Home Assignment 4

Dictionaries

General instructions:

- Read questions thoroughly and make sure your programs fulfill the required task.
- The assignment should be solved on your own!
- Follow the submission guidelines written on the website. In particular, all questions ought to be submitted in the attached template file ex4_012345678.py, after replacing the digits 012345678 with your 9-digit ID number (Teudat Zehut if you have one, otherwise it's typically a number beginning with 9).
- Submission is due by: see website.
- Task: in this assignment you need to add your code to the given template file. For each of the provided functions, replace the statement "pass" (=no operation) with your implementation. You may add tests calling your functions in the test section of the file (see next point).
- Self testing: for your convenience, the examples shown below appear as test cases in the template file. Each successful test prints True. Naturally, these examples don't cover all possible cases so you need to test your code on various inputs to verify that the output is correct and that your program never crashes.
- We often use automated testing of assignments, so your outputs must match the required format **exactly** (whitespace and capitalization matter!)
  Follow the output format provided in the examples.
- Do not remove comments from the template file.
- Do not change any variable or function name in the template file.
  Your code may use additional variables and functions, if needed.
- Unless stated otherwise, you may suppose that the input received by the functions is correct.
- You are not allowed to use external modules (e.g., numpy) in this exercise.

Question 1

Implement the function `most_popular_character(my_string)`, which gets the string argument `my_string` and returns its most frequent letter. In case of a tie, break it by returning the letter of smaller ASCII value.

Note that lowercase and uppercase letters are considered different (e.g., 'A' < 'a').

You may assume `my_string` consists of English letters only, and is not empty.

Example 1:

```
>>> most_popular_character("HelloWorld")
'l'
```

Explanation: the letter *ell* occurs 3 times, the letter *oh* appears twice, and each other letter appears once. Thus, ell is the most frequent.

Example 2:

```
>>> most_popular_character("gggcccbb")
'c'
```

Explanation: *cee* and *gee* appear three times each (and bee twice), but *cee* precedes *gee* lexicographically.

Hints (you may ignore these):

- Build a dictionary mapping letters to their frequency;
- Find the largest frequency;
- Find the smallest letter having that frequency.

Question 2

In recitation we saw a dictionary-based implementation of a sparse matrix. In such a representation, for each non-zero matrix entry $A_{ij} = x$ we keep in the dictionary the mapping from the key `i,j` (a tuple) to the value `x`.

Implement the function `diff_sparse_matrices(lst)` which gets a list `lst` of dictionaries, each representing a sparse matrix, and returns a dictionary representing the difference matrix.

Notes:

- We may assume `lst` contains at least two matrices (each represented as a dictionary).
- When `lst` contains more than two matrices, you should subtract from the first one all the others; e.g., for `lst = [A, B, C]` you should return a representation of $A - B - C$.
- All given matrices are of the same dimensions (which is not given to you as a parameter).
- Each matrix may have a different number of non-zero entries, so the dictionaries representing them may be of different lengths.

Example 1:

```
>>> diff_sparse_matrices([{(1, 3): 2, (2, 7): 1},
                          {(1, 3): 6}])
{(1, 3): -4, (2, 7): 1}
```

Example 2:

```
>>> diff_sparse_matrices([{(1, 3): 2, (2, 7): 1},
                          {(1, 3): 2}])
{(2, 7): 1}
```

Explanation: $(A - B)_{13} = 2 - 2 = 0$ so it is not stored explicitly in the dictionary.

Example 3:

```
>>> diff_sparse_matrices([{(1, 3): 2, (2, 7): 1},
                          {(1, 3): 6, (9, 10): 7},
                          {(2, 7): 0.5, (4, 2): 10}])
{(1, 3): -4, (2, 7): 0.5, (9, 10): -7, (4,2): -10}
```

Question 3

Implement a function `find_substring_locations(s, k)` that gets a string `s` and an integer `k`, representing a desired length of substrings, and returns the following dictionary.

Keys are all length-`k` substrings (=contiguous subsequences) of `s`. Each key `t` is mapped to a list of offsets such that `t` occurs in `s` at these offsets. For example, "`ge`" appears at offset 2 in "`drge`". All offsets are non-negative.

Note: the function should handle `k` between `1` and `len(s)`, inclusive.

Example 1:

```
>>> find_substring_locations('TTAATTAGGGGCGC', 2)
{'TT': [0, 4], 'TA': [1, 5], 'AA': [2], 'AT': [3], 'AG': [6],
'GG': [7, 8, 9], 'GC': [10, 12], 'CG': [11]}
```

Example 2:

```
>>> find_substring_locations('TTAATTAGGGGCGC', 3)
{'TTA': [0, 4], 'TAA': [1], 'AAT': [2], 'ATT': [3], 'TAG': [5],
'AGG': [6], 'GGG': [7, 8], 'GGC': [9], 'GCG': [10], 'CGC': [11]}
```

Example 3:

```
>>> find_substring_locations('Hello World', 3)
{'Hel': [0], 'ell': [1], 'llo': [2], 'lo ': [3], 'o W': [4],
' Wo': [5], 'Wor': [6], 'orl': [7], 'rld': [8]}
```

Question 4

a) Implement a function `courses_per_student(tuples_lst)`, whose input is a list of tuples `tuples_lst`. Each tuple in it consists of two strings: the 1st is a name of a student, and the 2nd is a name of a course this student took. The function returns a dictionary mapping each student name (i.e., these are the keys) to the list of the courses she took.

Notes:

- You may assume `tuples_lst` is non-empty.
- Students who took more than one course will appear in multiple tuples of `tuples_lst`.
- You may assume tuples are all distinct (i.e., same student and same course won't appear twice).
- Courses may appear in any order within the output lists.
- Names (of students and/or courses) may appear in mixed case; you should convert everything to lowercase.

Example:

```
>>> stud_dict = courses_per_student([('Tom', 'Math'),
    ('Oxana', 'Chemistry'), ('Scoobydoo', 'python'),
    ('Tom', 'pYthon'), ('Oxana', 'biology')])
>>> stud_dict
{'tom': ['math', 'python'], 'oxana': ['chemistry', 'biology'],
'scoobydoo': ['python']}
```

b) Implement a function `num_courses_per_student(stud_dict)`, which gets a dictionary `stud_dict`, assumed to be returned by the function `courses_per_student` from part a) above. The function **modifies** the given dictionary such that the new value associated with each key (=student name) would be the **number** of courses she took.
Note: the function should **not** return a dictionary!

You may assume the input dictionary is non-empty.

Example:

```
>>> num_courses_per_student(stud_dict)    # see example above
>>> stud_dict
{'tom': 2, 'oxana': 2, 'scoobydoo': 1}
```

Good luck!