Example code for indexer part 2

```
"""
Indexer part 2 for CS 3308: Information Retrieval

This rather long section of code provides an example of the implementation
of an indexer.   This code implements some, but not all, of the requirements
for the Unit 4 development assignment.  One key element that it does not
implement is blocking of the data.  Your indexer must have the capability
to process very large collections.  This code maintains the entire index in
memory and then writes the entire index to disk at the end.

Your indexer must process no more than 50,000 terms at a time in memory
(as was specifioed in the Unit 2 assignment), write them to disk, and then
be able to process the next 50,000 terms.  To simplify the code, the 50,000
term limit is approximate and you may want to write to disk only at the end
of the current document being indexed.

In this part 2 indexer assignment, there is important new functionality
required to be added to your part 1 assignment.

First you must define a list of between 50 and 100 stop words and develop
the functionality within your program to ignore any term that matches a stop
word.  If you recall, stop words are common words that have little discriminating
value in the index and so we do not include them to improve the processing
efficiency of our inverted index for searches

Second your indexer must perform more editing on the terms including the following:
    Ignore any term that begins with a punctionary character
    Ignore any term that is a number
    Ignore any term that is 2 characters or shorter in length

Third your indexer must compute and store the tf-idf 'vector' calculated
for each term stored in the dictionary

Finally your indexer must implment the data model outlined as part of the assignment

"""
import sys,os,re
import math
import sqlite3
import time


# the database is a simple dictionnary
database = {}

# regular expression for: extract words, extract ID from path, check for hexa value
chars = re.compile(r'\W+')
pattid= re.compile(r'(\d{3})/(\d{3})/(\d{3})')

# the higher ID
tokens = 0
documents = 0
terms = 0


#
# We will create a term object for each unique instance of a term
#
class Term():
    termid = 0
```

```
        termfreq = 0
        docs = 0
        docids = {}

# split on any chars
def splitchars(line) :
        return chars.split(line)

# process the tokens of the source code
def parsetoken(line):
        global documents
        global tokens
        global terms

        # this replaces any tab characters with a space character in the line
        # read from the file
        line = line.replace('\t',' ')
        line = line.strip()

        #
        # This routine splits the contents of the line into tokens
        l = splitchars(line)

        # for each token in the line process
        for elmt in l:
                # This statement removes the newline character if found
                elmt = elmt.replace('\n','')

                # This statement converts all letters to lower case
                lowerElmt = elmt.lower().strip()

                #
                # Increment the counter of the number of tokens processed.  This value will
                # provide the total size of the corpus in terms of the number of terms in the
                # entire collection
                #
                tokens += 1


                # if the term doesn't currently exist in the term dictionary
                # then add the term
                if not (lowerElmt in database.keys()):
                        terms+=1
                        database[lowerElmt] = Term()
                        database[lowerElmt].termid = terms
                        database[lowerElmt].docids = dict()
                        database[lowerElmt].docs = 0

                # if the document is not currently in the postings
                # list for the term then add it
                #
                if not (documents in database[lowerElmt].docids.keys()):
                        database[lowerElmt].docs += 1
                        database[lowerElmt].docids[documents] = 0

                # Increment the counter that tracks the term frequency
                database[lowerElmt].docids[documents] += 1

        return l

#
# Open and read the file line by line, parsing for tokens and processing.  All of the tokenizing
```

```
# is done in the parsetoken() function.  You should design your indexer program keeping the tokenizing
# as a separate function that will process a string as you will be able to reuse code for
# future assignments
#
def process(filename):
	try:
		file = open(filename, 'r')
	except IOError:
		print "Error in file %s" % filename
		return False
	else:
		for l in file.readlines():
			parsetoken(l)
	file.close()

#
# This function will scan through the specified directory structure selecting
# every file for processing by the tokenizer function
# Notices how this is a recursive function in that it calls itself to process
# sub-directories.
#
def walkdir(cur, dirname):
	global documents
	all = {}
	all = [f for f in os.listdir(dirname) if os.path.isdir(os.path.join(dirname, f)) or os.path.isfile(os.path.join(dirname, f))]
	for f in all:
		if os.path.isdir(dirname + '/' + f):
			walkdir(cur, dirname + '/' + f)
		else:
			documents += 1
			cur.execute("insert into DocumentDictionary values (?, ?)", (dirname+'/'+f, documents))
			process(dirname + '/' + f)
	return True


"""
================================================================================================
	>>> main

	This section is the 'main' or starting point of the indexer program. The python
	interpreter will find this 'main' routine and execute it first.
