COM661 Full Stack Techniques and Development

Practical A3: Complex Data Structures

Aims

- To introduce Web Scraping as a technique for harvesting information from web pages
- To demonstrate complex list elements
- To demonstrate the representation of a search engine index of content as a list
- To present the serialization of complex Python data structures
- To measuse the time performance of Python code
- To analyse algorithm performance with respect to future load
- To introduce the dictionary as a complex data structure
- To compare time performance of data retrieval from Python lists and dictionaries

Contents

A3.1 WEB SCRAPING 2

A3.2 COMPLEX LISTS 4

A3.3 PERFORMANCE AND EFFICIENCY 7

A3.4 PYTHON DICTIONARY STRUCTURES 11

A3.5 FURTHER INFORMATION 15

A3.1 Web Scraping

In the last practical, we developed a Web Crawler application that will form the basis for our Python search engine, ***Poodle***. In this exercise, we will investigate techniques for extracting the content from the pages discovered by the Crawler and evaluate alternative data structures that can be used to store the information gathered.

Web Scraping is "*the process of automatically extracting information from the Word Wide Web*" (Source: http://en.wikipedia.org/wiki/Web_scraping). Sometimes it is used to extract specific information from web pages, but in the context of search engines, it typically involves the gathering of all non-tag content – i.e., the text that makes up the content of the page. Consider the following code that extracts the content from the University Belfast campus information page at http://www.ulster.ac.uk/campuses/belfast, splitting the text into individual words and storing the words as a list.

File: A3/get_page_text.py

```
import urllib.request

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

page_text, page_words = "", []
html = html[html.find("<body") + 5:html.find("</body>")]

finished = False
while not finished:
   next_close_tag = html.find(">")
   next_open_tag = html.find("<", next_close_tag + 1)
   if next_open_tag > -1:
      content = " ".join(html[next_close_tag+1:next_open_tag] \
                .strip().split())
      page_text = page_text + " " + content
      html = html[next_open_tag + 1:]
   else:
      finished = True

for word in page_text.split():
   if word.isalnum() and word not in page_words:
      page_words.append(word)

print(page_words)
print("{} unique words found".format(len(page_words)))
```

Here, we use **urllib.request** to read the entire source of the web page into a string **`html`**. We then use a combination of the **`find()`** method and a string slice operator to obtain any the text between the **`<body>`** **…** **`</body>`** tags – where the page content will be located.

The main **`while`** loop operates by locating pairs of '>' and '<' characters and extracting the text between them. For example, the string

<p>This is a <b>sample</b> paragraph</p>

would be parsed by extracting text as follows:

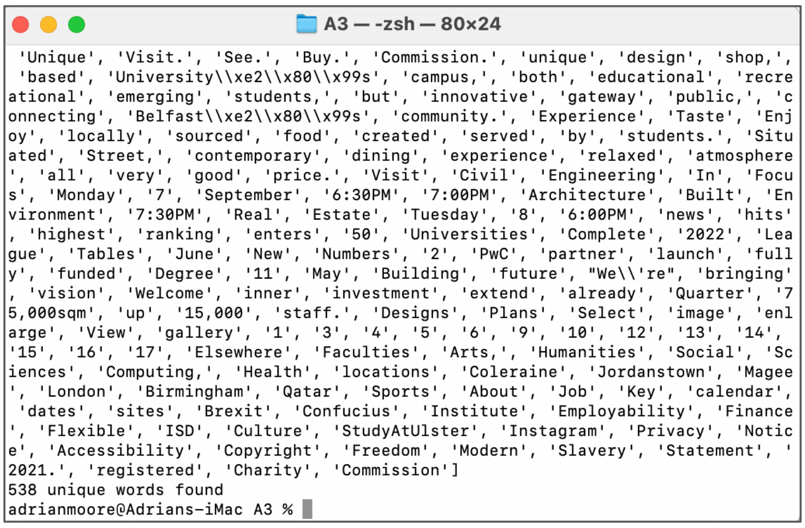
Stage 1: Locate the next '>' and '<' pair and grab the text between them
<p>**This is a** <b>sample</b> paragraph</p>

