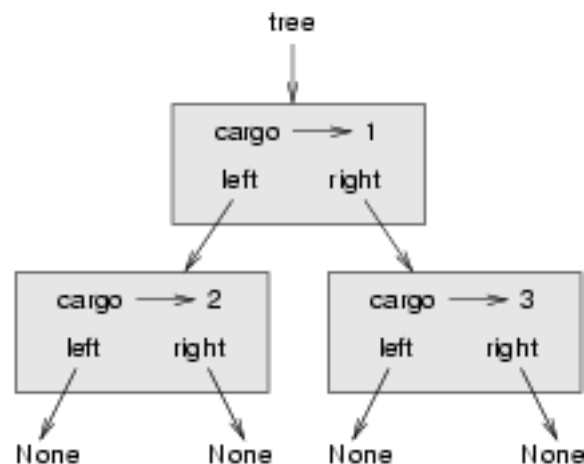# Chapter 28: Trees

Like linked lists, trees are made up of nodes. A common kind of tree is a **binary tree**, in which each node contains a reference to two other nodes (possibly None). These references are referred to as the left and right subtrees. Like list nodes, tree nodes also contain cargo. A state diagram for a tree looks like this:

To avoid cluttering up the picture, we often omit the Nones.

The top of the tree (the node tree refers to) is called the **root**. In keeping with the tree metaphor, the other nodes are called branches and the nodes at the tips with null references are called **leaves**. It may seem odd that we draw the picture with the root at the top and the leaves at the bottom, but that is not the strangest thing.

To make things worse, computer scientists mix in another metaphor: the family tree. The top node is sometimes called a **parent** and the nodes it refers to are its **children**. Nodes with the same parent are called **siblings**.

Finally, there is a geometric vocabulary for talking about trees. We already mentioned left and right, but there is also up (toward the parent/root) and down (toward the children/leaves). Also, all of the nodes that are the same distance from the root comprise a **level** of the tree.

We probably don't need three metaphors for talking about trees, but there they are.

Like linked lists, trees are recursive data structures because they are defined recursively. A tree is either:

1. the empty tree, represented by None, or
2. a node that contains an object reference (cargo) and two tree references.

28.1. Building trees

The process of assembling a tree is similar to the process of assembling a linked list. Each constructor invocation builds a single node.

```
1  class Tree:
2      def __init__(self, cargo, left=None, right=None):
3          self.cargo = cargo
4          self.left = left
5          self.right = right
6
7      def __str__(self):
8          return str(self.cargo)
```

The cargo can be any type, but the left and right parameters should be tree nodes. left and right are optional; the default value is None.

To print a node, we just print the cargo.

One way to build a tree is from the bottom up. Allocate the child nodes first:

```
1  left = Tree(2)
2  right = Tree(3)
```

Then create the parent node and link it to the children:

```
1  tree = Tree(1, left, right)
```

We can write this code more concisely by nesting constructor invocations:

```
1  >>> tree = Tree(1, Tree(2), Tree(3))
```

Either way, the result is the tree at the beginning of the chapter.

28.2. Traversing trees

Any time you see a new data structure, your first question should be, "How do I traverse it?" The most natural way to traverse a tree is recursively. For example, if the tree contains integers as cargo, this function returns their sum:

```
1  def total(tree):
2      if tree is None: return 0
3      return total(tree.left) + total(tree.right) + tree.cargo
```

The base case is the empty tree, which contains no cargo, so the sum is 0. The recursive step makes two recursive calls to find the sum of the child trees. When the recursive calls complete, we add the cargo of the parent and return the total.

28.3. Expression trees

A tree is a natural way to represent the structure of an expression. Unlike other notations, it can represent the computation unambiguously. For example, the infix expression 1 + 2 * 3 is ambiguous unless we know that the multiplication happens before the addition.

