LEARNING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 6
TREES

BY FRED ANNEXSTEIN, PHD

PYTHON PROGRAMMING

8. TREES

BY FRED ANNEXSTEIN, PHD

CHAPTER 8 OVERVIEW

In this module we will be covering the abstract data types of trees. We will examine the creation and processing of these objects using a functional programming style as well as an object oriented style. We will apply our tree implementations to the solution of several programming problems.

CHAPTER 8 OBJECTIVES

By the end of this module, you will be able to...

- Understand the notion of an abstract data type using the examples of trees.
- Understand a functional approach to the implementation and processing of trees.
- Understand a object oriented approach to the implementation and processing of trees.
- Understand how to use recursion to effectively process trees.
- Understand how to use trees as a data structure to solve programming problems.
- Understand how to represent complex objects with strings
- Apply the Tree ADT to the problem of question answering using decision style binary tree.

TREES

The tree is a generalization of linked list and is a fundamental data abstraction that imposes regularity on how hierarchical values are structured and processed. A tree has a root and a sequence of branches.

Each branch of a tree is also a tree called a subtree. A tree with no branches is called a leaf. Any tree contained within a tree is called a sub-tree of that tree, such as a branch. A tree with exactly two branches on each internal node is called a binary tree. The root of each sub-tree of a tree is called a node in that tree.

FUNCTIONAL IMPLEMENTATION OF TREES

The data abstraction for a tree consists of collection of functions including a constructor and selectors. We begin with simplified versions of these methods.

```
def tree(root_obj, branches=[]):
  return [root_obj] + list(branches)


def root(t):
  return t[0]
```

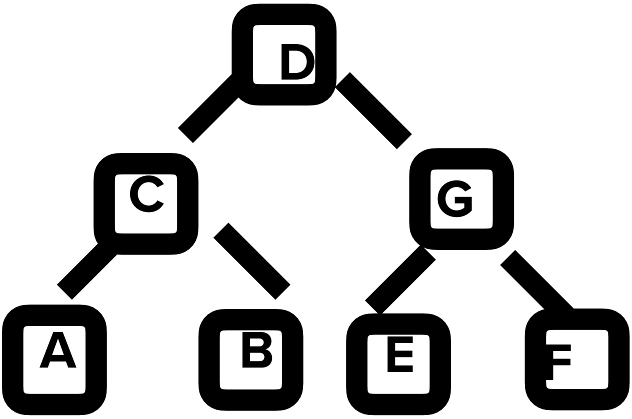
```
def branches(t):
  return t[1:]

def is_leaf(t):
  return not branches(t)

def add_branch(t,x):
  return tree(root(t),branches(t) +
[tree(x)])

alpha_tree =
  tree('D', [
   tree(‘C’, [tree('A'),tree('B')]),
   tree('G', [tree('E'),tree('F')])])
```

Here is a graphical representation of the alpha_tree object.



Traversing a tree means to visit each node of the tree exactly once. Here is a function to do just that while printing each root value.

```
def traverse(t):
  print(root(t))
  if not is_leaf(t):
    for b in branches(t):
         traverse(b)

>>> traverse(alpha_tree)
D  C  A  B  G  E  F
```

When a node of the tree is first reached the function prints out the root value. It then checks if the node is a leaf node (this is the base case). If it is not a leaf, a recursive call is made on each tree in the list of branches. Traverse always visits nodes in a depth-first order.

If we want to reach a random leaf of the tree we can do a random walk as follows. At each node, check if it is a leaf. If it is not we can choose a random branch to continue the walk.

We use the randrange function from the standard random library to choose a random branch as we go down the tree.

```
from random import randrange

def random_walk(t):
  print(root(t), end= ' ')
  if not is_leaf(t):
    random_walk(branches(t)
        [randrange(len(branches(t)))])

>>> random_walk(alpha_tree)
D C B

>>> random_walk(alpha_tree)
D G E

>>>random_walk(alpha_tree)
D G F
```

