13.2 Recursion on Trees

When we introduced tree terminology in the previous section, we kept on repeating the same question: "What's the relationship between a tree's X and the X of its subtrees?" Understanding the relationship between a tree and its subtrees—that is, its recursive structure—allows us to write extremely simple and elegant recursive code for processing trees, just as it did with nested lists and `RecursiveList` in the previous chapter.

Tree size

Here's a first example: suppose we want to calculate the size of a tree. We can approach this problem by following the recursive definition of a tree, being either empty or a root connected to a list of subtrees.

Let T be a tree, and let $size$ be a function mapping any tree to its size.

- If T is empty, then its size is 0: we can write $size(T) = 0$.
- If T is non-empty, then it consists of a root value and collection of subtrees $T_0, T_1, \ldots, T_{k-1}$ (for some $k \in \mathbb{N}$). In this case, the size of T is the *sum* of the sizes of its subtrees, plus 1 for the root:

$$size(T) = 1 + \sum_{i=0}^{k-1} size(T_i)$$

We can combine these two observations to write a recursive mathematical definition for our $size$ function:

$$size(T) = \begin{cases} 0, & \text{if } T \text{ is empty} \\ 1 + \sum_{i=0}^{k-1} size(T_i), & \text{if } T \text{ has subtrees } T_0, T_1, \ldots, T_{k-1} \end{cases}$$

This is quite similar to how we defined the size function for nested lists, and translates naturally into a recursive Python method `Tree.__len__`:

```
class Tree:
    def __len__(self) -> int:
        """Return the number of items contained in this tree.

        >>> t1 = Tree(None, [])
        >>> len(t1)
        0
        >>> t2 = Tree(3, [Tree(4, []), Tree(1, [])])
```

```
        >>> len(t2)
        3
        """
        if self.is_empty():
            return 0
        else:
            return 1 + sum(subtree.__len__() for subtree in self._subtrees)
```

This method is again recursive, similar to the recursive functions we studied in the previous chapter.[1] The base case is when `self.is_empty()`, and the recursive step occurs

[1] In fact, it is a good idea when reviewing to explicitly compare your `Tree` recursive methods with the recursive functions on nested lists. They'll have a lot in common!

when the tree is non-empty, and we need to recurse on each subtree.

While the built-in aggregation function `sum` is certainly convenient, we often will want to implement a custom aggregation operation using a loop. For example, we could have written our above else branch using a loop:

```
        else:
            size_so_far = 1
            for subtree in self._subtrees:
                size_so_far += subtree.__len__()
            return size_so_far
```

So in general, we have the following code template for recursing on trees:

Tree recursion code template.

```
class Tree:
    def method(self) -> ...:
        if self.is_empty():
            ...
        else:
            ...
            for subtree in self._subtrees:
                ... subtree.method() ...
            ...
```

Note: often the `...` will use `self._root` as well!

An explicit size-one case

Often when first dealing with trees, students like to think explicitly about the case where the tree consists of just a single item. We can modify our `__len__` implementation to handle this case separately by adding an extra check:

```
class Tree:
    def __len__(self):
        if self.is_empty():         # tree is empty
            return 0
        elif self._subtrees == []:  # tree is a single item
            return 1
        else:                       # tree has at least one subtree
            return 1 + sum(subtree.__len__() for subtree in self._subtrees)
```

Sometimes, this check will be *necessary*: we'll want to do something different for a tree with a single item than for either an empty tree or one with at least one subtree. And sometimes, this check will be *redundant*: the action performed by this case is already handled by the recursive step. In the case of `__len__`, the latter situation applies, because when there are no subtrees the built-in `sum` function returns `0`.[2]

[2] Or in the for loop version, the loop does not execute, so `size_so_far` remains `1` when it is returned.

However, the possibility of having a redundant case shouldn't discourage you from starting off by including this case. Treat the detection and coalescing of redundant cases as part of the code editing process. Your first draft might have some extra code, but that can be removed once you are confident that your implementation is correct. For your reference, here is the three-case recursive `Tree` code template:

Tree recursion code template (with size-one case).

```
class Tree:
    def method(self) -> ...:
        if self.is_empty():         # tree is empty
            ...
        elif self._subtrees == []:  # tree is a single value
            ...
        else:                       # tree has at least one subtree
            ...
            for subtree in self._subtrees:
                ... subtree.method() ...
            ...
```

Extended example: `Tree.__str__`

Because the elements of a list have a natural order, lists are pretty straightforward to traverse, meaning (among other things) that it's easy to write a `__str__` method that will return a `str` containing all of the elements. With trees, there is a non-linear ordering on the elements. How might we write a `Tree.__str__` method?