This expression tree represents the same computation:
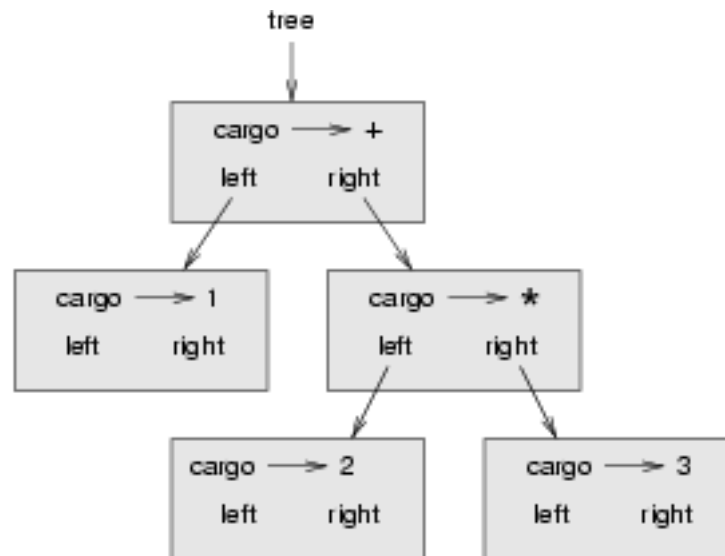
The nodes of an expression tree can be operands like 1 and 2 or operators like + and *. Operands are leaf nodes; operator nodes contain references to their operands. (All of these operators are **binary**, meaning they have exactly two operands.)

We can build this tree like this:

```
1  >>> tree = Tree("+", Tree(1), Tree("*", Tree(2), Tree(3)))
```

Looking at the figure, there is no question what the order of operations is; the multiplication happens first in order to compute the second operand of the addition.

Expression trees have many uses. The example in this chapter uses trees to translate expressions to postfix, prefix, and infix. Similar trees are used inside compilers to parse, optimize, and translate programs.

28.4. Tree traversal

We can traverse an expression tree and print the contents like this:

```
1  def print_tree(tree):
2      if tree is None: return
3      print(tree.cargo, end=" ")
4      print_tree(tree.left)
5      print_tree(tree.right)
```

In other words, to print a tree, first print the contents of the root, then print the entire left subtree, and then print the entire right subtree. This way of traversing a tree is called a **preorder**, because the contents of the root appear *before* the contents of the children. For the previous example, the output is:

```
1  >>> tree = Tree("+", Tree(1), Tree("*", Tree(2), Tree(3)))
2  >>> print_tree(tree)
3  + 1 * 2 3
```

This format is different from both postfix and infix; it is another notation called **prefix**, in which the operators appear before their operands.

You might suspect that if you traverse the tree in a different order, you will get the expression in a different notation. For example, if you print the subtrees first and then the root node, you get:

```
1  def print_tree_postorder(tree):
2      if tree is None: return
3      print_tree_postorder(tree.left)
4      print_tree_postorder(tree.right)
5      print(tree.cargo, end=" ")
```

The result, 1 2 3 * +, is in postfix! This order of traversal is called **postorder**.

Finally, to traverse a tree **inorder**, you print the left tree, then the root, and then the right tree:

```
1  def print_tree_inorder(tree):
2      if tree is None: return
3      print_tree_inorder(tree.left)
4      print(tree.cargo, end=" ")
5      print_tree_inorder(tree.right)
```

The result is `1 + 2 * 3`, which is the expression in infix.

To be fair, we should point out that we have omitted an important complication. Sometimes when we write an expression in infix, we have to use parentheses to preserve the order of operations. So an inorder traversal is not quite sufficient to generate an infix expression.

Nevertheless, with a few improvements, the expression tree and the three recursive traversals provide a general way to translate expressions from one format to another.

If we do an inorder traversal and keep track of what level in the tree we are on, we can generate a graphical representation of a tree:

```python
def print_tree_indented(tree, level=0):
    if tree is None: return
    print_tree_indented(tree.right, level+1)
    print("  " * level + str(tree.cargo))
    print_tree_indented(tree.left, level+1)
```

The parameter `level` keeps track of where we are in the tree. By default, it is initially 0. Each time we make a recursive call, we pass `level+1` because the child's level is always one greater than the parent's. Each item is indented by two spaces per level. The result for the example tree is:

```
>>> print_tree_indented(tree)
    3
  *
    2
+
  1
```

If you look at the output sideways, you see a simplified version of the original figure.