THE FIBONACCI TREE

The following function recursively builds and returns a binary tree that is constructed out of the fibonacci sequence. A binary tree has exactly two branches from each non-leaf node. Like the fibonacci sequence, the fibonacci function works recursively. The base case is when the value n is 1 or 0, and the function returns the single leaf with value 1. Otherwise, the function constructs a tree from the two sub-trees fibtree(n-1) and fibtree(n-2), setting them as two branches from root. The value stored in the root is simply the sum of the roots of the two branches.

```
def fibtree(n):
  if n ==1 or n==0:
    return tree(1)
  else:
    t1= fibtree(n-1)
    t2= fibtree(n-2)
    return tree(root(t1) + root(t2),
                       [t1] + [t2])

>>> f= fibtree(5)
>>> root(f)
8

>>> traverse(f)
8  5  3  2  1  1  1  2  1  1  3  2  1
1  1

>>> random_walk(f)
8 5 2 1

>>> random_walk(f)
8 5 3 1

>>> random_walk(f)
8 3 1

>>> random_walk(f)
8 3 2 1
```

OBJECT-ORIENTED IMPLEMENTATION OF TREE ADT

Trees can also be represented by instances of user-defined classes, rather than nested instances of built-in sequence types. Recall that a tree is a data structure that has as an data attribute and an attribute that is a sequence of branches that are also trees.

Previously, we defined trees in a functional style. Here is an alternative class definition, which we will use to illustrate new features that are magic methods.

We define the trees so that they have internal values at the roots of each subtree. The internal value is called a label in the tree. The Tree class below represents such trees, in which each tree has a sequence of branches that are also trees. A repr function is defined to return a string that could be evaluated to a Tree.

```
class Tree:
   def __init__(self, root, branches=()):
       self.root = root
       for branch in branches:
           assert type(branch) == Tree
       self.branches = branches

   def is_leaf(self):
            return not self.branches

   def traverse(self):
        print(self.root, end=' ')
        if self.branches:
            for b in self.branches:
                b.traverse()


def fib_tree(n):
        if n == 1:
            return Tree(0)
        elif n == 2:
            return Tree(1)
        else:
            left = fib_tree(n-2)
            right = fib_tree(n-1)
            return Tree(left.root +
right.root, (left, right))
```

The Tree class can represent, for instance, the values computed in an expression tree for the recursive implementation of fib, the function for computing Fibonacci numbers. The function fib_tree(n) below returns a Tree that has the nth Fibonacci number as its label and a trace of all previously computed Fibonacci numbers within its branches.

```
>>> f.fib_tree(5)
>>> f
Tree(3,(Tree(1,(Tree(0), Tree(1))),
Tree(2,(Tree(1), Tree(1,(Tree(0),
Tree(1)))))))

>>> f.traverse()
3 1 0 1 2 1 1 0 1
```

REPRESENTING OBJECTS WITH STRINGS

Our implementation is basically complete, but an instance of the Tree class is currently difficult to inspect. To help with debugging, we can also define a function to convert a Tree to a string expression.

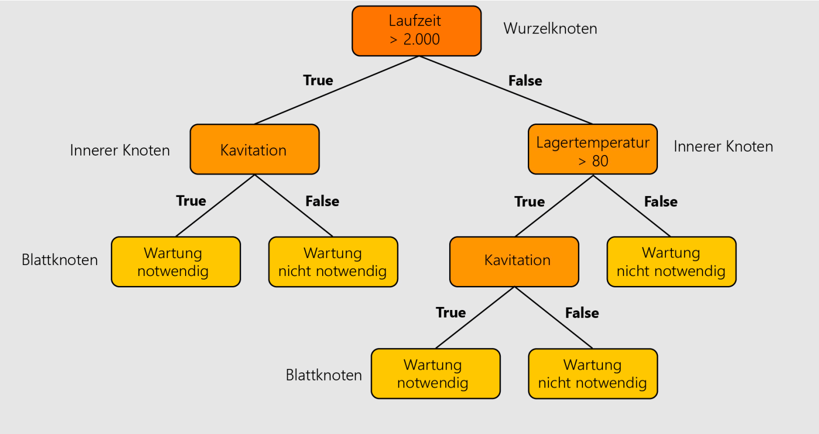
In Python, there are a few methods that you can implement in your class definition to customize how built-in functions that return representations of your class behave.

