12.4 Nested Lists and Structural Recursion

In the previous section, we ended by articulating a fundamental limitation of our `sum_list` functions: they cannot handle non-uniformly nested lists like

```
[[1, [2]], [[[3]]], 4, [[5, 6], [[[7]]]]]
```

In this section, we'll overcome this limitation by using a recursive approach: breaking down an object or problem into *smaller instances with the same structure as the original*.

A recursive definition for nested lists

Before we can hope to correctly implement functions that work on nested lists, we need to pin down precisely what we even mean by a nested list. Our goal is to formalize the idea of "list that contains other lists". We define a **nested list of integers** as one of two types of values:[1]

[1] This is a *structurally recursive definition*, like the kind you'll see more of in CSC236/CSC240.

- For all $n \in \mathbb{Z}$, the single integer n is a nested list of integers.
- For all $k \in \mathbb{N}$ and nested lists of integers $a_0, a_1, \dots, a_{k-1}$, the list $[a_0, a_1, \dots, a_{k-1}]$ is also a nested list of integers.

  Each of the a_i is called a *sublist* of the outer list.

It may seem a bit odd that we include "single integers" as nested lists; after all, `isinstance(3, list)` is False in Python! As we'll see a few times in this section, it is very convenient to include this part of our recursive definition, and makes both the rest of the definition and the subsequent code we'll write much more elegant.

A recursive definition for nested list sum

Our recursive definition of nested lists gives us a structure for how to recursively define *functions* that operate on nested lists. Let's return to the problem we introduced in the previous section: computing the sum of a nested list. We previously defined the sum of a nested list as "the sum of all the integers in the nested list". But there's another way to define this as a mathematical function, using notation that should look familiar to you:

$$nested\_sum(x) = \begin{cases} x, & \text{if } x \in \mathbb{Z} \\ \displaystyle\sum_{i=0}^{k-1} nested\_sum(a_i), & \text{if } x = [a_0, a_1, \dots, a_{k-1}] \end{cases}$$

In other words, the sum of a nested list that's an integer is simply the value of that integer itself. The sum of a nested list of the form $[a_0, a_1, \dots, a_{k-1}]$ is equal to the sum of each of the a_i's added together.

Just as we saw in Section 12.2, we can take this mathematical definition and traslate it naturally into a Python function. We'll show two ways of doing so, using a loop (just like the `sum_list` functions from the previous section), and using a comprehension, which more closely mimics the mathematical notation.

```python
def sum_nested_v1(nested_list: Union[int, list]) -> int:
    """Return the sum of the given nested list.

    This version uses a loop to accumulate the sum of the sublists.
    """
    if isinstance(nested_list, int):
        return nested_list
    else:
        sum_so_far = 0
        for sublist in nested_list:
            sum_so_far += sum_nested_v1(sublist)
        return sum_so_far


def sum_nested_v2(nested_list: Union[int, list]) -> int:
    """Return the sum of the given nested list.

    This version uses a comprehension and the built-in sum aggregation
        function.
    """
    if isinstance(nested_list, int):
        return nested_list
    else:
        return sum(sum_nested_v2(sublist) for sublist in nested_list)
```

Both versions of this function have the same structure as a typical recursive function, but the details differ from the numeric functions we looked at earlier. Now, our base case is not when `n == 0`, but rather when `isinstance(nested_list, int)`. The recursive step makes not just one recursive call, but many, depending on the number of sublists in `nested_list`.

This base case and recursive step structure didn't just come out of nowhere: it follows the mathematical definition of *nested_sum* from above, which in turn was informed by the recursive definition of nested lists themselves. *The structure of the data informs the structure of our code*. Just as the definition of nested lists separates integers and lists of nested lists into two cases, so too do the functions `sum_nested_v1` and `sum_nested_v2`. And because a nested list can have an arbitrary number of sublists, we use a loop/comprehension to make a recursive call on *each* sublist and aggregate the results.

