CSE/ISE 337: Scripting Languages
Stony Brook University
Programming Assignment #1 (Python)
Fall 2024

Assignment Due: Monday, September 30th, 2024 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 3. Learn how to construct custom data structures
- Some advanced features of Python such as Lambda. Exceptions, Comprehension, and I/O etc.
- You have three weeks to complete it. Please use your time wisely and smartly.

Instructions

- Read the problem descriptions carefully. All the details are in the description.
- 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 test cases. You will get credit for every passing test case. Tentative rubric:
    - 5% for software engineering: comments and coding style
    - 30% for your test cases. You need to write 15 test cases including the one given in the documents as examples. TAs will judge the quality of your test cases as well.
    - 65% for test cases prepared by the TAs.

Problem 1 (20 points) (Suyi Chen)

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" (string) if *s* is valid; "NO" (string) otherwise.

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

Sample Test Cases

- **isValid('aabbcd') → NO**
- **isValid('aabbcdddeefghi') → NO**
- **isValid('abcdefghhgfedecba') → YES**

Problem 2 (20 points) (Sai Nakul Reddy Manne)

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 are 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

- It contains no unmatched brackets
- The subset of brackets enclosed within the confines of a matched pair of brackets is also a matched pair of brackets.

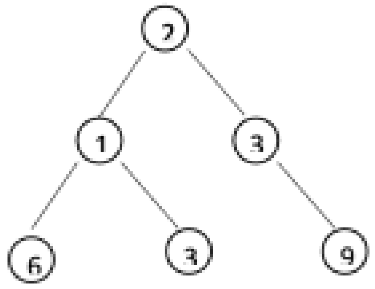
Function Description Write a function *isBalanced()* that takes a string, where each character in the string is a bracket, and returns YES if the brackets are balanced; otherwise, returns NO

Sample Test Case

- **isBalanced('{[()]}') → YES**
- **isBalanced('{[(])}') → NO**
- **isBalanced('{{[[(())]]}}') → YES**
- **isBalanced('[{}]()') → YES**
- **isBalanced('[{}()]') → YES**
- **isBalanced('[{}]()') → YES**

Problem 3 (40 + 10 = 50 points) (Yoosung Jang and Manthan Singh)

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 (40 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 the right child of the instance is *None*. By this logic, a leaf node will have both its left and right child set to *None*. Here, the left and right child should be given instance variable name *leftChild* and *rightChild* respectively. 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 a right child node with label N2

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

There are three ways to traverse a tree – *preorder*, *inorder*, and *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 *postorder* 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:

- Preorder–216339
- Inorder–613239
- Postorder–631932

Method Description Write 3 methods in class *Node*, one for each type of traversal. The method *preOrder()* of class *Node* returns a list of node labels in the tree in preorder form. Similarly, the inOrder() and postOrder() methods within this class should each return a list containing node labels from the tree in their respective traversal orders, namely, inorder and postorder.

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*]*

Hint:

- Review Chapter 12 on Recursion (Introduction to Python Programming Reference Book).

Part 2 (10 points) Write a method *sumTree()* in class *Node* that 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.

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

Problem 4 (20 points) (Anusha Avulapati)

Write a Python program that reads a text file containing multiple lines of text, processes the content, and outputs the following information:

1. The total number of lines in the file.
2. The total number of words in the file.
3. The total number of characters (including whitespace) in the file.
4. A new file named reversed_lines.txt contains all the lines from the original file in reverse order.

Steps:

1. **Read the file:** Use file input to read the contents of a text file named input.txt.
2. **Count lines, words, and characters:**
   - Lines: Count the number of lines in the file.
   - Words: Count the total number of words in the file.
   - Characters: Count the total number of characters in the file, including spaces and newline characters.
3. **Write reversed lines to a new file:** Write all the lines from the original file to a new file (reversed_lines.txt) in reverse order (the last line should appear first).

Example Input (input.txt):

```
Hello world!
Python is great.
I love coding.
```

Expected Output:

```
Total number of lines: 3
Total number of words: 8
Total number of characters: 45
```

Contents of reversed_lines.txt:

```
I love coding.
Python is great.
Hello world!
```

Requirements:

- Use with statement for file handling.
- Ensure to handle exceptions that might occur during file I/O (e.g., file not found).

Hints:

- Use the readlines() method to get all lines in the file as a list.
- You can use the split() method to count the words in each line.
- Remember to open the file in the correct mode ('r' for reading and 'w' for writing).