CSE/ISE 337: Scripting Languages

Stony Brook University

Programming Assignment #1

Fall 2020

Assignment Due: Friday, September 28, 2020 by 11:59 pm

Learning Outcomes

After completion of this programming project, you should be able to:

- Design and implement algorithms in Python
- Understand how to use and manipulate existing data structures in Python
- Learn how to construct custom data structures

Instructions

- Read the problem descriptions carefully. All the details are in the description.
- Note the function names and class names you need to implement. Since the code you submit will be auto-graded, *incorrect names may lead to your submission not being graded*.
- *Submit one python file called* ***hw1.py****.* All your solutions should be in this file.
- We will import your submission as *import hw1* and test the output of your program against our own test cases. You will get credit for every passing test case. A rubric will be posted after grading is complete.

Problem 1 (30 points)

A string is considered to be valid if all characters of the string appear the same number of times. It is also valid if *exactly* 1 character is removed, and the remaining characters occur the same number of times. Given a string *s*, determine if it is valid. If valid, return YES; otherwise, return NO.

As an example, consider the string *s = abc*. The string *s* is valid since every character occurs exactly once, {‘a’ : 1, ‘b’: 1, ‘c’: 1}. Similarly, the string *abcc* is also valid since removing one occurrence of the character *c* will make all occurrences of the remaining characters equal. However, the string *abccc* is not valid because removing exactly one character will not make all occurrences of the remaining characters equal. For example, if we remove the character *c*, the string will become *abcc*. The string *abcc* does not match the criteria of being valid since not all characters occur equally.

Function Description Write a function *isValid()* that takes a parameter *s* and returns YES if *s* is valid; NO otherwise.

Additional Constraints The string provided as parameter to the function *isValid()* will only contain characters [a - z].

Sample Test Cases
1. isValid('aabbcd') = NO
2. isValid('aabbcdddeefghi') = NO
3. isValid('abcdefghhgfedecba') = YES

Problem 2 (10 points)

A bracket is considered to be any one of *(,), {, }, [, or]*.

Two brackets are considered matched if an opening bracket (i.e., *(, {, [*) is followed by a closing bracket (i.e., *), },]*) of the exact same type. There 3 types of brackets – parenthesis, that is, (), braces, that is, *{}*, and square brackets, that is *[]*.

A matching pair of brackets is not balanced if the set of brackets it encloses are not matched. For example, *{ [( ]) }* is not balanced because the set of brackets between *{ }* is not balanced. The pair of square brackets encloses a single unbalanced open parenthesis *(*, and the pair of parenthesis encloses a single unbalanced closing square bracket *]*.

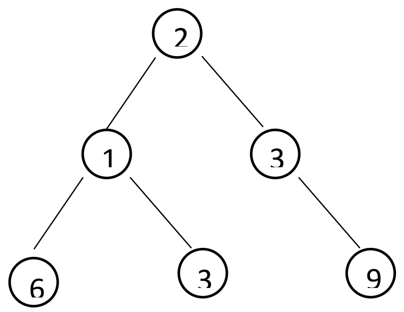
Hence, a sequence of brackets is balanced if the following conditions are met:
1. It contains no unmatched brackets
2. The subset of brackets enclosed within the confines of a matched pair of brackets is also a matched pair of brackets.

Function Description Write a function *isBalanced()* that takes a string, where each character in the string is a bracket, and returns YES if the brackets are balanced; otherwise, returns NO

Sample Test Case
1. isBalanced('{[()]}') = YES
2. isBalanced('{[(])}') = NO
3. isBalanced('{{[[(())]]}}') = YES

Problem 3 (10 + 30 + 20 = 60 points)

A binary tree is a data structure where each node has at most two children. Nodes that have no children are called leaf nodes. For example, the following structure is a binary tree because each node has 0, 1, or 2 children:



Part 1 (10 points) Design a class called *Node* to indicate the node of a binary tree. Each node must have an integer label and at most 2 child nodes. If an instance of a *Node* does not have a left child, then assume that the left child of that instance is *None*. Similarly, if the instance of a *Node* does not have a right child, then assume that right child of the instance is *None*. By this logic, a leaf node will have both its left and right child set to *None*. A valid implementation of the *Node* class can be instantiated in any one of the following ways:

1. Node(N) will create a node with label N and no children, that is, left and right child set to *None*
2. Node(N,(Node(N1)) will create a node with label N and a left child node with label N1, but no right child
3. Node(N, None, (Node(N1)) will create a node with label N and a right child node with label N1, but no left child
4. Node(N,(Node(N1), Node(N2)) will create a node with label N, a left child node with label N1 and right child node with label N2

Given a valid implementation of class *Node*, the binary tree shown above can be represented as *Node(2, Node(1, Node(6), Node(3)), Node(3, None, Node(9)).* You can assume that the node label will always be a number.

Part 2 (30 points) There are three ways to traverse a tree – *preorder*, *inorder*, *postorder*. In *preorder*, the root node of a tree is first visited, followed by the left node, and then the right node. In *inorder* traversal, the left node of a tree is first visited, followed by the root node, and then the right node. In *postoder* traversal, a tree is traversed by first visiting the left node, followed by the right node, and then the root node. For example, if we consider the tree shown above, the labels of the tree will be displayed as follows for each type of traversal:

- Preoder – 2 1 6 3 3 9
- Inorder – 6 1 3 2 3 9
- Postorder – 6 3 1 9 3 2

Function Description Write 3 functions in class *Node*, one for each type of traversal. The function *preOrder()* should take a tree as input and return a list of node labels in the tree in preorder form. The function *inorder()* should take a tree as input and return a list of node labels in the tree in inorder form. The function *postOrder()* should take a tree as input and return a list of node labels in the tree in postorder form. Each function returns an empty list if *None* is passed as the root of the tree.

Sample Test Cases

1. *root = Node(2, Node(1, Node(6), Node(3)), Node(3, None, Node(9))*
   *root.preOrder() = [*2,1,6,3,3,9*]*
   *root.inOrder() = [*6,1,3,2,3,9*]*
   *root.postOrder() = [*6,3,1,9,3,2*]*

2. *root = Node(1, Node(2, Node(3)), Node(4,None,(Node(5, None, Node(6, None, Node(7))))))*
   *root.preOrder() = [*1,2,3,4,5,6,7*]*
   *root.inOrder() = [*3,2,1,4,5,6,7*]*
   *root.postOrder() = [*3,2,7,6,5,4,1*]*

Part 3 (20 points) Write a function *sumTree()* in class *Node* that takes the root node of a tree as input, computes the sum of all the node labels in the tree, and returns the sum. For example, when the root node of the tree shown above, *Node(2, Node(1, Node(6), Node(3)), Node(3, None, Node(9))*, is provided as input to sum*Tree()*, it will return the sum 24 because 2+1+6+3+3+9 = 24.

Function Description *sumTree()* takes the root of the tree and returns a number indicating the sum of all the node labels in the tree. The function returns 0 if *None* is passed as input.

Sample Test Cases

1. *root = Node(2, Node(1, Node(6), Node(3)), Node(3, None, Node(9))*
   root.sumTree() = 24
2. *root = Node(1, Node(2, Node(3)), Node(4,None,(Node(5, None, Node(6, None, Node(7))))))*
   root.sumTree() = 28