Implementing a simple inverted index

In this exercise you will implement a simple inverted index in python.

We provide a boolean query parser for you. You can use the parser as follows

In [1]:

```python
from queryparser import parse_query
parse_query('Caesar AND Brutus')
```

Out[1]:

```
AST({'fst': AST({'fst': 'Caesar', 'op': None, 'parseinfo': ParseInfo(buffer=BooleanQueryBuffer@17, rule='notexpr', pos=0, endpos=6, line=0, endline=0)}, 'op': 'AND', 'lst': AST({'fst': 'Brutus', 'op': None, 'parseinfo': ParseInfo(buffer=BooleanQueryBuffer@17, rule='notexpr', pos=11, endpos=17, line=0, endline=1)}, 'op': None, 'parseinfo': ParseInfo(buffer=BooleanQueryBuffer@17, rule='andexpr', pos=0, endpos=17, line=0, endline=1))})
```

Note that the result of `parse_query` is an abstract syntax tree (AST). We provide another method `process_ast`, which will traverse the AST and produce a flattened version for some queries. The parser can deal with any valid boolean query which uses the operators `AND`, `OR`, and `NOT`. However your solution will not have to be able to handle all the possible queries.

To handle errors in the query parser, you can catch `ParseException`. For example, the query `Clown AND OR Circus` is not a well-formed boolean query. To not have your code crash with an unhandled exception you can do something like the following.

In [2]:

```python
from queryparser import ParseException
try:
    ast = parse_query('Clown AND OR Circus')
except ParseException as e:
    print("Failed to parse query:\n", e)
```

Failed to parse query:
```
(1:13) "OR" is a reserved word :
```

```
word
primaryexpr
notexpr
andexpr
expr
start
```

For the purposes of this exercise we provide you with the collected works of Shakespeare as retrieved from [gutenberg.org](http://www.gutenberg.org/files/100/100-0.txt) and split up into individual text files which you can find in `shared/corpus` in the root directory of your Jupyter account (which is your home directory on that machine).
In a future exercise, we will ask you to implement an algorithm for splitting up such a text file into smaller documents yourselves, but for this week, we have done the preprocessing for you.

To iterate over all the text files, for example to add them to your inverted index, you can use python’s `glob` library which allows you to list files using regular Unix shell `glob syntax` (https://en.wikipedia.org/wiki/Glob_(programming)).

In [3]:

```python
import glob
for file in glob.glob("../shared/corpus/*.txt"):
    print(file)
```

```
../shared/corpus/the_two_gentlemen_of_verona.txt
../shared/corpus/king_henry_the_eighth.txt
../shared/corpus/the_tempest.txt
../shared/corpus/the_tragedy_of_julius_caesar.txt
../shared/corpus/the_tragedy_of_king_lear.txt
../shared/corpus/loves_labours_lost.txt
../shared/corpus/king_john.txt
../shared/corpus/the_second_part_of_king_henry_the_sixth.txt
../shared/corpus/the_merchant_of_venice.txt
../shared/corpus/the_life_of_timon_of_athens.txt
../shared/corpus/a_lovers_complaint.txt
../shared/corpus/the_tragedy_of_hamlet_prince_of_denmark.txt
../shared/corpus/cymbeline.txt
../shared/corpus/king_richard_the_second.txt
../shared/corpus/a_midsummer_nights_dream.txt
../shared/corpus/alls_well_that_ends_well.txt
../shared/corpus/king_richard_the_third.txt
../shared/corpus/the_sonnets.txt
../shared/corpus/the_life_of_king_henry_the_fifth.txt
../shared/corpus/the_comedy_of_errors.txt
../shared/corpus/the_tragedy_of_macbeth.txt
../shared/corpus/the_tragedy_of_coriolanus.txt
../shared/corpus/the_tragedy_of_romeo_and_juliet.txt
../shared/corpus/the_passionate_pilgrim.txt
../shared/corpus/the_third_part_of_king_henry_the_sixth.txt
../shared/corpus/the_first_part_of_henry_the_sixth.txt
../shared/corpus/measure_for_measure.txt
../shared/corpus/the_rape_of_lucrece.txt
../shared/corpus/the_merry_wives_of_windsor.txt
../shared/corpus/the_tragedy_of_antony_and_cleopatra.txt
../shared/corpus/the_taming_of_the_shrew.txt
../shared/corpus/the_phoenix_and_the_turtle.txt
../shared/corpus/pericles_prince_of_tyre.txt
../shared/corpus/the_second_part_of_king_henry_the_fourth.txt
../shared/corpus/the_first_part_of_king_henry_the_fourth.txt
../shared/corpus/the_tragedy_of_titus_andronicus.txt
../shared/corpus/the_winters_tale.txt
../shared/corpus/the_two_noble_kinsmen.txt
../shared/corpus/twelfth_night_or_what_you_will.txt
../shared/corpus/the_history_of_troilus_and_cressida.txt
../shared/corpus/as_you_like_it.txt
../shared/corpus/much_ado_about_nothing.txt
../shared/corpus/the_tragedy_of_othello_moor_of_venice.txt
../shared/corpus/venus_and_adonis.txt
```

