Find one tagging error in each of the following sentences that are tagged with the Penn Treebank tag set:

a. I/PRP need/VBP a/DT flight/NN from/IN Atlanta/NN

>> Atlanta/NN

b. Does/VBZ this/DT flight/NN serve/VB dinner/NNS

>> dinner/NN

c. I/PRP have/VB a/DT friend/NN living/VBG in/IN Denver/NNP

>> have/VBP

d. Can/VBP you/PRP list/VB the/DT nonstop/JJ afternoon/NN flights/NNS

>> Can/MD

Use the Penn Treebank tag set to tag each word in the following sentences from Damon Runyon's short stories. You may ignore punctuation. Some of these are quite difficult; do your best:

a. It is a nice night.
>> It/PRP is/VBZ a/DT nice/JJ night/NN

b. This crap game is over a garage in Fifty-second Street. . .
>> This/DT crap/NN game/NN is/VBZ over/RP a/DT garage/NN in/IN Fifty-second/NNP Street/NNP

c. ...Nobody ever takes the newspapers she sells...
>> Nobody/NN ever/RB takes/VBZ the/DT newspapers/NNS she/PRP sells/VBZ

d. He is a tall, skinny guy with a long, sad, mean-looking kisser, and a mournful voice.

>> He/PRP is/VBZ a/DT tall/JJ skinny/NN guy/NN with/IN a/DT long/JJ sad/JJ mean-loo king/NN kisser/NN and/CC a/DT mournful/JJ voice/NN

e. ...I am sitting in Mindy's restaurant putting on the gefillte fish, which is a dish I am very fond of,...

>> I/PRP am/VBP sitting/VBG in/IN Mindy/NNP restaurant/NN putting/VBG on/IN the/ DT gefillte/NN fish/NN which/WDT is/VBZ a/DT dish/JJ I/PRP am/VBP very/RBfond/NN o f/IN

f. When a guy and a doll get to taking peeks back and forth at each other, why there you are indeed.

>> When/WRB a/DT guy/NN and/CC a/DT doll/NN get/NN to/TO taking/VBG peeks/N NS back/RB and/CC forth/NN at/IN each/DT other/JJ why/WRB there/EX you/PRP are/V BP indeed/RB

