COM661 Full Stack Strategies and Development

Practical A2: Web Programming with Python

Aims

- To present Python as a language for development of serious web applications
- To introduce the **`urllib.request`** library for remote access and retrieval of web pages
- To present the key components of search engines
- To demonstrate identification and extraction of specific elements of web pages
- To demonstrate the structure and implementation of a web crawler application
- To present a technique for limiting the scope of a web crawler
- To introduce the Python set operators and their application to lists

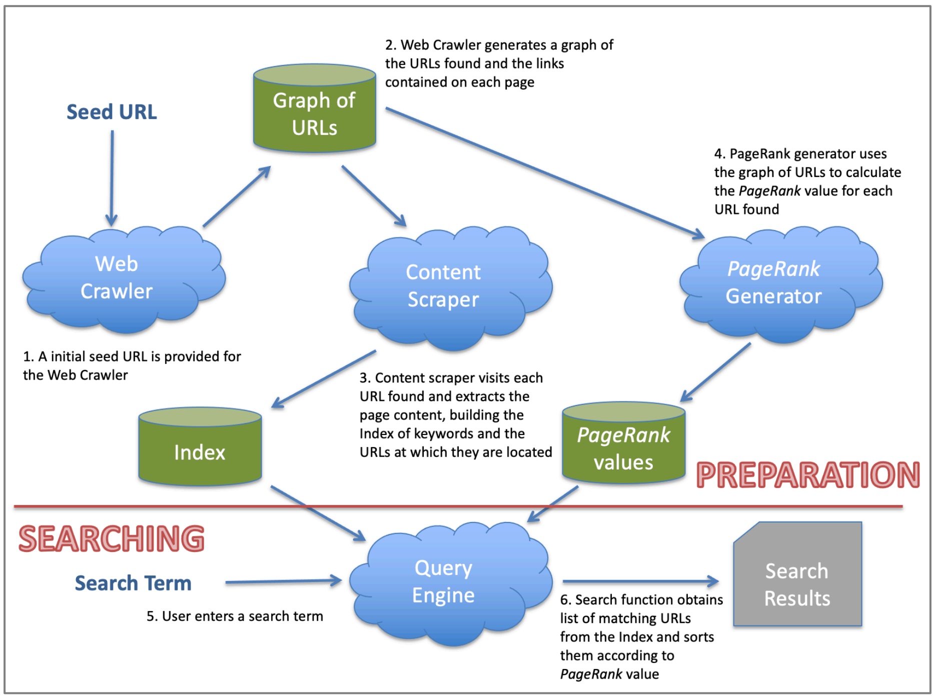
Contents

A2.1 Python and Web Programming

During this section of the module, we will examine the power of Python-based web programming by building the component elements of ***Poodle*** – a small Python-powered search engine. Search engines typically consist of 3 components:

i) a directory of web pages
ii) an index of content with links to the page at which the content is located
iii) a page ranking algorithm that determines the page that best satisfies a user query

These components are described in Figure A2.1 below, which identifies how they interact and the main data structures that are produced. It is these data structures that we will generate and manipulate over this and the following 2 practical sessions.



2. Web Crawler generates a graph of the URLs found and the links contained on each page
Seed URL
Graph of URLs
4. PageRank generator uses the graph of URLs to calculate the *PageRank* value for each URL found
Web Crawler
Content Scraper
PageRank Generator
1. A initial seed URL is provided for the Web Crawler
3. Content scraper visits each URL found and extracts the page content, building the Index of keywords and the URLs at which they are located
Index
PageRank values
PREPARATION
SEARCHING
Search Term
5. User enters a search term
Query Engine
6. Search function obtains list of matching URLs from the Index and sorts them according to *PageRank* value
Search Results

Figure A2.1 Anatomy of a Search Engine

The preparation of a Search Engine is illustrated by steps 1-4 in Figure A2.1. First, a starting point, or seed URL, is specified and a Web Crawler identifies and stores all the hyperlinks on this page. Then, it visits all the hyperlinks found and retrieves all the links on those pages. This process continues until there are no more hyperlinks left to visit, or (more likely) some agreed termination condition has been reached. The output from this is a graph of URLs, representing all the pages found and the pattern of hyperlinks between them.