Now that we have a way to parse a boolean query, and a corpus of documents, what is left to do is to implement a simple inverted index. For this exercise you can assume that the dictionary, and the postings fit
into main memory.

We also provide you with a very simple method which handles both the tokenizing of a document and preprocessing of the resulting tokens. This method will not do much linguistic preprocessing but rather just strips each token of special characters such as quotes, punctuation marks, etc.

We will discuss linguistic preprocessing in more details in future exercises.

Below is a very simple example of how to use the `tokenize_document` function.

In [4]:

```python
from textutils import tokenize_document
for token in tokenize_document('..//shared/corpus/the_phoenix_and_the_turtle.txt):
    print(token)
```

Herald
sad
trumpet
be
To
whose
sound
chaste
wings
obey
But
thou
shrieking
harbinger
Foul
precursor
fiend
Augur
fever’s
end

Now all that is left to do is to implement some code which will create an inverted index. We give you a very simple skeleton which outlines the implementation work that you will have to do.

In the index we want to refer to documents by a document ID. We define variables `documents` which we will assume to be the datastructure mapping document IDs to documents and `the_index` which will be the variable holding the inverted index.

Note that while we currently initialize `the_index` to `None`, you will have to choose a data structure to represent the inverted index, and initialize the variable accordingly.

In [5]:

```python
documents = dict()
the_index = None
```

The easiest way to do this for now, is to simply have a counter which we increment every time we add a document to the index. In order to be able to produce human-readable results you will need to also keep track of which document ID refers to which document. The function we provide to print results expects that `documents` is a `dict` mapping document IDs to document file names.
In [6]:

documentid_counter = 1
def add_document(path):
    '''
    Add a document to the inverted index. Return the document's document ID.
    Remember the mapping from document ID to document in the `documents`
data structure.
    '''
    # make sure that we access the global variables we have defined
global the_index, documents, documentid_counter
    print("Adding '%s' to index" % path)
    pass

Assuming you have implemented `add_document` correctly you should be able to create an inverted index as follows.
In [7]:

```python
import glob
for file in glob.glob('../*/*.txt'):
    add_document(file)
```

