CS116 Winter 2011 Assignment 7
Due: Tuesday, March 15 at 10:00AM

The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.

Important Information:

1. Read the course Style Guide for Python (Appendix C in the course notes) for information on assignment policies and how to organize and submit your work. In particular, your solutions should be placed in files a7qY.py, where Y is a value from 1 to 3.
2. Be sure to download the interface file to get started.
3. The interface file contains several strings, which you should use to format your output. Do not change these values or you will lose correctness marks on your assignment.
4. ***Do not use loops in your solutions to these problems. Any repetition should be accomplished with recursion.***
5. You are encouraged to use helper functions in your solutions as needed. Simply include them in the same file with your solution, but do not declare them locally. You should follow the full design recipe for your helper functions (which includes testing).
6. If you use print statements for debugging purposes, be sure to remove them (or comment them out) before you submit your solutions.
7. Do not copy the purpose directly from the assignment description – it should be in your own words.
8. You may assume that all consumed data will satisfy any stated assumptions, unless indicated otherwise.

Language level: Any Python in the range 2.5.0 to 2.7.1. (Do **not** use Python 3.0 or higher)

Coverage: Module 7

Provided files: `a7interface.py`

Question 1: Partial Search

Searching for words in one or more strings is a very common task for computers (you can probably press Ctrl+F right now and your PDF reader will perform this task). In most situations, it's quicker and easier if you only have to type in part of the word you want to search for. For example, searching for "superc" instead of "supercalifragilisticexpialidocious" will save you some time.

a) Write a function `partial_list_search` that will consume a **list of strings** to search and a string representing a search term, `target`. It will produce a list of the strings in the original list that contain `target`.

For example:

```
list_of_words = ['apples','bananas','oranges']
partial_list_search(list_of_words,'banana') => ['bananas']
partial_list_search(list_of_words,'an') => ['bananas','oranges']
partial_list_search(list_of_words,'grapes') => []
partial_list_search(list_of_words,'')
     => ['apples','bananas','oranges']
```

b) Write a function `partial_str_search` that consumes a **string** to search and a string representing a search term, `target`. In the original string, we will consider any consecutive groups of non-whitespace characters to be a word. It will produce a list of the words in the string that contain `target`. In addition:

- The search should be case insensitive – that is, "orange", "OranGe" and "ORANGE" should be treated the same.
- Words may only appear once in the returned list, and should all be in lowercase.
- Calling `partial_str_search` should print out lowercase duplicate words that contain `target` as they are encountered.
    - If a word appears more than twice, print out a duplication message for each time it is encountered after the first instance.

For example:

Calling `partial_str_search('Apples Bananas ORANGES', 'banana')` does not print out anything, and returns the value `['bananas']`.

Calling `partial_str_search('Apples OranGes Bananas ORANGES oranges', 'an')` prints out

```
Duplicate word: oranges
Duplicate word: oranges
```

and returns the value `['oranges','bananas']`.

Question 2: Caesar-cipher revisited

Recall that in assignment 5, we had you implement encryption of a plaintext string and decryption of an encrypted string with the Caesar cipher. Last time you did this in Scheme – now, let's try doing it in Python.

In the Caesar cipher, the encryption process takes each character in the plaintext (which is the message to be encrypted) and replaces it with the character in the alphabet that appears 3 places to the **right** of that character. When "3 places to the right" goes past the end of the alphabet, we wrap around to the start of the alphabet. In this problem, our alphabet consists of the underscore character "_", the period "." and the 26 lower-case Latin characters, in that order. We will refer to this alphabet as $\mathscr{A}$. The encryption process for each character in $\mathscr{A}$ is described as follows:

_	.	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	_	.	a

To decrypt (or undo) the ciphertext, we replace each character in the ciphertext with the character in the alphabet that appears 3 places to the **left** of it. In this case, we wrap around to the end of the alphabet when the ciphertext character is one of the first three characters in the alphabet. The decryption rule for each character in $\mathscr{A}$ is also given in the above table, where we replace each ↓ with ↑ .

a) Write a function `caesar_encrypt`, which consumes a string, `plain`, and produces a string that is the encrypted version of `plain`.

For example,
`caesar_encrypt('hello._goodbye.') => 'khoorcbjrrge.hc'`

b) Write a function `caesar_decrypt`, which consumes a string, `cipher`, and produces a string that is the decrypted version of `cipher`.

For example,
`caesar_decrypt('khoorcbjrrge.hc') => 'hello._goodbye.'`

Note that in Python, the `ord` function will do the same thing that the Scheme function `char->integer` does, and the `chr` function will do the same thing that the Scheme function `integer->char` does. For example, `ord('a') => 97, chr(97) => 'a'`.