Next, the graph of URLs is passed as input to a Content Scraper which visits each separate hyperlink found by the Web Crawler and stores and indexes all of the content. This index is the key data structure of the search engine as it associates page content with the URLs at which this content was found. In this way, if a search user later requests a particular search term, the index is able to provide the complete list of URLs at which that particular term was found.

The final element in the construction of the search engine is the PageRank generator, that associates a value to each URL that represents the "importance" of the page. This is the value used when determining the order in which search results should be returned to a user.

Once the Index and table of PageRank values are in place, the actual search engine is a relatively simple process. The engine accepts a search term from the user, determines from the Index the URLs at which that term is found and then presents that list of URLs sorted by PageRank value.

The first component of a search engine is therefore the ability to generate and manage a collection of links to web pages. In today's practical we will build a Web Crawler application that trawls the web and builds ***Poodle***'s collection of links.

A2.2 Extracting Elements from a HTML Stream

To harvest the links from the source of a web page (a technique known as **scraping**), we need to read the HTML code into a string variable for parsing. The Python library **urllib.request** (part of the larger **urllib** module) provides a very simple interface for fetching URLs using a variety of different protocols.

File*: A2/find_links.py*

```
import urllib.request
response = urllib.request.urlopen( \
                   'http://www.ulster.ac.uk/campuses/belfast')
html = str(response.read())
```

The **`urlopen()`** method takes a single parameter representing the complete URL of the resource we want to read (in this case the University's Belfast Campus information page) and returns a response object for the URL requested. The response is a file-like object, meaning that we can simply **`read()`** it just as we would a text file. Note that the data returned by the **`read()`** method in this context is a byte stream that we convert to a string by the **`str()`** function. The effect of this code is that **`html`** is a string variable containing the **entire contents** of the file at the specified URL.

Note: **`urlopen()`** can also accept protocols other than **http**. For example, we could obtain the contents of URLs that begin **file://**, **ftp://**, etc.

Having read the HTML source into the variable **`html`**, we now parse that content using the Python string processing functions. The following code iterates over an HTML sequence, extracting and printing the **`href`** attribute of each **`<a>`** hyperlink element discovered.

Note: For simplicity, we assume here that all hyperlinks are specified in the HTML in the form

```
<a href="http://url.of.web.page">Text for link</a>
```

such that the **`href`** property is immediately following the opening **`<a`** tag and is separated from the **`<a`** by exactly one space. We also assume that the URL is enclosed in double inverted comma characters.

If we were building a "real" web scraper, we would need to cater for all possible code formats – and it is even more likely that we would use a dedicated web scraping package such as BeautifulSoup.

File: A2/find_links.py

```
...

pos = 0
all_found, links_found = False, 0
while not all_found:
   tag_start = html.find("<a href=", pos)
   if tag_start > -1:
      href = html.find('"', tag_start + 1)
      end_href = html.find('"', href + 1)
      url = html[href + 1:end_href]
      print(url)
      links_found = links_found + 1
      close_tag = html.find("</a>", tag_start)
      pos = close_tag + 1
   else:
      all_found = True

print("{} hyperlinks found".format(linksFound))
```

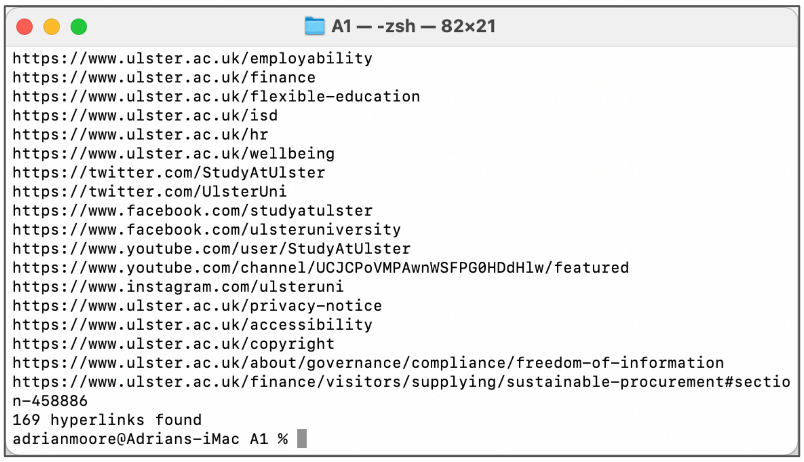
The code works by using an integer variable **`pos`** to act as a pointer into the string. Within the **`while`** loop, we first use the **`find()`** method to search for the first occurrence of the character sequence "**`<a href=`**", denoting the beginning of a hyperlink.