Stage 2: Locate the next '>' and '<' pair and grab the text between them
<p>This is a <b>**sample**</b> paragraph</p>

Stage 3: Locate the next '>' and '<' pair and grab the text between them
<p>This is a <b>sample</b> **paragraph**</p>

Stage 4: No more '>' and '<' pairs, so split the extracted text and return as a list
['This', 'is', 'a', 'sample', 'paragraph']

Do it now!	Run *get_page_text.py* and see how the content on the web page is extracted and displayed as shown in Figure A3.1. 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



```
 'Unique', 'Visit.', 'See.', 'Buy.', 'Commission.', 'unique', 'design', 'shop,',
 'based', 'University\\xe2\\x80\\x99s', 'campus,', 'both', 'educational', 'recre
ational', 'emerging', 'students,', 'but', 'innovative', 'gateway', 'public,', 'c
onnecting', 'Belfast\\xe2\\x80\\x99s', 'community.', 'Experience', 'Taste', 'Enj
oy', 'locally', 'sourced', 'food', 'created', 'served', 'by', 'students.', 'Situ
ated', 'Street,', 'contemporary', 'dining', 'experience', 'relaxed', 'atmosphere
', 'all', 'very', 'good', 'price.', 'Visit', 'Civil', 'Engineering', 'In', 'Focu
s', 'Monday', '7', 'September', '6:30PM', '7:00PM', 'Architecture', 'Built', 'En
vironment', '7:30PM', 'Real', 'Estate', 'Tuesday', '8', '6:00PM', 'news', 'hits'
, 'highest', 'ranking', 'enters', '50', 'Universities', 'Complete', '2022', 'Lea
gue', 'Tables', 'June', 'New', 'Numbers', '2', 'PwC', 'partner', 'launch', 'full
y', 'funded', 'Degree', '11', 'May', 'Building', 'future', "We\\'re", 'bringing'
, 'vision', 'Welcome', 'inner', 'investment', 'extend', 'already', 'Quarter', '7
5,000sqm', 'up', '15,000', 'staff.', 'Designs', 'Plans', 'Select', 'image', 'enl
arge', 'View', 'gallery', '1', '3', '4', '5', '6', '9', '10', '12', '13', '14',
'15', '16', '17', 'Elsewhere', 'Faculties', 'Arts,', 'Humanities', 'Social', 'Sc
iences', 'Computing,', 'Health', 'locations', 'Coleraine', 'Jordanstown', 'Magee
', 'London', 'Birmingham', 'Qatar', 'Sports', 'About', 'Job', 'Key', 'calendar',
 'dates', 'sites', 'Brexit', 'Confucius', 'Institute', 'Employability', 'Finance
', 'Flexible', 'ISD', 'Culture', 'StudyAtUlster', 'Instagram', 'Privacy', 'Notic
e', 'Accessibility', 'Copyright', 'Freedom', 'Modern', 'Slavery', 'Statement', '
2021.', 'registered', 'Charity', 'Commission']
538 unique words found
adrianmoore@Adrians-iMac A3 %
```