28.5. Building an expression tree

In this section, we parse infix expressions and build the corresponding expression trees. For example, the expression `(3 + 7) * 9` yields the following tree:
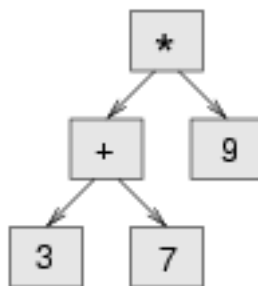
Notice that we have simplified the diagram by leaving out the names of the attributes.

The parser we will write handles expressions that include numbers, parentheses, and the operators + and *. We assume that the input string has already been tokenized into a Python list (producing this list is left as an exercise). The token list for (3 + 7) * 9 is:

```
1   ["(", 3, "+", 7, ")", "*", 9, "end"]
```

The end token is useful for preventing the parser from reading past the end of the list.

The first function we'll write is get_token, which takes a token list and an expected token as parameters. It compares the expected token to the first token on the list: if they match, it removes the token from the list and returns True; otherwise, it returns False:

```
1   def get_token(token_list, expected):
2       if token_list[0] == expected:
3           del token_list[0]
4           return True
5       return False
```

Since token_list refers to a mutable object, the changes made here are visible to any other variable that refers to the same object.

The next function, get_number, handles operands. If the next token in token_list is a number, get_number removes it and returns a leaf node containing the number; otherwise, it returns None.

```
1   def get_number(token_list):
2       x = token_list[0]
3       if type(x) != type(0): return None
4       del token_list[0]
5       return Tree(x, None, None)
```

Before continuing, we should test get_number in isolation. We assign a list of numbers to token_list, extract the first, print the result, and print what remains of the token list:

```
1   >>> token_list = [9, 11, "end"]
2   >>> x = get_number(token_list)
3   >>> print_tree_postorder(x)
4   9
5   >>> print(token_list)
6   [11, "end"]
```

The next method we need is get_product, which builds an expression tree for products. A simple product has two numbers as operands, like 3 * 7.

Here is a version of get_product that handles simple products.

```
1  def get_product(token_list):
2      a = get_number(token_list)
3      if get_token(token_list, "*"):
4          b = get_number(token_list)
5          return Tree("*", a, b)
6      return a
```

Assuming that `get_number` succeeds and returns a singleton tree, we assign the first operand to `a`. If the next character is *, we get the second number and build an expression tree with `a`, `b`, and the operator.

If the next character is anything else, then we just return the leaf node with `a`. Here are two examples:

```
1  >>> token_list = [9, "*", 11, "end"]
2  >>> tree = get_product(token_list)
3  >>> print_tree_postorder(tree)
4  9 11 *
```

```
1  >>> token_list = [9, "+", 11, "end"]
2  >>> tree = get_product(token_list)
3  >>> print_tree_postorder(tree)
4  9
```

The second example implies that we consider a single operand to be a kind of product. This definition of product is counter-intuitive, but it turns out to be useful.

Now we have to deal with compound products, like like `3 * 5 * 13`. We treat this expression as a product of products, namely `3 * (5 * 13)`. The resulting tree is:
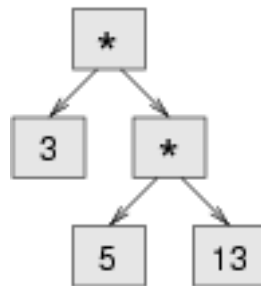
With a small change in `get_product`, we can handle an arbitrarily long product:

```
1   def get_product(token_list):
2       a = get_number(token_list)
3       if get_token(token_list, "*"):
4           b = get_product(token_list)         # This line changed
5           return Tree("*", a, b)
6       return a
```

In other words, a product can be either a singleton or a tree with ∗ at the root, a number on the left, and a product on the right. This kind of recursive definition should be starting to feel familiar.