If the search has been successful (any value other than -1 is returned by the **`find()`** method), we use **`find()`** twice more to locate the positions of the inverted comma characters that bound the URL within the **`<a>`** tag. By using the string slice operator **`[:]`**, we can then extract and print the text between the "..." characters.

Finally, we use **`find()`** once more to locate the **`</a>`** tag for the current link, and update **`pos`** before beginning the search for the next hyperlink.

If any search for "**`<a href=`**" returns -1, then we have already found all hyperlinks, so the loop can terminate.

Do it now!	Run *find_links.py* in the Python interpreter and see how all the links on the specified page are extracted and displayed as shown in Figure A2.1. Load the URL http://www.ulster.ac.uk/campuses/belfast into a web browser and see how the output of the Python script relates to the content of the web page.



```
https://www.ulster.ac.uk/employability
https://www.ulster.ac.uk/finance
https://www.ulster.ac.uk/flexible-education
https://www.ulster.ac.uk/isd
https://www.ulster.ac.uk/hr
https://www.ulster.ac.uk/wellbeing
https://twitter.com/StudyAtUlster
https://twitter.com/UlsterUni
https://www.facebook.com/studyatulster
https://www.facebook.com/ulsteruniversity
https://www.youtube.com/user/StudyAtUlster
https://www.youtube.com/channel/UCJCPoVMPAwnWSFPG0HDdHlw/featured
https://www.instagram.com/ulsteruni
https://www.ulster.ac.uk/privacy-notice
https://www.ulster.ac.uk/accessibility
https://www.ulster.ac.uk/copyright
https://www.ulster.ac.uk/about/governance/compliance/freedom-of-information
https://www.ulster.ac.uk/finance/visitors/supplying/sustainable-procurement#sectio
n-458886
169 hyperlinks found
adrianmoore@Adrians-iMac A1 %
```

Figure A2.1 Extracting links from a web page

Try it now!	Modify *find_links.py* so that the user is prompted for a URL to check from the command line. Test the code with a variety of URLs.

The basic version of *find_links.py* successfully locates all the hyperlinks on the page, but we could use some filtering and moderation of the output. There are some links that we would prefer not to have reported – for example links beginning **mailto:** do not refer to a visit-able webpage and links beginning **ftp://** typically refer to binary files - and there is potential ambiguity with links ending in the forward-slash character /. (Later, we will remove any duplicates from the list and make sure that *http:/www.somewhere.com* and *http://www.somewhere.com/* both refer to the same URL).

Try it now!

Modify *find_links.py* so that only URLs beginning in **http://** or **https://** are accepted and displayed. Also, where a URL ends in a trailing **/** character, the last character of the link should be removed.

Try it now!

Create the script *find_images.py* that accepts prompts the user to enter a URL and generates a list of the URLs of the images found at that URL. Only the image filename is required, rather than the full URL or the directory path.

You can assume that all images are specified in the form

```
   <img src="filename.img">
```

where the **`src`** component is the first attribute of the **`<img>`** tag, is separated from **`<img`** by a single space and the filename is enclosed in double inverted comma characters.

Hint: if the **`src`** attribute of the image begins contains a **/** character, then the string represents either the complete URL or includes a directory path (e.g. **myfiles/images/picture.gif**). In this case, only the portion of the string following the last **/** character should be returned (i.e. **picture.gif**).

Our application *find_links.py* identifies and displays the links embedded on a webpage, but ***Poodle*** will require that the links are stored in a data structure for later processing. Examine the code for *store_links.py*, which builds a list object containing the links.

