12.3 Introduction to Nested Lists

In the previous section, we learned about recursively-defined functions with domain $\mathbb{N}$. While this is arguably the most common domain of recursively-defined functions in mathematics, in programming we see recursive functions on a variety of different data types.

Whereas recursively-defined functions over $\mathbb{N}$ naturally followed the principle of induction (going from n down to $n - 1$), we'll need to be more careful when trying to extend this technique to other data types. Our first extension will be writing recursive functions that operate on nested lists, which we'll discuss in this section and the next.

A motivating example

Consider the problem of computing the sum of a list of numbers. Easy enough:[1]

[1] Even easier to use the built-in `sum` function, but we aren't using it here because we want to illustrate a code pattern in across the next few examples.

```
def sum_list(numbers: list[int]) -> int:
    """Return the sum of the numbers in the given list.

    >>> sum_list([1, 2, 3])
    6
    """
    sum_so_far = 0
    for num in numbers:
        sum_so_far += num
    return sum_so_far
```

But what if we make the input structure a bit more complex: a list of lists of numbers? After a bit of thought, we might arrive at using a nested loop to process individual items in the nested list:

```
def sum_list2(lists_of_numbers: list[list[int]]) -> int:
    """Return the sum of the numbers in the given list of lists.

    >>> sum_list2([[1], [10, 20], [1, 2, 3]])
    37
    """
    sum_so_far = 0
    for numbers in lists_of_numbers:
        for num in numbers:
```

```
            sum_so_far += num
    return sum_so_far
```

And now what happens if we want yet another layer, and compute the sum of the items in a list of lists of lists of numbers? Some more thought leads to a "nested nested loop":

```
def sum_list3(lists_of_lists_of_numbers: list[list[list[int]]]) -> int:
    """Return the sum of the numbers in the given list of lists of lists.

    >>> sum_list3([[[1], [10, 20], [1, 2, 3]], [[2, 3], [4, 5]]])
    51
    """
    sum_so_far = 0
    for lists_of_numbers in lists_of_lists_of_numbers:
        for numbers in lists_of_numbers:
            for num in numbers:
                sum_so_far += num
    return sum_so_far
```

Of course, you see where this is going: every time we want to add a new layer of nesting to the list, we add a new layer to the `for` loop. Note that this is quite interesting from a "meta" perspective: the structure of the data is mirrored in the structure of the code which operates on it.

Simplifying using helpers

You might have noticed the duplicate code above: in fact, we can use `sum_list` as a helper for `sum_list2`, and `sum_list2` as a helper for `sum_list3`:

```
def sum_list(numbers: list[int]) -> int:
    """Return the sum of the numbers in the given list.

    >>> sum_list([1, 2, 3])
    6
    """
    sum_so_far = 0
    for num in numbers:
        sum_so_far += num
    return sum_so_far


def sum_list2(lists_of_numbers: list[list[int]]) -> int:
    """Return the sum of the numbers in the given list of lists.

    >>> sum_list2([[1], [10, 20], [1, 2, 3]])
    37
    """
```

```
    sum_so_far = 0
    for numbers in lists_of_numbers:
        # numbers is a list[int]
        sum_so_far += sum_list(numbers)
    return sum_so_far

def sum_list3(lists_of_lists_of_numbers: list[list[list[int]]]) -> int:
    """Return the sum of the numbers in the given list of lists of lists.

    >>> sum_list3([[[1], [10, 20], [1, 2, 3]], [[2, 3], [4, 5]]])
    51
    """
    sum_so_far = 0
    for lists_of_numbers in lists_of_lists_of_numbers:
        # lists_of_numbers is a list[list[int]]
        sum_so_far += sum_list2(lists_of_numbers)
    return sum_so_far
```

While this is certainly a simplification, it still does not generalize elegantly. If we wanted to implement `sum_list10`, a function which works on lists with ten levels of nesting, our only choice with this approach would be to first define `sum_list4`, `sum_list5`, etc., all the way up to `sum_list9`.

Non-uniform nesting

There is an even bigger problem: no function of this form can handle nested lists with a non-uniform level of nesting among its elements, like the list

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

Each of the `sum_list`, `sum_list2`, etc. functions operate on a list of a specific structure, requiring that each list element itself have the same level of nesting of its elements. And yet, the type of nesting in the above list should still strike you as somehow recursive in nature: each of its elements, e.g. `[1, [2]]` and `[[5, 6], [[[7]]]]`, are themselves a nested list with the same overall structure as the whole list, just a bit smaller.

It is this observation that is critical for writing a recursive function that calculates the sum of *any* nested list of integers, regardless of how many levels deep each integer is nested, and allow for non-uniform levels of nesting as well. Proceed to the next section to see this how this works!