SMALL SEARCH ENGINE

Software Assignment:
PART II: Searching Keywords

Introduction to Python Programming Course
Winter Semester 2011/2012

Submitted by:
Sevtap Fernengel-Sienerth

Submitted to:
Annemarie Friedrich
Stefan Thater

26. January 2012, Saarbrücken

Usage

In order to search the document index for the nyt199501 corpus file, simply type the following at the terminal:

To start a session:

python3.2 indexSearcher.py

This will display the following messages:

```
$ python3.2 indexSearcher.py
************************************************************
* Welcome to the Python Christmas Project, Simple Search Engine *
************************************************************
```

To ask a query:

```
Please enter your search query, press ENTER to quit
Query >> AND(teacher NOT(OR(student students)))
In total 9 documents meet your criteria:

NYT19950117.0366
NYT19950118.0260
NYT19950125.0322
NYT19950125.0323
NYT19950131.0377
NYT19950131.0378
NYT19950131.0448
NYT19950105.0080
NYT19950105.0301

Please enter your search query, press ENTER to quit
Query >> AND(multimedia NOT(megabytes))
In total 10 documents meet your criteria:
...

Please enter your search query, press ENTER to quit
Query >> AND(students)
In total 51 documents meet your criteria:
...
```

To end the session:

```
Please enter your search query, press ENTER to quit
Query >>

*******************************************
* Thanks for using our search engine!      *
* Bye :)                                   *
*******************************************
```

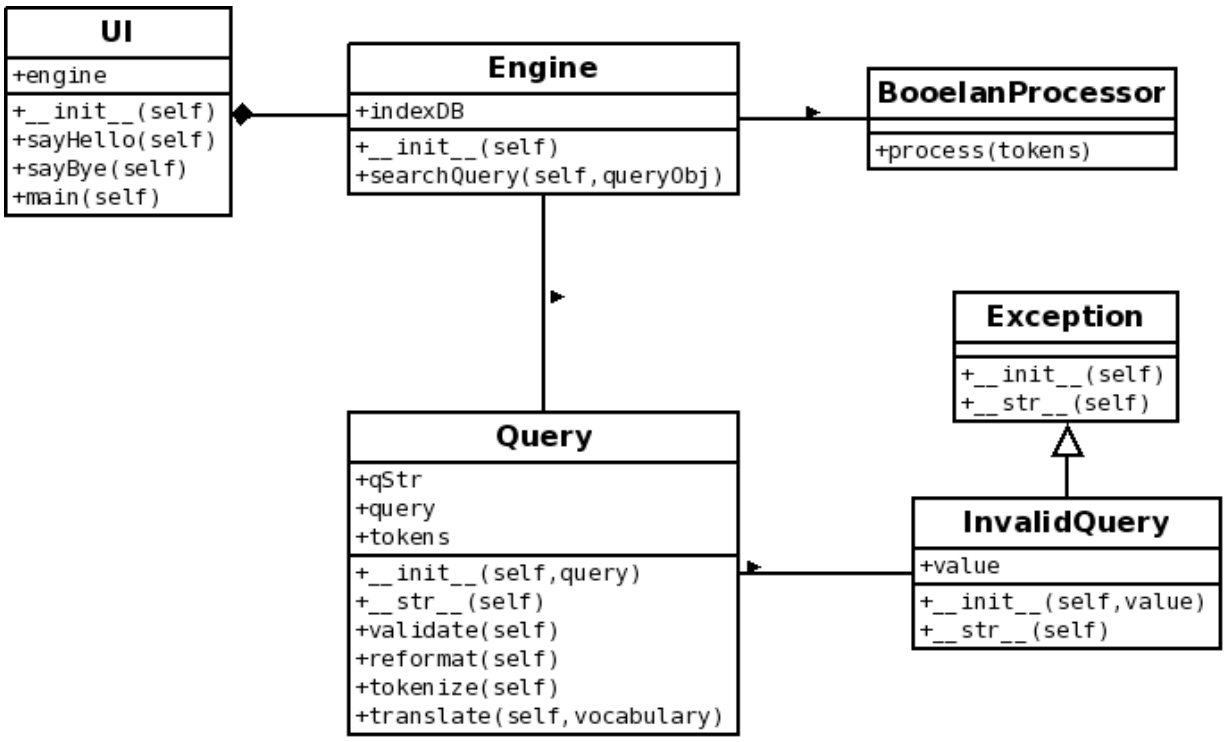
To see the response when search query is wrongly formatted:

```
Please enter your search query, press ENTER to quit
Query >> AND(students0
'Unbalanced parentheses in your query ## AND(students0 ##'

Please enter your search query, press ENTER to quit
Query >> students
Unable to process this query students
```

Class Diagram

Here are the classes and relations among them:



Textual Description

The indexSearcher.py code consists of 5 classes:

1. UI

This class does the main interaction with the user. That involves:

- giving the welcome message
- getting the search query
- validating the search query
- informing user if the query is invalid
- if the query is valid, asking engine to perform the search
- once the search is complete, presenting the results to the user
- at the end, showing the goodbye message to the user

2. Query

This class is used for converting string search query into a well formatted tokens for later processing. It also performs some validation however this validation is very limited that it only catches unbalanced number of parentheses and nothing more.