File: A2/store_links.py

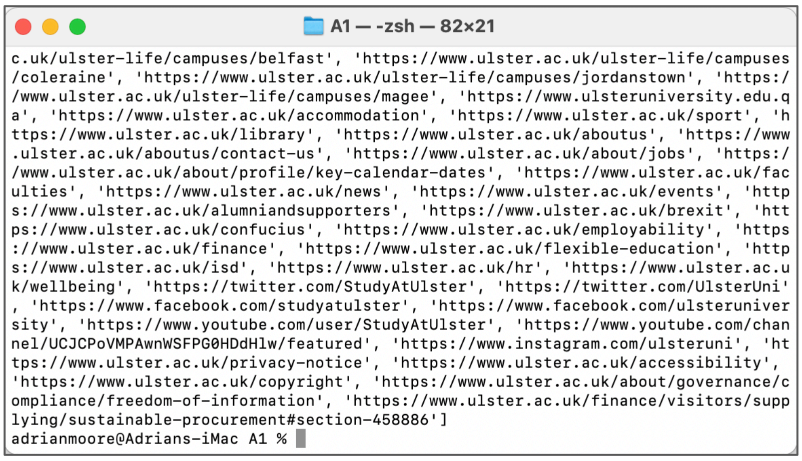
```
import urllib.request
response = urllib.request.urlopen( \
                  http://www.ulster.ac.uk/campuses/belfast')
html = str(response.read())

links, pos = [], 0
all_found, links_found = False, 0
while not all_found:
   tag_start = html.find("<a href=", pos)
   if tag_start > -1:
      href = html.find('"', tag_start + 1)
      end_href = html.find('"', href + 1)
      url = html[href + 1:end_href]
      if url[:7] == "http://" or url[:8] == "https://":
         if url[-1] == "/":
            url = url[:-1]
         links.append(url)
         print(url)
         links_found = links_found + 1
         close_tag = html.find("</a>", tag_start)
         pos = close_tag + 1
   else:
      all_found = True

print("{} hyperlinks found".format(links_found))
print(links)
```

Here, we use a list object **`links`** to maintain the collection of links. When a new link is obtained, the **`append()`** method adds the new URL to the end of the list. Note that this version of the application also incorporates the improvements from the previous **Try it Now!** exercise.

Do it now!	Run *store_links.py* in the Python interpreter and see how the output such as that illustrated in Figure A2.2 is produced from the Belfast Campus information page http://www.ulster.ac.uk/campuses/belfast.



```
c.uk/ulster-life/campuses/belfast', 'https://www.ulster.ac.uk/ulster-life/campuses
/coleraine', 'https://www.ulster.ac.uk/ulster-life/campuses/jordanstown', 'https:/
/www.ulster.ac.uk/ulster-life/campuses/magee', 'https://www.ulsteruniversity.edu.q
a', 'https://www.ulster.ac.uk/accommodation', 'https://www.ulster.ac.uk/sport', 'h
ttps://www.ulster.ac.uk/library', 'https://www.ulster.ac.uk/aboutus', 'https://www
.ulster.ac.uk/aboutus/contact-us', 'https://www.ulster.ac.uk/about/jobs', 'https:/
/www.ulster.ac.uk/about/profile/key-calendar-dates', 'https://www.ulster.ac.uk/fac
ulties', 'https://www.ulster.ac.uk/news', 'https://www.ulster.ac.uk/events', 'http
s://www.ulster.ac.uk/alumniandsupporters', 'https://www.ulster.ac.uk/brexit', 'htt
ps://www.ulster.ac.uk/confucius', 'https://www.ulster.ac.uk/employability', 'https
://www.ulster.ac.uk/finance', 'https://www.ulster.ac.uk/flexible-education', 'http
s://www.ulster.ac.uk/isd', 'https://www.ulster.ac.uk/hr', 'https://www.ulster.ac.u
k/wellbeing', 'https://twitter.com/StudyAtUlster', 'https://twitter.com/UlsterUni'
, 'https://www.facebook.com/studyatulster', 'https://www.facebook.com/ulsteruniver
sity', 'https://www.youtube.com/user/StudyAtUlster', 'https://www.youtube.com/chan
nel/UCJCPoVMPAwnWSFPG0HDdHlw/featured', 'https://www.instagram.com/ulsteruni', 'ht
tps://www.ulster.ac.uk/privacy-notice', 'https://www.ulster.ac.uk/accessibility',
'https://www.ulster.ac.uk/copyright', 'https://www.ulster.ac.uk/about/governance/c
ompliance/freedom-of-information', 'https://www.ulster.ac.uk/finance/visitors/supp
lying/sustainable-procurement#section-458886']
adrianmoore@Adrians-iMac A1 % 
```