================================================================================================

"""
if __name__ == '__main__':

	#
	# Capture the start time of the routine so that we can determine the total running
	# time required to process the corpus
	#
	t2 = time.localtime()
	print 'Start Time: %.2d:%.2d' % (t2.tm_hour, t2.tm_min)

	#
	# The corpus of documents must be extracted from the zip file and placed into the C:\corpus
	# directory or another directory that you choose.  If you use another directory make sure that
	# you point folder to the appropriate directory.
	#
	folder = "c:/reuters_corpus"

	#
	# Create a sqlite database to hold the inverted index.  The isolation_level statment turns
```

```
# on autocommit which means that changes made in the database are committed automatically
#
con = sqlite3.connect("c:\indexer_part2.db")
con.isolation_level = None
cur = con.cursor()

#
# In the following section three tables and their associated indexes will be created.
# Before we create the table or index we will attempt to drop any existing tables in
# case they exist
#

# Document Dictionary Table
cur.execute("drop table if exists DocumentDictionary")
cur.execute("drop index if exists idxDocumentDictionary")
cur.execute("create table if not exists DocumentDictionary (DocumentName text, DocId int)")
cur.execute("create index if not exists idxDocumentDictionary on DocumentDictionary (DocId)")

# Term Dictionary Table
cur.execute("drop table if exists TermDictionary")
cur.execute("drop index if exists idxTermDictionary")
cur.execute("create table if not exists TermDictionary (Term text, TermId int)")
cur.execute("create index if not exists idxTermDictionary on TermDictionary (TermId)")

# Postings Table
cur.execute("drop table if exists Posting")
cur.execute("drop index if exists idxPosting1")
cur.execute("drop index if exists idxPosting2")
cur.execute("create table if not exists Posting (TermId int, DocId int, tfidf real, docfreq int, termfreq int)")
cur.execute("create index if not exists idxPosting1 on Posting (TermId)")
cur.execute("create index if not exists idxPosting2 on Posting (Docid)")

#
# The walkdir method essentially executes the indexer.  The walkdir method will
# read the corpus directory, Scan all files, parse tokens, and create the inverted index.
#
walkdir(cur, folder)

t2 = time.localtime()
print 'Indexing Complete, write to disk: %.2d:%.2d' % (t2.tm_hour, t2.tm_min)

#
#  Create the inverted index tables.
#
#  Insert a row into the TermDictionary for each unique term along with a termid which is
#  a integer assigned to each term by incrementing an integer
#
#  Insert a row into the posting table for each unique combination of Docid and termid
#
#



#
# Commit changes to the database and close the connection
#
 con.commit()
 con.close()

 #
 # Print processing statistics
 #
```

```
print "Documents %i" % documents
print "Terms %i" % terms
print "Tokens %i" % tokens
t2 = time.localtime()
print 'End Time: %.2d:%.2d' % (t2.tm_hour, t2.tm_min)
```

Porter Stemmer Class Code to use in your indexer

The following section contains code for a Python class that implements a Porter stemmer. As you will recall stemmer routines reduce variability of terms in the term dictionary by attempting to remove the endings of words. If you think about the following words "farm farms farmed and farming" they all have the same root of farm and are all related to each other in meaning. The stemmer upon seeing any of these words would return only the root which is 'farm'. You can include this code directly into your program to provide the stemming functionality that is a requirement. Keep in mind that this code is defined as a Python class which means that you need to create an instance of an object from this class and to use the stemmer you must invoke a method from within the object.

