Trees

* All work to be performed individually.
* Submissions are due during your lab period the week of Jan 21.
* No late submissions accepted.

Use the submit command, with "1" as the first arg.

1. Consider a general Tree, T, defined as:
T -> []
T -> [value, [T1, T2 ...]]

This means:
* T can be an empty list
* T can be a 2 element list where:
    * T[0] is a value
    * T[1] is a list of elements that are trees
      These are termed the successor trees of T
      and T is termed the ancestor of T1, T2, ...

2. How would you represent this type of structure in C?
Sketch: you would define a structure that was something like:
```
struct TreeNode {
   int value; /* for integer valued trees */
   a linked list of "struct TreeNode *" to point to the successors
}
```

3. How would you represent this type of structure in Python?
Here is a class definition for a tree:
```
class tree:
    def __init__(self,x):
        self.store = [x,[]]

    def AddSuccessor(self,x):
        self.store[1] = self.store[1] + [x]
        return True
```

Here is an example of usage:
```
x=tree(1000)
y=tree(2000)
z=tree(3000)
x.AddSuccessor(y)
x.AddSuccessor(z)
c=tree(5)
z.AddSuccessor(c)
```

Draw the structure of what was created from the above commands. Do NOT read
further until you have done this.

We can ask python (or in this case, ipython) what the structure consists of:
```
In [10]: x.store
Out[10]:
[1000,
 [<tree.tree instance at 0x107ac6f38>, <tree.tree instance at 0x107ad3cf8>]]
```

The above says that x is a list consisting of a value, and a list of trees (as expected).

```
In [11]: x.store[1]
Out[11]: [<tree.tree instance at 0x107ac6f38>, <tree.tree instance at 0x107ad3cf8>]
```

```
The above gets us access to the list of subtrees; recall the 0 element is the
value and the 1 element is the list of subtrees.

In [12]: x.store[1][0]
Out[12]: <tree.tree instance at 0x107ac6f38>

The above gets us the first subtree, which is a tree class (as we want: a subtree
is a tree).

In [13]: x.store[1][0].store
Out[13]: [2000, []]

The above gets us the store variable of the first subtree: note the value is
2000 (see the list of commands above: this is clearly y).

In [14]: x.store[1][1].store
Out[14]: [3000, [<tree.tree instance at 0x107b02128>]]

The above gets us the store variable of the second subtree: note the value is
3000 (see the list of commands above: this is clearly z).

In [17]: x.store[1][1].store[1]
Out[17]: [<tree.tree instance at 0x107b02128>]

The above shows us that the second subtree (we called it z) has a subtree itself...

In [18]: x.store[1][1].store[1][0].store
Out[18]: [5, []]

The above shows us that the second subtree (we called it z) indeed has
the subtree of c, as expected!

This the diagram for this tree is:
       1000
       /  \
    2000  3000
            |
            5

It would be very convenient to have a print function that would display trees.
It would be a slight bit of sadism, however, for me to ask you to print
trees in the above fashion: pretty (sure), clear (yup), ... but a potential
pain in the root to do. It's not impossible however, and is a great exercise.
An exercise we'll leave for an exam or something.

Note, if you read the optional (but recommended) reference by Knuth he spends
some time thinking about the best way to print a tree. One method, which
I like, and you will too (because it's easier to print), would illustrate the
above tree like:

1000
   2000
   3000
      5

I.e., using indentation to show subtrees.

ASSIGNMENT 1:
This is eminently doable: so the first assignment in this lab is to modify the
tree class to have a print statement:
   def Print_DepthFirst(self):
      ...

Grading: this will NOT be graded. This is so you can get started, and
```

have a debug tool.

ASSIGNMENT 2:
Add a function to the class called "Get_LevelOrder" which will do a level order traversal and return a flat linear list of the level order traversal values.
If you need any additional classes to help you, write them and import them.

Grading: this will be graded.

ASSIGNMENT 3:
Create a second class called "binary_tree", analogous to the tree class but restricted to binary trees (trees with only two subtrees).
B -> []
B -> [val BL BR]

That is, a binary tree is either [] or a size-3 list with a value, a left binary subtree and a right binary subtree.

Instead of AddSuccessor, it will have AddLeft, and AddRight with the behavior you would expect given the names.
Make sure this class has a Get_LevelOrder function.

Grading: Nothing to grade here.

ASSIGNMENT 4:
Create a function, ConvertToBinaryTree, for the tree class that will generate the equivalent binary tree representation of the tree, and return that binary tree.

Grading: I will grade this by calling the function, and then calling the Get_LevelOrder function of the binary tree.

ASSIGNMENT 5:
Create a function, ConvertToTree, for the binary tree class that will generate the associated tree representation.
The return value is a length-2 list. The first element of the list is True (if the conversion can be performed) or False (if otherwise). The second element of the list is the Tree that is created (if the first element is True; if the first element is False, then clearly the second element is undefined).

Note: the right subtree of the root node of a binary tree (that was converted from a general tree) will not be present.

Grading: I will grade this by calling the function, and then calling Get_LevelOrder on the resultant tree

SUBMISSION:
Submit:
   1. binary_tree.py
   2. tree.py
in which the class definitions contain all of the corresponding functions that were requested above in the assignments.
   3. test.py
      TEST YOUR CODE.
* write a test.py that has:
from tree import *
from binary_tree import *
and tests your code.
Submit test.py, even though it isn't graded.

submit with a '1' as the second arg to the submit command: this is not a suggestion.