Figure A2.2. Gather links and store in a list

We can see from running *store_links.py* that all links retrieved now begin **http://** or **https://** and all trailing forward-slash characters have been removed. However, where a link appears multiple times on a webpage, then it is recorded multiple times in our list. For the purposes of our Web Crawler, we require our list to contain unique URLs - with no duplicates.

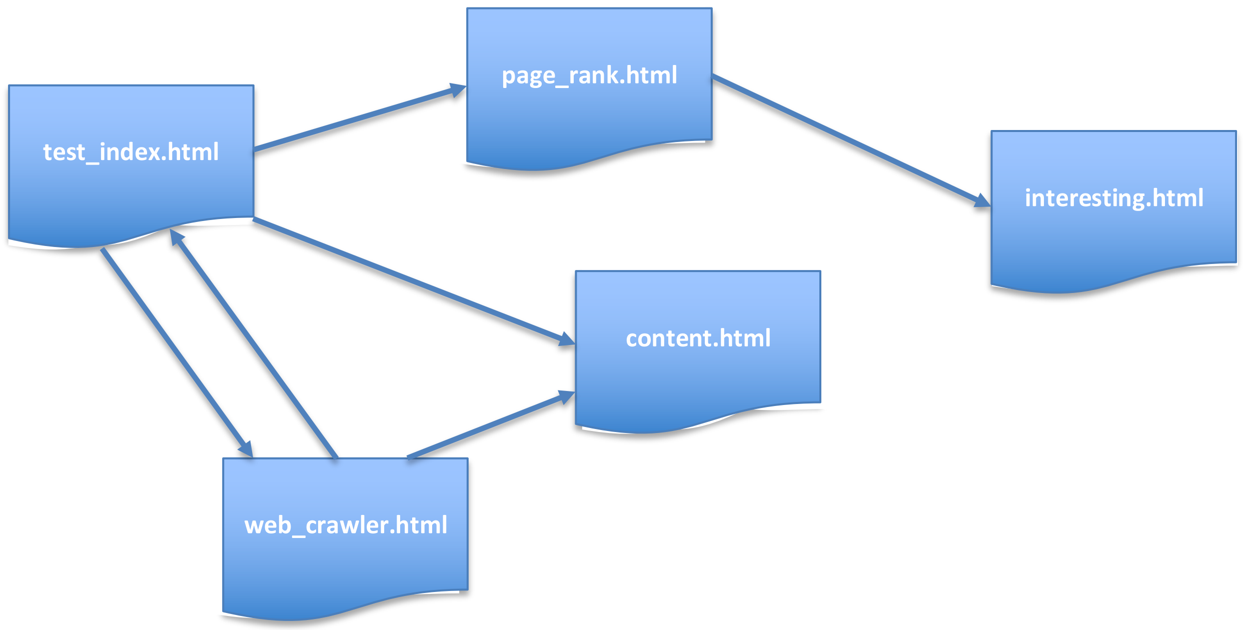
Try it now!	Modify *store_links.py* so that only unique links are saved in the list.

A2.3 Building a Web Crawler

A Web Crawler is "a computer program that browses the World Wide Web in a methodical, automated manner" (Source: http://en.wikipedia.org/wiki/Web_crawler). It starts with a URL to visit (a seed) and gathers all the URLs contained on that seed page. All these URLs are then recursively visited, gathering further URLs from these in turn. A Web Crawler forms the basis for a search engine – providing the collection of pages for which the content will be indexed.

As an example, consider the following small collection of linked pages:

- *test_index.html* – with links to *web_crawler.html*, *content.html* and *page_rank.html*
- *web_crawler.html* – with links to *content.html* and *test_index.html*
- *content.html* – with no links
- *page_rank.html* – with a link to *interesting.html*
- *interesting.html* – with no links



Do it now!	Load http://adrianmoore.net/com661/test_index.html into a web browser and explore the collection of linked web pages provided.