Adding '../shared/corpus/the_two_gentlemen_of_verona.txt' to index
Adding '../shared/corpus/king_henry_the_eighth.txt' to index
Adding '../shared/corpus/the_tempest.txt' to index
Adding '../shared/corpus/the_tragedy_of_julius_caesar.txt' to index
Adding '../shared/corpus/the_tragedy_of_king_lear.txt' to index
Adding '../shared/corpus/loves_labours_lost.txt' to index
Adding '../shared/corpus/king_john.txt' to index
Adding '../shared/corpus/the_second_part_of_king_henry_the_sixth.txt' to index
Adding '../shared/corpus/loves_labours_lost.txt' to index
Adding '../shared/corpus/king_john.txt' to index
Adding '../shared/corpus/the_merchant_of_venice.txt' to index
Adding '../shared/corpus/the_life_of_timon_of_athens.txt' to index
Adding '../shared/corpus/a_lovers_complaint.txt' to index
Adding '../shared/corpus/the_tragedy_of_hamlet_prince_of_denmark.txt' to index
Adding '../shared/corpus/cymbeline.txt' to index
Adding '../shared/corpus/king_richard_the_second.txt' to index
Adding '../shared/corpus/a_midsummer_nights_dream.txt' to index
Adding '../shared/corpus/alls_well_that_ends_well.txt' to index
Adding '../shared/corpus/king_richard_the_third.txt' to index
Adding '../shared/corpus/the_sonnets.txt' to index
Adding '../shared/corpus/king_john.txt' to index
Adding '../shared/corpus/the_life_of_king_henry_the_fifth.txt' to index
Adding '../shared/corpus/the_comedy_of_errors.txt' to index
Adding '../shared/corpus/the_tragedy_of_macbeth.txt' to index
Adding '../shared/corpus/the_tragedy_of_coriolanus.txt' to index
Adding '../shared/corpus/the_tragedy_of_romeo_and_juliet.txt' to index
Adding '../shared/corpus/the_passionate_pilgrim.txt' to index
Adding '../shared/corpus/the_third_part_of_king_henry_the_sixth.txt' to index
Adding '../shared/corpus/the_first_part_of_king_henry_the_sixth.txt' to index
Adding '../shared/corpus/measure_for_measure.txt' to index
Adding '../shared/corpus/the_rape_of_lucrece.txt' to index
Adding '../shared/corpus/the_merry_wives_of_windsor.txt' to index
Adding '../shared/corpus/the_tragedy_of_antony_and_cleopatra.txt' to index
Adding '../shared/corpus/the_taming_of_the_shrew.txt' to index
Adding '../shared/corpus/the_phoenix_and_the_turtle.txt' to index
Adding '../shared/corpus/pericles_prince_of_tyre.txt' to index
Adding '../shared/corpus/the_second_part_of_king_henry_the_fourth.txt' to index
Adding '../shared/corpus/the_first_part_of_king_henry_the_fourth.txt' to index
Adding '../shared/corpus/the_tragedy_of_titus_andronicus.txt' to index
Adding '../shared/corpus/the_winters_tale.txt' to index
Adding '../shared/corpus/the_two_noble_kinsmen.txt' to index
Adding '../shared/corpus/twelfth_night_or_what_you_will.txt' to index
Adding '../shared/corpus/the_history_of_troilus_and_cressida.txt' to index
Adding '../shared/corpus/as_you_like_it.txt' to index
Adding '../shared/corpus/much_ado_about_nothing.txt' to index
Adding '../shared/corpus/the_tragedy_of_othello_moor_of_venice.txt' to index
Adding '../shared/corpus/venus_and_adonis.txt' to index

https://informationretrieval.systems.ethz.ch/user/noraho/notebooks/01/inverted_index.ipynb
Now you need to implement a method to process boolean queries. For this exercise it is enough if your query processing can handle simple boolean query with at most one operation. Below is a selection of queries your implementation should be able to handle.

So you do not have to walk the AST you can use the method process_ast which was mentioned earlier to transform the AST into a flattened representation. Additionally we provide a method operation_is_complex, which will return True if the query is more complex than what we reasonably expect you to handle.

In [8]:

```python
from queryparser import process_ast
ast = parse_query('Brutus AND Calpurnia')
print("AST representation:", ast)
flat = process_ast(ast)
print("Flattened representation:", flat)
print(flat.op)
```

AST representation: AST({'fst': AST({{'fst': 'Brutus', 'op': None, 'parseinfo': ParseInfo(buffer=BooleanQueryBuffer@20, rule='notexpr', pos=0, endpos=6, line=0, endline=0)}), 'op': 'AND', 'lst': AST({'fst': 'Calpurnia', 'op': None, 'parseinfo': ParseInfo(buffer=BooleanQueryBuffer@20, rule='notexpr', pos=11, endpos=20, line=0, endline=1)}))}

Flattened representation: AND: args=['Brutus', 'Calpurnia']

AND
['Brutus', 'Calpurnia']

As you can see, it is easier to extract the operation and operands from the flat representation. The flattened representation is composed from Operation objects, and strings. An Operation object is a representation of a boolean query operation. The op field represents the operation type which should be one of AND, OR, NOT, or LOOKUP. The args field is a list of one or more arguments for the operation. NOT and LOOKUP operations are expected to have a argument list of length one. AND, and OR can have arbitrarily long argument lists, but the argument list for such an operation should contain at least two elements. For AND and OR, we represent operands of the form 'NOT ' as the string '-' to allow more efficient execution of queries of the form 'a AND NOT b' and 'a OR NOT b'.

Before directly diving into the full query processing code, you will have to implement methods that compute the intersection and union of two postings lists. We give you method declarations for these operations, but if you need to change the method arguments feel free to do so.

While we do not give hard requirements on the method arguments for either operation, you should probably consider making it a requirement that the postings lists which you pass to intersect and union are sorted, as that will make implementing the functionality significantly easier.
In [9]:
```python
def intersect(p1, p2):
    '''
    Method to compute the intersection of two postings lists. Takes two
    postings lists as arguments and returns the intersection.
    '''
    pass

def union(p1, p2):
    '''
    Method to compute the union of two postings lists. Takes two
    postings lists as arguments and returns the union.
    '''
    pass
```

Below there are a few example calls to `intersect` and `union` which you can use to test your implementation.