The following shows how the object can be created:

```
p = PorterStemmer()
```

This statement creates an object p that is based upon the porter stemmer class. To use this object you would invoke a method as illustrated below:

```
stemmedWord = p.stem(wordToStem, 0,len(wordToStem)-1)
```

In this example we have a variable wordToStem which contains the term that we want to stem. We invoke the stemmer using the instance of the object that we created, p, and the method stem. The output of this method is the stemmed word. The code follows and you can download this code from the resources section of the course. Please keep in mind that because this is a class definition you must place the entire block of code BEFORE the 'main' section of your program.

```
"""
============================================================================
Porter Stemming Algorithm
This is the Porter stemming algorithm, ported to Python from the
version coded up in ANSI C by the author. It may be be regarded
as canonical, in that it follows the algorithm presented in

Porter, 1980, An algorithm for suffix stripping, Program, Vol. 14,
no. 3, pp 130-137,

only differing from it at the points maked --DEPARTURE-- below.

See also http://www.tartarus.org/~martin/PorterStemmer

The algorithm as described in the paper could be exactly replicated
by adjusting the points of DEPARTURE, but this is barely necessary,
because (a) the points of DEPARTURE are definitely improvements, and
(b) no encoding of the Porter stemmer I have seen is anything like
as exact as this version, even with the points of DEPARTURE!

Vivake Gupta (v@nano.com)

Release 1: January 2001

Further adjustments by Santiago Bruno (bananabruno@gmail.com)
to allow word input not restricted to one word per line, leading
```

to:

release 2: July 2008
==
"""

```
class PorterStemmer:

    def __init__(self):
        """The main part of the stemming algorithm starts here.
        b is a buffer holding a word to be stemmed. The letters are in b[k0],
        b[k0+1] ... ending at b[k]. In fact k0 = 0 in this demo program. k is
        readjusted downwards as the stemming progresses. Zero termination is
        not in fact used in the algorithm.

        Note that only lower case sequences are stemmed. Forcing to lower case
        should be done before stem(...) is called.
        """

        self.b = ""  # buffer for word to be stemmed
        self.k = 0
        self.k0 = 0
        self.j = 0   # j is a general offset into the string

    def cons(self, i):
        """cons(i) is TRUE <=> b[i] is a consonant."""
        if self.b[i] == 'a' or self.b[i] == 'e' or self.b[i] == 'i' or self.b[i] == 'o' or self.b[i] == 'u':
            return 0
        if self.b[i] == 'y':
            if i == self.k0:
                return 1
            else:
                return (not self.cons(i - 1))
        return 1

    def m(self):
        """m() measures the number of consonant sequences between k0 and j.
        if c is a consonant sequence and v a vowel sequence, and <..>
        indicates arbitrary presence,

           <c><v>       gives 0
           <c>vc<v>     gives 1
           <c>vcvc<v>   gives 2
           <c>vcvcvc<v> gives 3
           ....
        """
        n = 0
        i = self.k0
        while 1:
            if i > self.j:
                return n
            if not self.cons(i):
                break
            i = i + 1
        i = i + 1
        while 1:
            while 1:
                if i > self.j:
                    return n
                if self.cons(i):
                    break
                i = i + 1
```

```
        i = i + 1
        n = n + 1
        while 1:
            if i > self.j:
                return n
            if not self.cons(i):
                break
            i = i + 1
        i = i + 1

    def vowelinstem(self):
        """vowelinstem() is TRUE <=> k0,...j contains a vowel"""
        for i in range(self.k0, self.j + 1):
            if not self.cons(i):
                return 1
        return 0

    def doublec(self, j):
        """doublec(j) is TRUE <=> j,(j-1) contain a double consonant."""
        if j < (self.k0 + 1):
            return 0
        if (self.b[j] != self.b[j-1]):
            return 0
        return self.cons(j)

    def cvc(self, i):
        """cvc(i) is TRUE <=> i-2,i-1,i has the form consonant - vowel - consonant
        and also if the second c is not w,x or y. this is used when trying to
        restore an e at the end of a short  e.g.

           cav(e), lov(e), hop(e), crim(e), but
           snow, box, tray.