The Web Crawler builds up its list of URLs by maintaining two lists **`to_crawl`** and **`crawled`** as follows:

Stage 1: initialize the **`to_crawl`** list with the seed URL (**`to_crawl`** list is now [*test_index.html*]).
Initialize the **`crawled`** list to be empty **`[]`**

Stage 2: Remove an entry from the **`to_crawl`** list (*test_index.html*). **`to_crawl`** list is now empty **`[]`**
Add this entry to the **`crawled`** list (**`crawled`** is now [*test_index.html*])

Stage 3: Get all the links from that page (*test_index.html*)
Add any links not already discovered to the **`to_crawl`** list (**`to_crawl`** is now [*web_crawler.html*, *content.html*, *page_rank.html*]

Stage 4: Remove an entry from the **`to_crawl`** list (*page_rank.html*). **`to_crawl`** list is now [*web_crawler.html*, *content.html*]
Add this entry to the **`crawled`** list (**`crawled`** is now [test_index.html, page_rank.html])

Stage 5: Get all the links from that page (*page_rank.html*)

Add any links not already discovered to the **`to_crawl`** list (**`to_crawl`** is now [*web_crawler.html*, *content.html*, *interesting.html*]

Stage 6: Remove an entry from the **`to_crawl`** list (*interesting.html*)
Add this entry to the **`crawled`** list (**`crawled`** is now [*test_index.html*, *page_rank.html*, *interesting.html*])

Stage 7: Get all the links from that page (*interesting.html*)
Add any links not already discovered to the **`to_crawl`** list (**`to_crawl`** is now [*web_crawler.html*, *content.html*]

Stage 8: Remove an entry from the **`to_crawl`** list (*content.html*)
Add this entry to the **`crawled`** list (**`crawled`** is now [*test_index.html*, *page_rank.html*, *interesting.html*, *content.html*])

Stage 9: Get all the links from that page (*content.html*)
Add any links not already discovered to the **`to_crawl`** list (**`to_crawl`** is now [*web_crawler.html*]

Stage 10: Remove an entry from the **`to_crawl`** list (*web_crawler.html*)
Add this entry to the **`crawled`** list (**`crawled`** is now [*test_index.html*, *page_rank.html*, *interesting.html*, *content.html*, *web_crawler.html*])

Stage 11: Get all the links from that page (*web_crawler.html*)
Add any links not already discovered to the **`to_crawl`** list (**`to_crawl`** is now empty **`[]`**

Stage 12: As **`to_crawl`** is now empty, **`crawled`** contains the complete list of pages [*test_index.html*, *page_rank.html*, *interesting.html*, *content.html*, *web_crawler.html*]

Do it now!	Follow the 12-stage sequence by tracing the content of the files in the ***test_web*** example. Be sure that you understand the behavior of the Web crawler before continuing.

Thankfully, the Python version of the Web Crawler code is slightly less complex than the narrative version…

File: A2/web_crawler.py

```
import urllib.request

to_crawl = \
       ["http://adrianmoore.net/com661/test_index.html"]
crawled = []
while to_crawl:
        url = to_crawl.pop()
        crawled.append(url)
        new_links = get_all_new_links_on_page(url, crawled)
        to_crawl = list( set(to_crawl) | set(new_links))

print(crawled)
```

The Web Crawler above is implemented as follows

- First, we initialize the **`to_crawl`** list to the seed URL, and the **`crawled`** list to empty (Stage 1, above).
- For as long as there are items remaining in the **`to_crawl`** list, we remove (**`pop`**) the last element from the **`to_crawl`** list and **`append`** it to the **`crawled`** list. (Stages 2,4,6,8,10)
- We call the function **`get_all_new_links_on_page()`** to extract all undiscovered links in the source HTML of the current page and merge those links with the **`to_crawl`** list. (Stages 3,5,7,9,11)
- Once the **`to_crawl`** list is empty, the loop terminates as all links have been found. (Stage 12)

Note in particular the last line of this code fragment, which merges a pair of lists while eliminating duplicates. It works by first converting the lists to **set**s and then combining the sets with the **set union** operator (**|**). The union of the two sets is then converted back to a **list**.