There are two string method traditionally used. One is called “str” and another called “repr”. The difference is that “repr” generally returns a string that can be processed by the interpreter to re-produce an exact copy of the object.

```
def Tree.__repr__(self):
       if self.branches:
           return ‘Tree({0},
               {1})'.format(self.label,
               repr(self.branches))
       else:
           return
               'Tree({0})'.format(repr(self.
               label))

>>> f.fib_tree(5)
>>> repr(f)
’Tree(3,(Tree(1,(Tree(0), Tree(1))),
Tree(2,(Tree(1), Tree(1,(Tree(0),
Tree(1)))))))'

>>> f2 = eval(repr(f))
>>> type(f2) == Tree
True
>>> f2
```



```
Tree(3,(Tree(1,(Tree(0), Tree(1))),
Tree(2,(Tree(1), Tree(1,(Tree(0),
Tree(1)))))))
```

APPLICATION: DECISION TREE CONSTRUCTION

Decision trees are labeled binary tree that can answer questions and are useful in classification problems. Decision trees contain at each non-leaf node an yes/no question of if-then-else rule followed by new branches of the tree determined by the outcome of the rule or answer to question. The leaf nodes determine an outcome determined by the root to leaf path answers.

As we will see in later chapters decision trees are often used in classification and regression problems. The goal of a decision tree is to create a model that can accurately predict the value of a target variable by applying easy decision rules.

The basic structure of a decision tree is shown in the figure below, where the yellow leaf nodes label the class, and the orange internal nodes are labeled with T/F decision test.

Here we develop an decision tree application using our Tree data structure to store a set of questions in the internal nodes of the tree, and a collection of animal names at the leaves of the tree. Our program asks questions until a leaf is reached and a guess is made. If the guess is correct we print a message, otherwise we ask the user to input a questions that would distinguish the guessed animal from the animal picked by the user. We add this question to the tree and we create branches to the old guess and the newly added animal.

```
def decision_tree():

    root = Tree(“Bird",[None,None])

    # Loop until the user quits
    while True:
        if not yes(“Thinking of an animal?
"): break
        # Walk the tree
        tree = root
        while tree.branches[0] is not None:
            prompt = tree.root + "? "
            if yes(prompt):
                tree = tree.branches[0]
            else:
                tree = tree.branches[1]

        # Make a guess
        guess = tree.root
        prompt = "Is it a " + guess + "? "
        if yes(prompt):
            print("Guessed it!!!”)
            continue

        # Get new information
        prompt  = "What is the animal's 
name? "
        animal  = input(prompt)
        prompt  = "What question would 
distinguish a {0} from a {1}? "
        question = 
input(prompt.format(animal, guess))

        # Add new information to the tree
        tree.root = question
        prompt = "If the animal were {0} the 
answer would be? "
        if yes(prompt.format(animal)):
            tree.branches[1] = Tree(guess, 
[None,None])
            tree.branches[0] = Tree(animal, 
[None,None])
        else:
            tree.branches[1] = Tree(animal, 
[None,None])
            tree.branches[0] = Tree(guess, 
[None,None])

def yes(question):
    ans = input(question).lower()
    return ans[0] == “y"

>>> guessing_game()
Are you thinking of an animal? y
Is it a Bird? n
What is the animals name? Dog
What question would distinguish a Dog from a
Bird? Can it fly
If the animal were dog the answer would be?
n

Are you thinking of an animal? y
Can it fly? n
Is it a Dog? n
What is the animals name? Cat
What question would distinguish a Cat from a
Dog? Does it bark
If the animal were cat the answer would be?
n

Are you thinking of an animal? y
Can it fly? n
Does it bark? y
Is it a Dog? y
Guessed it!!!

Are you thinking of an animal? n
```

CHAPTER 8 EXERCISES

Question 1: Height

Define the function height, which takes in a tree as an argument and returns the number of branches it takes start at the root node and reach the leaf that is farthest away from the root. Note: given the definition of the height of a tree, if height is given a leaf, what should it return?

Question 2: Sprout leaves

Define a function sprout_leaves that, given a tree, t, and a list of values, vals, and produces a tree with every leaf having new children that contain each of the items in vals. Do not add new children to non-leaf nodes.