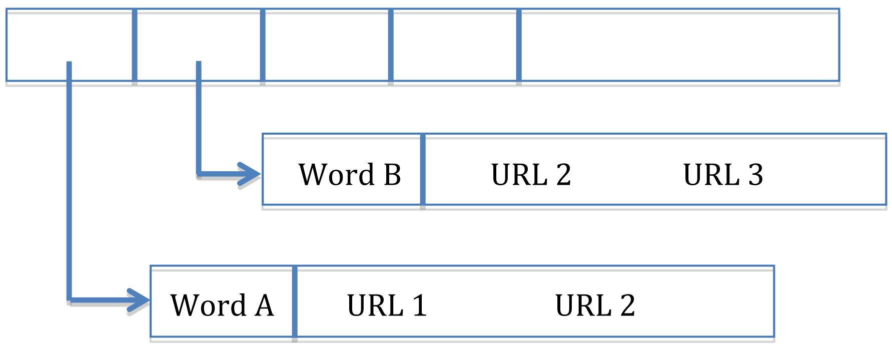
Figure A3.1 Result of the Web Scraper

Try it now!	Make the following modifications to *get_page_text.py* to improve its performance and assist with its deployment within our search engine: 1) Re-structure the code so that the application contains a function **`get_page_text()`** that takes a URL as a parameter and returns a list containing all of the unique words found at that URL. 2) Have the function ignore any content contained within **`<script>`** … **`</script>`** tags. Verify that it works by scraping the content of any page that includes a **`<script>`** element. You will find such a page at http://adrianmoore.net/COM661/scriptTest.html. 3) Create a text file *ignorelist.txt* that contains a sequence of words such as "a", "the", "on", etc. that we want the Web Scraper to ignore. Modify *get_page_text.py* so that it will return words of any length except for those present in this file.

A3.2 Complex Lists

Our Web Scraper will successfully return all the words found on a web page, but to incorporate that page content within ***Poodle***, we need also to store the URLs at which the content can be found.

Our initial version of the ***Poodle*** web index will be structured as a list, where each element is itself a list containing 2 items – a keyword and a list of URLs where that keyword can be found.



In the illustration above, the index contains 2 items, a keyword (“Word A”) that has been discovered at URL1 and URL 2; and a keyword (“Word 2”) that has been discovered at URL2 and URL3. The Python representation of this would be as shown below

```
[ [“Word A”, [“URL 1”,“URL 2”] ], [ “Word 2”, [“URL 2”,“URL 3”] ] ]
```

When adding a keyword to the index, we first need to establish if it is already present – if it is, we append the new URL to the list for that keyword; if not, we create a new element. Examine the following code that defines a new index and stores the content scraped from the URL provided.

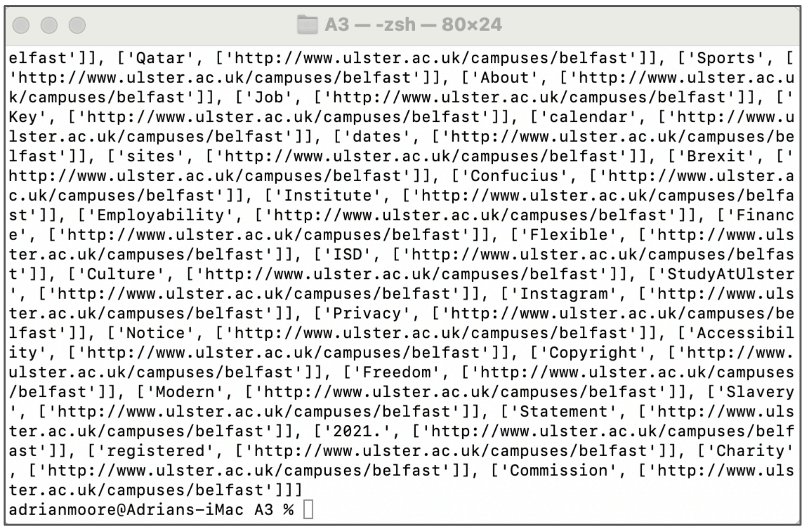
File: A3/add_page_to_index.py

```
def add_word_to_index(index, keyword, url):
    for entry in index:
        if entry[0] == keyword:
            entry[1].append(url)
            return
    index.append([keyword, [url]])

def add_page_to_index(index, url):
    page_words = get_page_text(url)
    for word in page_words:
        add_word_to_index(index, word, url)

index, page_words = [], []
url = "http://www.ulster.ac.uk/campuses/belfast"
add_page_to_index(index, url)
```

Here, we separate the web scraper into its constituent functions:

`get_page_text()`	(previously defined), which scrapes the URL provided as a parameter and returns a list of keywords in the list **`page_words`**
`add_page_to_index()`	which iterates across the **`page_words`** list and presents each in turn to…
`add_word_to_index()`	which accepts a word, a URL and an index and checks to see if the word is already included in the index. If so, then the URL is added to that keywords list of sources. If the keyword is not already included, then a new entry is appended to the index.