The **`get_all_new_links_on_page()`** function accepts a URL and a list of previously visited URLs and returns a list containing all links in the source of the page that do not appear in the previously visited list. This is essentially the code from the earlier example *store_links.py*, except that we have an additional input **`prev_links`** – a list of previously discovered URLs (i.e. the **`crawled`** list). When a link has been extracted from the source code, we check that it is not already present in the list of previously discovered links before adding it to the collection generated from that URL. This prevents the situation where pages that reference each other cause the crawler to be trapped in an infinite loop.

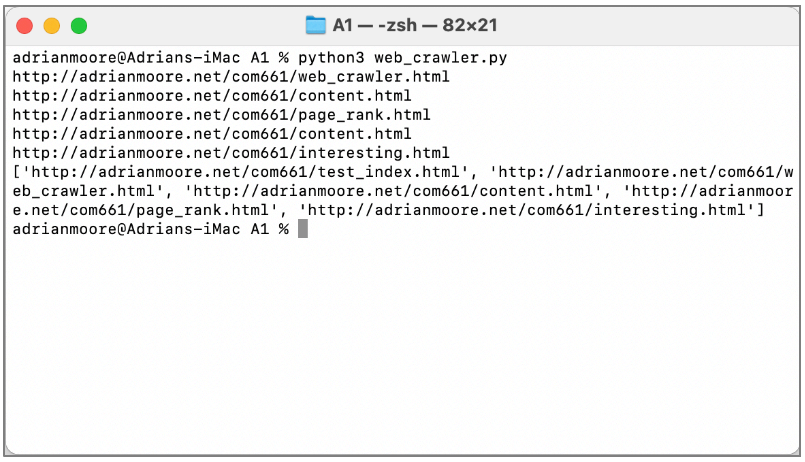
Do it now!	Complete the implementation of the Web Crawler by adding the code for **`get_all_new_links_on_page()`** (presented below) to the top of the source file. Now, run *web_crawler.py* and observe its output.

File: A2/web_crawler.py

```
def get_all_new_links_on_page(page, prev_links):
   response = urllib.request.urlopen(page)
   html = str(response.read())

   links, pos, all_found = [], 0, False
   while not all_found:
      tag_start = html.find("<a href=", pos)
      if tag_start > -1:
         href = html.find('"', tag_start + 1)
         end_href = html.find('"', href + 1)
         url = html[href + 1:end_href]
         if url[:7] == "http://" or url[:8] == "https://":
            if url[-1] == "/":
               url = url[:-1]
            if url not in links and url not in prev_links:
               links.append(url)
               print(url)
         close_tag = html.find("</a>", tag_start)
         pos = close_tag + 1
      else:
         all_found = True
      return links

...
```



```
adrianmoore@Adrians-iMac A1 % python3 web_crawler.py
http://adrianmoore.net/com661/web_crawler.html
http://adrianmoore.net/com661/content.html
http://adrianmoore.net/com661/page_rank.html
http://adrianmoore.net/com661/content.html
http://adrianmoore.net/com661/interesting.html
['http://adrianmoore.net/com661/test_index.html', 'http://adrianmoore.net/com661/w
eb_crawler.html', 'http://adrianmoore.net/com661/content.html', 'http://adrianmoor
e.net/com661/page_rank.html', 'http://adrianmoore.net/com661/interesting.html']
adrianmoore@Adrians-iMac A1 %
```