Let's test the new version with a compound product:

```
1   >>> token_list = [2, "*", 3, "*", 5 , "*", 7, "end"]
2   >>> tree = get_product(token_list)
3   >>> print_tree_postorder(tree)
4   2 3 5 7 * * *
```

Next we will add the ability to parse sums. Again, we use a slightly counter-intuitive definition of sum. For us, a sum can be a tree with + at the root, a product on the left, and a sum on the right. Or, a sum can be just a product.

If you are willing to play along with this definition, it has a nice property: we can represent any expression (without parentheses) as a sum of products. This property is the basis of our parsing algorithm.

get_sum tries to build a tree with a product on the left and a sum on the right. But if it doesn't find a +, it just builds a product.

```
1   def get_sum(token_list):
2       a = get_product(token_list)
3       if get_token(token_list, "+"):
4           b = get_sum(token_list)
5           return Tree("+", a, b)
6       return a
```

Let's test it with 9 * 11 + 5 * 7:

```
1   >>> token_list = [9, "*", 11, "+", 5, "*", 7, "end"]
2   >>> tree = get_sum(token_list)
3   >>> print_tree_postorder(tree)
4   9 11 * 5 7 * +
```

We are almost done, but we still have to handle parentheses. Anywhere in an expression where there can be a number, there can also be an entire sum enclosed in parentheses. We just need to modify get_number to handle **subexpressions**:

```
1  def get_number(token_list):
2      if get_token(token_list, "("):
3          x = get_sum(token_list)          # Get the subexpression
4          get_token(token_list, ")")       # Remove the closing parenthesis
5          return x
6      else:
7          x = token_list[0]
8          if type(x) != type(0): return None
9          del token_list[0]
10         return Tree(x, None, None)
```

Let's test this code with `9 * (11 + 5) * 7`:

```
1  >>> token_list = [9, "*", "(", 11, "+", 5, ")", "*", 7, "end"]
2  >>> tree = get_sum(token_list)
3  >>> print_tree_postorder(tree)
4  9 11 5 + 7 * *
```

The parser handled the parentheses correctly; the addition happens before the multiplication.

In the final version of the program, it would be a good idea to give `get_number` a name more descriptive of its new role.

28.6. Handling errors

Throughout the parser, we've been assuming that expressions are well-formed. For example, when we reach the end of a subexpression, we assume that the next character is a close parenthesis. If there is an error and the next character is something else, we should deal with it.

```
1  def get_number(token_list):
2      if get_token(token_list, "("):
3          x = get_sum(token_list)
4          if not get_token(token_list, ")"):
5              raise ValueError('Missing close parenthesis")
6          return x
7      else:
8          # The rest of the function omitted
```

The `raise` statement throws the exception object which we create. In this case we simply used the most appropriate type of built-in exception that we could find, but you should be aware that you can create your own more specific user-defined exceptions if you need to. If the function that called `get_number`, or one of the other functions in the traceback, handles the exception, then the program can continue. Otherwise, Python will print an error message and quit.

28.7. The animal tree

In this section, we develop a small program that uses a tree to represent a knowledge base.

The program interacts with the user to create a tree of questions and animal names. Here is a sample run:

```
 1  Are you thinking of an animal? y
 2  Is it a bird? n
 3  What is the animals name? dog
 4  What question would distinguish a dog from a bird? Can it fly
 5  If the animal were dog the answer would be? n
 6
 7  Are you thinking of an animal? y
 8  Can it fly? n
 9  Is it a dog? n
10  What is the animals name? cat
11  What question would distinguish a cat from a dog? Does it bark
12  If the animal were cat the answer would be? n
13
14  Are you thinking of an animal? y
15  Can it fly? n
16  Does it bark? y
17  Is it a dog? y
18  I rule!
19
20  Are you thinking of an animal? n
```