Do it now!	Run *add_page_to_index.py* and see how all of the content on the University home page is extracted and displayed as shown in Figure A3.2. Examine the structure of the output and verify that it fits the index structure discussed earlier.



```
elfast']], ['Qatar', ['http://www.ulster.ac.uk/campuses/belfast']], ['Sports', [
'http://www.ulster.ac.uk/campuses/belfast']], ['About', ['http://www.ulster.ac.u
k/campuses/belfast']], ['Job', ['http://www.ulster.ac.uk/campuses/belfast']], ['
Key', ['http://www.ulster.ac.uk/campuses/belfast']], ['calendar', ['http://www.u
lster.ac.uk/campuses/belfast']], ['dates', ['http://www.ulster.ac.uk/campuses/be
lfast']], ['sites', ['http://www.ulster.ac.uk/campuses/belfast']], ['Brexit', ['
http://www.ulster.ac.uk/campuses/belfast']], ['Confucius', ['http://www.ulster.a
c.uk/campuses/belfast']], ['Institute', ['http://www.ulster.ac.uk/campuses/belfa
st']], ['Employability', ['http://www.ulster.ac.uk/campuses/belfast']], ['Financ
e', ['http://www.ulster.ac.uk/campuses/belfast']], ['Flexible', ['http://www.uls
ter.ac.uk/campuses/belfast']], ['ISD', ['http://www.ulster.ac.uk/campuses/belfas
t']], ['Culture', ['http://www.ulster.ac.uk/campuses/belfast']], ['StudyAtUlster
', ['http://www.ulster.ac.uk/campuses/belfast']], ['Instagram', ['http://www.uls
ter.ac.uk/campuses/belfast']], ['Privacy', ['http://www.ulster.ac.uk/campuses/be
lfast']], ['Notice', ['http://www.ulster.ac.uk/campuses/belfast']], ['Accessibil
ity', ['http://www.ulster.ac.uk/campuses/belfast']], ['Copyright', ['http://www.
ulster.ac.uk/campuses/belfast']], ['Freedom', ['http://www.ulster.ac.uk/campuses
/belfast']], ['Modern', ['http://www.ulster.ac.uk/campuses/belfast']], ['Slavery
', ['http://www.ulster.ac.uk/campuses/belfast']], ['Statement', ['http://www.uls
ter.ac.uk/campuses/belfast']], ['2021.', ['http://www.ulster.ac.uk/campuses/belf
ast']], ['registered', ['http://www.ulster.ac.uk/campuses/belfast']], ['Charity'
, ['http://www.ulster.ac.uk/campuses/belfast']], ['Commission', ['http://www.uls
ter.ac.uk/campuses/belfast']]]
adrianmoore@Adrians-iMac A3 % 
```

Figure A3.2. Add Page to index

Try it now!	Modify *add_page_to_index.py* so that data from 3 different URLs is added to the same index. Add some Python code to return the list of all keywords that appear on all three pages.

Between them, our Web Crawler and Web Scraper are able to visit a collection of web pages and retrieve and index their content. However, this effort is pointless without some way of storing our index so that we don't have to generate it from scratch every time we want to perform a search. We could traverse our index list, writing the data to a file – but this would lose the list structure that our code has worked so hard to generate.

Fortunately, Python provides **pickle** – a powerful module that serializes and de-serializes data structures so that they can be written to/read from a text file in a single operation. Consider the code below, which uses **pickle**'s **`dump()`** method to write an index list to a text file, before using the **pickle `load()`** method to read the list back into a Python variable. Note that we use "**`wb`**" and "**`rb`**" as the file mode when opening the file for writing and reading, respectively, rather than the more usual "**`w`**" and "**`r`**". The additional "**`b`**" flag denotes that the information being written and read is a byte stream, rather than simple text.

File: A3/add_page_to_index_pickled.py

```
import pickle

...

print(index)
fout = open("index.txt", "wb")
pickle.dump(index, fout)
fout.close()

print("------------")

fin = open("index.txt", "rb")
new_index = pickle.load(fin)
fin.close()
print(new_index)
```

Do it now! Run *add_page_to_index_pickled.py* in the Python interpreter and verify that the index is successfully written to file and then read in again. Open the *index.txt* file that is generated and examine its structure. Note that the file contents are encoded – pickled files are not designed to be human-readable.

