Homework 6: Args, Kwargs and Recursion

Total Points: 100 (Correctness and Style) + 3 EC (Checkpoint)

Due Date:

- **Entire Assignment: Tuesday, February 18th, 11:59pm**
- **Checkpoint (read below): Sunday, February 16th, 11:59pm**

Starter Files

Download hw06.py.zip Inside the archive, you will find starter files for the questions of this homework. You cannot import anything to solve the problems.

Part 1. Coding and Docstring Style

Link to the requirements (I moved it to the separate page from now on).

Checkpoint submission + redemption

Due Date: Sunday, February 14th, 11:59pm (SD time)

You can earn **3 points extra credit** by submitting the checkpoint by the due date above. In the checkpoint submission, you should complete **Question 1, 2** and submit ***hw06.py*** file only (without the *outfiles/* folder) to gradescope. Checkpoint submission is graded by completion, which means you can get full points if your code can pass some simple sanity check (no tests against edge cases). Note that in your final submission, you should still submit these questions, and you may modify your implementation if you noticed any errors.

1. **DO NOT IMPORT ANY PACKAGES.**
2. Please add your own doctests **(at least three)** as the given doctests are not sufficient. You will be graded on the doctests.
3. No assert statements are required for all questions. You can assume that all arguments will be valid.
4. You must solve questions 3, 4 and 5 with **RECURSION (no loops!)**. You are not allowed to create inner functions or lambda functions. This implies that you must make recursive calls to the function itself, instead of creating recursive inner functions. (Side note: creating recursive inner functions is a useful trick since it gives you more room to customize the arguments and separate recursive logic from other parts. However, we banned this trick for this homework since we want you to practice more under restricted settings.)

Question 1:

Write a function that takes a series of arguments and returns a dictionary where the keys are the data types and the values are lists of items, organized according to the rules below.

If the type is a:

- `string`: keep the characters at the even indices of the string, i.e. (0th, 2nd, 4th, 6th index and so on…)
- `int`: if even cast to *True*, if odd cast to *False*.
- `float`: if negative, convert to equivalent positive value, if non-negative, change it into `int` by cutting off everything after the decimal.
- `list`: use its length as a value for a corresponding dictionary list.
- Anything else: key is `'garbage'`, and use unchanged arguments as values for a corresponding dictionary list.

Example:

```
>>> randomize(1, 2.3, False, 'DSC20')
{'int': [False], 'float': [2], 'garbage': [False], 'str': ['DC0']}
```

Arg	Output	Cumulative output	Explanation
1	False	`{'int': [False]}`	1 is odd
2.3	2	`{'int': [False], 'float': [2]}`	2.3 is positive, changed to 2
False	False	`{'int': [False], 'float': [2], 'garbage': [False]}`	Not a string, int, float or list
'DSC20'	'DC0'	`{'int': [False], 'float': [2], 'garbage': [False], 'str': ['DC0']}`	Characters at even indices

(new edit): If you solve it using recursion, EC will be awarded. But recursion is not required.

```
def randomize(*args):
    """
     >>> randomize(1, 2.3, False, 'DSC20')
```

```
    {'int': [False], 'float': [2], 'garbage': [False], 'str': ['DC0']}
    >>> randomize(True, 4, 'ABC', -9.8, [1,2,3], 'a', False)
    {'garbage': [True, False], 'int': [True], 'str': ['AC', 'a']\
, 'float': [9.8], 'list': [3]}
     >>> randomize(False, True, 'DS', True, 'abc', -3.2, 5, {'a': 1},
-2, ' .')
     {'garbage': [False, True, True, {'a': 1}], 'str': ['D', 'ac', '
']\
, 'float': [3.2], 'int': [False, True]}
    >>> randomize()
    {}
    >>> randomize(True)
    {'garbage': [True]}
    """
```

Question 2:

Write a function that combines the positional arguments (*args) and keyword arguments (**kwargs) into a list of tuples. Each tuple in the output should include:

- The type of argument (positional or keyword),
- The position of the argument within *args or **kwargs (using 0-based indexing),
- The value held by the argument.

The specific format is shown below:

Example:

Input: `rearrange_args(10, False, player1 = [25, 30], player2 = [5, 50])`

The first two arguments (in blue) are positional, so the corresponding tuples are:

- `('positional_0', 10)  # 10 is at the 0th place`
- `('positional_1', False)  #First is at the 1st place`

The last two arguments are keyword arguments (in red), and the output tuples are:

- ("keyword_0_player1", [25, 30])
    - indicates the first keyword argument (at index 0) is called player1 and holds the value [25, 30]
- ('keyword_1_player2', [5, 50])
    - indicates the second keyword argument (at index 1) is called player2 and holds the value [5, 50]

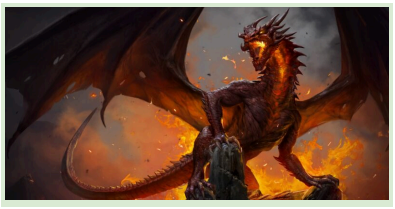
Hints:

- It may be easier to take items from the back of each list instead of the front.
- You can use f-string to build the string at each level for cleaner code.
- `enumerate` might be useful here

(new edit): If you solve it using recursion, EC will be awarded. But recursion is not required.

```
def rearrange_args(*args, **kwargs):
    """
    >>> rearrange_args(10, False, player1=[25, 30], player2=[5, 50])
    [('positional_0', 10), ('positional_1', False), \
('keyword_0_player1', [25, 30]), ('keyword_1_player2', [5, 50])]
    >>> rearrange_args('L', 'A', 'N', 'G', L='O', I='S')
    [('positional_0', 'L'), ('positional_1', 'A'), ('positional_2',
'N'), \
('positional_3', 'G'), ('keyword_0_L', 'O'), ('keyword_1_I', 'S')]
    >>> rearrange_args(no_positional=True)
    [('keyword_0_no_positional', True)]
    """
    # YOUR CODE GOES HERE #
```

Question 3.1:



You were so impressed to learn that there is a dragon at the HDSI building that you decided to check her out. Unfortunately, the entrance is password protected.