You do not have to use `ord` and `chr` if you don't want to; however, as in assignment 5, solutions in which all 28 individual encryption rules are explicitly included in the function will receive no marks.

You are allowed to take your Scheme solution to this problem or the solution provided on the CS116 website and simply translate it into Python. Alternatively, you may create your own solution from scratch; whichever seems more straightforward to you.

Question 3: Working with CSV files

Almost all programs work with data in some sense. Taking data from one program and using it in another can sometimes be a difficult task. Because of this, common formats for representing large amounts of data have been created to facilitate data sharing.

One simple format is called the Comma-Separated Values (CSV) format. In it, values within a row are separated by commas; rows are separated by line breaks.

For example, the following information is in CSV format. The values represent the name of a student, their midterm mark, and their final mark.
Trevor, 87.5, 84.5
Bruce, 73.0, 75.0
Lucy, 96.4, 94.9

Spreadsheet programs like Microsoft Excel and LibreOffice Calc can open files created in this format and represent the data in simple tables. The table produced by the data above would look like the following.

Trevor	87.5	84.5
Bruce	73.0	75.0
Lucy	96.4	94.9

We can represent the CSV format data in Python with a list of strings. We will refer to a list of strings that represent data in CSV format as a *CSV List*.
For example:

```
csv_data = ['Trevor, 87.5, 84.5',
            'Bruce, 73.0, 75.0',
            'Lucy, 96.4, 94.9']
```

Note that a CSV List can be of any length, but the strings inside the list *must* contain the same number of commas – that is, there are the same number of columns in each row, but there can be any number of rows.

We can more easily manipulate this data in Python with a hierarchical list – specifically, a two-dimensional list. We will refer to a list of lists of strings that represents data as a *Nested List*.
For example:

```
list_data = [['Trevor','87.5','84.5'],
             ['Bruce','73.0','75.0'],
             ['Lucy','96.4','94.9']]
```

Note that a Nested List can be of any length, but the lists inside the list *must* contain the same number of strings – that is, there are the same number of columns in each row, but there can be any number of rows.

The Nested List representation allows us to easily access information if we know the row and column that it resides in.

a) Write a function `csv_to_list` that consumes a list of strings in CSV format (a CSV List), and produces a list of lists of strings (a Nested List) that represents the same information.

For example,

```
csv_to_list(csv_data) => [['Trevor','87.5','84.5'],
['Bruce','73.0','75.0'], ['Lucy','96.4','94.9']]
```

b) Write a function `list_to_csv` that consumes a list of lists of strings (a Nested List), and produces a list of strings in CSV format (a CSV List) that represents the same information.

For example,

```
list_to_csv(list_data) =>
  ['Trevor, 87.5, 84.5','Bruce, 73.0, 75.0','Lucy, 96.4, 94.9']
```

c) Write a function `change_value` that consumes a list of lists (a Nested List), two natural numbers `row` and `col` representing the row and column of the value to be changed, and a string `new_val`, the new value. `change_value` does not produce anything, it only changes the list of lists and prints out an informative message.

For example, `change_value(list_data,2,1,'95.0')` returns nothing, but prints out

```
Value at row 2 column 1 changed.
```

and has the effect of changing `list_data`; that is, `list_data[2][1] => '95.0'`

d) A common (and handy!) feature of most spreadsheet programs is to sort information by a particular column. In the example above, it might be beneficial to sort the information alphabetically by the student's name, or in descending order of their midterm marks.

If we are using the nested list representation, we can sort any nested list alphabetically by the ***first*** column (names, in the example above) using the following line of code.

```
list_data.sort(lambda x,y: cmp(x[0], y[0]), reverse=False)
```

`sort` consumes a function that will be used to compare the values in the list. `cmp` is a function built into Python. It compares two values, whether they are integers, string, or anything else, returning -1 if the first argument is smaller, 0 if they are equal, or 1 if the first argument is larger.

Write a function `sort_csv` that consumes a list of strings *in CSV format*, a natural number `sort_index` that indicates the column that we want to sort by, and a boolean `ascending` which denotes if the data should be sorted in ascending order (`ascending=True`) or descending order (`ascending=False`). The function will produce a list of strings *in CSV format* containing the same data, but sorted based on `ascending` and `sort_index`.

For example:

```
sort_csv(csv_data, 0, True) =>
  ['Bruce, 73.0, 75.0','Lucy, 96.4, 94.9','Trevor, 87.5, 84.5']
sort_csv(csv_data, 1, False) =>
  ['Lucy, 96.4, 94.9','Trevor, 87.5, 84.5','Bruce, 73.0, 75.0']
```

You may use your solutions to parts a) and b) in answering this question.