A3.3 Performance and Efficiency

Even from the examples so far, we can appreciate the size of the data structures that are generated by indexing Web scraped content. Before proceeding further with the design and implementation of ***Poodle***, it is necessary to investigate how the performance of the search will be affected by an increasing data set.

The Python **`time`** module provides a way of measuring execution time. Examine the following code, which returns the run time of some Python code passed as a parameter.

File: A3/timed_create_index.py

```
import time

def time_execution(code):
    start = time.process_time()
    eval(code)
    run_time = time.process_time() - start
    return run_time
```

Here, we measure execution time by taking a reading from **`time.process_time()`** immediately before and after execution of the code. The Python function **`eval()`** takes a string of Python code as a parameter, and returns the result that is generated when Python executes this code. For example, the effect of

```
print(time_execution("add_page_to_index(url)"))
```

would be to print the time (in seconds) required to execute the **add_page_to_index()** function.

To analyse the performance of our index structure, we will write code (shown below) to generate a large index where every keyword is different, and then measure the time taken to search the index in the worst-case scenario. By repeating the test with a range of index sizes, we can estimate the future search performance of ***Poodle***.

File: A3/timed_create_index.py

```
def make_string(list_of_letters):
    str = ""
    for e in list_of_letters:
        str = str + e
    return str

def make_big_index(index, size):
    letters = ['a','a','a','a','a','a','a','a']
    while len(index) < size:
        word = make_string(letters)
        add_word_to_index(index, word, "dummyURL")
        for i in range(len(letters) - 1, 0, -1):
            if letters[i] < 'z':
                letters[i] = chr(ord(letters[i]) + 1)
                break
            else:
                letters[i] = 'a'
    return index
```

The Python code above presents a pair of functions that generate an index where each keyword is different. The function **`make_big_index()`** defines a list of letters which is initialised to contain all 'a' characters. Using a **`for`** loop, we iterate across the letters in reverse order, incrementing the letter and storing the new word in the index, until that letter is 'z'. We then move back to the previous letter and repeat the sequence, generating a series such as
'aaaaaaaa', 'aaaaaaab', 'aaaaaaac', 'aaaaaaad', 'aaaaaaae', …, 'aaaaaaay', 'aaaaaaaz', 'aaaaaabz', 'aaaaaacz', …

Note the use of the **`ord()`** and **`chr()`** functions, which convert single letters to and from their ASCII representations, respectively.

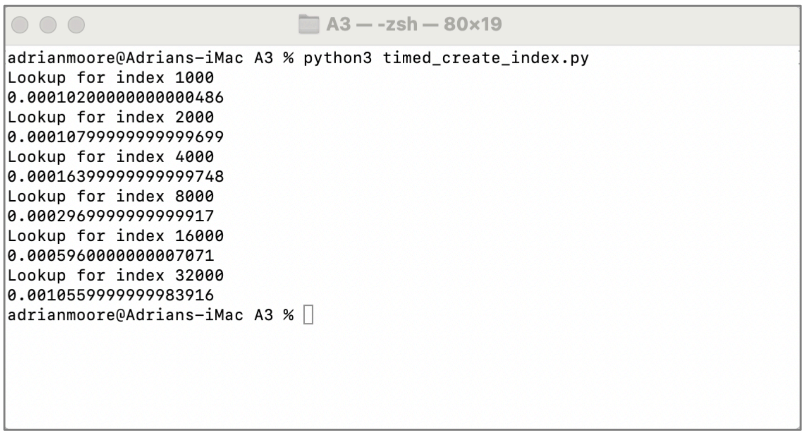
Now that we have our large index of dummy values, we want to simulate a ***Poodle*** search on the index. This is presented by the **`lookup()`** function below, which checks each keyword, and when a match is found, returns the associated list of URLs. We combine this with the **`time_execution()`** function to measure how long it takes to perform a worst-case scenario search – one where the keyword being searched for does not appear in the index.

