl._create_unverified_context()
```

See the posted script for details.

online data

Course archives at UIC are available via the web site
for the office of student system services (osss).

The base of the URL is

`http://osss.uic.edu/ims/classschedule/`

followed by `S`, `SUM`, or `F`
for spring, summer, or fall semester,

followed by year as 4-digit number, e.g. `2010`,

followed by the subject, e.g.: `LAT`.

Parsing HTML and Web Crawlers

1. Web Clients
   - alternatives to web browsers
   - opening a web page and copying its content
2. Course Listings at UIC
   - grabbing course offerings
   - the `HTMLParser` module
   - overriding methods in HTMLParser
   - filtering attributes of tags
3. Web Crawlers and Scrapers
   - making requests recursively
   - scraping with `BeautifulSoup`

the `HTMLParser` module to parse html code

In the standard Python distribution:

```
>>> from html.parser import HTMLParser
>>> help(HTMLParser)
```

The class `HTMLParser`

- allows to override handlers of tags,
- provides a `feed` method

$\rightarrow$ the `feed` method handles buffering

using HTMLParser

```
from http.parser import HTMLParser
from urllib import urlopen

class OurHTMLParser(HTMLParser):

    def __init__(self):

    def handle_starttag(self, tag, attrs):

    def handle_endtag(self, tag):


def main():
    f = urlopen(page)
    p = OurHTMLParser()

    while True:
        data = f.read(80)
        if data == '': break
        p.feed(data)
    p.close()
```

Tallying the Tags

using the class HTMLParser

Gather basic statistics about a page:

1. what types of tags are used,
2. count number of occurrences for each tag.

At end of each tag the tally is updated.

Data structure for the tally: dictionary.

- keys: string with type of tag
- values: natural number counts #occurrences

The tally is an object data attribute.

Parsing HTML and Web Crawlers

1. Web Clients
   - alternatives to web browsers
   - opening a web page and copying its content
2. Course Listings at UIC
   - grabbing course offerings
   - the `HTMLParser` module
   - overriding methods in HTMLParser
   - filtering attributes of tags
3. Web Crawlers and Scrapers
   - making requests recursively
   - scraping with `BeautifulSoup`

the class `TagTally`

```
from html.parser import HTMLParser
from urllib.request import urlopen

class TagTally(HTMLParser):
    """
    Makes a tally of ending tags.
    """
    def __init__(self):
        """
        Initializes the dictionary of tags.
        """
    def __str__(self):
        """
        Returns the string representations of tags.
        """
    def handle_endtag(self, tag):
        """
        Maintains a tally of the tags.
        """
```

the `main()` of script `tallytags.py`

```
def main():
    """
    Opens a web page and parses it.
    """
    url = 'http://www.uic.edu'
    print('opening %s ...' % url)
    page = urlopen(url)
    tags = TagTally()
    while True:
        data = page.read(80).decode()
        if data == '':
            break
        tags.feed(data)
    tags.close()
    print('the tally of tags :')
    print(tags)
```

constructor and string representation

If overriding `__init__`, we must initialize parent class.

```
def __init__(self):
    """
    Initializes the dictionary of tags.
    """
    HTMLParser.__init__(self)
    self.tags = {}

def __str__(self):
    """
    Returns the string representation of tags.
    """
    result = ''
    for tag in self.tags:
        result += str(tag) + ':' + str(self.tags[tag]) + '\n'
    return result[:-1]
```

update of the tally

First check if there is already a tag...