Here's an idea: start with the value of the root, then recursively add on the `__str__` for each of the subtrees. That's pretty easy to implement. The base case is when the tree is empty, and in this case the method returns an empty string.

```
def __str__(self) -> str:
    """Return a string representation of this tree.
    """
    if self.is_empty():
        return ''
    else:
        # We use newlines ('\n') to separate the different values.
        s = f'{self._root}\n'
        for subtree in self._subtrees:
            s += subtree.__str__()  # equivalent to str(subtree)
        return s
```

Consider what happens when we run this on the following tree structure:

```
>>> t1 = Tree(1, [])
>>> t2 = Tree(2, [])
>>> t3 = Tree(3, [])
>>> t4 = Tree(4, [t1, t2, t3])
>>> t5 = Tree(5, [])
>>> t6 = Tree(6, [t4, t5])
>>> print(t6)  # This automatically calls `t6.__str__` and prints the result
6
4
1
2
3
5
```

We know that `6` is the root of the tree, but it's ambiguous how many children it has. In other words, while the *items* in the tree are correctly included, we lose the *structure* of the tree itself.

Drawing inspiration from how PyCharm (among many other programs) display the folder structure of our computer's files, we're going to use *indentation* to differentiate between the different levels of a tree. For our example tree, we want `__str__` to produce this:

```
>>> # (The same t6 as defined above.)
>>> print(t6)
6
  4
    1
    2
    3
  5
```

In other words, we want `__str__` to return a string that has 0 indents before the root value, 1 indent before its children's values, 2 indents before their children's values, and so on. But how do we do this? We need the recursive calls to act differently—to return strings with more indentation the deeper down in the tree they are working. In other words, we want information from where a method is called to influence what happens inside the method. This is *exactly* the problem that parameters are meant to solve!

So we'll pass in an extra parameter for the *depth* of the current tree, which will be used to add a corresponding number of indents before each value in the `str` that is returned. We can't change the public interface of the `__str__` method itself, but we can define a helper method that has this extra parameter:

```
def _str_indented(self, depth: int) -> str:
    """Return an indented string representation of this tree.

    The indentation level is specified by the <depth> parameter.
    """
    if self.is_empty():
        return ''
    else:
        s = '  ' * depth + f'{self._root}\n'
        for subtree in self._subtrees:
            # Note that the 'depth' argument to the recursive call is
         modified.
            s += subtree._str_indented(depth + 1)
        return s
```

Now we can implement `__str__` simply by making a call to `_str_indented`:

```
def __str__(self) -> str:
    """Return a string representation of this tree.
    """
    return self._str_indented(0)
```

```
>>> t1 = Tree(1, [])
>>> t2 = Tree(2, [])
>>> t3 = Tree(3, [])
```

```
>>> t4 = Tree(4, [t1, t2, t3])
>>> t5 = Tree(5, [])
>>> t6 = Tree(6, [t4, t5])
6
  4
    1
    2
    3
  5
```

Technical note: optional parameters

One way to customize the behaviour of functions is to make a parameter **optional** by giving it a **default value**. This can be done for any function, recursive or non-recursive. The syntax for doing so is quite simple; we use it in this revised version of `_str_indented` to give a default value for `depth`:

```
def _str_indented(self, depth: int = 0) -> str:
    """Return an indented string representation of this tree.

    The indentation level is specified by the <depth> parameter.
    """
    if self.is_empty():
        return ''
    else:
        s = '  ' * depth + f'{self._root}\n'
        for subtree in self._subtrees:
            # Note that the 'depth' argument to the recursive call is
         modified.
            s += subtree._str_indented(depth + 1)
        return s
```

In this version of `_str_indented`, `depth` is an optional parameter that can either be included or not included when this method is called.

So we can call `t._str_indented(5)`, which assigns its `depth` parameter to `5`, as we would expect. However, we can also call `t._str_indented()` (no argument for `depth` given), in which case the method is called with the `depth` parameter assigned to `0`.

Optional parameters are a powerful Python feature that allows us to write more flexible functions and methods to be used in a variety of situations. Two important points to keep in mind, though:

- All optional parameters must appear *after* all of the required parameters in the function header.
- **Do NOT** use mutable values like lists for your optional parameters. (If you do, the code will appear to work, until it mysteriously doesn't. Feel free to ask more

about this during office hours.) Instead, use optional parameters with immutable values like integers, strings, and `None`.

Traversal orders

The `__str__` implementation we gave visits the values in the tree in a fixed order:

1. *First* it visits the root value.
2. *Then* it recursively visits each of its subtrees, in left-to-right order. (By convention, we think of the `_subtrees` list as being ordered from left to right.)

This visit order is known as the **(left-to-right) preorder** tree traversal, named for the fact that the root value is visited before any values in the subtrees. Often when this traversal is described, the "left-to-right" is omitted.

There is another common tree traversal order known as **(left-to-right) postorder**, which—you guessed it—is so named because it visits the root value *after* it has visited every value in its subtrees. Here is how we might have implemented `_str_indented` in a postorder fashion; note that the only difference is in where the root value is added to the accumulator string `s`.

```
def _str_indented_postorder(self, depth: int = 0) -> str:
    """Return an indented *postorder* string representation of this tree.

    The indentation level is specified by the <depth> parameter.
    """
    if self.is_empty():
        return ''
    else:
        s = ''
        for subtree in self._subtrees:
            # Note that the 'depth' argument to the recursive call is
         modified.
            s += subtree._str_indented_postorder(depth + 1)

        s += '  ' * depth + f'{self._root}\n'
        return s
```