File: A3/timed_create_index.py

```
def lookup(keyword, index):
    for e in index:
        if e[0] == keyword:
            return e[1]

index = []

make_big_index(index,1000)
print("Lookup for index 1000")
print(time_execution("lookup('xxx', index)"))

make_big_index(index,2000)
print("Lookup for index 2000")
print(time_execution("lookup('xxx', index)"))
```

Do it now! *Run timed_create_index.py* in the Python interpreter and observe the output as presented in Figure A3.3.

Note: The time to create the index is much longer than the time to search it. However, the search time is the real measure of performance in a search engine, and this is the only time component that is measured.



```
adrianmoore@Adrians-iMac A3 % python3 timed_create_index.py
Lookup for index 1000
0.00010200000000000486
Lookup for index 2000
0.00010799999999999699
Lookup for index 4000
0.00016399999999999748
Lookup for index 8000
0.0002969999999999917
Lookup for index 16000
0.0005960000000007071
Lookup for index 32000
0.0010559999999983916
adrianmoore@Adrians-iMac A3 % 
```

Figure A3.3 Timing results for index list lookup

The output illustrated Figure A3.3 presents the results generated for the worst-case scenario search for indexes of 1000, 2000, 4000, 8000, 16000 and 32000 elements. We can see from these results that doubling the number of elements in the index (roughly) doubles the time taken for the worst-case search. This is as expected, since a list search involves visiting each element in turn until the element required is discovered.

Note:	Your results may be slightly different to those illustrated above – and may even vary between different runs of the same code. The exact figures produced will vary according to factors such as operating system housekeeping, other live applications, and other requests for processor resource – although you should be able to observe a rough doubling of the execution time in each case.

Regardless of the precise timings, it is clear that a list is not a suitable structure in which to maintain **Poodle**'s index. We require a data structure where the search time will remain constant (or as close as possible) regardless of the number of elements.

A3.4 Python Dictionary Structures

A dictionary is a data structure that can store any number of Python objects. Dictionaries consist of pairs of keys and their corresponding values.

Do it now!	Open the Python command line interpreter and verify the following sequence of Python dictionary manipulation commands.

First, we will create a new dictionary. Note that dictionaries are enclosed by **{   }** brackets.

```
>>> modules = {"COM668": "Project", "COM661": "Full Stack
Strategies and Development", "COM682": "Cloud Native
Development"}
```

This creates a dictionary containing 3 items, each consisting of a module code as a key and a module title as a value. To retrieve the value associated with any key, we can refer to is as…

```
>>> modules["COM668"]
'Project'
```

If we wish to add a new item to the dictionary, we simply assign a value to its key, e.g.

```
>>> modules["COM662"] = "Data Analytics"
>>> modules
{'COM668': 'Project', 'COM661': 'Full Stack Strategies and
Development', 'COM682': 'Cloud Native Development', 'COM662':
'Data Analytics'}
```

Note: In versions of Python before 3.7, that the order of elements in a dictionary was entirely arbitrary and not governed by the order in which they are added. From v3.7 on, dictionary order is preserved, but it is not considered good practice to depend on it.

Each key in a dictionary can correspond to only one value. If a second value for any key is presented, then it over-writes the previous entry.

```
>>> modules[“COM682”] = “Cloud Development”
>>> modules
{‘COM668’: ‘Project’, ‘COM661’: ’Full Stack Strategies and
Development’, ‘COM682’: ‘Cloud Development’, ‘COM662’: ‘Data
Analytics’}
```

