CSE 101: Introduction to Computational and Algorithmic Thinking

Homework #3-3

Fall 2018

Assignment Due: Monday, December 10, 2018, by 11:59 PM

Getting Started

This assignment contains a "bare bones" file named `manipulation.py`. This file contains one or more function and/or class *stubs* and a few tests you can try out to see if your code seems to be correct. You must fill in the missing code as directed in order to complete the assignment. **Do not, under any circumstances, change the names of any classes or functions (including those functions' parameter lists) that we provide in the starter code.** The automated grading system will be looking for exactly those class and function names (you are more than free to add your own helper functions and classes if you desire). **Please note that the test cases and data files that we give you are not necessarily the same ones we will use during grading!**

Directions

This assignment is worth a total of 16 points. The automated grading system will execute each function in your solution several times, using different input values each time. Each test that produces the correct/expected output will earn 1 or more points, adding up to the total point value available for that function. You must earn at least 12 points (75%) in order to pass (receive credit for) this assignment.

This assignment counts towards Course Outcome 3: Using Algorithms to Develop and Express Solutions to Computational Problems.

- At the top of the `manipulation.py` file, include the following information **in full-line comments**, with each item on a separate line:
  - your full name ***AS IT APPEARS IN BLACKBOARD***
  - your Stony Brook ID #
  - your Stony Brook NetID (your Blackboard username)
  - the course number (CSE 101) and semester (Fall 2018)
  - the assignment name and number (Assignment #3-3)

⚠ Any functions that we tell you to complete (i.e., that we provide stubs for in the starter code) **MUST** use the names and parameter lists indicated in the starter code file. Submissions with different function names (or whose functions contain different parameter lists) can't be graded by our automated grading system.

⚠ Be sure to submit your final work as directed by the indicated due date and time. Late work will not be accepted for grading. Work is late if it is submitted after the due date and time.

⚠ Programs that crash will likely receive a grade of zero, so make sure you thoroughly test your work before submitting it.

Sequence Manipulation: Strings and Recursion

Part I: An Enhanced `split()` Function (8 points)

We've already seen and worked with the Python string method `split()`, which allows us to divide a string into parts based on a single delimiter string. Our goal with this problem is to enhance this functionality by developing a new function that can divide a string into parts using a set of multiple delimiters; each (single-character) delimiter is equally capable of breaking up the original string. Our function, named `enhancedSplit()`, takes three arguments: a string to process, a list of strings representing valid delimiter characters, and a Boolean value indicating whether the delimiters should be returned in addition to the pieces of the original string. The function returns a list of string pieces, possible including the delimiter characters

For example, consider the string `"when in the course of human events"` and the delimiter list `["e", "t", "u"]`. Our function will break up the source string whenever it encounters one of the characters in the delimiter list, giving us the list of substrings `["wh", "n in ", "h", " co", "rs", " of h", "man ", "v", "n", "s"]` — every 'e', 't', and 'u' character has been removed and used to separate the original string into smaller substrings (if we chose to include the delimiters in the result, our list would be `["wh", "e", "n in ", "t", "h", "e", " co", "u", "rs", "e", " of h", "u", "man ", "e", "v", "e", "n", "t", "s"]` instead).

As a general strategy, consider the following:

- Create three variables: an empty list to hold the end product and two empty strings (one for the current set of valid (non-delimiter) characters and one for the last delimiter seen).
- For each character in the source string:
    - If the current character is a delimiter:
        - If your "valid characters" variable contains at least one character, append it to the result list.
        - Reset the "valid characters" variable to the empty string
        - Set your "delimiter characters" variable to the current character
    - Otherwise:
        - If you are including delimiter characters and your "delimiter characters" variable contains at least one character:
            - Append the "current delimiters" variable to your result list
            - Reset your "current delimiters" variable to the empty string
        - Append the current character to your "valid characters" variable
- After your main loop ends, check to see if you still have any non-delimiter characters that haven't been added to your list. If so, append that string to your list.
- Finally, return your list of substrings

NOTE: Consecutive delimiter characters will be listed as a single substring in the list of pieces (if delimiters are included in the result).