        """
        if i < (self.k0 + 2) or not self.cons(i) or self.cons(i-1) or not self.cons(i-2):
            return 0
        ch = self.b[i]
        if ch == 'w' or ch == 'x' or ch == 'y':
            return 0
        return 1

    def ends(self, s):
        """ends(s) is TRUE <=> k0,...k ends with the string s."""
        length = len(s)
        if s[length - 1] != self.b[self.k]: # tiny speed-up
            return 0
        if length > (self.k - self.k0 + 1):
            return 0
        if self.b[self.k-length+1:self.k+1] != s:
            return 0
        self.j = self.k - length
        return 1

    def setto(self, s):
        """setto(s) sets (j+1),...k to the characters in the string s, readjusting k."""
        length = len(s)
        self.b = self.b[:self.j+1] + s + self.b[self.j+length+1:]
        self.k = self.j + length

    def r(self, s):
        """r(s) is used further down."""
        if self.m() > 0:
            self.setto(s)
```

```
def step1ab(self):
    """step1ab() gets rid of plurals and -ed or -ing. e.g.

       caresses  ->  caress
       ponies    ->  poni
       ties      ->  ti
       caress    ->  caress
       cats      ->  cat

       feed      ->  feed
       agreed    ->  agree
       disabled  ->  disable

       matting   ->  mat
       mating    ->  mate
       meeting   ->  meet
       milling   ->  mill
       messing   ->  mess

       meetings  ->  meet
    """
    if self.b[self.k] == 's':
        if self.ends("sses"):
            self.k = self.k - 2
        elif self.ends("ies"):
            self.setto("i")
        elif self.b[self.k - 1] != 's':
            self.k = self.k - 1
    if self.ends("eed"):
        if self.m() > 0:
            self.k = self.k - 1
    elif (self.ends("ed") or self.ends("ing")) and self.vowelinstem():
        self.k = self.j
        if self.ends("at"):   self.setto("ate")
        elif self.ends("bl"): self.setto("ble")
        elif self.ends("iz"): self.setto("ize")
        elif self.doublec(self.k):
            self.k = self.k - 1
            ch = self.b[self.k]
            if ch == 'l' or ch == 's' or ch == 'z':
                self.k = self.k + 1
        elif (self.m() == 1 and self.cvc(self.k)):
            self.setto("e")

def step1c(self):
    """step1c() turns terminal y to i when there is another vowel in the stem."""
    if (self.ends("y") and self.vowelinstem()):
        self.b = self.b[:self.k] + 'i' + self.b[self.k+1:]

def step2(self):
    """step2() maps double suffices to single ones.
    so -ization ( = -ize plus -ation) maps to -ize etc. note that the
    string before the suffix must give m() > 0.
    """
    if self.b[self.k - 1] == 'a':
        if self.ends("ational"):   self.r("ate")
        elif self.ends("tional"):  self.r("tion")
    elif self.b[self.k - 1] == 'c':
        if self.ends("enci"):      self.r("ence")
        elif self.ends("anci"):    self.r("ance")
    elif self.b[self.k - 1] == 'e':
```

```
        if self.ends("izer"):      self.r("ize")
    elif self.b[self.k - 1] == 'l':
        if self.ends("bli"):       self.r("ble") # --DEPARTURE--
        # To match the published algorithm, replace this phrase with
        #   if self.ends("abli"):      self.r("able")
        elif self.ends("alli"):    self.r("al")
        elif self.ends("entli"):   self.r("ent")
        elif self.ends("eli"):     self.r("e")
        elif self.ends("ousli"):   self.r("ous")
    elif self.b[self.k - 1] == 'o':
        if self.ends("ization"):   self.r("ize")
        elif self.ends("ation"):   self.r("ate")
        elif self.ends("ator"):    self.r("ate")
    elif self.b[self.k - 1] == 's':
        if self.ends("alism"):     self.r("al")
        elif self.ends("iveness"): self.r("ive")
        elif self.ends("fulness"): self.r("ful")
        elif self.ends("ousness"): self.r("ous")
    elif self.b[self.k - 1] == 't':
        if self.ends("aliti"):     self.r("al")
        elif self.ends("iviti"):   self.r("ive")
        elif self.ends("biliti"):  self.r("ble")
    elif self.b[self.k - 1] == 'g': # --DEPARTURE--
        if self.ends("logi"):      self.r("log")
    # To match the published algorithm, delete this phrase

def step3(self):
    """step3() dels with -ic-, -full, -ness etc. similar strategy to step2."""