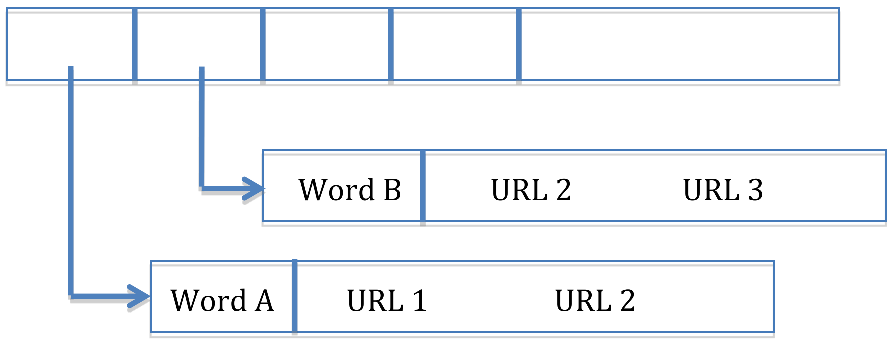
Elements can be removed from a dictionary by the **`del`** command.

```
>>> del modules[“COM662”]
>>> modules
{‘COM668’: ‘Project’, ‘COM661’: ’Full Stack Strategies and
Development’, ‘COM682’: ‘Cloud Development’ }
```

We can also test for the presence of a key using the powerful **`in`** operator.

```
>>> ‘COM662’ in modules
False
>>> ‘COM661’ in modules
True
```

For our ***Poodle*** index, we will replace the list structure by a dictionary where the key value is a keyword scraped from a web page and the value is a list of URLs where the key value is found.



The Python representation of this dictionary is

```
>>> dictionary = {
                  "Word A": [ "URL 1", "URL 2" ],
                  "Word B": [ "URL 2", "URL 3" ]
                }

>>> dictionary["Word 2"]
['URL 2', 'URL 3']
```

To measure the performance of dictionaries, we repeat the worst-case scenario search experiment, by re-defining the **`add_word_to_index()`** function to accept **`index`** as a dictionary rather than a list.

File: A3/timed_create_dictionary.py

```
def add_word_to_index(index, keyword, url):
    if keyword in index:
        index[keyword].append(url)
    else:
        index[keyword] = [url]

...

index = {}
```

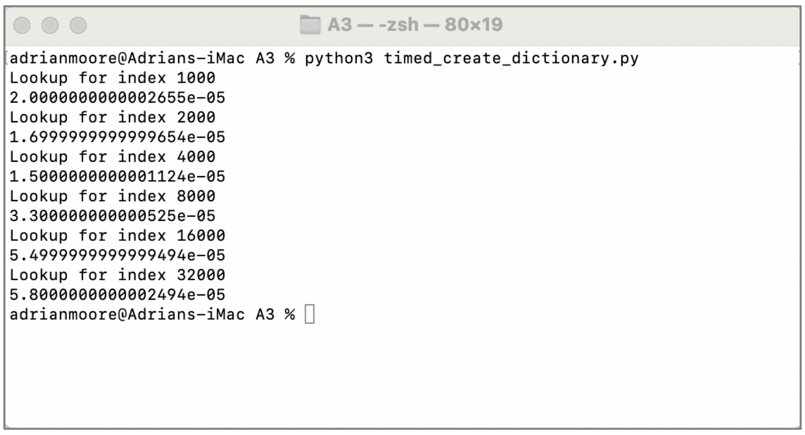
Here, we use the **`in`** operator to test for the presence of the keyword in the **`index`** dictionary. If it exists, we add the URL to the list at that entry. If the keyword is not found, then a new entry is created - where the dictionary value at the new keyword is a list containing the URL at which the word was found.

We also change the **`lookup()`** function to reflect that the **`index`** is stored as a dictionary. Note that we no longer need to traverse across each entry, but can use the **in** operator to check for the presence of a dictionary key value.

File: A3/timed_create_dictionary.py

```
def lookup(keyword, index):
    if keyword in index:
        return index[keyword]
    else:
        return None
```

Do it now!	Run *timed_create_dictionary.py* in the Python interpreter and observe the results generated when the index is represented as a dictionary