In [10]:
```python
print(intersect([1,2,3,6], [1,3,4])) # should produce [1,3]
print(intersect([1,6], [3,4])) # should produce []
print(intersect([1,2,6], [1,6])) # should produce [1,6]
print(intersect([], [1,3])) # should produce []

print(union([1,2], [3,4])) # should produce [1,2,3,4]
print(union([1,2], [1,2,4])) # should produce [1,2,4]
print(union([1,2,6], [2,4,5])) # should produce [1,2,4,5,6]
print(union([], [1,3])) # should produce [1,3]
```

None
None
None
None
None
None
None

Now you should be fairly well-equipped to implement a boolean query processor. Feel free to split your implementation into multiple methods if you find that `execute_query` is getting too big for your taste.

Note that we expect you to implement intersections and unions of more than two terms in the way discussed in the lecture and the book.

Queries that are conjunctions where all terms are negated (i.e. NOT Brutus AND NOT Calpurnia) and queries that are disjunction where at least one term is negated (i.e. Brutus OR NOT Calpurnia OR Hamlet) are often not implemented in real-world, large-scale, general purpose systems. We do not expect you to implement them either, your code can just return None in these cases.
In [11]:

```python
from queryparser import operation_is_complex

def execute_query(query):
    '''
    Execute a boolean query on the inverted index. This method should return a list of document ids which satisfy the query. The list does not need to be in a particular order.
    '''
    try:
        ast = parse_query(query)
    except ParseException as e:
        print("Failed to parse query '{0}':\n" % query, e)

    # flatten query AST
    flat = process_ast(ast)

    if operation_is_complex(flat):
        print("Complex query '{0}'" % query)
        return None

    # TODO: implement query processing
    print("Processing", flat)
    pass
```

If you can process all the queries given below, you have made a good start on building an inverted index and boolean query processor. A good way to improve your query processing is to directly handle `AND NOT`.

You can either try to extend the `intersect` method which you implemented previously, or you can implement `AND NOT` in a separate method.

In order to get human readable output, the following function translates document IDs back to file names, assuming you've correctly populated the `documents` dict.

In [12]:

```python
def print_result(docs):
    if not docs:
        print("No documents found")
        print()
        return

    # If we got some results, print them
    for doc in docs:
        print(documents[doc])

    print()
```
In [13]:

```python
print_result(execute_query('Caesar'))
print_result(execute_query('hello'))
print_result(execute_query('Brutus AND Calpurnia'))
print_result(execute_query('Brutus OR Hamlet'))
print_result(execute_query('Brutus AND NOT Calpurnia'))
print_result(execute_query('Caesar AND Brutus AND Calpurnia'))
print_result(execute_query('Caesar AND Brutus AND NOT Calpurnia'))
```

Processing LOOKUP: args=['Caesar']
No documents found

Processing LOOKUP: args=['hello']
No documents found

Processing AND: args=['Brutus', 'Calpurnia']
No documents found

Processing OR: args=['Brutus', 'Hamlet']
No documents found

Processing AND: args=['Brutus', '-Calpurnia']
No documents found

Processing AND: args=['Caesar', 'Brutus', 'Calpurnia']
No documents found

Processing AND: args=['Caesar', 'Brutus', '-Calpurnia']
No documents found

If you wish, you can try to improve your query processor to handle more complex queries, such as the examples below. This part is optional.

Note that the query parser gives NOT precedence over AND, and AND precedence over OR in the absence of parentheses in queries.

In [14]:

```python
print_result(execute_query('(Caesar OR Brutus) AND Calpurnia'))
print_result(execute_query('(Caesar OR Brutus) AND (Calpurnia OR clown)'))
print_result(execute_query('(Brutus AND Calpurnia) OR (Romeo AND Juliet)'))
print_result(execute_query('(Caesar OR Brutus) AND Calpurnia OR clown')
```

Complex query '(Caesar OR Brutus) AND Calpurnia'
No documents found

Complex query '(Caesar OR Brutus) AND (Calpurnia OR clown)'
No documents found

Complex query '(Brutus AND Calpurnia) OR (Romeo AND Juliet)'
No documents found

Complex query 'Caesar OR Brutus AND Calpurnia'
No documents found

If everything so far was a walk in the park for you, you can even try to implement a method which converts any
query into conjunctive normal form (CNF). Note, however, that converting a boolean formula to CNF is NP hard, and as such we do not require you to do this.