Here is the tree this dialog builds:
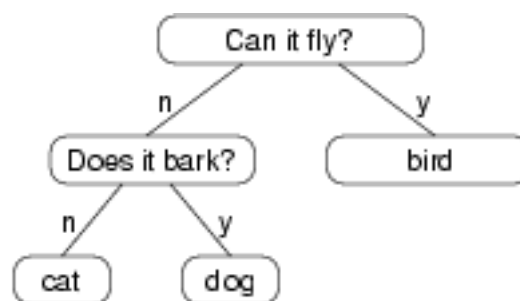
At the beginning of each round, the program starts at the top of the tree and asks the first question. Depending on the answer, it moves to the left or right child and continues until it gets to a leaf node. At that point, it makes a guess. If the guess is not correct, it asks the user for the name of the new animal and a question that distinguishes the (bad) guess from the new animal. Then it adds a node to the tree with the new question and the new animal.

Here is the code:

```python
1   def yes(ques):
2       ans = input(ques).lower()
3       return ans[0] == "y"
4
5   def animal():
6       # Start with a singleton
7       root = Tree("bird")
8
9       # Loop until the user quits
10      while True:
11          print()
12          if not yes("Are you thinking of an animal? "): break
13
14          # Walk the tree
15          tree = root
16          while tree.left is not None:
17              prompt = tree.cargo + "? "
18              if yes(prompt):
19                  tree = tree.right
20              else:
21                  tree = tree.left
22
23          # Make a guess
24          guess = tree.cargo
25          prompt = "Is it a " + guess + "? "
26          if yes(prompt):
27              print("I rule!")
28              continue
29
30          # Get new information
31          prompt  = "What is the animal's name? "
32          animal  = input(prompt)
33          prompt  = "What question would distinguish a {0} from a {1}? "
34          question = input(prompt.format(animal, guess))
35
36          # Add new information to the tree
37          tree.cargo = question
38          prompt = "If the animal were {0} the answer would be? "
39          if yes(prompt.format(animal)):
40              tree.left = Tree(guess)
41              tree.right = Tree(animal)
```

```
42          else:
43              tree.left = Tree(animal)
44              tree.right = Tree(guess)
```

The function yes is a helper; it prints a prompt and then takes input from the user. If the response begins with *y* or *Y,* the function returns True.

The condition of the outer loop of animal is True, which means it will continue until the break statement executes, if the user is not thinking of an animal.

The inner while loop walks the tree from top to bottom, guided by the user's responses.

When a new node is added to the tree, the new question replaces the cargo, and the two children are the new animal and the original cargo.

One shortcoming of the program is that when it exits, it forgets everything you carefully taught it! Fixing this problem is left as an exercise.

28.8. Glossary

```
1   binary operator
2       An operator that takes two operands.
3
4   binary tree
5       A tree in which each node refers to zero, one, or two dependent nodes.
6
7   child
8       One of the nodes referred to by a node.
9
10  leaf
11      A bottom-most node in a tree, with no children.
12
13  level
14      The set of nodes equidistant from the root.
15
16  parent
17      The node that refers to a given node.
18
19  postorder
20      A way to traverse a tree, visiting the children of each node before the
21      node itself.
22
23  prefix notation
24      A way of writing a mathematical expression with each operator appearing
```

```
25        before its operands.
26
27  preorder
28        A way to traverse a tree, visiting each node before its children.
29
30  root
31        The topmost node in a tree, with no parent.
32
33  siblings
34        Nodes that share a common parent.
35
36  subexpression
37        An expression in parentheses that acts as a single operand in a larger
38        expression.
```

28.9. Exercises

1. Modify `print_tree_inorder` so that it puts parentheses around every operator and pair of operands. Is

the output correct and unambiguous? Are the parentheses always necessary?

2. Write a function that takes an expression string and returns a token list.

3. Find other places in the expression tree functions where errors can occur and add appropriate `raise`

statements. Test your code with improperly formed expressions.

4. Think of various ways you might save the animal knowledge tree in a file. Implement the one you think is

easiest.