Inductive reasoning revisited

Because the recursive step for these functions are more complex than previous examples, let's take a moment to review how to reason this code using the *inductive approach* or *partial tracing*.

For example, suppose we want to trace the call:[2]

[2] We'll use the loop-based version in this example, but similar reasoning applies to the comprehension version as well.

```
>>> sum_nested_v1([1, [2, [3, 4], 5], [6, 7], 8])
36
```

In this case, the recursive step executes, which is the code shown below:

```
... # Above code omitted
else:
    sum_so_far = 0
    for sublist in nested_list:
        sum_so_far += sum_nested_v1(sublist)
    return sum_so_far
```

Because there is more than one recursive call, and these calls happen during a loop, we'll set up a *loop accumulation table* to keep track of what's going on. We'll use the loop accumulation table structure we saw in Section 4.4, but with one additional column added to represent the return value of the recursive call:

Iteration	`sublist`	`sum_nested_v1(sublist)`	Accumulator `sum_so_far`
0	N/A	N/A	`0`
1	`1`		
2	`[2, [3, 4], 5]`		
3	`[6, 7]`		
4	`8`		

Now, rather than tracing the loop and accumulator row by row, we will *assume* that each recursive call is correct, and use this assumption to fill in the third column of the table directly. Note that this assumption depends only on the specification of `sum_nested_v1` found in its docstring, and *not* the actual implementation.

Iteration	`sublist`	`sum_nested_v1(sublist)`	Accumulator `sum_so_far`
0	N/A	N/A	`0`
1	`1`	`1`	
2	`[2, [3, 4], 5]`	`14` (`2 + 3 + 4 + 5`)	
3	`[6, 7]`	`13` (`6 + 7`)	
4	`8`	`8`	

Finally, we trace through the loop, updating the accumulator `sum_so_far` using the values of the recursive calls we wrote down. Remember that the accumulator column shows the value of `sum_so_far` at the *end* of that row's iteration; the final entry shows the value of `sum_so_far` after the loop is complete.

Iteration	`sublist`	`sum_nested_v1(sublist)`	Accumulator `sum_so_far`
0	N/A	N/ A	`0`
1	`1`	`1`	`1`
2	`[2, [3, 4], 5]`	`14`	`15`
3	`[6, 7]`	`13`	`28`
4	`8`	`8`	`36`

From our table, we see that after the loop completes, the final value of `sum_so_far` is `36`, and this is the value returned by our original call to `sum_nested_v1`. It also happens to be the correct value!

Recursive function design recipe for nested lists

What we've learned in this section for `sum_nested_v1` is a general technique that can be used to design functions that operate on nested lists. Here is a general recursive function design recipe for working with nested lists:

1. Write a doctest example to illustrate the *base case* of the function, when the function is called on a single `int` value.

2. Write a doctest example to illustrate the *recursive step* of the function.

    - Pick a nested list with around 3 sublists, where at least one sublist is a single `int`, and another sublist is a `list` that contains other lists.
    - Your doctest should show the correct return value of the function for this input nested list.

3. Use the following *nested list recursion code template* to follow the recursive structure of nested lists:

```
def f(nested_list: Union[int, list]) -> ...:
    if isinstance(nested_list, int):
        ...
    else:
        accumulator = ...

        for sublist in nested_list:
            rec_value = f(sublist)
            accumulator = ... accumulator ... rec_value ...

        return accumulator
```

4. Implement the function's base case, using your first doctest example to test. Most base cases are pretty straightforward to implement, though this depends on the exact function you're writing.
5. Implement the function's recursive step by doing two things:
    - Use your second doctest example to write down the relevant sublists and recursive function calls (these are the second and third columns of the loop accumulation table we showed above). Fill in the recursive call output based on the function specification, not any code you have written!
    - Analyse the output of the recursive calls and determine how to combine them to return the correct value for the original call. This will almost certainly involve some *aggregation* of the recursive call return values.

In lecture and tutorial this week, you'll get more practice applying this design recipe to writing recursive functions that operate on nested lists.