If a query is valid then it goes through the following stages of processing:

Query:
AND(OR(students teachers) NOT(music))

Reformatting:
In this step, spaces are added around left and right parentheses.

AND (OR (students teachers) NOT (music))

Tokenization:
This is where the string is split into tokens and the resulting list is reversed.

[ ),), music, (, NOT,), teachers, students, (, OR, (, AND]

Translation:
Given a vocabulary like this one {teachers:True, school:True, ...} we replace keywords in the query with True if the keyword exists in the vocabulary, with False otherwise.
Here music and students do not exist in the vocabulary, therefore they are replaced with False, and since the teacher keyword exists in the vocabulary it is replaced with True in the search query.

[ ),), False, (, NOT,), True, False, (, OR, (, AND]

3. Engine

This class has the data we need for searching and also the processor for searching.

Data part:

During class initialization the index.txt file is read into memory as a nested dictionary object. It looks like this:

```
indexDB = {doc_id1: {keyword1:True, keyword2:True, ...}, doc_id2:
{keyword3:True, keyword4:True, ...}, ...}
```

We use this data object heavily to see if a certain keyword exists in a certain document. The way we check that works like this:

First, we obtain the dictionary for a specific document for example, “doc_id1” using `indexDB[doc_id1]` and call this object our vocabulary.

Second, we use the vocabulary object to check if a given keyword, for example, “keyword1”, exists in this vocabulary by using the library function “get” for dictionaries, `vocabulary.get(keyword1, False)`. This function will return `True` if the “keyword1” exists in the vocabulary, `False` otherwise. This method is used in the query translation part.

Searching:

The searching is performed by the searchQuery function of this class. We search through a list of documents represented with their identification numbers, for documents that meet our search criteria. For this we go through the whole list of documents one by one and if a particular keyword index for that document contains the given keywords we add the document identification number to our result list.

Once we are done going through the whole list of available documents we return the list with the document ids back.

To verify if a document meets the search criteria, we use the static function BooleanProcessor.process(). If this function returns True, we add the document to our list.

4. BooleanProcessor

This class consists of one static function, process(). This function knows how to evaluate boolean functions. Those functions can be simple AND(), OR(), NOT() functions with one or more parameters. Also the result of any boolean function can be a parameter for an encompassing boolean function, such as:
AND(True OR(True False)), so to process this query we first evaluate OR(True False) and put the result back into AND(True True) and finally we get True.

Our boolean processor can evaluate nested boolean functions with one or more parameters. Some examples are:

AND(True)
OR(False)
NOT(True)
AND(True False)
OR(False True False)
NOT(True False False False True)
AND(True OR(False NOT(OR(True))) AND(False) NOT(False))

5. InvalidQuery

This class extends the Exception class and is used for stopping the execution of a query if it is ill-formed. In our code, it is only raised when the number of left parentheses does not match the number of right parentheses. In the general case, however, it could be used for other formatting related problems that occur during query validation

Improvement Suggestions

Although the second part of the code implements the required functionality including the deep nested boolean search queries mentioned in the bonus section, there are still points which can be improved if the need arises. These are:

1. A better tokenizer; at the moment, I add spaces around the search query before splitting it into its constituents, better tokenizer however could solve the problem more elegantly by using regular expressions.

2. A better query validation; the validator as it is now, is far from being a complete validator. Therefore if there is any problem with the query which is not caught during this step, will cause the whole query to fail during the searching step, which is not desired. Here is another opportunity to use regular expressions to implement code for full-fledged validation.

3. Faster data structures for searching; the search functionality relies on one nested dictionary and a simple iteration through this dictionary's items to verify if a given document should be in the resulting list. For this project, the queries do not take too long to process, however for a more realistic corpus, it might be a better idea to use other data structures or special databases for querying the index information.

4. Better boolean processor; while evaluating any boolean function with many parameters, sometimes it is sufficient to cut the processing once we know that further processing of the parameters is not going to change the output. Python has any() and all() functions for this purpose. I have not implemented my functions that way, because I need to remove the remaining operands until I reach the matching parentheses. But given some more time, this inefficiency can be easily fixed.

References

The project description document: http://goo.gl/IH3Jq
The Python Documentation: http://docs.python.org/py3k/library/