    if self.b[self.k] == 'e':
        if self.ends("icate"):     self.r("ic")
        elif self.ends("ative"):   self.r("")
        elif self.ends("alize"):   self.r("al")
    elif self.b[self.k] == 'i':
        if self.ends("iciti"):     self.r("ic")
    elif self.b[self.k] == 'l':
        if self.ends("ical"):      self.r("ic")
        elif self.ends("ful"):     self.r("")
    elif self.b[self.k] == 's':
        if self.ends("ness"):      self.r("")

def step4(self):
    """step4() takes off -ant, -ence etc., in context <c>vcvc<v>."""
    if self.b[self.k - 1] == 'a':
        if self.ends("al"): pass
        else: return
    elif self.b[self.k - 1] == 'c':
        if self.ends("ance"): pass
        elif self.ends("ence"): pass
        else: return
    elif self.b[self.k - 1] == 'e':
        if self.ends("er"): pass
        else: return
    elif self.b[self.k - 1] == 'i':
        if self.ends("ic"): pass
        else: return
    elif self.b[self.k - 1] == 'l':
        if self.ends("able"): pass
        elif self.ends("ible"): pass
        else: return
    elif self.b[self.k - 1] == 'n':
        if self.ends("ant"): pass
        elif self.ends("ement"): pass
```

```
            elif self.ends("ment"): pass
            elif self.ends("ent"): pass
            else: return
        elif self.b[self.k - 1] == 'o':
            if self.ends("ion") and (self.b[self.j] == 's' or self.b[self.j] == 't'): pass
            elif self.ends("ou"): pass
            # takes care of -ous
            else: return
        elif self.b[self.k - 1] == 's':
            if self.ends("ism"): pass
            else: return
        elif self.b[self.k - 1] == 't':
            if self.ends("ate"): pass
            elif self.ends("iti"): pass
            else: return
        elif self.b[self.k - 1] == 'u':
            if self.ends("ous"): pass
            else: return
        elif self.b[self.k - 1] == 'v':
            if self.ends("ive"): pass
            else: return
        elif self.b[self.k - 1] == 'z':
            if self.ends("ize"): pass
            else: return
        else:
            return
        if self.m() > 1:
            self.k = self.j

    def step5(self):
        """step5() removes a final -e if m() > 1, and changes -ll to -l if
        m() > 1.
        """
        self.j = self.k
        if self.b[self.k] == 'e':
            a = self.m()
            if a > 1 or (a == 1 and not self.cvc(self.k-1)):
                self.k = self.k - 1
        if self.b[self.k] == 'l' and self.doublec(self.k) and self.m() > 1:
            self.k = self.k -1

    def stem(self, p, i, j):
        """In stem(p,i,j), p is a char pointer, and the string to be stemmed
        is from p[i] to p[j] inclusive. Typically i is zero and j is the
        offset to the last character of a string, (p[j+1] == '\0'). The
        stemmer adjusts the characters p[i] ... p[j] and returns the new
        end-point of the string, k. Stemming never increases word length, so
        i <= k <= j. To turn the stemmer into a module, declare 'stem' as
        extern, and delete the remainder of this file.
        """
        # copy the parameters into statics
        self.b = p
        self.k = j
        self.k0 = i
        if self.k <= self.k0 + 1:
            return self.b # --DEPARTURE--

        # With this line, strings of length 1 or 2 don't go through the
        # stemming process, although no mention is made of this in the
        # published algorithm. Remove the line to match the published
        # algorithm.
```

```
        self.step1ab()
        self.step1c()
        self.step2()
        self.step3()
        self.step4()
        self.step5()
        return self.b[self.k0:self.k+1]
```

Output Produced Against the CS 3308 Corpus

```
>>>
Start Time: 07:21
Indexing Complete, write to disk: 07:28
Documents 2476
Terms 9167
Tokens 576045
End Time: 09:15
>>>
```