Read Norvig (2007) and implement one of the extensions he suggests to his Python noisy channel spell checker.

```
import re
import collections
from itertools import product, imap

VERBOSE = True
vowels = set('aeiouy')
alphabet = set('abcdefghijklmnopqrstuvwxyz')

### IO

def log(*args):
    if VERBOSE: print ''.join([ str(x) for x in args])

def words(text):

    return re.findall('[a-z]+', text.lower())

def train(text, model=None):

    model = collections.defaultdict(lambda: 0) if model is None else model
    for word in words(text):
        model[word] += 1
    return model

def train_from_files(file_list, model=None):
    for f in file_list:
        model = train(file(f).read(), model)
    return model

### UTILITY FUNCTIONS

def numberofdupes(string, idx):

    # "abccdefgh", 2  returns 1
    initial_idx = idx
    last = string[idx]
    while idx+1 < len(string) and string[idx+1] == last:
        idx += 1
    return idx-initial_idx

def hamming_distance(word1, word2):
    if word1 == word2:
        return 0
    dist = sum(imap(str.__ne__, word1[:len(word2)], word2[:len(word1)]))
    dist = max([word2, word1]) if not dist else dist+abs(len(word2)-
len(word1))
    return dist

def frequency(word, word_model):
    return word_model.get(word, 0)

### POSSIBILITIES ANALYSIS
```

```
def variants(word):

    splits     = [(word[:i], word[i:]) for i in range(len(word) + 1)]
    deletes    = [a + b[1:] for a, b in splits if b]
    transposes = [a + b[1] + b[0] + b[2:] for a, b in splits if len(b)>1]
    replaces   = [a + c + b[1:] for a, b in splits for c in alphabet if b]
    inserts    = [a + c + b for a, b in splits for c in alphabet]
    return set(deletes + transposes + replaces + inserts)

def double_variants(word):

    return set(s for w in variants(word) for s in variants(w))

def reductions(word):

    word = list(word)
    # ['h','i', 'i', 'i'] becomes ['h', ['i', 'ii', 'iii']]
    for idx, l in enumerate(word):
        n = numberofdupes(word, idx)
        # if letter appears more than once in a row
        if n:
            # generate a flat list of options ('hhh' becomes
['h','hh','hhh'])
            flat_dupes = [l*(r+1) for r in xrange(n+1)][:3] # only take up to
3, there are no 4 letter repetitions in english
            # remove duplicate letters in original word
            for _ in range(n):
                word.pop(idx+1)
            # replace original letter with flat list
            word[idx] = flat_dupes

    # ['h',['i','ii','iii']] becomes 'hi','hii','hiii'
    for p in product(*word):
        yield ''.join(p)

def vowelswaps(word):

    word = list(word)
    # ['h','i'] becomes ['h', ['a', 'e', 'i', 'o', 'u', 'y']]
    for idx, l in enumerate(word):
        if type(l) == list:
            pass                        # dont mess with the reductions
        elif l in vowels:
            word[idx] = list(vowels)    # if l is a vowel, replace with all
possible vowels

    # ['h',['i','ii','iii']] becomes 'hi','hii','hiii'
    for p in product(*word):
        yield ''.join(p)

def both(word):

    for reduction in reductions(word):
        for variant in vowelswaps(reduction):
            yield variant

### POSSIBILITY CHOOSING
```

```
def suggestions(word, real_words, short_circuit=True):

    word = word.lower()
    if short_circuit:   # setting short_circuit makes the spellchecker much 
faster, but less accurate in some cases
        return ({word}                      & real_words or   #  caps 
"inSIDE" => "inside"
                set(reductions(word))       & real_words or   #  repeats 
"jjoobbb" => "job"
                set(vowelswaps(word))       & real_words or   #  vowels 
"weke" => "wake"
                set(variants(word))         & real_words or   #  other 
"nonster" => "monster"
                set(both(word))             & real_words or   #  both 
"CUNsperrICY" => "conspiracy"
                set(double_variants(word))  & real_words or   #  other 
"nmnster" => "manster"
                {"NO SUGGESTION"})
    else:
        return ({word}                      & real_words or
                (set(reductions(word))  | set(vowelswaps(word)) | 
set(variants(word)) | set(both(word)) | set(double_variants(word))) & 
real_words or
                {"NO SUGGESTION"})

def best(inputted_word, suggestions, word_model=None):


    suggestions = list(suggestions)

    def comparehamm(one, two):
        score1 = hamming_distance(inputted_word, one)
        score2 = hamming_distance(inputted_word, two)
        return cmp(score1, score2)  # lower is better

    def comparefreq(one, two):
        score1 = frequency(one, word_model)
        score2 = frequency(two, word_model)
        return cmp(score2, score1)  # higher is better

    freq_sorted = sorted(suggestions, cmp=comparefreq)[10:]     # take the 
top 10
    hamming_sorted = sorted(suggestions, cmp=comparehamm)[10:]  # take the 
top 10
    print 'FREQ', freq_sorted
    print 'HAM', hamming_sorted
    return ''

if __name__ == '__main__':
    # init the word frequency model with a simple list of all possible words
    word_model = train(file('/usr/share/dict/words').read())
    real_words = set(word_model)

    # add other texts here, they are used to train the word frequency model
    texts = [
        'sherlockholmes.txt',
        'lemmas.txt',
    ]
    # enhance the model with real bodies of english so we know which words
are more common than others
    word_model = train_from_files(texts, word_model)

    log('Total Word Set: ', len(word_model))
    log('Model Precision: %s' %
(float(sum(word_model.values()))/len(word_model)))
    try:
        while True:
            word = str(raw_input('>'))

            possibilities = suggestions(word, real_words,
short_circuit=False)
            short_circuit_result = suggestions(word, real_words,
short_circuit=True)
            if VERBOSE:
                print [(x, word_model[x]) for x in possibilities]
                print best(word, possibilities, word_model)
                print '---'
            print [(x, word_model[x]) for x in short_circuit_result]
            if VERBOSE:
                print best(word, short_circuit_result, word_model)

    except (EOFError, KeyboardInterrupt):
        exit(0)
```