Figure. A2.3. Web Crawler output

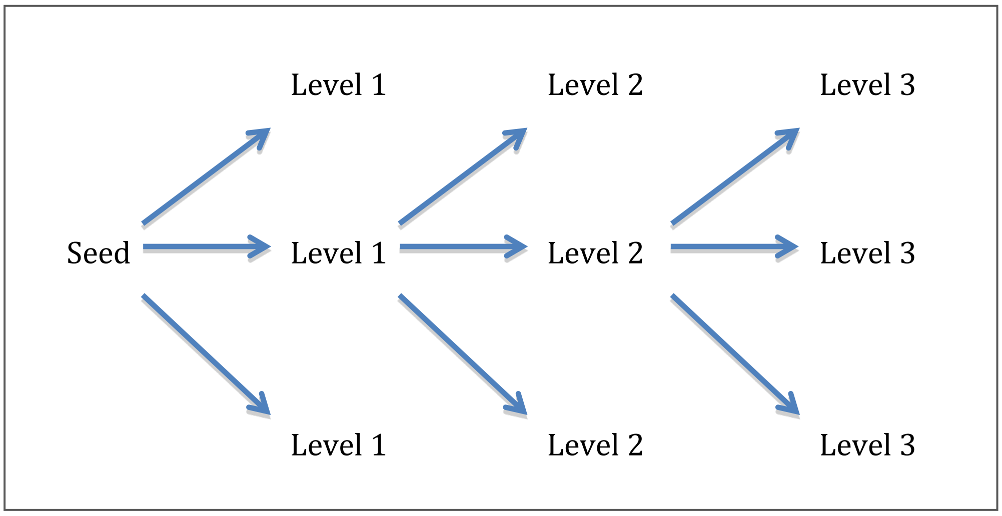
Up to now, we have tested our Web Crawler on a "closed" environment – the small **test_web** example that contains no links to the outside world. We could, in theory, provide any URL as the seed for the Web Crawler (**DON'T DO THIS!**), but there are two significant reasons why this would not be a good idea…

i) Our Web Crawler program will continue until there are no more pages to find. Currently, Google's index is estimated to contain in the region of 55 billion webpages (Source: http://www.worldwidewebsize.com) so we will soon overload the memory and processing capability of our PC – leading to potentially unpredictable results and the potential for…

ii) A **Denial of Service Attack** is where a host is bombarded with requests that prevent it from performing its intended service. An out-of-control Web Crawler making rapid-fire requests to (even different pages on) the same server can easily fall into this category.

Note: A Denial of Service Attack is a form of hacking and is a violation of the IAB (Internet Architecture Board) Internet proper use policy. It also violates the acceptable use policy of virtually all Internet Service Providers – and Ulster University! - as well as being illegal. **Please do not run un-tested Web Crawlers on external web sites**.

One way to make our Crawler acceptable to external web sites is to limit its scope. If we keep track of how many steps removed from the original seed URL we have moved, and terminate execution once the limit of steps has been reached, then the Crawler is guaranteed to exit gracefully (as long as we choose a sensible limit). For example, consider the following tree representing URLs discovered by a Crawler.



In the diagram above, nodes labeled "Level 1" represent pages one click away from the seed URL, "Level 2" represents nodes two clicks away from the seed URL, and so on. If the Crawler keeps track of the number of levels of removal from the original seed, then we can ensure that it terminates gracefully when the limit is reached.

Try it now!

Modify *web_crawler.py* so that it contains a constant value **`MAX_DEPTH`** that limits the number of steps from the original seed that can be taken. Once all pages within **`MAX_DEPTH`** links of the seed have been visited, the crawler should terminate.

Hint: Each entry in the **`to_crawl`** list will need to store the level at which that page was found as well as the URL for the page. One way to do this is to make each entry in the **`to_crawl`** list a list of 2 values rather than a simple URL. For example, the **`to_crawl`** list might be:

```
[ ['http://somewhere.com',2],
  ['http://somewhere.else.com',3],
  ...
]
```

where the first element of each entry is the URL and the second element is the level at which that URL was found.

A2.4 Further Information

- https://docs.python.org/3/library/urllib.request.html
  Python manual – the **urllib.request** library

- http://wiki.python.org/moin/WebProgramming
  Python for Web programming

- http://programmers.stackexchange.com/questions/12189/how-do-i-learn-python-from-zero-to-web-development
  Links for learning Python for Web development

- http://en.wikipedia.org/wiki/Web_crawler
  Web Crawler definition from Wikipedia

- https://www.crummy.com/software/BeautifulSoup/bs4/doc/
  BeautifulSoup Documentation

- http://www.worldwidewebsize.com
  Measuring the size of the World Wide Web

- http://tudr.thapar.edu:8080/jspui/bitstream/10266/620/3/T620.pdf
  Web Crawling Approaches in Search Engines (M.Eng. Thesis, Thapar University, Patiala, India)