Examples:

Function Call	Return Value
`enhancedSplit("repetition", [], False)`	`["repetition"]`
`enhancedSplit("repetition", ["e"], False)`	`["r", "p", "tition"]`
`enhancedSplit("i am the very model of a modern major-general", ["p"], False)`	`["i am the very model of a modern major-general"]`
`enhancedSplit("why is the sun yellow and not blue", ["m", "x", "z"], False)`	`["why is the sun yellow and not blue"]`
`enhancedSplit("vampires and werewolves are mainly nocturnal creatures", ["a", "m", "n", "x", "z"], False)`	`["v", "pires ", "d werewolves ", "re ", "i", "ly ", "octur", "l cre", "tures"]`
`enhancedSplit("repetition", [], True)`	`["repetition"]`
`enhancedSplit("repetition", ["e"], True)`	`["r", "e", "p", "e", "tition"]`
`enhancedSplit("i am the very model of a modern major-general", ["p"], True)`	`["i am the very model of a modern major-general"]`
`enhancedSplit("why is the sun yellow and not blue", ["m", "x", "z"], True)`	`["why is the sun yellow and not blue"]`
`enhancedSplit("vampires and werewolves are mainly nocturnal creatures", ["a", "m", "n", "x", "z"], True)`	`["v", "am", "pires ", "an", "d werewolves ", "a", "re ", "ma", "i", "n", "ly ", "n", "octur", "na", "l cre", "a", "tures"]`

Part II: Recursively Flattening Nested Lists (8 points)

Recall that a Python list can contain any type of element, even another list. As a result, we can have lists of lists, lists of lists of lists, and so on, with ever-increasing levels of nesting. This problem intends to put a stop to that sort of nonsense.

Complete the `flatten()` function, which takes a single list as its argument. The function examines its list argument and **RECURSIVELY**[1] generates (and returns) a new single-level (non-nested) list that contains every element from the original list, in the same order it appeared in that list. For example, the list [2, [4, 6], 8] contains a list as one of its elements; `flatten()` will process this list and return the new single-level list [2, 4, 6, 8], where the inner list has been eliminated and its elements have replaced the original sublist.

One possible recursive strategy resembles the following:

- If the current list is an empty list, simply return an empty list. (This is your base case)
- Otherwise:
    - If the first element of the current list is a list, return the result of flattening that list plus the result of flattening the remainder of the list. (This is your first recursive case)
    - Otherwise, return a list containing just the first element plus the result of flattening the remainder of the list. (This is your second recursive case)

[1] We will check to make sure that your solution to this problem uses a recursive algorithm; **if it does not, you will not receive ANY credit for this problem!**

Hint: Python has a special function named `isinstance()` that can be extremely helpful here. `isinstance()` takes two arguments: a value to examine, and the name of a Python type (in this case, `list`). It returns `True` or `False` depending on whether the first argument belongs to the specified type. For example, the function call `isinstance(12, int)` returns `True`, while the function call `isinstance("3.14", float)` returns `False`.

Examples:

Function Call	**Return Value**
`flatten([])`	`[]`
`flatten([2])`	`[2]`
`flatten([3, 6, 1, 0])`	`[3, 6, 1, 0]`
`flatten([[5]])`	`[5]`
`flatten([1, 3, 5, [7], 9, 11])`	`[1, 3, 5, 7, 9, 11]`
`flatten([[6], 32, 36, 59, [93, 54], 98, [88]])`	`[6, 32, 36, 59, 93, 54, 98, 88]`
`flatten([[[[3]]]])`	`[3]`
`flatten([65, [52, 41], [7, [14, 35], 93], 73, 33, 94, 44, [63, 24, [[27, 88], [78, 75, 13], 31], 84]])`	`[65, 52, 41, 7, 14, 35, 93, 73, 33, 94, 44, 63, 24, 27, 88, 78, 75, 13, 31, 84]`