Please write a **recursive** function that takes a list of strings as the first parameter and a string `password` as the second parameter. This function must **recursively** compute the number of times the `password` appears in the list.

Requirement: Your function must be recursive! No loops!

```
def count_the_password(lst, password):
    """
    >>> count_the_password(["cooldragon", "dragon", "gold"], "dragon")
    1
    >>> count_the_password(["DRAGON", "dragon!!"], "dragon")
    0
    >>> count_the_password([], "dragon")
```

```
    0
    >>> count_the_password(["dragon "], "dragon")
    0
    >>> count_the_password(["dragon", "likes", "recursions", "right", \
"dragon", "?"],"dragon")
    2
    """
```

Question 3.2:

The dragon does not like visitors. She knows how to locate suspicious lists containing passwords and wants to corrupt them so that the information becomes unreadable.

Write a **recursive** function that takes in a single string, a character `to_insert` and returns a corrupted new string where each character is followed by a `to_insert` character.

Requirement: Your function must be recursive! No loops!

```
def corrupt_password(input, to_insert):
     """
     >>> corrupt_password('dragon', '#')
    'd#r#a#g#o#n#'
    >>> corrupt_password('', '@')
    ''
    >>> corrupt_password('I can help', '-')
    'I- -c-a-n- -h-e-l-p-'
     """
```

Question 3.3:

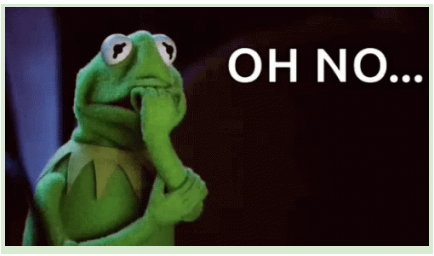
You decided to outsmart the dragon! Write a **recursive** function that takes a list of strings, a password to look for and an element to insert. Your function should only corrupt the strings that do not match a given *password* but leaves the *password* untouched. See doctests for more examples.

Requirement: Your function must be recursive! No loops!

```
def outsmart_dragon(lst, password, to_insert):
    """
    >>> outsmart_dragon(['dragon'], 'dragon','#')
```

```
    ['dragon']
    >>> outsmart_dragon([], 'dragon','@')
    []
    >>> outsmart_dragon(['help me', 'dragon'], 'dragon','-')
    ['h-e-l-p- -m-e-', 'dragon']
    >>> outsmart_dragon(['help me', 'dear dragon'], 'dragon','-')
    ['h-e-l-p- -m-e-', 'd-e-a-r- -d-r-a-g-o-n-']
    >>> outsmart_dragon(['DrAgOn', 'Dragon'], 'dragon','-')
    ['D-r-A-g-O-n-', 'D-r-a-g-o-n-']
    """
```

Question 4:



Your corruption method worked for a few days but…but the smart dragon guessed your trick and created another password corruption.

Write a **recursive** function that removes vowels from an input string. You can assume input is always string. Vowels are the letters (a, e, i, o, u) . Not case-sensitive.

Requirement: Your function must be recursive! No loops!

```
def corrupt_with_vowels(input):
    """
    >>> corrupt_with_vowels('buy and sell')
    'by nd sll'
    >>> corrupt_with_vowels('gold gold gold')
    'gld gld gld'
    >>> corrupt_with_vowels('AeI oU')
    ' '
    """
```

Question 5:

Phew, finally you were able to get it and it is time for you to meet the Dragon. Unfortunately, the place is hard to navigate around so you decided to visit every room in order.

Write a **recursive** function that takes three parameters:

- integers *point1* and *point2*,
- string *separator*

then it returns a string with **all** integers between *point1* and *point2* (both ends are included) separated by a third parameter.

- When *point1* < *point2*, then the numbers in the string are in *ascending* order.
- When *point1* > *point2*, then the numbers in the string are in *descending* order.
- When *point1* == *point2*, just return the string representation of the bound itself.

Requirement: Your function must be recursive! No loops!

Hint:

- Solve for one case (*point1* < *point2* or *point1* > *point2*) first. Then, you only need to change your implementation slightly to cover the other case.

```
def where_to_go(point1, point2, separator):
    """
    >>> where_to_go(17, 17, 'left')
    '17'
    >>> where_to_go(1, 8, ',')
    '1,2,3,4,5,6,7,8'
    >>> where_to_go(8, 1, '->')
    '8->7->6->5->4->3->2->1'
    """
```

Submission

Please submit the homework via Gradescope. You may submit more than once before the deadline, and only the final submission will be graded. Please refer to the Gradescope FAQ if you have any issues with your submission.