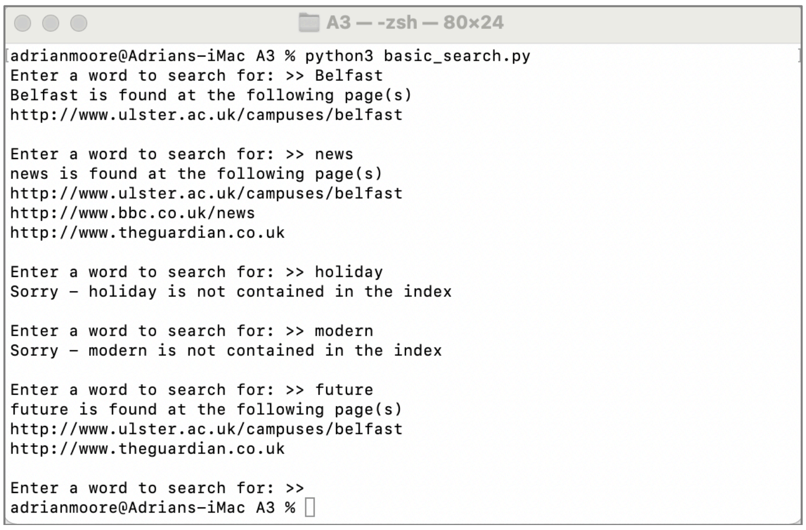
```
adrianmoore@Adrians-iMac A3 % python3 timed_create_dictionary.py
Lookup for index 1000
2.0000000000002655e-05
Lookup for index 2000
1.6999999999999654e-05
Lookup for index 4000
1.5000000000001124e-05
Lookup for index 8000
3.300000000000525e-05
Lookup for index 16000
5.4999999999999494e-05
Lookup for index 32000
5.8000000000002494e-05
adrianmoore@Adrians-iMac A3 %
```

Figure A3.4 Timing results for index dictionary lookup

The results from Figure A3.4 demonstrate that representing the index as a dictionary results in a (roughly) constant search time regardless of the size of the index. Also, the execution is (again, roughly) 10 times quicker than observed in the search of the 1000 element list. There can be no doubt that a dictionary is a much more efficient structure for the organisation of ***Poodle***'s index of web content.

Try it now!	Convert your enhanced Web Scraper application (your version of *add_page_to_index.py* that indexes the contents of 3 source files) so that the index is stored using a dictionary rather than a list. Pickle the dictionary and print its contents to verify the operation of the code.

Try it now!	Write the script *basic_search.py* that reads a ***Pickle***d index (generated from the previous "Try it now!") and prompts the user for a search term as input. The application should print the list of URLs in the index at which that search term was found. If the search term is not contained in the index, the application should print an appropriate message. The user should continue to be prompted for a search term until empty input is provided. Sample output from *basic_search.py* is provided in Figure A3.5. below.



```
adrianmoore@Adrians-iMac A3 % python3 basic_search.py
Enter a word to search for: >> Belfast
Belfast is found at the following page(s)
http://www.ulster.ac.uk/campuses/belfast

Enter a word to search for: >> news
news is found at the following page(s)
http://www.ulster.ac.uk/campuses/belfast
http://www.bbc.co.uk/news
http://www.theguardian.co.uk

Enter a word to search for: >> holiday
Sorry - holiday is not contained in the index

Enter a word to search for: >> modern
Sorry - modern is not contained in the index

Enter a word to search for: >> future
future is found at the following page(s)
http://www.ulster.ac.uk/campuses/belfast
http://www.theguardian.co.uk

Enter a word to search for: >>
adrianmoore@Adrians-iMac A3 %
```

Figure A3.5 Output from Basic Search

A3.5 Further Information

- http://en.wikipedia.org/wiki/Web_scraping
  Wikipedia definition of web scraping

- http://scrapy.org
  Scrapy – an open source Web Scraping framework for Python

- http://en.wikipedia.org/wiki/Search_algorithm
  Search algorithms

- http://docs.python.org/library/pickle.html
  Python Pickle module

- http://wiki.python.org/moin/UsingPickle
  Using Pickle

- https://www.w3schools.com/python/python_lists.asp
  Python Lists

- http://www.tutorialspoint.com/python/python_dictionary.htm
  Python dictionary tutorial

- http://www.i-programmer.info/programming/python/3990-the-python-dictionary.html
  The Python dictionary: i-programmer tutorial

- http://www.greenteapress.com/thinkpython/html/book012.html
  Think Python! Dictionaries