```
def handle_endtag(self, tag):
    """
    Maintains a tally of the tags.
    """
    if tag in self.tags:
        self.tags[tag] = self.tags[tag] + 1
    else:
        self.tags.update({tag: 1})
```

If no tag present, update the dictionary.

Parsing HTML and Web Crawlers

1. Web Clients
   - alternatives to web browsers
   - opening a web page and copying its content
2. Course Listings at UIC
   - grabbing course offerings
   - the `HTMLParser` module
   - overriding methods in HTMLParser
   - filtering attributes of tags
3. Web Crawlers and Scrapers
   - making requests recursively
   - scraping with `BeautifulSoup`

get the links a page refers to

Recall our code for a web crawler:

- double quoted strings starting with `http`

could be misleading at times, cumbersome code.

A more proper way to get the hyperlinks:

1. look for tags of type `'a'`
2. name of attribute is `href`
3. get hyperlink corresponding to `href`

the Class HTMLrefs

```
from html.parser import HTMLParser
from urllib.request import urlopen
class HTMLrefs(HTMLParser):
    """
    Makes a list of all html links.
    """
    def __init__(self):
        """
        Initializes the list of links.
        """
    def __str__(self):
        """
        Returns the string rep of the links.
        """
    def handle_starttag(self, tag, attrs):
        """
        Looks for tags equal to 'a' and
        stores links for href attributes.
        """
```

the `main()` of the script `htmlrefs.py`

```
def main():
    """
    Opens a web page and parses it.
    """
    url = 'http://www.uic.edu/'
    print('opening %s ...' % url)
    page = urlopen(url)
    refs = HTMLrefs()
    while True:
        data = page.read(80).decode()
        if data == '':
            break
        refs.feed(data)
    refs.close()
    print('all html links :')
    print(refs)
```

constructor and string representation

We use a list as object data attribute.

```
def __init__(self):
    """
    Initializes the list of links.
    """
    HTMLParser.__init__(self)
    self.refs = []

def __str__(self):
    """
    Returns the string rep of the links.
    """
    result = ''
    for link in self.refs:
        result += link + '\n'
    return result[:-1]
```

filtering attributes

Attributes are lists of tuples: `[(.,.),(.,.),...]`

e.g.: `[('href', 'learning.shtml'), ...]`
→ link is the `y` in a `(x,y)` tuple

```
def handle_starttag(self, tag, attrs):
    """
    Looks for tags equal to 'a' and
    stores links for href attributes.
    """
    print(attrs)
    if tag == 'a':
        F = [x_y for x_y in attrs if x_y[0] == 'href']
        L = [y for (x, y) in F]
        self.refs = self.refs + L
```

Parsing HTML and Web Crawlers

1. Web Clients
   - alternatives to web browsers
   - opening a web page and copying its content
2. Course Listings at UIC
   - grabbing course offerings
   - the `HTMLParser` module
   - overriding methods in HTMLParser
   - filtering attributes of tags
3. Web Crawlers and Scrapers
   - making requests recursively
   - scraping with `BeautifulSoup`

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.

$\rightarrow$ continue recursively, *crawling* the web

exception handling

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

```
Lec24> python webcrawler.py
crawling the web ...
Give URL : http://www.uic.edu
give maximal depth : 2
opening http://www.uic.edu ...
```

.. it takes a while ..

```
total #locations : 689
Lec24>
```

modular design of a web crawler

```
from scanhttplinks import httplinks
```

Still left to write:

1. management of list of server locations,
2. recursive function to crawl the web.

retain only new Locations

```
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:
            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`

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

We must provide a 6-tuple as argument ...

the function `main()`

```
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))
```

code for the crawler

```
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 = httplinks(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
```

Parsing HTML and Web Crawlers

1. Web Clients
   - alternatives to web browsers
   - opening a web page and copying its content
2. Course Listings at UIC
   - grabbing course offerings
   - the `HTMLParser` module
   - overriding methods in HTMLParser
   - filtering attributes of tags
3. Web Crawlers and Scrapers
   - making requests recursively
   - scraping with `BeautifulSoup`

web scraping

Ryan Mitchell: *Web Scraping with Python. Collecting Data from the Modern Web.* O'Reilly, 2015.

`BeautifulSoup` is a Python library

- for pulling data out of HTML and XML files;
- with support for Python's HTML parser and alternative parsers such as `lxml` and `html5lib`.

These alternative parsers have to be installed separately.

Summary + Exercises

Exercises:

1. 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`.
2. 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.
3. Write an iterative version for the web crawler.
4. 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.