CS 113
Homework Assignment 2

Given: Oct 01, 2009 **Due:** October 22, 2009

The assignment is due by 11:59 p.m. of the due date. The point value of each problem is shown in [ ]. Each solution must be the student's own work. Assistance should only be sought or accepted from the course instructor. Any violation of this rule will be dealt with harshly. Follow the documentation guidelines while writing and documenting your programs. Programs that are not well-documented will be penalized heavily. Your program must work on *all* possible inputs.

---

[70] 1. Rational Numbers. Given below is a sample client program for a class `Rational` that implements fractions. You must implement the `Rational` class so that when the client program given below is compiled with your `Rational` class, the output matches the one listed below. Your class definition must work with other similar client programs too. Your submission must contain a file `rational.py` that contains the `Rational` class definition.

```
import rational

def main():
    first = Rational()
    print ‘first:  ’, first
    second = Rational(4)
    print ‘second:  ’, second
    third = Rational(25,10)
    print "third:
    first += third
    print ‘first:  ’, first
    third *= 8
    print ‘third:  ’, third
    third /= first
    print ‘third:  ’, third
    fourth = Rational(18,158)
    print ‘fourth:  ’, fourth
    third -= fourth
    print ‘third:  ’, third
    fourth = first + second
    print ‘fourth:  ’, fourth

    if fourth == third:
       print ‘The two rationals are the same’
    elif fourth > third:
       print ‘fourth rational is greater than the third rational’
    else:
       print ‘fourth rational is smaller than the third rational’
```

The output of the above client program is as follows.

```
first:   0/1
second:   4/1
third:   5/2
first:   5/2
third:   20/1
third:   8/1
fourth:   9/79
third:   623/79
fourth:   13/2
fourth rational is smaller than the third rational.
```

Note that the numerator and the denominator of the rational numbers must be reduced to their smallest possible values. It may be helpful to implement a function that computes the Greatest Common Divisor of two numbers. You can make this a nested function in the constructor for the `Rational` class.

[130] 2. Spell Checker. This is the first in a series of 2-3 programming projects at the end of which you will have constructed a spell checker – program that checks and possibly corrects spelling errors in documents. In this project you will take a first step, by building a somewhat primitive spell checker. It is my hope that this exercise will give you a glimpse into some of difficulties involved in constructing good programs for checking and correcting spellings.

Overview
As part of this project you will have to write two separate programs: `buildDict` and `spellCheck`. As the name suggests, the program `buildDict` builds a small dictionary of words, while the program `spellCheck` uses this dictionary of words to determine which words in a given document are misspelled. The task that `buildDict` performs can be thought of as a preprocessing step; `buildDict` will be used rarely as compared to `spellCheck`. Once a dictionary has been built, then there is no reason to use `buildDict` again until the performance of `spellCheck` is found to be poor enough to require a modification of the dictionary. The implication is that `buildDict` can expend more time and space resources on its task as compared to `spellCheck`.

Building a Dictionary
Building a good dictionary is a rather hard task and in this first attempt, the dictionary we build, will most likely lead to rather poor performance by `spellCheck`. However, in

subsequent attempts we will improve and enlarge our dictionary. In this project, we will build a dictionary by extracting words from large on-line documents known to be error free. I am making available the following three famous novels in text form:

- Lewis Carroll, Alice in Wonderland: Through The Looking Glass.
- Charles Dickens, A Christmas Carol.
- Robert Louis Stevenson, Dr. Jekyl and Mr. Hyde.

The program `buildDict` should read each of these documents, extract words from them, and insert them into a dictionary. It is important to have a precise understanding of what we mean by a word before we proceed. As far as this project is concerned, a word is a contiguous sequence of 2 or more letters (lower case or upper case) immediately followed by a non-alphabetic character. For this project we will ignore the difference between upper and lower case letters and therefore the words `The` and `the` will be considered identical. To understand the above definition, consider the following text:

```
 The giant telephone company AT&T bought an IBM360 computer last
year.  Since then the company has had an error-free run in
the telecommunications industry.
```

The distinct words extracted from this text are listed below in alphabetical order:

an	computer	had	industry	telecommunications	year
at	error	has	last	telephone	
bought	free	ibm	run	the	
company	giant	in	since	then	

Table 1: Dictionary for the sample text

Notice that the word `at` has been extracted from AT&T and the hyphenated word `error-free` has contributed two words `error` and `free` to the above list. Each of the extracted words needs to be inserted into a data structure that can also be searched quickly (so that the same word is not inserted twice).

The program `buildDict` should begin by asking the user for a list of text files to read from. The user should respond by typing the file names as `[filename1] [white space] [filename2] [white space] [filename3]...` In this version of the project you will be using only three files: `alice.txt`, `carol.txt`, and `hyde.txt`. Then `buildDict` will read all specified text files and build a dictionary of all distinct words found. It should then write the contents of this dictionary into a file called `words.dat`. This file constitutes the dictionary that will subsequently be used by the program `spellCheck`. The file `words.dat` should contain words in alphabetically sorted order.

The spellCheck program

After the `buildDict` program completes its task and creates a dictionary file `words.dat`, the `spellCheck` program is ready to take over. This program should start by reading the list of words in the file `words.dat` into a data structure that can be quickly searched. To support this you should define and implement a class called `Dictionary` that can be used by the `spellCheck` program. In addition to the words in `words.dat`, valid single letter words such as 'A' and 'I' should also be added to the data structure.

After the list of words have been read from `words.dat` it should prompt the user for the name of a document she wants spell-checked. Then `spellCheck` should read words from this user specified document, one-by-one and test if the words appears in its dictionary. Suppose that the user wants a file called `myLetter` to be spell-checked. As `spellCheck` reads from the file `myLetter`, it writes into a file called `myLetter.out`. After spellCheck has completely processed `myLetter`, the file `myLetter.out` will be the corrected version of `myLetter`. Note that `myLetter.out` is identical to `myLetter` except for misspelled words being replaced by correctly spelled words. The `spellCheck` program should make sure that all the white spaces and punctuation marks that appear in `myLetter` appear exactly in the same manner in `myLetter.out`. As the program scans the user specified document, it may encounter a word that is not present in its dictionary. It takes such a word to be misspelled and responds by (i) displaying the misspelled word and (ii) producing the following prompt:

```
Replace(R), replace all(P), ignore(I), ignore all(N), or exit(E)?
```

The user is expected to respond by typing an `R` or `r`, `P` or `p`, `I` or `i`, `N` or `n`, or `E` or `e`. If the user types anything else, the program should simply ask the question again. Here is what your program should do for each of the possible responses from the user.

User input is `R` or `r`. In this case, your program should ask the user to type a replacement word. The program should read the replacement word and replace the current occurance of the misspelled word by the replacement word. If the same misspelled word occurs later in the document, the user needs to be prompted again. For example, if the document `myLetter` contains the word `colour`, then when `spellCheck` encounters this word, it searches for it in its dictionary, does not find it, and asks the user what he/she wants to do. Assuming that the user responds by typing `R` and then types the replacement word `color`, the program writes `color` instead of `colour` in `myLetters.out`.

User input is `P` or `p`. In this case also, your program should ask the user to type a replacement word. The program should read the replacement word and replace all future occurances of the misspelled word by the replacement word. This means that the user will not be prompted when another occurance of the misspelled word is encountered later in the document.

User input is `I` or `i`. In this case, your program should just ignore the misspelled word and move on to the next word. If the same misspelled word occurs later in the document, the user needs to be prompted again.

User input is `N` or `n`. In this case, your program should ignore all future occurances of the misspelled word. This means that the user will not be prompted when another occurance of the misspelled word is encountered later in the document.

User input is `E` or `e`. In this case the program should simply terminate without changing the text file at all from that point.

The actions corresponding to the responses `R`, `I`, and `E` are easy to implement. The actions corresponding to the responses `P` and `N` require additional data structures. For example, if the user asks that all future occurrances of `colour` be replaced by `color`, then the program needs to remember this so that whenever subsequent occurances of the word `colour` are encountered they are replaced by `color`. Similarly, for `N`. These data structures must be maintained by the `Dictionary` class. This organization means that when a word is encountered, the `spellCheck` program should first check if the spelling of the word is to be ignored or if there is a replacement word for it and only if it does not find the word there, it should look in the dictionary. The `Dictionary` class must provide at least the following member functions.

- `init`. The constructor must read the words from the dictionary into a data structure.
- `verify`. This method verifies the word supplied by the `spellCheck` program. If it finds word in the data structures that it maintains then it returns a 'new word' that must be output. Otherwise, it returns `None`.
- `update`. This method updates the data structures in response to the user input of `P` or `p` and `N` or `n`.

Suppose the contents of the file to be spell-checked, say `myLetter` is as follows.

```
Computers fromm IBM are better than those Fromm BMI.
```

The following is a sample execution when we run the program `spellCheck.py` using the dictionary in Table 1.

```
 Name of the document to be spell-checked:  myLetter
                          Computers
replace(R), replace all(P), ignore(I), ignore all(N), exit(E): i

                          fromm
replace(R), replace all(P), ignore(I), ignore all(N), exit(E): G
replace(R), replace all(P), ignore(I), ignore all(N), exit(E): p
Replacement word:  from

                          are
replace(R), replace all(P), ignore(I), ignore all(N), exit(E): I

                          better
replace(R), replace all(P), ignore(I), ignore all(N), exit(E): N

                          than
replace(R), replace all(P), ignore(I), ignore all(N), exit(E): I

                          those
replace(R), replace all(P), ignore(I), ignore all(N), exit(E): n
```

```
                          BMI
replace(R), replace all(P), ignore(I), ignore all(N), exit(E): I
```

After running `spellCheck`, the contents of the file `mytext.out` will be:

```
    Computers from IBM are better than those from BMI.
```

Submission Guidelines. You must create a directory called P2_<last name> under which you must create two more directories for the two programs. The directories must be named "rationalNumbers" and "spellChecker". The "rationalNumbers" directory must contain the file `rational.py` that contains the `Rational` class definition and the client program. The "spellChecker" directory must contain the files `buildDictionary.py`, `dictionary.py` that contains the `Dictionary` class definition, and `spellCheck.py` that contains the spell checker program. Furthermore, this directories should not contain files that are not directly relevant to the problem. Make sure that there are no executable files in the directory.

Before you submit your program, you must create a *compressed*, *tar* file of the directory P2_<last name>. This can be done in two steps as follows.

```
% tar cvf P2_<last name>.tar P2_<last name>
% gzip P2_<last name>.tar
```

The above commands should create a file P2_<last name>.tar.gz . You must e-mail the above file as an attachment to `rajivg@clam.rutgers.edu`. The programs are due by 11:59p.m. of the due date. Hard copies of your programs must be turned in by the end of the first class after the deadline.

Some other commands that may be useful are the following.
Uncompressing a file can be done using the `gunzip` command.

```
% gunzip <file name>.gz
```

Extracting files from an archive can be done as follows.

```
% tar xvf <file name>.tar
```

To list all the files in the archive do the following.

```
% tar tf <file name>.tar
```